An Optimised Algorithm for Determinisation and Completion of Finite Tree Automata

John P. Gallagher
Mai Ajspur
Bishoksan Kafle

SEPTEMBER 2014 ROSKILDE UNIVERSITY COMPUTER SCIENCE RESEARCH REPORT #145

John P. Gallagher, Mai Ajspur, and Bishoksan Kafle

Computer Science
Roskilde University
P. O. Box 260
DK–4000 Roskilde
Denmark

Telephone: +45 4674 3839
Telefax: +45 4674 3072
Internet: http://www.ruc.dk/dat_en/
E-mail: datalogi@ruc.dk

ISSN 0109–9779

Research reports are available electronically from:

http://www.ruc.dk/dat_en/research/reports/

An Optimised Algorithm for Determinisation and Completion of Finite Tree Automata*

John P. Gallagher[1,2], Mai Ajspur[1], and Bishoksan Kafle[1]

[1] Roskilde University, Denmark
[2] IMDEA Software Institute, Madrid
Email: {jpg,ajspur,kafle}@ruc.dk

Abstract. Determinisation is an important concept in the theory of finite tree automata. However the complexity of the textbook procedure for determinisation is such that it is not viewed as a being a practical procedure for manipulating tree automata, even fairly small ones. The computational problems are exacerbated when an automaton has to be both determinised and completed, for instance to compute the complement of an automaton. In this paper we develop an algorithm for determinisation and completion of finite tree automata, whose worst-case complexity remains unchanged, but which performs dramatically better than existing algorithms in practice. The algorithm is developed in stages by optimising the textbook algorithm. A critical aspect of the algorithm is that the transitions of the determinised automaton are generated in a potentially very compact form called product form, which can often be used directly when manipulating the determinised automaton. The paper contains an experimental evaluation of the algorithm on a large set of tree automata examples. Applications of the algorithm include static analysis of term rewriting systems and logic programs, and checking containment of languages defined by tree automata such as XML schemata.

1 Introduction

Finite tree automata (FTAs) are mathematical machines that define so-called recognisable tree languages, possibly infinite sets of terms that have desirable properties such as closure under Boolean set operations, and decidability of membership and emptiness. In the paper we will give a brief overview of the relevant features of FTAs, but the main goal of the paper is to focus on two operations on FTAs, namely *determinisation* and *completion*. These operations play a key role in the theory of FTAs, for example in showing that recognisable tree languages are closed under Boolean operations. Potentially, they also play a practical role in systems that manipulate sets of terms, but their complexity has so far discouraged their widespread application.

* Supported by Danish Natural Science Research Council (FNU) project 10-084290 (NUSA)

In the paper we develop an optimised algorithm for determinisation, and analyse its properties. Experiments show that it performs well, though the worst case remains intractable. We also discuss applications of finite tree automata that exploit the determinisation algorithm. In Section 2 the essentials, for our purposes, of finite tree automata are introduced. The operations of determinisation and completion are defined. Section 3 presents the optimised algorithm for determinising an FTA. It is developed in a series of stages starting from the textbook algorithm for determinisation. In Section 4 a compact representation called *product form* of the set of transitions of an FTA is described, and it is shown how the algorithm in Section 3 can output transitions in product form. The performance of the algorithm is analysed in Section 5. In Section 6 we discuss the combination of determinisation and completion of an FTA and show that the performance of the algorithm generating product form is as effective when generating a complete determinised automaton. Section 7 reports on the performance of the algorithm on a large number of example tree automata. Section 8 discusses potential applications of the algorithm, Section 9 contains a discussion of related work and finally in Section 10 we summarise the outlook for further work and applications.

2 Preliminaries

A *finite tree automaton* (FTA) is defined as a quadruple $\langle Q, Q_f, \Sigma, \Delta \rangle$, where

1. Q is a finite set called *states*,
2. $Q_f \subseteq Q$ is called the set of accepting (or final) states,
3. Σ is a set of function symbols and
4. Δ is a set of *transitions*.

Each function symbol $f \in \Sigma$ has an arity $n \geq 0$, written $\mathsf{ar}(f) = n$. Function symbols with arity 0 are called *constants*. Q and Σ are disjoint. $\mathsf{Term}(\Sigma)$ is the set of *ground terms* (also called *trees*) constructed from Σ where $t \in \mathsf{Term}(\Sigma)$ iff $t \in \Sigma$ is a constant or $t = f(t_1, \ldots, t_n)$ where $\mathsf{ar}(f) = n$ and $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma)$. Similarly $\mathsf{Term}(\Sigma \cup Q)$ is the set of terms/trees constructed from Σ and Q, treating the elements of Q as constants.

Each transition in Δ is of the form $f(q_1, \ldots, q_n) \to q$, where $\mathsf{ar}(f) = n$ and $q, q_1, \ldots, q_n \in Q$.

To define acceptance of a term by the FTA $\langle Q, Q_f, \Sigma, \Delta \rangle$ we first define a *context* for the FTA. A context is a term from $\mathsf{Term}(\Sigma \cup Q \cup \{\bullet\})$ containing exactly one occurrence of $\bullet$ (which is a constant not in Σ or Q). Let c be a context and $t \in \mathsf{Term}(\Sigma \cup Q)$; $c[t]$ denotes the term resulting from the replacement of $\bullet$ in c by t.

The binary relation $\Rightarrow$ represents one step of a run for the FTA. It is defined as follows; $c[l] \Rightarrow c[r]$ iff c is a context and $l \to r \in \Delta$. The reflexive, transitive closure of $\Rightarrow$ is denoted $\Rightarrow^*$.

A run for $t \in \mathsf{Term}(\Sigma)$ exists if $t \Rightarrow^* q$ where $q \in Q$. The run is *successful* if $q \in Q_f$ and in this case t is *accepted* by the FTA. A tree automaton R defines a

set of terms, that is, a tree language, denoted $L(R)$, as the set of all terms that it accepts. We also write $L(q)$ to be the set of terms t such that $t \Rightarrow^* q$ in a given FTA.

Definition 1. *An FTA $\langle Q, Q_f, \Sigma, \Delta\rangle$ is called* bottom-up deterministic *if and only if Δ contains no two transitions with the same left hand side. A bottom-up deterministic FTA is abbreviated as a DFTA.*

Runs of a DFTA are deterministic in the following sense; for every context c and term of form $c[t]$ there is at most one term $c[t']$ such that $c[t] \Rightarrow c[t']$. It follows that for every $t \in \mathsf{Term}(\Sigma)$ there is at most one $q \in Q$ such that $t \Rightarrow^* q$. As far as expressiveness is concerned we can limit our attention to DFTAs. For every FTA R there exists a DFTA R' such that $L(R) = L(R')$[3].

Definition 2. *An automaton $R = \langle Q, Q_f, \Sigma, \Delta\rangle$ is* complete *if for all n-ary functions $f \in \Sigma$ and states $q_1, \ldots, q_n \in Q$, there exists a state q such that $f(q_1, \ldots, q_n) \to q \in \Delta$.*

It follows that in a complete FTA every term t has at least one run and furthermore since in a complete DFTA each t has a run to exactly one state, a complete DFTA defines a partition of $\mathsf{Term}(\Sigma)$, namely $\{L(q) \mid q \in Q\} \setminus \emptyset$.

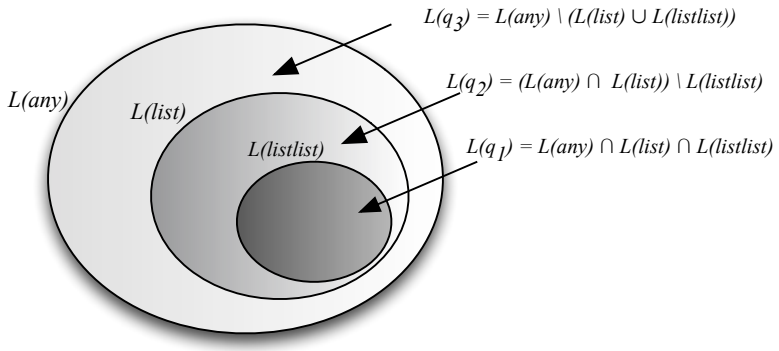

Fig. 1. The disjoint languages from Example 1

Definition 3. *Let Σ be any signature and* any *a state. We define $\Delta^{\Sigma}_{\mathsf{any}}$ to be the following set of transitions.*

$$\{f(\overbrace{\mathsf{any}, \ldots, \mathsf{any}}^{n \text{ times}}) \to \mathsf{any} \mid f^n \in \Sigma\}$$

[3] FTAs are sometimes denoted NFTAs in the literature where N stands for nondeterministic.

procedure FTA DETERMINISATION (Input: $\langle Q, \Sigma, Q_f, \Delta \rangle$)
  $\mathcal{Q}_d \leftarrow \emptyset$
  $\Delta_d \leftarrow \emptyset$
  repeat
    $\mathcal{Q}_d \leftarrow \mathcal{Q}_d \cup \{Q_0\}$,
    $\Delta_d \leftarrow \Delta_d \cup \{f(Q_1, \ldots, Q_n) \to Q_0\}$
    where
      $f^n \in \Sigma$, $Q_1, \ldots, Q_n \in \mathcal{Q}_d$,
      $Q_0 = \{q_0 \mid \exists q_1 \in Q_1, \ldots, q_n \in Q_n, (f(q_1, \ldots, q_n) \to q_0) \in \Delta\}$
  until no rules can be added to Δ_d
  $\mathcal{Q}_f \leftarrow \{Q' \in \mathcal{Q}_d \mid Q' \cap Q_f \neq \emptyset\}$
  return $(\mathcal{Q}_d, \Sigma, \mathcal{Q}_f, \Delta_d)$
end procedure

Fig. 2. Textbook Determinisation Algorithm

Clearly, given an FTA $\langle Q, Q_f, \Sigma, \Delta \rangle$ *with* $\mathsf{any} \in Q$ *and* $\Delta^{\Sigma}_{\mathsf{any}} \subseteq \Delta$*, there is a run* $t \Rightarrow^* \mathsf{any}$ *for any* $t \in \mathsf{Term}(\Sigma)$*, that is,* $L(\mathsf{any}) = \mathsf{Term}(\Sigma)$.

We normally drop the superscript in $\Delta^{\Sigma}_{\mathsf{any}}$ as Σ is usually clear from the context.

Example 1. Let $\Sigma = \{[], [.|.], 0\}$, $Q = \{list, listlist, \mathsf{any}\}$, $Q_f = \{list, listlist\}$ and $\Delta = \{[] \to list, [\mathsf{any}|list] \to list, [] \to listlist, [list|listlist] \to listlist\} \cup \Delta_{\mathsf{any}}$. $L(list)$ is the set of lists of any terms, while $L(listlist)$ is the set of lists whose elements are themselves lists. Clearly $L(listlist)$ is contained in $L(list)$, which is contained in $L(\mathsf{any})$.

The automaton is not bottom-up deterministic; a determinisation algorithm (see Section 3) yields the DFTA $\langle Q', Q'_f, \Sigma, \Delta' \rangle$, where $Q' = \{q_1, q_2, q_3\}$, $Q'_f = \{q_1, q_2\}$ and $\Delta' = \{[] \to q_1, [q_1|q_1] \to q_1, [q_2|q_1] \to q_1, [q_1|q_2] \to q_2, [q_2|q_2] \to q_2, [q_3|q_2] \to q_2, [q_3|q_1] \to q_2, [q_2|q_3] \to q_3, [q_1|q_3] \to q_3, [q_3|q_3] \to q_3, 0 \to q_3\}$. The states q_1, q_2 and q_3 are abbreviations for elements of the powerset of the states of the original FTA; here $q_1 = \{\mathsf{any}, list, listlist\}$, $q_2 = \{\mathsf{any}, list\}$ and $q_3 = \{\mathsf{any}\}$. This automaton is also complete. □

In Example 1, $L(q_1) = L(\mathsf{any}) \cap L(list) \cap L(listlist)$, $L(q_2) = (L(list) \cap L(\mathsf{any})) \setminus L(listlist)$, and $L(q_3) = L(\mathsf{any}) \setminus (L(list) \cup L(listlist))$. The relationship between the languages corresponding to the FTA and DFTA states in Example 1 is shown in Figure 1.

3 Development of an Optimised Determinisation Algorithm

In this section we present the textbook algorithm for FTA determinisation, and then proceed to optimise it. The determinisation algorithm in Figure 2 is the one presented (apart from some renaming of variables) in [2]. In each of the figures from Figure 2 to Figure 11 we show successive modifications of the algorithm, where the changed lines are marked on the right hand side.

```
procedure FTA Determinisation (Input: ⟨Q, Σ, Q_f, Δ⟩)
    Q_d ← ∅
    Δ_d ← ∅
    repeat
        Q_d^old ← Q_d                                                    ▷
        Δ_d^old ← Δ_d                                                    ▷
        for all f ∈ Σ do                                                 ▷
            for all (Q_1, ... Q_n) ∈ (Q_d^old × ··· × Q_d^old) do        ▷
                Q_0 = {q_0 | ∃q_1 ∈ Q_1, ..., q_n ∈ Q_n, (f(q_1, ..., q_n) → q_0) ∈ Δ}
                if Q_0 ≠ ∅ then                                          ▷
                    Q_d ← Q_d ∪ {Q_0}
                    Δ_d ← Δ_d ∪ {f(Q_1, ..., Q_n) → Q_0}
                end if                                                   ▷
            end for                                                      ▷
        end for                                                          ▷
    until Δ_d = Δ_d^old                                                  ▷
    Q_f ← {Q' ∈ Q_d | Q' ∩ Q_f ≠ ∅}
    return (Q_d, Σ, Q_f, Δ_d)
end procedure
```

Fig. 3. Restructured algorithm with explicit iteration (Step 1)

We note first a small ambiguity in the algorithm as presented in [2]. In the assignment $Q_0 = \{q_0 \mid \exists q_1 \in Q_1, \ldots, q_n \in Q_n, (f(q_1, \ldots, q_n) \to q_0) \in \Delta\}$ the right hand side is implicitly assumed to evaluate to a non-empty set, otherwise it is ignored. Although allowing the variable Q_0 to take the value $\emptyset$ would return a correct result, many redundant transitions of the form $f(Q_1, \ldots, Q_n) \to \emptyset$ would be generated. In our transformed algorithm we make this assumption explicit and eliminate such transitions.

Note that the states of the computed DFTA are elements of 2^Q where Q is the set of states of the input FTA.

3.1 Step 1: Minor restructuring

First we apply some minor restructuring to the algorithm. The **repeat** ... **where** ... **until** loop is rewritten to iterate explicitly over Σ and $\mathcal{Q}_d$, and the termination condition "no rules can be added to Δ_d" is rewritten to compare values of Δ_d on successive iterations. Using this restructuring we obtain the algorithm in Figure 3.

3.2 Step 2: Introduction of functional notation

Let $\langle Q, \Sigma, Q_f, \Delta \rangle$ be an FTA. Let $t = f(q_1, \ldots, q_n) \to q, n \geq 0$ be a transition in Δ. Define the following selector functions on t.

$\mathsf{rhs} : \Delta \to Q$	$\mathsf{lhs}_i : \Delta \hookrightarrow Q$	$\mathsf{func} : \Delta \to Q$
$\mathsf{rhs}(t) = q$	$\mathsf{lhs}_i(t) = q_i, 1 \leq i \leq n$	$\mathsf{func}(t) = f$

The lhs_i functions are partial functions on Δ since lhs_i is not defined for every transition for a given i. In particular the lhs_i functions are undefined on transitions whose function symbol has arity zero.

The inverse mappings $\mathsf{lhs}_i^{-1} : Q \to 2^{\Delta}$ and $\mathsf{func}^{-1} : \Sigma \to 2^{\Delta}$ are defined respectively as $\mathsf{lhs}_i^{-1}(q) = \{t \mid \mathsf{lhs}_i(t) = q\}$, $\mathsf{func}^{-1}(f) = \{t \mid \mathsf{func}(t) = f\}$. Using these, $\mathsf{lhsf}_i : (\Sigma \times Q) \to 2^{\Delta}$ is defined as $\mathsf{lhsf}_i(f, q) = \mathsf{lhs}_i^{-1}(q) \cap \mathsf{func}^{-1}(f)$.

$\mathsf{lhsf}_i(f, q)$ can be regarded as an index for Δ returning the set of transitions whose function symbol is f and whose left hand side has q in the i^{th} position. The mappings lhsf_i are lifted to sets of states, giving Lhsf_i defined as follows.

$$\mathsf{Lhsf}_i : (\Sigma \times 2^Q) \to 2^{\Delta}$$
$$\mathsf{Lhsf}_i(f, S) = \bigcup_{s \in S} \mathsf{lhsf}_i(f, s)$$

We also lift rhs to sets of transitions, giving the function $\mathsf{Rhs} : 2^{\Delta} \to 2^Q$, where $\mathsf{Rhs}(T) = \{\mathsf{rhs}(t) \mid t \in T\}$.

We now apply the notation introduced above, and use the following property, obtaining the result shown in Figure 4.

Property 1. The following expressions are equal for all $f \in \Sigma$ and $Q_1, \ldots Q_n \in 2^Q$.

- $\{q_0 \mid \exists q_1 \in Q_1, \ldots, q_n \in Q_n, (f(q_1, \ldots, q_n) \to q_0) \in \Delta\}$
- if $\mathsf{ar}(f) = 0$ then $\mathsf{Rhs}(\mathsf{func}^{-1}(f))$ else $\mathsf{Rhs}(\mathsf{Lhsf}_1(f, Q_1) \cap \cdots \cap \mathsf{Lhsf}_n(f, Q_n))$

Proof. Application of the definitions of Rhs, Lhsf_i and set operations. □

3.3 Step 3: Modifying the termination condition, and delaying computation of transitions

Examining the loop beginning "**for all** $f \in \Sigma$ ", we observe that the values of Δ_d^{old} and $\mathcal{Q}_d^{old}$ are assigned to Δ_d and $\mathcal{Q}_d$ respectively just before the loop, and the values of Δ_d^{old} and $\mathcal{Q}_d^{old}$ do not change in the body of the **for all** loop. By a simple dependency analysis we can establish that the values of Δ_d and $\mathcal{Q}_d$ at the end of the loop body depend only on the value of $\mathcal{Q}_d^{old}$ at the start of the loop body. Furthermore the values of Δ_d and $\mathcal{Q}_d$ are only incremented in the loop body. Thus we can assert the following invariants immediately before the **until** statement:

$$\Delta_d = \Delta_d^{old} \ \cup \ \phi_1(\mathcal{Q}_d^{old}) \tag{1}$$
$$\mathcal{Q}_d = \mathcal{Q}_d^{old} \ \cup \ \phi_2(\mathcal{Q}_d^{old}) \tag{2}$$

where $\phi_1 : 2^{2^Q} \to 2^{\Delta}$ and $\phi_2 : 2^{2^Q} \to 2^{2^Q}$ represent the loop body, projected onto Δ_d and $\mathcal{Q}_d$ respectively.

Rename the values of $\mathcal{Q}_d$ and Δ_d on the i^{th} iteration ($i = 1, 2, \ldots$) of the **repeat** loop, just before the **until** statement, as $\mathcal{Q}_d^i$ and Δ_d^i respectively. Assume that $\mathcal{Q}_d^0 = \Delta_d^0 = \emptyset$. $\mathcal{Q}_d^{old}$ on the i^{th} iteration is the same as the final value of

```
procedure FTA Determinisation (Input: ⟨Q, Σ, Q_f, Δ⟩)
    Q_d ← ∅
    Δ_d ← ∅
    repeat
        Q_d^old ← Q_d
        Δ_d^old ← Δ_d
        for all f ∈ Σ do
            for all (Q_1, ... Q_n) ∈ (Q_d^old × ··· × Q_d^old) do
                if (ar(f) = 0) then
                    Q_0 ← Rhs(func^{-1}(f))                          ▷
                else
                    Q_0 ← Rhs(Lhsf_1(f, Q_1) ∩ ··· ∩ Lhsf_n(f, Q_n))  ▷
                end if
                if Q_0 ≠ ∅ then
                    Q_d ← Q_d ∪ {Q_0}
                    Δ_d ← Δ_d ∪ {f(Q_1, ..., Q_n) → Q_0}
                end if
            end for
        end for
    until Δ_d = Δ_d^old
    Q_f ← {Q' ∈ Q_d | Q' ∩ Q_f ≠ ∅}
    return (Q_d, Σ, Q_f, Δ_d)
end procedure
```

Fig. 4. Algorithm after applying Property 1 (Step 2)

$\mathcal{Q}_d$ on the previous iteration, that is, $\mathcal{Q}_d^{i-1}$; similarly $\Delta_d^{i-1} = \Delta_d^{old}$. With this notation, (1) and (2) are rewritten as

$$\Delta_d^i = \Delta_d^{i-1} \cup \phi_1(\mathcal{Q}_d^{i-1}) \tag{3}$$

$$\mathcal{Q}_d^i = \mathcal{Q}_d^{i-1} \cup \phi_2(\mathcal{Q}_d^{i-1}) \tag{4}$$

We next show that the following invariants hold.

Property 2 (Loop Invariant). The following assertions hold just before the **until** statement in Figure 4.

$$\Delta_d^i = \phi_1(\mathcal{Q}_d^{i-1})$$
$$\mathcal{Q}_d^i = \phi_2(\mathcal{Q}_d^{i-1}).$$

Proof. ϕ_1 and ϕ_2 are monotonic functions and $\mathcal{Q}_d^i \subseteq \mathcal{Q}_d^{i+1}$, $i = 0, 1, \ldots$ since the value of $\mathcal{Q}_d$ is only incremented in the algorithm. The proof is by induction on i; for Δ_d^i, the base case $i = 1$ holds using (3) and the fact that $\Delta_d^0 = \emptyset$. In the inductive step, assume that $\Delta_d^i = \phi_1(\mathcal{Q}_d^{i-1})$ and show that $\Delta_d^{i+1} = \phi_1(\mathcal{Q}_d^i)$. We have

$$\begin{aligned}\Delta_d^{i+1} &= \Delta_d^i \cup \phi_1(\mathcal{Q}_d^i), \text{by (3)} \\ &= \phi_1(\mathcal{Q}_d^{i-1}) \cup \phi_1(\mathcal{Q}_d^i), \text{by ind. hyp.} \\ &= \phi_1(\mathcal{Q}_d^i), \text{since } \mathcal{Q}_d^{i-1} \subseteq \mathcal{Q}_d^i \text{ and } \phi_1 \text{ is monotonic}\end{aligned}$$

□

The proof of the invariant for $\mathcal{Q}_d^i$ is similar.

Corollary 1. *For all* $i > 0$, $\mathcal{Q}_d^{i-1} = \mathcal{Q}_d^i \Rightarrow (\Delta_d^i = \Delta_d^{i+1} \wedge \mathcal{Q}_d^i = \mathcal{Q}_d^{i+1})$.

The corollary states that if the value of $\mathcal{Q}_d$ stabilises on the i^{th} iteration, that is, $\mathcal{Q}_d^{i-1} = \mathcal{Q}_d^i$ then the value of Δ_d stabilises at the latest on the $i{+}1^{th}$ iteration. We exploit this fact to modify the termination condition to $\mathcal{Q}_d = \mathcal{Q}_d^{old}$, and possibly reduce the number of iterations by 1. In this case the set of transitions might be incomplete when the **repeat** loop terminates, but since $\Delta_d^i = \phi_1(\mathcal{Q}_d^{i-1})$, that is, Δ_d depends only on the final value of $\mathcal{Q}_d$, we can remove the computation of Δ_d from the **repeat** loop entirely, delaying it until after the termination of the loop. We return to this point in Section 3.7.

3.4 Step 4: Separate handling of 0-arity functions

The processing of 0-arity functions depends only on the original FTA, and so can be precomputed before entering the **repeat** loop. The next stage of the transformed algorithm after applying Steps 3 and 4 is displayed in Figure 5.

The transformations so far are fairly superficial and have little bearing on the efficiency of the algorithm. However they enable us to focus on the iterations of the inner **for all** loop, with a view to more substantial efficiency improvements.

3.5 Step 5: Inner Loop Optimisation

The fact that we no longer need to compute transitions in the inner loop can lead to major savings since we can focus on the computation of $\mathcal{Q}_d$. Let us suppose that $|\mathcal{Q}_d^{old}| = k$. Then for a function symbol f of arity n, there are k^n tuples $(Q_1, \ldots Q_n)$ in the cartesian product $(\mathcal{Q}_d^{old} \times \cdots \times \mathcal{Q}_d^{old})$ and so the function $\mathsf{Lhsf}_i(f, Q_j)$ is called $n * k^n$ times. On the other hand, within the loop there are only $k * n$ different calls of the form $\mathsf{Lhsf}_i(f, Q_j)$ and therefore it is worth precomputing these $k * n$ values outside the loop and avoid recomputing the same call many times. Furthermore, cases of $\mathsf{Lhsf}_i(f, Q_j)$ that evaluate to the empty set can be ignored since they cannot contribute to a non-empty value of Q_0 within the loop, since $\mathsf{Rhs}(\emptyset) = \emptyset$.

We precompute the $\mathsf{Lhsf}_i(f, Q_j)$ values by introducing a function called $\mathcal{T}_i : (\Sigma \times 2^{2^Q}) \rightarrow 2^{2^\Delta}$ defined as

$$\mathcal{T}_i(f, \mathcal{Q}') = \{\mathsf{Lhsf}_i(f, Q') \mid Q' \in \mathcal{Q}'\} \setminus \emptyset.$$

```
procedure FTA Determinisation (Input: ⟨Q, Σ, Q_f, Δ⟩)
    Q_d ← ∅
    for all f ∈ Σ do                                                    ▷
        if (ar(f) = 0) then                                             ▷
            Q_0 ← Rhs(func^{-1}(f))                                     ▷
            if Q_0 ≠ ∅ then                                             ▷
                Q_d ← Q_d ∪ {Q_0}                                       ▷
            end if                                                      ▷
        end if                                                          ▷
    end for                                                             ▷
    repeat
        Q_d^old ← Q_d
        for all f ∈ Σ do
            if (ar(f) > 0) then
                for all (Q_1, ... Q_n) ∈ (Q_d^old × ··· × Q_d^old) do
                    Q_0 ← Rhs(Lhsf_1(f, Q_1) ∩ ··· ∩ Lhsf_n(f, Q_n))
                    if Q_0 ≠ ∅ then
                        Q_d ← Q_d ∪ {Q_0}
                    end if
                end for
            end if
        end for
    until Q_d = Q_d^old                                                 ▷
    Compute the set of transitions Δ_d (see Section 3.7)                ▷
    Q_f ← {Q' ∈ Q_d | Q' ∩ Q_f ≠ ∅}
    return (Q_d, Σ, Q_f, Δ_d)
end procedure
```

Fig. 5. Algorithm after Steps 3 and 4

This function is defined for $1 \leq i \leq n$ for a function of arity n. The inner **for all** loop is then rewritten to iterate over tuples of sets of transitions chosen from the product $\mathcal{T}_1(f, \mathcal{Q}_d^{old}) \times \cdots \times \mathcal{T}_n(f, \mathcal{Q}_d^{old})$ instead of $(\mathcal{Q}_d^{old} \times \cdots \times \mathcal{Q}_d^{old})$. It is clear that exactly the same non-empty values of Q_0 are generated within the loop; we are just choosing precomputed values of $\mathsf{Lhsf}_i(f, Q_j)$ from the $\mathcal{T}_i(f, \mathcal{Q}_d^{old})$ sets, which also omits the empty values. Applying this transformation, we get the result shown in Figure 6.

The transformation of the inner loop is significant in typical applications. Instead of k^n iterations of the loop, where $k = |\mathcal{Q}_d^{old}|$, there are $\prod_{i=i}^{n} |\mathcal{T}_i(f, \mathcal{Q}_d^{old})|$ iterations which is usually much smaller. Note that in many FTAs the size of the set $\mathcal{T}_i(f, \mathcal{Q}_d^{old})$ is usually much smaller than k (and is often zero) since the states of the input automaton tend to appear in only a few argument positions of function symbols.

```
procedure FTA Determinisation (Input: ⟨Q, Σ, Q_f, Δ⟩)
    Q_d ← ∅
    for all f ∈ Σ do
        if (ar(f) = 0) then
            Q_0 ← Rhs(func^{-1}(f))
            if Q_0 ≠ ∅ then
                Q_d ← Q_d ∪ {Q_0}
            end if
        end if
    end for
    repeat
        Q_d^{old} ← Q_d
        for all f ∈ Σ do
            if (ar(f) > 0) then
                (Ψ_1, ..., Ψ_n) ← (T_1(f, Q_d^{old}), ..., T_n(f, Q_d^{old}))      ▷
                for all (Δ_1, ... Δ_n) ∈ (Ψ_1 × ··· × Ψ_n) do                      ▷
                    Q_0 ← Rhs(Δ_1 ∩ ··· ∩ Δ_n)                                    ▷
                    if Q_0 ≠ ∅ then
                        Q_d ← Q_d ∪ {Q_0}
                    end if
                end for
            end if
        end for
    until Q_d = Q_d^{old}
    Compute the set of transitions Δ_d (see Section 3.7)
    Q_f ← {Q' ∈ Q_d | Q' ∩ Q_f ≠ ∅}
    return (Q_d, Σ, Q_f, Δ_d)
end procedure
```

Fig. 6. Algorithm after Step 5

3.6 Step 6: Tracking new values on each iteration

We now apply an optimisation that further reduces the computation in the innermost loop. As it stands in Figure 6, any value of Q_0 generated on some iteration is also generated on all subsequent iterations of the **repeat** loop, since elements are added to $\mathcal{Q}_d$ but never removed. To avoid this, we note that when evaluating the statement $Q_0 \leftarrow \mathsf{Rhs}(\Delta_1 \cap \cdots \cap \Delta_n)$ in some iteration of the **repeat** loop, a new value is obtained for Q_0 only when at least one of $\Delta_1, \ldots, \Delta_n$ is a new value, that is, one that was not available on the previous iteration. We therefore try to avoid re-evaluating old values of Δ_i for each i.

Some bookkeeping is needed to keep track of new values. A variable $\mathcal{Q}_d^{new}$ represents the new elements of $\mathcal{Q}_d$ produced on some iteration. (The termination condition of the **repeat** loop is altered to $\mathcal{Q}_d^{new} = \emptyset$). We introduce variables Ψ_i^f, which has the value of $\mathcal{T}_i(f, \mathcal{Q}_d)$. The variables are initialised to $\emptyset$ and their values are augmented on each iteration. The statement $(\Phi_1, \ldots, \Phi_n) \leftarrow$

$(\mathcal{T}_1(f, \mathcal{Q}_d^{new}) \setminus \Psi_1^f, \ldots, \mathcal{T}_n(f, \mathcal{Q}_d^{new}) \setminus \Psi_n^f)$ computes the new sets of transitions from which the Δ_i sets can be chosen.

The innermost **for all** statement now iterates over the union of sets of tuples Z where $Z = \bigcup_{i=1}^{n}(\Psi_1^f \times \cdots \times \Psi_{i-1}^f \times \Phi_i \times \Psi_{i+1}^{f,new} \times \cdots \times \Psi_n^{f,new})$, which consists of exactly those tuples which contain at least one new value (that is, from one of the Φ_i). Note in particular that if no new values for any argument are produced for some f on some iteration, then the iteration set for f on the next iteration is empty, since all the variables $(\Phi_1, \ldots, \Phi_n)$ have the value $\emptyset$.

```
procedure FTA Determinisation (Input: ⟨Q, Σ, Q_f, Δ⟩)
    Q_d ← ∅
    for all f ∈ Σ do
        if (ar(f) = 0) then
            Q_0 ← Rhs(func^{-1}(f))
            if Q_0 ≠ ∅ then
                Q_d ← Q_d ∪ {Q_0}
            end if
        end if
        Ψ_1^f, ..., Ψ_n^f = ∅, ..., ∅                                   ▷
    end for
    Q_d^{new} ← Q_d                                                     ▷
    repeat
        Q_d^{old} ← Q_d
        for all f ∈ Σ do
            if (ar(f) > 0) then
                (Φ_1, ..., Φ_n) ← (T_1(f, Q_d^{new}) \ Ψ_1^f, ..., T_n(f, Q_d^{new}) \ Ψ_n^f)   ▷
                (Ψ_1^{f,new}, ..., Ψ_n^{f,new}) ← (Ψ_1^f ∪ Φ_1, ..., Ψ_n^f ∪ Φ_n)               ▷
                Z ← ⋃_{i=1}^{n}(Ψ_1^f × ··· × Ψ_{i-1}^f × Φ_i × Ψ_{i+1}^{f,new} × ··· × Ψ_n^{f,new})   ▷
                for all (Δ_1, ... Δ_n) ∈ Z do                           ▷
                    Q_0 ← Rhs(Δ_1 ∩ ··· ∩ Δ_n)
                    if Q_0 ≠ ∅ then
                        Q_d ← Q_d ∪ {Q_0}
                    end if
                end for
            end if
            (Ψ_1^f, ..., Ψ_n^f) ← (Ψ_1^{f,new}, ..., Ψ_n^{f,new})       ▷
        end for
        Q_d^{new} ← Q_d \ Q_d^{old}
    until Q_d^{new} = ∅                                                 ▷
    Compute the set of transitions Δ_d (see Section 3.7)
    Q_f ← {Q' ∈ Q_d | Q' ∩ Q_f ≠ ∅}
    return (Q_d, Σ, Q_f, Δ_d)
end procedure
```

Fig. 7. Algorithm after Step 6

3.7 Computing the transitions of the DFTA

As noted in Section 3.3, the set of transitions Δ_d of the determinised automaton can be computed from the final set of states $\mathcal{Q}_d$ and the functions Rhs, Lhsf and func^{-1} defined on the original FTA transitions Δ. Again, we start from a naive computation and proceed to transform it. The code in Figure 8 arises straightforwardly by extracting the computation of the transitions from within the **repeat** loop of the original algorithm (see Figure 4).

```
Δ_d ← ∅
for all f ∈ Σ do
    for all (Q_1, ... Q_n) ∈ (Q_d × ··· × Q_d) do
        if (ar(f) = 0) then
            Q_0 ← Rhs(func^{-1}(f))
        else
            Q_0 ← Rhs(Lhsf_1(f, Q_1) ∩ ··· ∩ Lhsf_n(f, Q_n))
        end if
        if Q_0 ≠ ∅ then
            Δ_d ← Δ_d ∪ {f(Q_1, ..., Q_n) → Q_0}
        end if
    end for
end for
```

Fig. 8. Computation of transitions I

We can now repeat the optimisation discussed in Section 3.5 above, in order to precompute the values of $\mathsf{Lhsf}_i(f, Q_i)$ using the function $\mathcal{T}_i(f, \mathcal{Q}) = \{\mathsf{Lhsf}_i(Q, f) \mid Q \in \mathcal{Q}\} \setminus \emptyset$, thus reducing $n * k^n$ calls to Lhsf_i for an n-ary function f to $k * n$ calls, where $k = |\mathcal{Q}|$.

We now turn to the question of how to return the final set of transitions Δ_d. If we want an explicit set of transitions, then we need to generate the tuples $(Q_1, \ldots, Q_n)$ which are used to build the left hand side of a transition inside the loop. Given an n-ary function f and a tuple of sets of transitions $(\Delta_1, \ldots \Delta_n)$, we want to generate all tuples $(Q_1, \ldots, Q_n)$ such that $\mathsf{Lhsf}_i(f, Q_i) = \Delta_i$, for $1 \leq i \leq n$. To do this we use an "inverse" function $\mathcal{T}_i^{-1} : (\Sigma \times 2^{2^Q} \times 2^{\Delta}) \to 2^{2^Q}$, defined as follows.

$$\mathcal{T}_i^{-1}(f, \mathcal{Q}, \Delta') = \{Q' \mid \mathsf{Lhsf}_i(f, Q') = \Delta', Q' \in \mathcal{Q}\}$$

This leads to the code in Figure 9 for generating transitions.

However, enumerating the final set Δ_d, which is often far larger than the original set of transitions of the FTA, could nullify the optimisations already presented. In particular, when the DFTA is complete the size of the transition set explodes. For a function symbol f of arity n, and set of DFTA states of size k, there are k^n transitions for f in the complete DFTA. In practice this means that explicitly generating the transitions is prohibitively expensive except for automata with low-arity function symbols.

```
Δ_d ← ∅
for all f ∈ Σ do
    if (ar(f) = 0) then
        Q_0 ← Rhs(func^{-1}(f))
        if Q_0 ≠ ∅ then
            Δ_d ← Δ_d ∪ {f → Q_0}                                   ▷
        end if
    else
        n = ar(f)
        Ψ_1^f, ..., Ψ_n^f = T_1(f, Q_d), ..., T_n(f, Q_d)           ▷
        for all (Δ_1, ... Δ_n) ∈ (Ψ_1^f × ··· × Ψ_n^f) do           ▷
            Q_0 ← Rhs(Δ_1 ∩ ··· ∩ Δ_n)                              ▷
            if Q_0 ≠ ∅ then
                Q_1, ..., Q_n = T_1^{-1}(f, Q_d, Δ_1), ..., T_n^{-1}(f, Q_d, Δ_n)   ▷
                for all (Q_1, ... Q_n) ∈ (Q_1 × ··· × Q_n) do       ▷
                    Δ_d ← Δ_d ∪ {f(Q_1, ..., Q_n) → Q_0}
                end for                                             ▷
            end if
        end for
    end if
end for
```

Fig. 9. Optimised computation of transitions II

4 Product representation sets of transitions

A *product transition* is of the form $f(Q_1, \ldots, Q_n) \to q$ where $Q_1, \ldots, Q_n$ are sets of states and q is a state. This product transition denotes the set of transitions $\{f(q_1, \ldots, q_n) \to q \mid q_1 \in Q_1, \ldots, q_n \in Q_n\}$. Thus $\prod_{i=1}^{n} |Q_i|$ transitions are represented by a single product transition. Alternatively, we can regard a product transition as introducing ϵ-transitions. An ϵ-transition has the form $q_1 \to q_2$ where q_1, q_2 are states. ϵ-transitions can be eliminated, if desired. Given a product transition $f(Q_1, \ldots, Q_n) \to q$, introduce n new non-final states $s_1, \ldots, s_n$ corresponding to $Q_1, \ldots, Q_n$ respectively and replace the product transition by the set of transitions $\{f(s_1, \ldots, s_n) \to q\} \cup \{q' \to s_i \mid q' \in Q_i, i = 1..n\}$. It can be shown that this transformation preserves the language of the FTA.

Example 2. The transitions of the DFTA generated in Example 1 can be represented in product transition form as follows.

$$\begin{aligned}
\Delta' = \{[] \to q_1 \quad & 0 \to q_1 \\
[\{q_1, q_2\}|\{q_1\}] \to q_1 \quad & [\{q_3\}|\{q_1\}] \to q_2 \\
[\{q_1, q_2\}|\{q_3\}] \to q_3 \quad & [\{q_3\}|\{q_3\}] \to q_3 \\
[\{q_1, q_2\}|\{q_2\}] \to q_2 \quad & [\{q_3\}|\{q_2\}] \to q_2\}
\end{aligned}$$

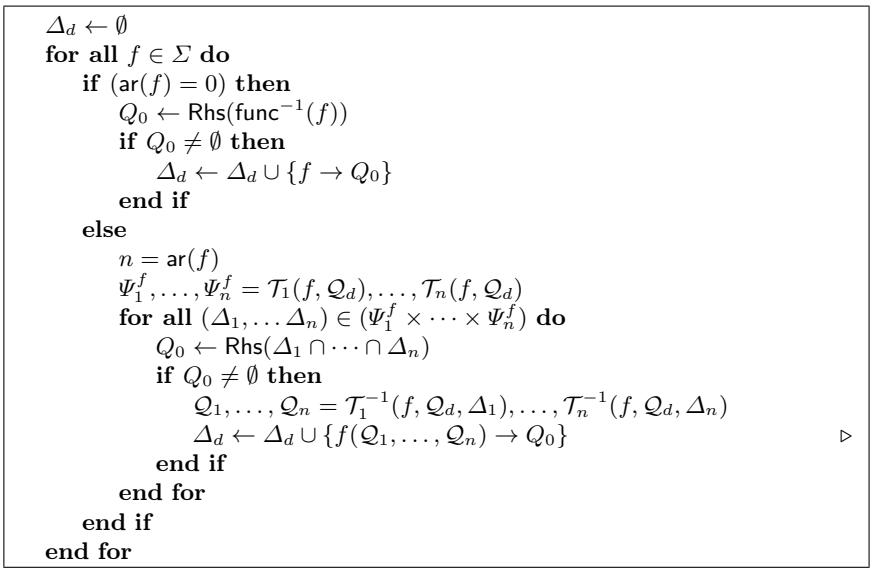

Fig. 10. Computation of product transitions

These 8 product transitions represent the 11 transitions shown in Example 1. There are more compact equivalent sets of product transitions, for example.

$$\begin{array}{ll} \Delta'' = \{[] \rightarrow q_1 & 0 \rightarrow q_1 \\ \quad [\{q_1, q_2, q_3\}|\{q_3\}] \rightarrow q_3 & [\{q_1, q_2, q_3\}|\{q_2\}] \rightarrow q_2 \\ \quad [\{q_1, q_2\}|\{q_1\}] \rightarrow q_1 & [\{q_3\}|\{q_1\}] \rightarrow q_2\} \end{array}$$

□

4.1 Determinisation algorithm giving transitions in product form

Product form can be obtained directly from optimised algorithm. We take the code in Figure 9 and simply omit the enumeration of the tuples $Q_1, \ldots, Q_n$ in the inner loop. This gives the algorithm in Figure 10 for generating Δ_d in product form. In the expression $f(\mathcal{Q}_1, \ldots, \mathcal{Q}_n) \rightarrow Q_0$ in Figure 10, $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ are sets of DFTA states. Rather than enumerating their cartesian product we simply return the product transition.

For the input FTA in Example 1 the output Δ' in Example 2 is then obtained. Later, in Section 6.3 we will see that this can be improved, yielding the output Δ'' in Example 2.

This can be a huge saving as will be seen from the experimental results, but obviously it would not be worth anything if it just delays the problem, that is, if we have to enumerate the cartesian product anyway later when we use the DFTA in some application. We will show that product form can often be used directly and in such cases the product never needs to be expanded.

procedure FTA DETERMINISATION (Input: $\langle Q, \Sigma, Q_f, \Delta \rangle$)
  $\mathcal{Q}_d \leftarrow \emptyset$ ▷ Compute DFTA states
  for all $f \in \Sigma$ **do**
    if $(\mathsf{ar}(f) = 0)$ **then**
      $Q_0 \leftarrow \mathsf{Rhs}(\mathsf{func}^{-1}(f))$
      if $Q_0 \neq \emptyset$ **then**
        $\mathcal{Q}_d \leftarrow \mathcal{Q}_d \cup \{Q_0\}$
      end if
    end if
  end for
  $\Psi_1^f, \ldots, \Psi_n^f = \emptyset, \ldots, \emptyset$
  $\mathcal{Q}_d^{new} \leftarrow \mathcal{Q}_d$
  repeat
    $\mathcal{Q}_d^{old} \leftarrow \mathcal{Q}_d$
    for all $f \in \Sigma$ **do**
      if $(\mathsf{ar}(f) > 0)$ **then**
        $(\Phi_1, \ldots, \Phi_n) \leftarrow (\mathcal{T}_1(f, \mathcal{Q}_d^{new}) \setminus \Psi_1^f, \ldots, \mathcal{T}_n(f, \mathcal{Q}_d^{new}) \setminus \Psi_n^f)$
        $(\Psi_1^{f,new}, \ldots, \Psi_n^{f,new}) \leftarrow (\Psi_1^f \cup \Phi_1, \ldots, \Psi_n^f \cup \Phi_n)$
        $Z \leftarrow \bigcup_{i=1}^{n} (\Psi_1^f \times \cdots \times \Psi_{i-1}^f \times \Phi_i \times \Psi_{i+1}^{f,new} \times \cdots \times \Psi_n^{f,new})$
        for all $(\Delta_1, \ldots \Delta_n) \in Z$ **do**
          $Q_0 \leftarrow \mathsf{Rhs}(\Delta_1 \cap \cdots \cap \Delta_n)$
          if $Q_0 \neq \emptyset$ **then**
            $\mathcal{Q}_d \leftarrow \mathcal{Q}_d \cup \{Q_0\}$
          end if
        end for
      end if
      $(\Psi_1^f, \ldots, \Psi_n^f) \leftarrow (\Psi_1^{f,new}, \ldots, \Psi_n^{f,new})$
    end for
    $\mathcal{Q}_d^{new} \leftarrow \mathcal{Q}_d \setminus \mathcal{Q}_d^{old}$
  until $\mathcal{Q}_d^{new} = \emptyset$ ▷
  $\Delta_d \leftarrow \emptyset$ ▷ Compute DFTA transitions
  for all $f \in \Sigma$ **do**
    if $(\mathsf{ar}(f) = 0)$ **then**
      $Q_0 \leftarrow \mathsf{Rhs}(\mathsf{func}^{-1}(f))$
      if $Q_0 \neq \emptyset$ **then**
        $\Delta_d \leftarrow \Delta_d \cup \{f \to Q_0\}$
      end if
    else
      $n = \mathsf{ar}(f)$
      for all $(\Delta_1, \ldots \Delta_n) \in (\Psi_1^f \times \cdots \times \Psi_n^f)$ **do**
        $Q_0 \leftarrow \mathsf{Rhs}(\Delta_1 \cap \cdots \cap \Delta_n)$
        if $Q_0 \neq \emptyset$ **then**
          $\mathcal{Q}_1, \ldots, \mathcal{Q}_n = \mathcal{T}_1^{-1}(f, \mathcal{Q}_d, \Delta_1), \ldots, \mathcal{T}_n^{-1}(f, \mathcal{Q}_d, \Delta_n)$
          $\Delta_d \leftarrow \Delta_d \cup \{f(\mathcal{Q}_1, \ldots, \mathcal{Q}_n) \to Q_0\}$
        end if
      end for
    end if
  end for
  $\mathcal{Q}_f \leftarrow \{Q' \in \mathcal{Q}_d \mid Q' \cap Q_f \neq \emptyset\}$
  return $(\mathcal{Q}_d, \Sigma, \mathcal{Q}_f, \Delta_d)$ ▷ Return DFTA
end procedure

Fig. 11. Optimised FTA determinisation algorithm returning product form

4.2 Further optimisation

We note that in the above code for computing Δ_d, the values of $\Psi_1^f, \ldots, \Psi_n^f$ are available from the main **repeat** loop and do not need to be recomputed. Also, the values of the expressions $\mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta_i)$ can be computed in the main loop of the algorithm by tabulating computed values of Lhsf_i; more precisely, whenever an expression $\mathsf{Lhsf}_i(f, Q')$ is evaluated and yields a non-empty value Δ', Q' is added to the set of values $\mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta')$. The complete determinisation algorithm returning transitions in product form is shown in Figure 11 (without the tabulation of the computation of $\mathcal{T}_i^{-1}$ just mentioned).

5 Performance of the optimised algorithm

Worst Case. Consider first the worst case running time for determinising the FTA $\langle Q, Q', \Sigma, \Delta \rangle$. The size of the input is measured by $|\Sigma|$, $|Q|$ and n, the maximum arity of the elements of Σ.

The main **repeat** loop of the algorithm in Figure 11 can be traversed up to $2^{|Q|}$ times, which is the upper bound of the number of states in the DFTA. For each $f \in \Sigma$ within the main loop there are up to $(2^{|Q|})^n$ iterations, since the size of $\mathcal{T}_i(f, \mathcal{Q}_d^{old})$ can be up to $|\mathcal{Q}_d| = 2^{|Q|}$. Combining all three nested loops, the complexity of the main loop of the algorithm is $O(|\Sigma|.2^{|Q|.(n+1)})$.

Considering the product transitions, the maximum number of iterations of the transition-generation loop for each n-ary $f \in \Sigma$ is $(2^{|Q|})^n$ and there is at most one product transition generated in each iteration. Hence the number of product transitions generated in the worst case is $O(|\Sigma|.2^{n.|Q|})$.

Regarding both the running time and the size of the output, the optimised algorithm performs no better in the worst case than the textbook algorithm with explicit enumeration of the DFTA transitions.

Running time in practice. The worst case of $2^{|Q|}$ for the number of DFTA states seems to be approached only in unusual situations; to achieve it, for every pairs of states q, q' of the original automaton it would have to be the case that $L(q) \cap L(q')$, $L(q) \setminus L(q')$ and $L(q') \setminus L(q)$ are all nonempty (since the states of the generated DFTA include only ones accepting some term, see Lemma 1). For such a pair of states, it is more common either that $L(q)$ and $L(q')$ are disjoint or that one includes the other. If this is the case for all such pairs, then the number of DFTA states is at most the number of original states. In Example 1 the number of DFTA states is the same as the number of original states; our experiments (Section 7) show a tendency for the size of the set of states of the FTA and the corresponding DFTA to be close for examples where the FTA is a description of program data. The set of DFTA states may even be smaller, if the original FTA has redundant or duplicated states - which sometimes happens with automatically generated FTAs.

Replacing $2^{|Q|}$ by $|\mathcal{Q}_d|$ in the complexity expressions gives a running time of $O(|\Sigma|.|\mathcal{Q}_d|^{n+1})$ and $O(|\Sigma|.|\mathcal{Q}_d|^n)$ for the number of product transitions in the output. Even if $|\mathcal{Q}_d|$ is small, we can see that high-arity function symbols still

present a potential for blow-up. Again, in practice this danger is often greatly reduced by the structure of the input FTA. As already noted, the size of $\mathcal{T}_i(f, \mathcal{Q}_d)$, whose worst-case size is $|\mathcal{Q}_d|$ is usually much smaller than $\mathcal{Q}_d$. This is due to the natural "typing" of function symbols. A function argument position in the original FTA is typically associated with a small number of states. However, there are applications where the size of $\mathcal{T}_i(f, \mathcal{Q}_d)$ is larger and for these is a danger of blow-up for high-arity function symbols. Such applications will be discussed in Section 7.

6 Complete DFTAs

Recall that in a complete FTA (Definition 2) every term $t \in \mathsf{Term}(\Sigma)$ has a run $t \Rightarrow^* q$ where $q \in Q$. An FTA can always be completed [2], by adding an extra state to Q and adding extra transitions to Δ.

Example 3. Consider the following FTA $\langle Q, \Sigma, Q_f, \Delta\rangle$ where $\Sigma = \{[], [.|.], 0, s(.)\}$, $Q = \{list, num\}$, $Q_f = \{list\}$ and $\Delta = \{[] \to list, [num|list] \to list, 0 \to num, s(num) \to num\}$. The FTA is not complete; for instance there is no transition with left hand side $s(list)$ or $[num|num]$. Thus the terms $s([])$ and $[s(0)|0]$, for example, have no run. To complete it, we can add an extra non-final state, say e (for *error*), and add the following transitions. $\Delta_e = \{s(list) \to e, s(e) \to e, [list|list] \to e, [num|num] \to e, [list|num] \to e, [e|list] \to e, [list|e] \to e, [e|e] \to e, [num|e] \to e, [e|num] \to e\}$. The FTA $\langle Q \cup \{e\}, \Sigma, Q_f, \Delta \cup \Delta_e\rangle$ is complete and accepts the same language as the original FTA (the set of lists of numbers). Any term that did not have a run in the original now has a run to e (which is not an accepting state).

One can see that the number of transitions in the completed FTA is determined by the arity of the function symbols and the number of states, and that it is exponential in the arity of the function symbols.

6.1 Simultaneous completion and determinisation using Δ_{any}

Recall that given a finite signature Σ we define the set of transitions Δ_{any} as $\{f(\overbrace{\mathsf{any}, \ldots, \mathsf{any}}^{n \text{ times}}) \to \mathsf{any} \mid f^n \in \Sigma\}$. Clearly for any term $t \in \mathsf{Term}(\Sigma)$ there exists a run $t \Rightarrow^* \mathsf{any}$. The following lemma shows that we can use Δ_{any} to obtain a complete DFTA from a given FTA, in other words, we can perform determinisation and completion simultaneously. First we establish an important property of the DFTAs generated by the algorithm - that they contain only "non-empty" states.

Lemma 1. *Given an FTA, for every state Q' of the DFTA obtained by determinising it using the given algorithm, $L(Q') \neq \emptyset$.*

Proof. We consider the algorithm in Figure 4 for simplicity, rather than the final optimised algorithm. If the algorithm terminates in one iteration then the set of states of the DFTA is empty and the lemma holds trivially. Otherwise, consider a DFTA state Q' that is generated in the algorithm. We reason by induction on the number of iterations of the main loop of the algorithm

- Base case ($i = 1$): If Q' is generated on the first iteration, then there exists some 0-arity function f such that $Q' = \mathsf{Rhs}(\mathsf{func}^{-1}(f))$. The transition $f \to Q'$ is generated on iteration 1 hence there is a run $f \Rightarrow Q'$.
- Induction ($i > 1$): Assume that the lemma holds for all states generated up to the $i-1^{th}$ iteration, and that Q' is a new state generated on the i^{th} iteration. Then there exist states $Q_1, \ldots, Q_n$ which were generated in the first $i-1$ iterations, such that a transition $f(Q_1, \ldots, Q_n) \to Q'$ is constructed on the i^{th} iteration. By the inductive hypothesis, there exist terms $t_1, \ldots, t_n$ such that $t_i \Rightarrow^* Q_i, 1 \leq i \leq n$. Hence there is a run $f(t_1, \ldots, t_n) \Rightarrow^* Q'$.

Hence the lemma holds for states generated on any iteration. □

Lemma 2. *Let $\langle Q, Q_f, \Sigma, \Delta\rangle$ be an FTA such that every term $t \in \mathsf{Term}(\Sigma)$ has a run $t \Rightarrow^* q$ for some $q \in Q$. Then the DFTA obtained by determinising $\langle Q, Q_f, \Sigma, \Delta\rangle$ is complete.*

Proof. Let $\mathcal{Q}'$ be the set of states of the generated DFTA, and assume that it is not complete. Then there exist $Q_1, \ldots, Q_n \in \mathcal{Q}'$ and an n-ary function f such that for all Q_0 there is no transition $f(Q_1, \ldots, Q_n) \to Q_0$ in the DFTA. Let $t_1, \ldots, t_n \in \mathsf{Term}(\Sigma)$ be terms such that $t_i \Rightarrow^* Q_i, 1 \leq i \leq n$. The existence of such terms is guaranteed by Lemma 1. Since it is a DFTA, there cannot be any other runs $t_i \Rightarrow^* Q'_i$ where $Q_i \neq Q'_i$ respectively. Hence there is no run for the term $f(t_1, \ldots, t_n)$, which contradicts the assumption of the lemma. Hence the generated DFTA is complete. □

Thus we establish the following.

Lemma 3. *Let $\langle Q, Q_f, \Sigma, \Delta\rangle$ be an FTA, such that $\mathsf{any} \in Q$ and $\Delta_{\mathsf{any}} \subseteq \Delta$. Then the DFTA obtained by the determinisation algorithm with this input is complete.*

Proof. For every term $t \in \mathsf{Term}(\Sigma)$ the FTA has a run $t \Rightarrow^* \mathsf{any}$. The result follows from Lemma 2. □

6.2 Performance of the algorithm after adding Δ_{any}

Although by Lemma 3 we can obtain a complete DFTA from any given input FTA, simply by adding the state any and the transitions Δ_{any} to the input before running the algorithm, we may ask what is the impact on the performance of the determinisation algorithm.

The impact on the number of states of the DFTA is slight; at most one extra state $\{\mathsf{any}\}$ is generated, in the case that there are some terms accepted by any

but not by any other state. This state represents the "error" state of the classical completion procedure. Apart from this, the same states are generated but any is added to each one; it is easy to see that any must appear in every DFTA state since the state any appears in every possible left-hand-side position in Δ_{any}.

The main question is thus the impact on the generated product representation of the transitions. Let us analyse the effect of adding Δ_{any} on part of the algorithm generating the transitions of the DFTA in product form (Figure 11). Consider the effect of the introduction of Δ_{any} on the basic operations of the algorithm.

- $\mathsf{func}^{-1}(f)$ and $\mathsf{Lhsf}_i(f, Q)$: in each case the returned set contains at most one extra transition (that is, the transition $f(\mathsf{any}, \ldots, \mathsf{any}) \to \mathsf{any}$, in the case that $\mathsf{any} \in Q$).
- $\mathsf{Rhs}(T)$: the returned set contains at most one extra state any in the case that $T \cap \Delta_{\mathsf{any}} \neq \emptyset$.

Given this, the effect on the generation of transitions is a constant independent of the input FTA. In Section 7 we verify that the overhead of adding Δ_{any} is minimal. Thus we obtain the completed DFTA at almost no extra cost over obtaining the DFTA.

6.3 Don't-Care Arguments in Complete DFTAs

An underscore argument "_" is used as shorthand for $\mathcal{Q}_d$ in a product transition, for example $f(\mathcal{Q}_1, \ldots, \_, \ldots, \mathcal{Q}_n) \to Q_0$. This indicates that the choice of DFTA state in this argument is irrelevant in determining the right hand side Q_0. Product transitions of the form $f(\ldots, \_, \mathcal{Q}_i, \_, \ldots) \to Q_0$ in which all but one argument are don't-care arguments are especially interesting, since the right-hand-side of the transition is determined by just one argument. We call the elements of such arguments *deciding arguments*.

A typical case of deciding arguments arises in complete DFTAs constructed by adding Δ_{any} to the original FTA, where a state $\{\mathsf{any}\}$ is generated, which accepts the terms not accepted by any other state. $\{\mathsf{any}\}$ is a deciding argument; the presence of $\{\mathsf{any}\}$ in any argument in a DFTA transition is sufficient to ensure that the right hand side of the transition is $\{\mathsf{any}\}$. That is, there are DFTA product transitions of the form:

$$\begin{gathered} f(\{\{\mathsf{any}\}\}, \_, \ldots, \_, \_) \to \{\mathsf{any}\} \\ f(\_, \{\{\mathsf{any}\}\}, \_, \ldots, \_) \to \{\mathsf{any}\} \\ \ddots \\ f(\_, \_, \ldots, \_, \{\{\mathsf{any}\}\}) \to \{\mathsf{any}\} \end{gathered}$$

These product transitions overlap, obviously, since $\{\mathsf{any}\}$ is included in the don't-care arguments. However, this form might be much more compact than the product transitions generated by the determinisation algorithm. Furthermore, there could be other deciding arguments besides $\{\mathsf{any}\}$.

We prove two lemmas defining sufficient conditions for finding deciding arguments and generating the corresponding don't-care product transitions.

Lemma 4. *Let $\mathcal{Q}_d$ be the set of states of a compete DFTA and let $\Psi_1, \ldots, \Psi_n = \mathcal{T}_1(f, \mathcal{Q}_d), \ldots, \mathcal{T}_n(f, \mathcal{Q}_d)$ for some n-ary function f. Let $\Delta' \in \Psi_i$ and $\mathcal{Q}_i = \mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta')$. Then $\mathcal{Q}_i$ are* deciding arguments *for the i^{th} argument of f if*

$$\mathsf{Rhs}(\Delta' \cap \bigcap(\cap\Psi_1, \ldots, \cap\Psi_{i-1}, \cap\Psi_{i+1}, \ldots, \cap\Psi_n)) = \mathsf{Rhs}(\Delta').$$

Proof. Consider the set of right hand sides of transitions that can be built from $(\Psi_1 \times \cdots \times \Psi_n)$ using Δ' in the i^{th} position, say $\bar{Q}$. That is, $\bar{Q} = \mathsf{Rhs}(\Delta_1 \cap \cdots \cap \Delta' \cap \cdots \cap \Delta_n)$ where $(\Delta_1, \ldots, \Delta', \ldots, \Delta_n) \in (\Psi_1 \times \cdots \times \{\Delta'\} \times \cdots \Psi_n)$ with Δ' in the i^{th} position.

$\mathsf{Rhs}(\Delta')$ is an upper bound for $\bar{Q}$, since $\Delta_1 \cap \ldots, \Delta' \cap \ldots \cap \Delta_n \subseteq \Delta'$ and Rhs is monotonic. The expression $\mathsf{Rhs}(\Delta' \cap \bigcap(\cap\Psi_1, \ldots, \cap\Psi_{i-1}, \cap\Psi_{i+1}, \ldots, \cap\Psi_n))$ is a lower bound for $\bar{Q}$, since $\Delta' \cap \bigcap(\cap\Psi_1, \ldots, \cap\Psi_{i-1}, \cap\Psi_{i+1}, \ldots, \cap\Psi_n) \subseteq \Delta_1 \cap \cdots \cap \Delta' \cap \cdots \cap \Delta_n$ and Rhs is monotonic. If these two are equal, as in the statement of the property, then we can conclude that the value $\mathsf{Rhs}(\Delta')$ is the right hand side for any such transition since it is both an upper and a lower bound.

□

If we can find such a $\Delta' \in \mathcal{T}_i(f, \mathcal{Q}_d)$, a (product) transition $f(\ldots, \_, \mathcal{Q}_i, \_, \ldots) \to \mathsf{Rhs}(\Delta')$ is constructed, where $\mathcal{Q}_i = \mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta')$ and the underscore arguments stand for $\mathcal{Q}_d$.

A more specialised sufficient condition for deciding arguments for binary functions is given by the following lemma.

Lemma 5. *Let $\mathcal{Q}_d$ be the set of states of a complete DFTA and let $\Psi_1, \Psi_2 = \mathcal{T}_1(f, \mathcal{Q}_d), \mathcal{T}_2(f, \mathcal{Q}_d)$ for some 2-ary function f. Then a set of DFTA states $\mathcal{Q}_i \subseteq \mathcal{Q}_d$, where $\mathcal{Q}_i = \mathcal{T}_i^{-1}(f, \mathcal{Q}_d, \Delta')$ for some $\Delta' \in \Psi_i$, $i \in \{1, 2\}$ are* deciding arguments *for the i^{th} argument of f if $\mathsf{Rhs}(\Delta')$ is a singleton, and for all $\Delta'' \in \Psi_j$, $j \in \{1, 2\} \setminus \{i\}$,*

$$\Delta' \cap \Delta'' \neq \emptyset.$$

Proof. $\Delta' \cap \Delta'' \neq \emptyset$ implies that $\mathsf{Rhs}(\Delta_j \cap \Delta') \neq \emptyset$. Since $\mathsf{Rhs}(\Delta')$ is a singleton and $\mathsf{Rhs}(\Delta'' \cap \Delta') \subseteq \mathsf{Rhs}(\Delta')$, $\mathsf{Rhs}(\Delta'' \cap \Delta') = \mathsf{Rhs}(\Delta')$.

If such a Δ' is found, say in argument 2, we generate the product transition $f(\_, \mathcal{Q}_2) \to \mathsf{Rhs}(\Delta')$ where $\mathcal{Q}_2 = \mathcal{T}_2^{-1}(f, \mathcal{Q}_d, \Delta')$ and the underscore arguments stand for $\mathcal{Q}_d$.

We can easily add a check for deciding arguments using the conditions of Lemma 4 and 5 to the algorithm, just before generating product transitions. The calculation of the intersections is exponential in the arity of the function symbols, but does not alter the complexity of the overall algorithm and can save effort in generating product transitions. For each set of deciding arguments $\mathcal{Q}_i$ discovered, we generate a don't-care product transition of the form just shown, and the corresponding value Δ' is removed from Ψ_i when computing the remaining product transitions for f.

	Textbook algorithm		Optimised algorithm w/o don't cares		Optimised algorithm with don't cares	
	det	**det +compl**	**det**	**det +compl**	**det**	**det +compl**
solved	112	109	14663	14663	14663	14663
timeout	14584	14587	33	33	33	33
avg. secs.	0.55	3.35	0.07	0.20	0.07	0.14
% solved	0.76	0.74	99.77	99.77	99.77	99.77

Table 1. Comparison of textbook vs. product form algorithm for FTA determinisation and completion on 14,696 benchmark problems (timeout 60 seconds)

The problem of finding the minimum number of product transitions to represent the DFTA transitions seems to be intractable and is beyond the scope of this paper. In essence it can be stated as the problem of finding the minimum number of cartesian products whose union is a given relation.

7 Experiments

Tables 1 and 2 show experimental results comparing the determinisation algorithm in Figure 11 with the textbook algorithm. It also compares the effect of adding the detection of don't care arguments in the determinisation algorithm. The algorithms are implemented in Java; the textbook algorithm is a direct implementation of the program in Figure 4.

The 14,696 benchmark FTAs were obtained from the repository that is part of the tool libvata[4], which is a highly optimised non-deterministic finite tree automata library. Many of these FTAs originate in the Timbuk system [7]. The experiments were carried out with an Intel 2.9 GHz processor with 8 GB memory on a MacBook Pro running OS X 10.9.4.

7.1 Determinisation and completion

The columns in Table 1 show the overall effectiveness of three versions of the determinisation algorithm, for determinisation (**det**) and determinisation with completion (**def+compl**). The first version is the textbook algorithm, the second is the optimised algorithm returning product form, without detection of don't care arguments, and the third is with detection of don't cares.

The first notable point is that the textbook algorithm is able to solve less than 1% of the problems while the optimised algorithms solve nearly all of them. The running time of the textbook algorithm is far slower even considering only those problems that it could solve.

Completion is a significant overhead for the textbook algorithm mainly because usually it results in a much larger set of transitions, and fewer problems

[4] http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/

	Optimised algorithm w/o don't cares		Optimised algorithm with don't cares	
	det	**det+compl**	**det**	**det+compl**
average $\lvert Q\rvert$	58.3	59.3	58.3	59.3
average $\lvert \Delta\rvert$	286.3	301.8	286.3	301.8
average $\lvert \Sigma\rvert$	15.5	15.5	15.5	15.5
average $\lvert Q_d\rvert$	66.9	67.9	66.9	67.9
average $\lvert \Delta_d\rvert$	17614	2.9×10^{18}	-	2.9×10^{18}
average $\lvert \Delta_\Pi\rvert$	649	13211	663	3249

Table 2. Size statistics for output of optimised DFTA algorithm

were solvable within the timeout period. For the optimised algorithm, completion increases the running time by an average of approximately two to three times but does not affect the number of solvable problems. Detection of don't cares is not a substantial overhead, and in fact can lead to overall time savings when completion is performed since the number of product transitions can thereby be reduced.

The size of the input and output DFTAs is summarised in Table 2. The size of the set of states and transitions of the input FTA are $|Q|$ and $|\Delta|$. The number of DFTA states is $|Q_d|$ and the number of product transitions in the DFTA is $|\Delta_\Pi|$. For completed DFTAs, the precise size of the set of transitions depends only on the signature and can be calculated; this is shown in the table as $|\Delta_d|$. For non-complete DFTAs we can estimate the actual size of $|\Delta_d|$ by summing the product of the sizes of the product states in each transition. However, the same transition could be represented by more than one product transition so it is an over-estimate. In the case of product transitions with don't cares the over-estimation is so great that we omit it in Table 2. It can immediately be seen that the number of DFTA states is on average only slightly greater than the number of input FTA states, as discussed in Section 5. The average size of the set of transitions in completed DFTAs is extremely large, and shows immediately why the textbook algorithm fails. Regarding completion, the size of the set of states of both input and output automata is increased by exactly one. The size of the input set of transitions is increased by the size of Δ_{any} which is $|\Sigma|$.

The effect of computing don't cares is noticeable in the size of Δ_Π for complete DFTAs. The number of product transitions with don't cares is only about 25% of the number without don't cares and is reflected in the somewhat faster runtimes for don't cares with completion in Table 1.

8 Applications

The question of applicability depends partly on whether the product form of transitions is directly usable. As already pointed out, the product form is of little ultimate use if the transitions need to be explicitly enumerated in order to

use them. Fortunately, it appears that the product form can often be efficiently processed, and in some cases the transitions are not even needed – the set of states of the DFTA is all that is needed.

Determinisation and completion are needed to form the complement of an FTA. Applications in verification and analysis using tree automata to represent set of states could benefit from the availability of a practical complementation algorithm, e.g. [3,13,7,4,1]. Note that we obtain the complement with its transitions in product form, simply by switching the accepting and non-accepting states of the completed DFTA.

Gallagher *et al.* [5,6] showed that program properties of interest in analysis of logic program can be formulated as sets of terms defined by an FTA on the program signature and that a precise abstract domain for static analysis could then be constructed by determinising the FTA. The algorithm presented in detail here was first developed in the context of that work. The approach was made practical by encoding the product transitions directly as binary decision diagrams (BDDs) thus avoiding the need to enumerate transitions explicitly.

DFTA states encode useful information in themselves. The emptiness of intersections of input FTA states can be checked using the DFTA states. There exists a term accepted by every member of a set of states Q_1 in the FTA if and only if the corresponding DFTA includes a state Q_2 such that $Q_1 \subseteq Q_2$. Given the fact that the DFTA transitions are not needed for this check, the algorithm presented here could provide a useful basis for checking emptiness of intersections.

Another question that can be answered by examining the set of states is universality. An FTA A with signature Σ is universal if $L(A) = \mathsf{Term}(\Sigma)$. We add a non-final state any to the states of A and add Δ_{any} to the transitions of A and determinise the result. Then A is universal if and only if every final state in the resulting DFTA contains any.

Recently, the determinisation algorithm has been used to implement a refinement procedure for Horn clause verification [10]. An FTA is built to represent the set of derivations in a given set of Horn clauses. Infeasible traces can be eliminated from the FTA by application of the determinisation algorithm to construct the difference of two FTAs, which is then used to construct a new set of Horn clauses.

9 Related Work

The algorithm presented in this paper was sketched by Gallagher *et al.* in [6], including the concept of product transitions. Otherwise, we do not know of other attempts to design practical algorithms for determinisation. Previous work that used tree automata as a modelling formalism commented on the impracticality of handling complementation, due to the complexity of the determinisation and completion algorithm [13,8]. Available libraries for tree automata manipulation seem to implement the textbook algorithm [7,11,12].

A different but related problem is tree automata containment checking. The classical approach to checking containment of FTA A_1 in FTA A_2 is to check the emptiness of $A_1 \cap \bar{A}_2$ which involves the complementation of A_2. Other containment algorithms not requiring complementation have been presented [14,9]. It would appear that the optimised algorithm for determinisation gives renewed potential to the classical approach, especially since the transitions do not need to be generated to check containment. A_1 is contained in A_2 if and only if the states of the DFTA for $A_1 \cup A_2$ do not include any state containing a final state of A_1 but no final state of A_2. Currently we are comparing this approach with state-of-the-art containment checking algorithms.

10 Future Work

There remains interesting work to do both on the algorithm itself and its applications. Firstly, there seem to be opportunities for optimisation of the critical inner loop of the algorithm generating the DFTA states. A state can be generated many times, and it seems likely that there are conditions on the elements of the Φ and Ψ arrays in the algorithm that could be checked in order to avoid this. The challenge is to simplify the checks sufficiently to make them worthwhile as an optimisation. Perhaps a completely different representation of the Φ and Ψ arrays, such as some Boolean encoding, is needed. We are actively investigating this.

Secondly, we are looking at applications of the algorithm. The original motivating application, that of logic program analysis, is still interesting, since Horn clauses (pure logic programs) are increasingly used as a representation language for a variety of other languages and computational formalisms. Essentially the same analysis problems arise in term rewriting systems, where a system state is represented by a term, and an FTA expresses state properties of interest.

Finally, as mentioned in Section 9, we are performing an evaluation of the use of determinisation algorithm for the FTA containment problem. This has applications in XML language containment checking, among others.

Acknowledgements

We would like to thank Kim Steen Henriksen and Gourinath Banda for discussions in the early stages of this work.

References

1. E. Balland, Y. Boichut, P.-E. Moreau, and T. Genet. Towards an efficient implementation of tree automata completion. In *Algebraic Methodology and Software Technology, 12th International Conference, AMAST*, pages 67–82, 2008.
2. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

3. J. Elgaard, A. Møller, and M. I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proc. Programming Languages and Systems, 9th European Symposium on Programming, ESOP '00*, volume 1782 of *LNCS*, pages 182–194. Springer-Verlag, March/April 2000.
4. G. Feuillade, T. Genet, and V. V. T. Tong. Reachability analysis over term rewriting systems. *J. Autom. Reasoning*, 33(3-4):341–383, 2004.
5. J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, volume 3132 of *Springer-Verlag Lecture Notes in Computer Science*, pages 27–42, 2004.
6. J. P. Gallagher, K. S. Henriksen, and G. Banda. Techniques for scaling up analyses based on pre-interpretations. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming, ICLP'2005*, volume 3668 of *Springer-Verlag Lecture Notes in Computer Science*, pages 280–296, 2005.
7. T. Genet and V. V. T. Tong. Reachability analysis of term rewriting systems with Timbuk. In R. Nieuwenhuis and A. Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 2001.
8. N. Heintze. Using bottom-up tree automaton to solve definite set constraints. Unpublished. Presentation at Schloß Dagstuhl Seminar 9743, `http://www.informatik.uni-trier.de/~seidl/Trees.html`, 1997.
9. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.
10. B. Kafle and J. P. Gallagher. Tree automata-based refinement with application to Horn clause verification. Submitted for publication, 2014.
11. N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.
12. O. Lengál, J. Simácek, and T. Vojnar. VATA: A library for efficient manipulation of non-deterministic tree automata. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012*, pages 79–94, 2012.
13. D. Monniaux. Abstracting cryptographic protocols with tree automata. *Sci. Comput. Program.*, 47(2-3):177–202, 2003.
14. T. Suda and H. Hosoya. Non-backtracking top-down algorithm for checking tree automata containment. In *Implementation and Application of Automata, 10th International Conference, CIAA 2005*, pages 294–306, 2005.

RECENT RESEARCH REPORTS

#144 Magnus Rotvit Perlt Hansen. *Discovering the Process of User Expectating in a Pilot Implementation Expectations and Experiences in Information Systems Development*. PhD thesis, Roskilde, Denmark, June 2014.

#143 Keld Helsgaun. Solving the Bottleneck Traveling Salesman Problem Using the Lin-Kernighan-Helsgaun Algorithm. 42 pp. May 2014, Roskilde University, Roskilde, Denmark.

#142 Keld Helsgaun. Solving the Clustered Traveling Salesman Problem Using the Lin-Kernighan-Helsgaun Algorithm. 13 pp. May 2014, Roskilde University, Roskilde, Denmark.

#141 Keld Helsgaun. Solving the Equality Generalized Traveling Salesman Problem Using the Lin-Kernighan-Helsgaun Algorithm. 15 pp. May 2014, Roskilde University, Roskilde, Denmark.

#140 Anders Barlach. *Effekt-drevet IT udvikling Eksperimenter med effektdrevne systemudviklingsprojekter, der involverer CSC Scandihealth og kunder fra det danske sundhedsvæsen*. PhD thesis, Roskilde, Denmark, November 2013.

#139 Mai Lise Ajspur. *Tableau-based Decision Procedures for Epistemic and Temporal Epistemic Logics*. PhD thesis, Roskilde, Denmark, October 2013.

#138 Rasmus Rasmussen. *Electronic Whiteboards in Emergency Medicine Studies of Implementation Processes and User Interface Design Evaluations*. PhD thesis, Roskilde, Denmark, April 2013.

#137 Christian Theil Have. *Efficient Probabilistic Logic Programming for Biological Sequence Analysis*. PhD thesis, Roskilde, Denmark, January 2013.

#136 Sine Zambach. *Regulatory Relations Represented in Logics and Biomedical Texts*. PhD thesis, Roskilde, Denmark, February 2012.

#135 Ole Torp Lassen. *Compositionality in probabilistic logic modelling for biological sequence analysis*. PhD thesis, Roskilde, Denmark, November 2011.

#134 Philippe Blache, Henning Christiansen, Verónica Dahl, and Jørgen Villadsen, editors. *Proceedings of the 6th International Workshop on Constraints and Language Processing*, Roskilde, Denmark, October 2011.

#133 Jens Ulrik Hansen. *A logic toolbox for modeling knowledge and information in multi-agent systems and social epistemology*. PhD thesis, Roskilde, Denmark, September 2011.

#132 Morten Hertzum and Magnus Hansen, editors. *Proceedings of the Tenth Danish Human-Computer Interaction Research Symposium (DHRS2010)*, Roskilde, Denmark, November 2010.