Algorithms for Guided Tree Automata

Morten Biehl[1], Nils Klarlund[2], and Theis Rauhe[1]

[1] BRICS, Department of Computer Science,
University of Aarhus,
Ny Munkegade, Aarhus, Denmark
`{mbiehl,theis}@brics.dk`

[2] AT&T Labs - Research
600 Mountain Ave.,
Murray Hill, NJ 07974
`klarlund@research.att.com`

Abstract. When reading an input tree, a bottom-up tree automaton is "unaware" of where it is relative to the root. This problem is important to the efficient implementation of decision procedures for the Monadic Second-order Logic (M2L) on finite trees. In [7], it is shown how exponential state space blow-ups may occur in common situations. The analysis of the problem leads to the notion of *guided tree automaton* for combatting such explosions. The guided automaton is equipped with separate state spaces that are assigned by a top-down automaton, called the *guide*.

In this paper, we explore the algorithmic and practical problems arising from this relatively complicated automaton concept.

Our solutions are based on a BDD representation of automata [4], which allows the practical handling of automata on very large alphabets. In addition, we propose data structures for avoiding the quadratic size of transition tables associated with tree automata.

We formulate and analyze product, projection (subset construction), and minimization algorithms for guided tree automata. We show that our product algorithm for certain languages are asymptotically faster than the usual algorithm that relies on transition tables.

Also, we provide some preliminary experimental results on the use of guided automata vs. standard tree automata.

1 Introduction

The MONA tool [4] implements a decision procedure and counter model generator for a Monadic Second-order Logic on strings. This logic has first-order terms that denote positions in the string and limited arithmetic on such terms; in addition, second-order terms denote subsets of positions. The decision procedure follows the classical method of associating a *Deterministic Finite Automaton* (DFA) to each formula in M2L such that the DFA accept the strings satisfying the formula. The main idea is the use of an extended input alphabet such that a string encodes the value of all free variables. Naturally, this encoding leads to very large alphabets, whose representation becomes the major issue in the computational handling of such DFAs.

An extension of Mona to binary trees is currently under development at BRICS in Aarhus. The traditional way of deciding formulas in the Monadic Second-order Logic on Binary Trees is to associate a *Deterministic Finite Tree Automaton* (DFTA) to each subformula. Each automaton represents the set of interpretations that make the subformula true. The automata are calculated according to a simple correspondence between logical connectives and automata-theoretic operations. In [7], some sources of exponential and polynomial blow-ups in tree automata associated with M2L formulas are studied. It is shown there that a common source of state space explosion for DFTAs is their lack of knowledge about the position relative to the root of the subtree read by the automaton.

Among the proposals in [7], the asymptotically best one is to factorize the state space by a representation called a *guided tree automaton*. The factorization is carried out by a top-down automaton, called the *guide*, which assigns state space to the nodes of the input tree. The transition relation of the bottom-up automaton is thereby split into many components.

In [7], a high-level programming language Fido based on M2L and some conventional programming language concepts is proposed for the convenient expression of properties of parse trees. The compilation of Fido into M2L is described. Also, a concept of *universe* is proposed as an extension of M2L. A universe declaration defines a separate tree address space. In [7], it is argued how universes naturally arise from a Fido program and how they in turn give rise to guides that reduce state spaces.

The algorithmic aspects of guided tree automata are involved and were not discussed in [7].

In the present paper, we propose efficient data structures and algorithms for BDD-represented guided tree automata. A main problem addressed is how to avoid the inherent quadratic blow-up in the representation of a transition relation. This blow-up hinges on the property that for an n-state automaton, there are n^2 pairs for which a function from letters to new states have to be specified. With our solution, we can bound the running time of the product algorithm in certain situations: the total time required is $O(N^{\frac{3}{2}})$, where N is the number of states of the product automaton, whereas a conventional algorithm would use $\theta(N^2)$ time (and space).

Our solution to the quadratic blow-up problem relies on observations that we make about the nature of transition relations occurring during the decision procedure for M2L. We argue that transition relations tend to be *sparse*, at least for the important first-order fragment of M2L. Our representation uses essentially the same default idea as in [2], which in our case can be expressed as: for fixed left state q', the largest class of right states q'' such that that the transition function is constant on (q', q'') can be represented by a default transition. All other q'' must be represented by explicit entries for (q', q''). Our contribution is to solve a number of technical problems that must be overcome in order to use this idea efficiently, that is, so that asymptotic and practical gains can be achieved.

The rest of the paper is organized as follows. In Section 2, we provide the definitions of guided tree automata and discuss their relation to usual tree automata. In Section 3, we suggest efficient data structures for the representation of guided tree automata. In Section 4, 5, and 6, we give detailed accounts of algorithms for product, projection (including determinization), and minimization. Finally, we report on some preliminary experimental results in Section 7.

2 Guided tree automata

Let Σ be an alphabet. The *trees* T_Σ over Σ are denoted as follows. A *leaf* is identified with the empty tree ε. Internal nodes are identified with their subtrees, which are of the form $\alpha\langle t_1, t_2\rangle$, where $\alpha \in \Sigma$ is the *label* of the node and $t_1, t_2 \in T_\Sigma$ are the left and right subtrees. Leaves do not have labels. A *Deterministic Finite Tree Automaton* (DFTA) is a tuple $M = (Q, \Sigma, q_0, F, \delta)$, where Q is the finite set of states, $F \subseteq Q$ is the final states, $q_0 \in Q$ is the initial state and δ is the transition function $\delta : (Q \times Q) \rightarrow (\Sigma \rightarrow Q)$. The labeling function $\hat{\delta} : T_\Sigma \rightarrow Q$ is defined inductively by

$$\hat{\delta}(\varepsilon) = q_0$$
$$\hat{\delta}(\alpha\langle t_1, t_2\rangle) = \delta(\hat{\delta}(t_1), \hat{\delta}(t_2))(\alpha)$$

We say that M *accepts* the *input tree* t if $\hat{\delta}(t) \in F$. Thus informally, the automaton traverses the tree bottom-up while associating a state to each subtree. The set $L(M)$ of all trees accepted is the *language* accepted by M. A tree language is *regular* if and only if it is the language accepted by some DFTA.

According to the definition above, the traversal of a subtree is independent of where the subtree is positioned in the input tree. As argued in [7], it would often be beneficial to provide the automaton with “positional knowledge” indicating what part of the input tree, relative to the root, the automaton is currently traversing.

To see how such information may bring about a reduction in the state space, consider the tree language L consisting of all trees for which the number of occurrences of $\alpha \in \Sigma$ in the left subtree is divisible by n and the number of occurrences of $\beta \in \Sigma$ in the right subtree is divisible by m. A DFTA recognizing L then requires $n \cdot m$ states (when n and m are relative primes). The language is more efficiently recognized by means of three separate automata : M_L for traversing the left subtree (n states), M_R for traversing the right subtree (m states) and M_T for combining the results. The polynomial state space explosion ($n \cdot m$) is avoided, because each automaton has “positional knowledge.” For example, M_L “knows” that it is traversing the left subtree, and thus it does not need to keep track of occurrences of β.

Technically, “positional knowledge” can be provided by a top-down automaton that identifies regions of the input trees.

Definition 2.1. A *guide* $G = (D, \mu, d_0)$ consists of

> D, a finite set of *state space IDs*,
> $\mu : D \to D \times D$, the *guide function*, and
> $d_0 \in D$, the *initial ID*.

The *size* γ of G is the cardinality of D.

A simple guide that identifies whether a node is a left child, a right child, or the root of the input tree can be defined as $G = (D, \mu, d_0)$, where

$$\begin{aligned} D &= \{\textit{top}, \ \textit{left}, \ \textit{right}\}, \\ d_0 &= \textit{top}, \ \text{ and} \\ \mu(d) &= (\textit{left}, \ \textit{right}), \ \text{ for any } d \in D. \end{aligned}$$

A *guided tree automaton* defines a state space for each state space ID and a transition function for each transition of the guide:

Definition 2.2. Let $G = (D, \mu, d_0)$ be a guide. A *guided tree automaton* (GTA) M_G with guide G is of the form $(\{Q_d\}_{d\in D}, \Sigma, \{\delta_d\}_{d\in D}, \{\bar{q}_d\}_{d\in D}, F)$. The components of M_G are as follows.

- $\{Q_d\}_{d\in D}$ is a family of disjoint finite sets of states, one set for each state space ID. We often abbreviate this family $\{Q\}_D$.
- Σ is the alphabet.
- $\{\delta_d\}_{d\in D}$ is a family of transition functions, one for each state space ID, such that if $\mu(d) = (d', d'')$ for some $d, d', d'' \in D$, then δ_d is a transition function of the form $\delta_d : (Q_{d'} \times Q_{d''}) \to (\Sigma \to Q_d)$. We say that δ_d *is of type* $d' \times d'' \to d$, and call d' the *left ID of* δ_d, d'' the *right ID of* δ_d and d the *target ID of* δ_d. Similarly, we refer to $Q_{d'}$ as the *left states of* δ_d, $Q_{d''}$ as the *right states of* δ_d and Q_d as the *range states of* δ_d. We often abbreviate this family $\{\delta\}_D$.
- $\{\bar{q}_d\}_{d\in D}$ is the family of initial states, one for each state space ID. We often abbreviate this family $\{\bar{q}\}_D$.
- $F \subseteq Q_{d_0}$ is the set of final states.

The *size* n of M_G is the cardinality of the largest state space in M_G.

Note that if $G = (D, \mu, d_0)$ is a guide with $|D| = 1$, then any tree automaton guided by G is just an ordinary DFTA.

We will rely on notational shortcuts to improve readability. States and sets of states are usually subscripted by the state space ID, denoted by d, d', d'', etc. Where no confusion arises, we write q instead of q_d, q' instead of $q_{d'}$, q'' instead of $q_{d''}$ etc. Similar abbreviations are used for sets of states.

The intuition behind guided tree automata is quite simple. First, the guide labels each node in the tree with a state space ID ("positional knowledge"). Second, each leaf is labeled with the initial state of the state space indicated by the ID. Then in a bottom-up manner, every remaining internal node is labeled with a state of the space indicated by the ID according to the corresponding transition function.

To make these notions more precise, we let $T_{(\Sigma_1,\Sigma_2)}$ denote the set of trees whose leaves belong to the alphabet Σ_1 and for which all other nodes belong to the alphabet Σ_2. The ID labeling function $\hat{\mu} : T_\Sigma \rightarrow T_{(\{\bullet\}\times D, \Sigma\times D)}$ is now defined by $\hat{\mu}(t) = \tilde{\mu}(t, d_0)$, where

$$\tilde{\mu}(\varepsilon, d) = (\bullet, d),$$
$$\tilde{\mu}(\alpha\langle t', t''\rangle, d) = (\alpha, d)\langle\tilde{\mu}(t', d'), \tilde{\mu}(t'', d'')\rangle \text{ where } \mu(d) = (d', d'')$$

and $\bullet$ is a symbol not in Σ. The state labeling function $\hat{\delta} : T_{(\{\bullet\}\times D, \Sigma\times D)} \rightarrow \bigcup\{Q\}_D$ is defined by:

$$\hat{\delta}(\bullet, d) = \bar{q}_d \text{ and}$$
$$\hat{\delta}((\alpha, d)\langle(t', d'), (t'', d'')\rangle) = \delta_d(\hat{\delta}(t', d'), \hat{\delta}(t'', d''))(\alpha),$$

where δ_d is of type $d' \times d'' \rightarrow d$. Note that since $\hat{\mu}$ attaches d_0 to the root of a tree t, we have $\hat{\delta} \circ \hat{\mu}(t) \in Q_{d_0}$. A tree $t \in T_\Sigma$ is said to be accepted by a guided tree automata M_G if $\hat{\delta} \circ \hat{\mu}(t) \in F$, and the set of trees accepted by M_G is denoted $L(M_G)$.

The following proposition constructively shows how DFTAs can be simulated by GTAs and vice versa.

Proposition 2.3.

(a) Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFTA of size n and $G = (D, \mu, d_0)$ a guide. Then there is a GTA $M_G = (\{P\}_D, \Sigma, \{\gamma\}_D, \{\bar{p}\}_D, E)$ of size n such that $L(M) = L(M_G)$.

(b) Let $M_G = (\{Q\}_D, \Sigma, \{\delta\}_D, \{\bar{q}\}_D, F)$ be a GTA of size n guided by $G = (D, \mu, d_0)$ of size γ. Then there is a DFTA $M = (P, \Sigma, \gamma, p_0, E)$ of size n^γ such that $L(M_G) = L(M)$.

Proof. (Idea)

(a) We define M_G by making a copy of the state space and transition function for each $d \in D$.

(b) M simulates all the transition functions of M_G in parallel.

3 Data structures for guided automata

Decision procedures and large alphabets The decision procedure for M2L on strings associates a language over an alphabet of the form $\Sigma = \mathbb{B}^k$ (where $\mathbb{B} = \{0, 1\}$ is the Booleans) to each formula ϕ, see [11]. The idea is the following: for a word $w \in (\mathbb{B}^k)^*$ of length ℓ, each of the k components defines a bit pattern or *track* of length ℓ. Each free variable of ϕ is assigned to a track and is interpreted as the subset of positions in $\{0, \ldots, \ell - 1\}$ for which the track contains a 1. The language defined for ϕ is the set of strings that interpret ϕ to be true. Since the number of free variables can be large, say 100, the resulting alphabets can be of astronomical size, say 2^{100}.

Shared BDD representation In the MONA implementation of automata on finite strings [4], a shared multi-terminal BDD, called the *Σ-BDD* is used to represent the transition function.

BDDs were originally introduced in [1]. We use the variety defined as follows.

Notation A *Binary Decision Diagram* (BDD) is a rooted, directed graph. Each node ω is either an *internal node* or a *leaf*. A leaf ω defines a *leaf value* in V, where V is a finite set. An internal node ω possesses an *index* together with a *low successor* and a *high successor* such that the index of both successors is higher than the index of ω. Each node ω represents a function $\mathbb{B}^k \to V$ for some k, and we use ω to denote both the node and the function it represents. Thus, if ω is a root and $\mathbf{b}$ is a vector of k bits, then we denote the value of the function on $\mathbf{b}$ by $\omega(\mathbf{b})$.

The kind of BDD defined above is sometimes called a *multi-terminal* BDD. A *shared* BDD is a BDD with multiple roots. In the algorithms in the following sections, we use the *apply* and *restrict* operations described in [1], although we use the term *projection* in place of restriction.

If ω and ω' are roots, then we denote by $\omega * \omega'$ the pairing of functions ω and ω', that is, $\omega * \omega'(\mathbf{b}) = (\omega(\mathbf{b}), \omega'(\mathbf{b}))$. This function can be calculated by a binary BDD apply operation in time bounded by the product of the sizes of ω and ω'.

In our automaton representation, each state of the automaton points to a BDD node, and each leaf value is a state. Given a letter $\alpha \in \mathbb{B}^k$, we may find the value of the transition function by following the BDD nodes according to α from the node pointed to by the state. The leaf reached contains the name of the next state. A MONA representation of a DFA is depicted in Figure 1.

The BDD-based representation of automata on strings allows efficient implementation of standard operations on finite automata (except for minimization, where our current algorithm is quadratic although its behavior in practice is better than quadratic).

BDD-based tree automaton data structure We would like to use a similar representation to gain efficient algorithms for guided tree automata. Thus, we assume that a shared BDD is used for representing the alphabetic part of each transition relation, i.e. the part with signature $\Sigma \to Q$ of $\delta_d : (Q' \times Q'') \to (\Sigma \to Q)$. This BDD is called the *Σ-BDD* and we call the function $\Sigma \to Q$ that it represents the *Σ-behavior*. We use Ω to denote the nodes of the Σ-BDD.

A naïve approach for representing $\delta_d : (Q' \times Q'') \to (\Sigma \to Q)$ is to create an entry for each pair $\langle q', q'' \rangle \in Q' \times Q''$. An entry defines a σ-behavior as a BDD node in a Σ-BDD. This approach leads to an unfortunate quadratic growth, since each of the $|Q'| \times |Q''|$ entries must be explicitly represented.

An alternative approach is to define a binary encoding of the state spaces Q' and Q''. Each state $q'' \in Q''$ has a unique identifier in the range $\{0, \ldots |Q''|-1\}$, which can be encoded by means of a vector of $k = \log(|Q''|)$ bits. Thus for each

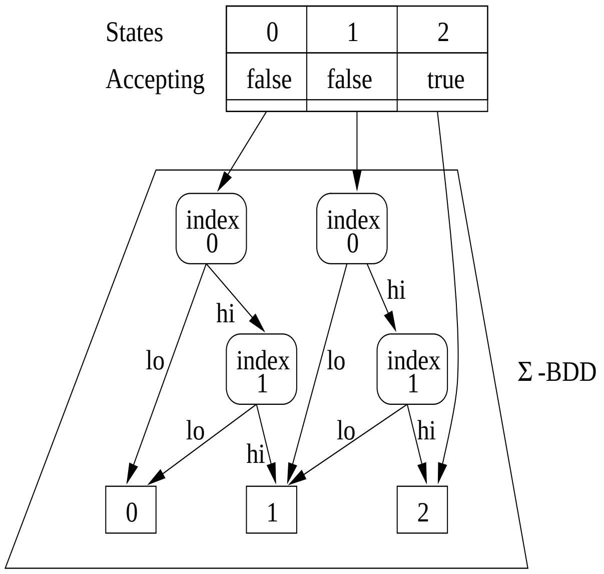

Figure 1. MONA representation of DFA that accepts all strings over $\mathbb{B}^2$ with at least two occurrences of the letter "11".

$q' \in Q'$, we can define a binary function

$$f_{q'}(b_0, \ldots, b_{k-1}) = \delta_d(q', q''),$$

where $b_0, \ldots, b_{k-1}$ is the binary encoding of q''. All these functions can now be represented in a shared BDD. Furthermore, we still use a shared BDD for representing the Σ-behavior $\delta_d(q', q'')$.

For a fixed q', this representation represents the functions $\delta_d(q', q'')$, where q'' ranges over all Q'', succinctly in the case that $\delta_d(q', q'')$ has the same value, which we call a *default Σ-behavior*, for almost all q''. For instance, if 99% of all q'' lead to the same Σ-BDD node, then the state BDD for q' has approximately $0.01 \cdot n$ nodes (when $.01 \cdot n >> \log n$). The total representation is then only one hundredth the size of a total transition table. Thus our notion of *sparsity* is that for any left state, very few right states are of interest—the rest all lead to the default behavior.

An experimental version of the extended MONA system has been implemented based on this representation with acceptable, although not astonishing results. Since the state encoding tends to be random, the compression occurs only for the situations just described. Also, there is a $O(\log n)$ penalty for looking up the transition function for a particular q'', since $O(\log n)$ BDD nodes must be followed. Our experience is that this factor creeps into automata algorithms based on this representation. Therefore, we will pursue in this paper a representation without a logarithmic overhead.

M2L decision procedure on trees Before we propose another representation, we review the decision procedure for M2L on trees in order to understand where sparseness in transition tables may occur.

In its most basic form, the M2L logic on finite trees consists of formulas made out of second-order variables P, logical connectives $\wedge$ and $\neg$, and the quantifier $\exists$. Also, there are a unary, postfix function symbols $\cdot l$ and $\cdot r$, and binary, infix function symbols $\cup$ and $\setminus$ (set difference). $T \cdot l$, where T is a second-order term, is interpreted as the set of left successors of positions in T. Similarly, $T \cdot r$ is the set of right successors of T. The interpretation of $T \cup T'$ is the union of the interpretations of T and T'. The Boolean connections and $\exists$ are interpreted in the usual manner.

Connectives like $\vee$, the quantifier $\forall$, the successor function for first-order terms, and first-order variables can be reduced to expressions in the basic form. An important trick is that a first-order term can be simulated by a second-order variable that is restrained to be a singleton. (We omit a further discussion, which would become very technical.)

We consider two kinds of semantics. The *skeleton semantics* defines the meaning of terms and formulas relative to a finite, binary tree t, called the skeleton. Second-order quantification is restricted to the skeleton, that is, a second-order variable denotes a subset of the nodes of this tree. Consider a formula ϕ with free variables $\mathcal{P} = \{P_1, \cdots, P_k\}$. A *value assignment* relative to t is a binary, labeled tree T that has the same shape as the skeleton and where each label determines set membership status of the variables in $\mathcal{P}$. Thus, T can be regarded as a mapping $T : t \rightarrow \mathbb{B}^k$, which determines the value of P_i to be the set of positions $p \in t$ such that the kth component of $T(p)$ is 1. If ϕ holds under the interpretation T, then we write $T \models_{skel} \phi$. We let $L_{skel}(\phi)$ denote the language of satisfying interpretations. Also, by convention, the successor functions *stutter* at leaves, that is, the left or right successor of a leaf in t is the leaf itself.

The *natural semantics* is that of WS2S, the weak-second order theory of two succesors, which also interprets second variables over only finite subsets. But quantification is not restricted by a skeleton. Also, the successor functions do not stutter anywhere. This semantic interpretation is denoted $\models_{nat}$. Note that for any interpretation of $\mathcal{P}$, there will be infinitely many trees T that describe the interpretation because of the *padding property*: any T can be padded with extra positions labeled $(0, \ldots, 0)$ while still denoting the same collection of finite subsets.

The decision procedure works as for strings in both cases. By structural induction on formulas, we construct automata $A^{\phi,\mathcal{P}}$ on alphabet $\mathbb{B}^k$, where $k = |\mathcal{P}|$, satisfying the correspondence:

$$T \models \phi \text{ iff } T \in L(A^{\phi,\mathcal{P}})$$

Thus, $A^{\phi,\mathcal{P}}$ accepts exactly the labeled trees T that make ϕ true, that is, $L(\phi)$.

All formulas can be assumed to contain basic formulas (those that are not composed from Boolean connectives or quantifiers) that are very simple, like

$P = Q \cdot l$ or $P = Q \cup R$. This is because complex terms can be decomposed by the introduction of more variables. For such simple basic formulas, it is straightforward to construct automata satisfying the correspondence. Also, it is not hard to see that for the conjunction $\phi \wedge \psi$, if $\mathfrak{P}$ corresponds to ϕ and $\mathfrak{Q}$ corresponds to ψ, then the automaton product $\mathfrak{P} \times \mathfrak{Q}$ corresponds to $\phi \wedge \psi$. Negation has an automata-theoretic formulation as complementation, which is achieved by making final states non-final and vice versa. For the case of existential quantification, a projection operation combined with the subset construction can be used under both semantics. Because of the padding property, however, the natural semantics demands an additional automata-theoretic operation discussed in 5.

Sparsity of M2L transition relations Typically, an arbitrary T would not make sense relative to the original formula if it contained first-order variables. For example, if the second-order variable P is used in ϕ to denote a single position (corresponding to the common case where quantification is first-order), then ϕ is trivially false if there are more than one position in the P-track containing a 1. Once a second such position is encountered by the automaton in its bottom-up parsing of the tree, it will go to a "reject-all state" (a graph-theoretic sink). So, intuitively, each state will contain information or *assumptions* specifying the "first-order status" of P, namely whether 0, 1, or more occurrences of a 1 have been encountered. Thus, if we consider a random left state q' and a random right state q'', then a transition to a state other than the "reject-all state" can happen only if q' and q'' for each variable make consistent assumptions about the its status. For example, in the case of a second-order variable P modeling a first-order variable, the "reject-all state" will result from any scenario where each of q' and q'' contains the first-order status assumption that a single 1 in the P-track has already occurred. Therefore, given k first-order variables, the chance that a random pair of states q' and q'' are consistent is ρ^k, where $\rho < 1$. (Here, we have assumed a uniform probability distribution and that the first-order status information is not masked by other information; thus, the number of states is also exponential in k.) This argument could be formalized so as to show that sparsity occurs under some rather representative circumstances when M2L is translated to tree automata.

Our representation Our representation exploits the assumed existence of a preponderant equivalence class by storing it implicitly in a manner similar to the incompletely specified transition functions of [2].

To make notions more precise, consider a transition function δ_d of type $d' \times d'' \to d$. A state $q' \in Q'$ induces an equivalence relation $\equiv_{q',\delta_d}$ on Q'' defined by

$$q''_1 \equiv_{q',\delta_d} q''_2 \ \text{ iff } \ \delta_d(q', q''_1)(\alpha) = \delta_d(q', q''_2)(\alpha) \ \text{ for all } \alpha \in \Sigma.$$

Sparsity means that one equivalence classes of $\equiv_{q',\delta_d}$ contains almost all of Q'', whereas the other equivalence classes only contain a few elements of Q''

each. An equivalence class that contains at least as many elements as any other equivalence class is referred to as a *largest* equivalence class.

In our representation, we store in an array the following information for each of the left states $q' \in Q'$:

- a node in the Σ-BDD, named $q'.\mathit{default}_d$, denoting the default behavior, and
- a set of pairs $(q'', \omega) \in Q'' \times \Omega$, named $q'.\mathit{explicit}_d$.

The subscript d indicates that the associations are with respect to the transition function with target ID d, i.e. the above associations are made for each transition function. Note that this representation is asymmetric in the sense that $\mathit{default}_d$ and $\mathit{explicit}_d$ are defined only for left states. (The choice of left is arbitrary.)

We represent a largest equivalence class $[q''_*]$ of $\equiv_{q',\delta_d}$ implicitly by setting $q'.\mathit{default}_d = \delta_d(q', q''_*)$. For all $q'' \notin [q''_*]$, the pair $(q'', \delta_d(q', q''))$ is stored in $q'.\mathit{explicit}_d$. Thus for arbitrary $q'' \in Q''$ and $\alpha \in \Sigma$, we have

$$\delta_d(q', q'')(\alpha) = \begin{cases} \omega(\alpha) & \text{if } (q'', \omega) \in q'.\mathit{explicit}_d \text{ for some } \omega \\ q'.\mathit{default}_d(\alpha) & \text{otherwise} \end{cases}$$

Thus determining $\delta_d(q', q'')$ amounts to a set lookup. The representation of a transition function is depicted in Figure 2.

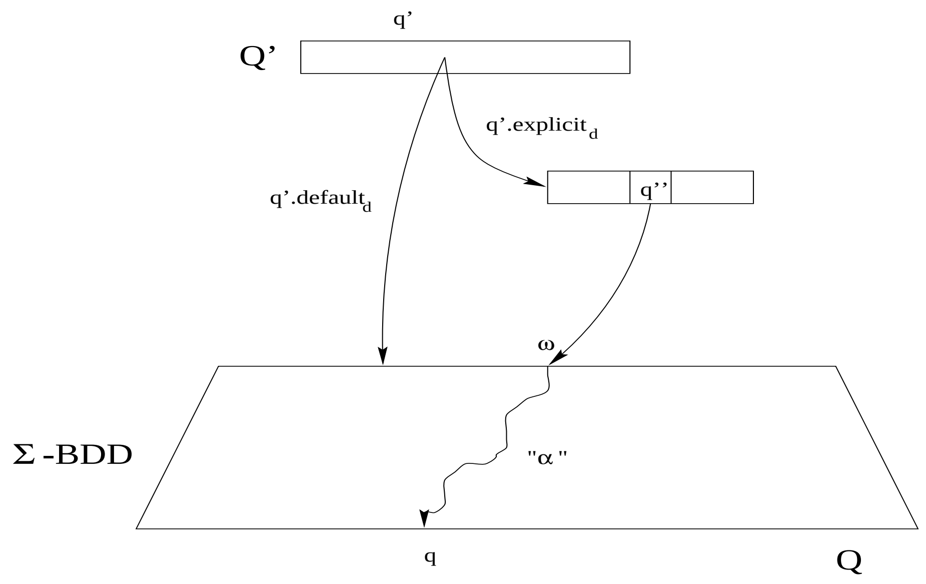

Figure 2. Representation of a transition function δ_d of type $d' \times d'' \to d$.

Representing the whole GTA A GTA $M_G = (\{Q\}_D, \Sigma, \Delta, \{\bar{q}\}_D, F)$ is implemented by a structure as described above for each transition function. We use a single BDD, shared among all transition functions, to encode Σ. In addition, the implementation has a bit vector of size $|Q_{d_0}|$ to represent F.

4 Product

Let $\mathfrak{P} = (\{P\}_D, \Sigma, \{\gamma\}_D, \{\bar{p}\}_D, E)$ and $\mathfrak{Q} = (\{Q\}_D, \Sigma, \{\phi\}_D, \{\bar{q}\}_D, F)$ be tree automata guided by $G = (D, \mu, d_0)$. The product automaton of $\mathfrak{P}$ and $\mathfrak{Q}$ is then $\mathfrak{R} = (\{R\}_D, \Sigma, \{\delta\}_D, \{\bar{r}\}_D, H)$, where

$$\begin{aligned}
R_d &= P_d \times Q_d \\
\delta_d &= \gamma_d \times \phi_d : ((P'_{d'} \times Q'_{d'}) \times (P''_{d''} \times Q''_{d''})) \to (\Sigma \to (P_d \times Q_d)), \text{where} \\
&\quad \gamma_d \times \phi_d(\langle p', q'\rangle, \langle p'', q''\rangle)(\alpha) = \langle \gamma_d(p', p'')(\alpha), \phi_d(q', q'')(\alpha)\rangle \\
\bar{r_d} &= \langle \bar{p}_d, \bar{q}_d\rangle \\
H &= E \times F
\end{aligned}$$

A relatively simple approach

We begin by describing a simple approach of performing a product of two guided tree automata. This approach does not construct the whole product spaces $P_d \times Q_d$; instead, only reachable pairs of states are calculated.

The algorithm maintains two sets for each $d \in D$:

- R_d now denotes subset of $P_d \times Q_d$ that consists of all pairs encountered so far during the computation. The set is initialized to $\{\langle \bar{p}_d, \bar{q}_d\rangle\}$.
- $\mathit{unprocessed}_d$ is the subset of the reached pairs R_d for which the transition function has not been calculated. Initially, $\mathit{unprocessed}_d$ is also the singleton set $\{\langle \bar{p}_d, \bar{q}_d\rangle\}$.

The transition function to be calculated of type $d' \times d'' \to d$ is represented by explicit sets of the form $\langle p', q'\rangle.\mathit{explicit}_d$ and default values $\langle p', q'\rangle.\mathit{default}_d$.

Consider a pair $\langle \hat{p}, \hat{q}\rangle$ from some set $\mathit{unprocessed}_{\hat{d}}$. For every transition function δ_d of type $\hat{d} \times d'' \to d$, we must calculate all transitions involving $\langle \hat{p}, \hat{q}\rangle$. We say that we process $\langle \hat{p}, \hat{q}\rangle$ under the *left view*. Specifically for each $\langle p'', q''\rangle \in R_{d''}$, we calculate

$$\omega = \gamma_d(\hat{p}, p'') * \phi_d(\hat{q}, q'')$$

by performing a binary BDD apply. Since ω equals $\delta_d(\langle \hat{p}, \hat{q}\rangle, \langle p'', q''\rangle)$, we can extend δ_d for this value by inserting $(\langle p'', q''\rangle, \omega)$ in $\langle \hat{p}, \hat{q}\rangle.\mathit{explicit}_d$.

We also have to process under the *right view*: for transition functions of type $d' \times \hat{d} \to d$, we compute for each pair $\langle p', q'\rangle \in R_{d'}$

$$\omega = \gamma_d(p', \hat{p}) * \phi_d(q', \hat{q})$$

and insert $(\langle \hat{p}, \hat{q}\rangle, \omega)$ into $\langle p', q'\rangle.\mathit{explicit}_d$.

After all transitions where $\langle \hat{p}, \hat{q}\rangle$ occurs have been considered, $\langle \hat{p}, \hat{q}\rangle$ is removed from $\mathit{unprocessed}_d$.

During computation of each BDD apply, all pairs that occur as BDD leaf values and that have not been encountered before are inserted into $\mathit{unprocessed}_d$ and R_d.

The algorithm continues until every set $\mathit{unprocessed}_d$ is empty. It is easy to verify that the algorithm maintains the *processed invariant*:

> For all $d \in D$, the transition functions δ_d are totally defined for $(R_{d'} \setminus \mathit{unprocessed}_{d'}) \times (R_{d''} \setminus \mathit{unprocessed}_{d''})$.

Hence upon termination, all transition functions are defined (through the sets $\mathit{explicit}_d$) for all reachable pairs.

To compress the representation of the state part of the transition relation, an extra sweep upon termination is necessary to adjust the $\mathit{default}_d$ values. Alternatively, this can be done on-line by keeping track of the frequency of BDD nodes inserted in the $\mathit{explicit}_d$ sets. In this way, space cost is decreased.

To analyze the above algorithm, assume for simplicity that the number of reachable states in any state space of the resulting automaton is bounded by N, i.e. $|R_d| \leq N$ for all d. The time used per state in the resulting automaton is $O(N)$, and hence in total the algorithm uses time $O(N^2)$. (In this paper, we ignore the size of the Σ-BDDs in our complexity estimates.)

A more efficient approach

The algorithm we propose in this paper is designed to take advantage of the $\mathit{default}_d$ values of the automata $\mathfrak{P}$ and $\mathfrak{Q}$. In certain cases, this method makes it possible to obtain an $O(N^{\frac{3}{2}})$ time bound in contrast to the quadratic bound above.

The main idea explained for the left view is: when processing a pair $\langle \hat{p}, \hat{q} \rangle$ for a transition function δ_d of type $\hat{d} \times d'' \to d$, we can often avoid considering all pairs $\langle p'', q'' \rangle$ of state space $R_{d''}$. This is accomplished by tentatively letting $\langle \hat{p}, \hat{q} \rangle.\mathit{default}$ equal $\hat{p}.\mathit{default}_d * \hat{q}.\mathit{default}_d$. If

$$\text{either } p'' \in \hat{p}.\mathit{explicit}_d \text{ or } q'' \in \hat{q}.\mathit{explicit}_d, \tag{1}$$

then we do have to consider $\langle p'', q'' \rangle$; otherwise, when (1) does not hold, it is the case that $\delta_d(\langle \hat{p}, \hat{q} \rangle, \langle p'', q'' \rangle) = \langle \hat{p}, \hat{q} \rangle.\mathit{default}$, and hence we do not need to insert $\langle p'', q'' \rangle$ into the set $\langle \hat{p}, \hat{q} \rangle.\mathit{explicit}_d$.

Similarly, under the right view, for transition function δ_d of type $d' \times \hat{d} \to d$, we need to insert $\langle \hat{p}, \hat{q} \rangle$ in $\langle p', q' \rangle.\mathit{explicit}_d$ only if

$$\text{either } \hat{p} \in p'.\mathit{explicit}_d \text{ or } \hat{q} \in q'.\mathit{explicit}_d. \tag{2}$$

Since the purpose of the efficient approach is to avoid considering data implied by the default behaviors, we need to introduce additional data structures in which the pairs that need consideration can be explicitly looked up.

To do this, we need to precompute some information for each transition function δ of type $d' \times d'' \to d$. For automaton $\mathfrak{P}$, the following is computed:

– To each state $p' \in P_{d'}$, we define the set

$$p' \rightarrow expl_d = \{p'' \mid (p'', \omega) \in p'.explicit\}.$$

These sets are calculated in order to maintain an explicit representation of the pairs that satisfy (1).

– To each state $p'' \in P_{d''}$ we let

$$p'' \leftarrow expl_d = \{p' \mid (p'', \omega) \in p'.explicit\}.$$

These sets are for the computational handling of requirement (2).

Note that $p'' \in p' \rightarrow expl_d$ if and only if $p' \in p'' \leftarrow expl_d$. Similar information is calculated for $\mathfrak{Q}$. These calculations can be carried out in linear time of the size of the representation.

In addition to the sets of reachable pairs R_d and the sets *unprocessed*, the algorithm maintains *critical pairs sets*, which are subsets of the reachable pairs sets. For automaton $\mathfrak{P}$, these sets are

– $R|_{p' \rightarrow expl_d} = \{\langle p'', q'' \rangle \in R_{d''} \mid p'' \in p' \rightarrow expl_d\}$ and
– $R|_{p'' \leftarrow expl_d} = \{\langle p', q' \rangle \in R_{d'} \mid p' \in p'' \leftarrow expl_d\}$

for all $p' \in P_{d'}$ and $p'' \in P_{d''}$ where $\mu(d) = (d', d'')$. Similar sets are defined for automaton $\mathfrak{Q}$. The *critical pairs invariant* is that the data structures representing the critical pairs sets satisfy these definitions.

The sets $R|_{p' \rightarrow expl_d}$ determine whether a pair $\langle p'', q'' \rangle$ belongs to the *explicit* list of $\langle \hat{p}, \hat{q} \rangle$ under the left view according to (1): if $\langle p'', q'' \rangle \in R|_{\hat{p} \rightarrow expl_d} \cup R|_{\hat{q} \rightarrow expl_d}$, then p'' is found in $\hat{p}.explicit_d$ or q'' is found in $\hat{q}.explicit_d$. In either case, the *explicit* set for $\langle \hat{p}, \hat{q} \rangle$ must contain $\langle p'', q'' \rangle$ (unless the Σ-behavior of $(\langle \hat{p}, \hat{q} \rangle, \langle p'', q'' \rangle)$ happens to be the default $\hat{p}.default_d * \hat{q}.default_d$). On the other hand, if $\langle p'', q'' \rangle \notin R|_{\hat{p} \rightarrow expl_d} \cup R|_{\hat{q} \rightarrow expl_d}$, then the value of $\delta_d(\langle \hat{p}, \hat{q} \rangle, \langle p'', q'' \rangle)$ is the default behavior. Thus processing of pair $\langle \hat{p}, \hat{q} \rangle$ under the left view need to involve only right states $\langle p'', q'' \rangle \in R|_{\hat{p} \rightarrow expl_d} \cup R|_{\hat{q} \rightarrow expl_d}$.

The sets $R|_{p'' \leftarrow expl_d}$ play a similar role for the processing under the right view.

If we assume that all $p \rightarrow expl_d$, $p \leftarrow expl_d$, $q \rightarrow expl_d$, and $q \leftarrow expl_d$ sets have been pre-computed, then Figure 3 shows how the *unprocessed* sets and the various versions of the sets of reachable states are extended as a result of new pairs that are generated during an apply operation:

Figure 4 summarizes how the main part of the product algorithm works. The correctness of the algorithms is argued below.

After the product automaton has been calculated, the product algorithm must also ensure that the default transition in each case corresponds to an equivalence class of maximal size. If not, a maximal equivalence class will be converted from an explicit representation to a default representation and the former default representation is made explicit. These calculations can be carried out in time linear to the size of the product automaton.

```
fun $apply\_and\_extend(\omega_1, \omega_2, \hat{d}) =$
    use the BDD apply operation to calculate $\omega = \omega_1 * \omega_2$
    for each $\langle \hat{p}, \hat{q} \rangle$ encountered (during the apply)
    that is not already in $R_{\hat{d}}$ do
        add $\langle \hat{p}, \hat{q} \rangle$ to $R_{\hat{d}}$ and to $unprocessed_{\hat{d}}$
        for $\hat{d} \times d'' \to d$ for some $d, d''$ do
            add $\langle \hat{p}, \hat{q} \rangle$ to $R|_{p'' \leftarrow expl_d}$ for each $p'' \in \hat{p} \to expl_d$
            add $\langle \hat{p}, \hat{q} \rangle$ to $R|_{q'' \leftarrow expl_d}$ for each $q'' \in \hat{q} \to expl_d$
        od
        for $d' \times \hat{d} \to d$ for some $d, d'$ do
            add $\langle \hat{p}, \hat{q} \rangle$ to $R|_{p' \to expl_d}$ for each $p' \in \hat{p} \leftarrow expl_d$
            add $\langle \hat{p}, \hat{q} \rangle$ to $R|_{q' \to expl_d}$ for each $q' \in \hat{q} \leftarrow expl_d$
        od
    od
    return $\omega$
```

Figure 3. Algorithm for auxiliary function *apply_and_extend*.

Correctness It can be seen that *apply_and_extend* satisfies both the processed invariant and the critical pairs invariant (the details of this argument are left to the reader).

The first **for**-loop in Figure 4 initializes the critical pairs set so that the critical pairs invariant holds. Also, since $R_d = unprocessed_d$, the processed invariant is trivially true.

Let us consider the apply operations that are executed after a pair $\langle \hat{p}, \hat{q} \rangle$ is removed from $unprocessed_{\hat{d}}$ under the left view $(\hat{d} \times d'' \to d)$. After these apply operations, it holds that δ_d is defined correctly for all $(\langle \hat{p}, \hat{q} \rangle, \langle p'', q'' \rangle)$ in $R_{critical}$. But $R_{critical}$ contains all $\langle p'', q'' \rangle$ in $R_{d''}$ (as evaluated before the operations) that are not known to induce the default behavior for $\langle \hat{p}, \hat{q} \rangle$. In particular, δ_d is defined for all $(\langle \hat{p}, \hat{q} \rangle, \langle p'', q'' \rangle)$, where $\langle p'', q'' \rangle \in R_{d''} \backslash unprocessed_{d''}$, where $R_{d''}$ and $unprocessed_{d''}$ stand for the sets before the apply operations (and after $\langle \hat{p}, \hat{q} \rangle$ has been removed from $unprocessed_{d''}$). This property also holds if $\hat{d} = d''$. Since $R_{d''} \backslash unprocessed_{d''}$ remains constant during *apply_and_extend*, the property also holds after the operations, and it can be seen that it is sufficient for guaranteeing the processed invariant. The critical pairs invariant also holds throughout the apply operations, and it is not affected by the removal of a product state from the *unprocessed* set.

Analysis We analyze the effect of the default representation here. So assume that there is only one state space ID d and that the spaces of $\mathfrak{P}$ and $\mathfrak{Q}$ are bounded by n. In addition, let $t(n)$ be a function of n that bounds the number

```
$R_d$, $unprocessed_d = \{\langle \bar{p}_d, \bar{q}_d \rangle\}$ for all $d \in D$
for each $\hat{d}$, let $\langle \hat{p}, \hat{q} \rangle$ be the state in $R_{\hat{d}}$ and do
    for $\hat{d} \times d'' \to d$ for some $d, d''$ do
        add $\langle \hat{p}, \hat{q} \rangle$ to $R|_{p'' \leftarrow expl_d}$ for each $p'' \in \hat{p} \to expl_d$
        add $\langle \hat{p}, \hat{q} \rangle$ to $R|_{q'' \leftarrow expl_d}$ for each $q'' \in \hat{q} \to expl_d$
    od
    for $d' \times \hat{d} \to d$ for some $d, d'$ do
        add $\langle \hat{p}, \hat{q} \rangle$ to $R|_{p' \to expl_d}$ for each $p' \in \hat{p} \leftarrow expl_d$
        add $\langle \hat{p}, \hat{q} \rangle$ to $R|_{q' \to expl_d}$ for each $q' \in \hat{q} \leftarrow expl_d$
    od
od
while $unprocessed_{\hat{d}} \neq \emptyset$ for some $\hat{d} \in D$ do
    remove a pair $\langle \hat{p}, \hat{q} \rangle$ from $unprocessed_{\hat{d}}$
    for $\hat{d} \times d'' \to d$ for some $d, d''$ do
        $R_{critical} = R|_{\hat{p} \to expl_d} \cup R|_{\hat{q} \to expl_d}$
        $\langle \hat{p}, \hat{q} \rangle.default_d = apply\_and\_extend(\hat{p}.default_d, \hat{q}.default_d, d)$
        for all $\langle p'', q'' \rangle \in R_{critical}$ do
            $\omega = apply\_and\_extend(\gamma_d(\hat{p}, p''), \phi_d(\hat{q}, q''), d)$
            if $\omega \neq \langle \hat{p}, \hat{q} \rangle.default_d$ then
                add $(\langle p'', q'' \rangle, \omega)$ to $\langle \hat{p}, \hat{q} \rangle.explicit_d$
            fi
        od
    od
    for $d' \times \hat{d} \to d$ for some $d, d'$ do
        $R_{critical} = R|_{\hat{p} \leftarrow expl_d} \cup R|_{\hat{q} \leftarrow expl_d}$
        for all $\langle p', q' \rangle \in R_{critical}$ do
            $\langle p', q' \rangle.default_d = apply\_and\_extend(p'.default_d, q'.default_d, d)$
            $\omega = apply\_and\_extend(\gamma_d(p', \hat{p}), \phi_d(q', \hat{q}), d)$
            if $\omega \neq \langle p', q' \rangle.default_d$ then
                add $(\langle \hat{p}, \hat{q} \rangle, \omega)$ to $\langle p', q' \rangle.explicit_d$
            fi
        od
    od
```

Figure 4. Algorithm for product construction for GTAs.

of elements in any of the sets $p \to expl_d$, $p \leftarrow expl_d$, $q \to expl_d$, $q \leftarrow expl_d$.

We then claim that our algorithm uses at most time $O(t(n)n)$ per state in the product automaton $\mathfrak{R}$. Consider the pair $\langle \hat{p}, \hat{q} \rangle$. The transition function δ_d of type $\hat{d} \times d'' \to d$ is extended under the left view for only those pairs $\langle p'', q'' \rangle$ from $R_{d''}$ for which either $p'' \in \hat{p}.explicit_d$ or $q'' \in \hat{q}.explicit_d$. The number of such pairs is bounded by $|\hat{p}.explicit_d| \cdot |Q_{d''}| + |\hat{q}.explicit_d| \cdot |P_{d''}| = O(t(n) \cdot n)$. (Similar considerations apply to the calculations done under the right view.) It can be argued that updating the critical pairs sets can be done within time $O(t(n)+n)$ per product state. Thus in total, the algorithm spends time $O(t(n) \cdot n)$ per product state.

In contrast, the simple algorithm visits all reachable pairs, so it uses time $O(n^2)$ per product automaton state when all states are reachable. Hence for $t(n) < o(n)$, our time of $O(n \cdot t(n))$ is asymptotically better.

To put this in a sharper light, assume $t(n) = O(1)$. The resulting product automata could have size $N = n^2$, and the simple algorithm would be of time complexity $O(N^2)$; in contrast, our algorithm uses time $O(t(n) \cdot n)$ per state of the resulting automata, e.g. it uses total time $O(n \cdot t(n) \cdot n^2) = O(n^3) = O(N^{\frac{3}{2}})$.

5 Projection and determinization

Existential quantification in M2L corresponds to the automata-theoretic operation of projecting the transition relation on a new alphabet where the quantified variable is no longer described. The resulting nondeterministic automaton must then be determinized by a subset construction.

Let π_i denote the *tuple projection on component* i, that is, $\pi_i(\hat{\mathbf{b}})$, where $\hat{\mathbf{b}} \in \mathbb{B}^k$, is the tuple $\mathbf{b}$ with the ith component removed $(1 \leq i \leq k)$. Intuitively, automaton projection on component i is the process of converting a guided tree automaton $\mathfrak{Q}$ recognizing a language L over $\mathbb{B}^k$ to a nondeterministic guided tree automaton $\mathfrak{Q}'$ over $\mathbb{B}^{k-1}$ by removing track i. Thus, $L(\mathfrak{Q}')$ consists of all trees over $\mathbb{B}^{k-1}$ that are the projections on component i of trees in L, ie., that are gotten from trees in L by applying tuple projection on i to each label. This language is denoted $\pi_i(L)$.

It can then be shown that $L_{skel}(\exists P_i : \phi) = \pi_i(L_{skel}(\phi))$, which derives our interest in the projection operation. (Later, we shall look at the modifications necessary to accommodate the natural semantics.)

The automaton $\mathfrak{Q}'$ is constructed by applying the BDD projection operation π on the Σ-BDD associated with each transition function. Applied to a Σ-BDD ω, the projection operation results in a BDD ω_π representing the function

$$\omega_\pi(\mathbf{b}) = \{\omega(\hat{\mathbf{b}}) \mid \pi_i(\hat{\mathbf{b}}) = \mathbf{b}\},$$

Note that for each $\mathbf{b}$, there are exactly two $\hat{\mathbf{b}}$s that satisfy the criterion above. Thus the leaves of $\pi(\omega)$ are sets with one or two elements. The nondeterministic automaton represented by the projection BDDs must then be determinized so that it can later be minimized. We will describe an operation that simultaneously carries out the projection and determinization of $\mathfrak{Q}$. If the automaton $\mathfrak{Q}$ denotes a language L, then our projection and determinization construction results in an automaton that represents the projected language $\pi_i(L)$.

To be more precise, consider a GTA $\mathfrak{Q} = (\{Q\}_D, \Sigma, \{\delta\}_D, \overline{p}_D, F)$ guided by $G = (D, \mu, d_0)$, where $\Sigma = \mathbb{B}^k$. The *i-projected power set automaton* is the automaton $(\mathcal{P}(Q_d), \Sigma, \gamma_d, \{\{\overline{p}_d\}\}_D, \{\{Q \mid Q \cap F_d \neq \emptyset\}\}_D)$, where γ_d is defined as

$$\gamma_d(Q', Q'')(\mathbf{b}) = \bigcup\{\delta_d(q', q'')(\hat{\mathbf{b}}) \mid \pi_i(\hat{\mathbf{b}}) = \mathbf{b}\}$$

In practice, we are of course interested in only calculating the transition function for the reachable subsets.

Algorithmically, γ can be calculated as follows. Consider $d' \times d'' \to d$, a subset Q' of $Q_{d'}$, and a subset Q'' of $Q_{d''}$. Then, $\gamma_d(Q', Q'')$ is the value of

$$\overset{\cup}{*}\{\pi(\delta(q', q'')) \mid q' \in Q' \text{ and } q'' \in Q''\}$$

where $\overset{\cup}{*}$, the *union apply operation*, calculates for a collection of BDDs that map into sets the BDD that maps each $\mathbf{b}$ to the union of the sets mapped to by the collection.

Adapting the product algorithm

To take advantage of the default representation, we adapt the techniques developed for the product algorithm. For a transition δ of type $d' \times d'' \to d$, we introduce *critical subset sets*, which are similar to the critical pairs sets of Section 4. The set $R|_{q' \to expl_d}$ denotes the critical subsets Q'' such that q' occur in $q''.explicit_d$ for some $q'' \in Q''$.

$$R|_{q' \to expl_d} = \{Q'' \in R_{d''} \mid Q'' \cap q' \to expl_d \neq \emptyset\}$$

Similarly, $R|_{q'' \leftarrow expl_d}$ denotes the subsets Q', where q'' occur in $q'.explicit_d$ for some $q' \in Q'$.

$$R|_{q'' \leftarrow expl_d} = \{Q' \in R_{d'} \mid Q' \cap q'' \leftarrow expl_d \neq \emptyset\}$$

We assume that we have at our disposal a binary version of $\overset{\cup}{*}$, which "unions" together the leaves of two BDDs. The subset construction can be implemented as shown in Figure 5 and Figure 6. This algorithm is a straightforward adaptation of the product algorithm. For example, the default subset state for a subset state Q' is the union apply of the projections of the default behaviors of states in Q'.

Improvements

A major expense in the algorithm just outlined is the repeated calculations of BDDs of the form $\overset{\cup}{*}\{\pi_\delta(q', q'') \mid q' \in Q', q'' \in Q''\}$, where $\pi_\delta(q', q'') = \pi(\delta_d(q', q''))$ for some appropriate d. We also denote this calculation as $\overset{\cup}{*}\{\pi_\delta(Q', Q'')\}$. It can be seen by an inductive argument that if a subset such as Q' is reachable, then there are subsets $Q'_1, \ldots, Q'_\ell$, $\ell \geq 2$, such that $Q' = Q'_1 \cup \cdots \cup Q'_\ell$ and each Q'_i is a singleton or is reachable [3]. (It is unfortunately the case that a set $\{q_1, q_2\}$ may be reachable without $\{q_1\}$ and $\{q_2\}$

```
fun $apply\_extend(\{w_1, \ldots, w_k\}, \hat{d})$ =
    calculate $\omega = \omega_1 \overset{\cup}{*} \omega_2 \overset{\cup}{*} \cdots \overset{\cup}{*} \omega_k$ (in some order)
    for each new subset of states $\hat{Q}$ not already in $R_{\hat{d}}$ obtained as
    the value of a leaf in the last apply operation above do
        insert $\hat{Q}$ into $R_{\hat{d}}$ and $unprocessed_{\hat{d}}$
        for $\hat{d} \times d'' \to d$ for some $d, d''$ do
            add $\hat{Q}$ to $R|_{q'' \leftarrow expl_d}$ for each $q'' \in \hat{q} \to expl_d$ for some $\hat{q} \in \hat{Q}$
        od
        for $d' \times \hat{d} \to d$ for some $d, d'$ do
            add $\hat{Q}$ to $R|_{q' \to expl_d}$ for each $q' \in \hat{q} \leftarrow expl_d$ for some $\hat{q} \in \hat{Q}$
        od
    od
    return the BDD for $\omega$
```

Figure 5.

being reachable; this may happen, for example, if the set $\{q_1, q_2\}$ is a leaf of a projection operation from the initial state.) For simplicity, we assume next that $\ell = 2$ and we say that Q'_1 and Q''_2 form a *binary decomposition* of Q'.

As a further simplification, assume that $|Q'| = |Q''| = n = 2^N$. Also, assume that each Q' has a *binary decomposition* into disjoint sets Q'_1 and Q'_2 and that a similar property holds for Q'' sets.

Then the value of $\overset{\cup}{*}\{\pi_\delta(Q', Q'')\}$ can be calculated as $\pi_\delta(Q'_1, Q''_1) \overset{\cup}{*} \pi_\delta(Q'_1, Q''_2) \overset{\cup}{*} \pi_\delta(Q'_2, Q''_2) \overset{\cup}{*} \pi_\delta(Q'_2, Q''_2)$. Thus, if $\alpha(N)$ is the total number of apply operations for sets of size 2^N, then $\alpha(N) = 4 \cdot \alpha(N-1) + 3$, which has the solution $\alpha(N) = 4^N - 1$. In contrast, the direct calculation involves $(2^N)^2$ projection applys and $(2^N)^2 - 1$ union applys, in total $2 \cdot 4^N - 1$ apply operations. Therefore, the decomposition method requires approximately half as many apply operations for $N \geq 3$ or $n \geq 8$.

Further benefits of decomposition It can be seen that there are 7 ways of arranging the three union applys in a calculation of $\overset{\cup}{*}\{\pi_\delta Q', Q''\}$ assuming a binary decomposition. Some are better than others if we assume that the two components of any reachable set are also reachable. For example, if

$$\omega_1 = \pi_\delta(Q'_1, Q''_1) \overset{\cup}{*} \pi_\delta(Q'_1, Q''_2) \text{ and}$$
$$\omega_2 = \pi_\delta(Q'_2, Q''_1) \overset{\cup}{*} \pi_\delta(Q'_2, Q''_2)$$

then $\overset{\cup}{*}\{\pi_\delta(Q', Q'')\} = \omega_1 \overset{\cup}{*} \omega_2$. The point is that both ω_1 and ω_2 are results of transition function calculations involving reachable sets. Thus, the extra cost

```
$R_d, unprocessed_d = \{\{\overline{q}_d\}\}$ for all $d \in D$
**for** each $\hat{d}$, let $\hat{Q}$ be the subset in $R_{\hat{d}}$ and **do**
    **for** $\hat{d} \times d'' \rightarrow d$ for some $d, d''$ **do**
        add $\hat{Q}$ to $R|_{q'' \leftarrow expl_d}$ for each $q'' \in \hat{q} \rightarrow expl_d$ for some $\hat{q} \in \hat{Q}$
    **od**
    **for** $d' \times \hat{d} \rightarrow d$ for some $d, d'$ **do**
        add $\hat{Q}$ to $R|_{q' \rightarrow expl_d}$ for each $q' \in \hat{q} \leftarrow expl_d$ for some $\hat{q} \in \hat{Q}$
    **od**
**od**
**while** $unprocessed_{\hat{d}} \neq \emptyset$ for some $\hat{d} \in D$ **do**
    remove a set $\hat{Q}$ from $unprocessed_{\hat{d}}$
    **for** $\hat{d} \times d'' \rightarrow d$ for some $d, d''$ **do**
        $\hat{Q}.default_d = apply\_extend(\{\pi(\hat{q}.default_d) \mid \hat{q} \in \hat{Q}\}, d)$
        $R_{critical} = \bigcup_{\hat{q} \in \hat{Q}} R|_{\hat{q} \rightarrow expl_d}$
        **for** all $Q'' \in R_{critical}$ **do**
                $\omega = apply\_extend(\{\pi(\delta_d(\hat{q}, q'')) \mid \hat{q} \in \hat{Q}, q'' \in Q''\}, d)$
                **if** $\omega \neq \hat{Q}.default_d$ **then**
                    add $(Q'', \omega)$ to $\hat{Q}.explicit_d$
                **endif**
        **od**
    **od**
    **for** $d' \times \hat{d} \rightarrow d$ for some $d, d'$ **do**
        $R_{critical} = \bigcup_{\hat{q} \in \hat{Q}} R|_{\hat{q} \leftarrow expl_d}$
        **for** all $Q' \in R_{critical}$ **do**
                $Q'.default_d = apply\_extend(\{\pi(q'.default_d) \mid q' \in Q'\}, d)$
                $\omega = apply\_extend(\{\pi(\delta_d(q', \hat{q})) \mid \hat{q} \in \hat{Q}, q' \in Q'\}, d)$
                **if** $\omega \neq Q'.default_d$ **then**
                    add $(\hat{Q}, \omega)$ to $Q'.explicit_d$
                **endif**
        **od**
    **od**
**od**
```

Figure 6. Algorithm for projection of GTAs.

of calculating $\overset{\cup}{*}\{\pi(\delta Q', Q'')\}$ is only one apply operation if all such results are cached.

In practice, we have chosen to work with only the binary decomposition. Subsets that have a decomposition with more than two subsets are forced into a form consisting of several binary decompositions. Consequently, the subsets for which we calculate transition functions are generally not reachable.

The quotient operation

Under the natural semantics, the automaton for $\psi \equiv \exists P_i : \phi$ cannot be obtained by the project and determinize operation just described. The problem is that a satisfying interpretation T of ψ could be "smaller" than a witness $\hat{T}$ satisfying ϕ. This happens when the domain of T is properly included in the domain of $\hat{T}$, which could be an interpretation that assigns elements to P_i outside T. More formally, we solve this problem as follows.

Let

$$\begin{array}{c} L\backslash L' = \{T'' \mid \text{for some } T \text{ and some } T_1' \in L', \ldots, T_n' \in L', \} \\ T'' = T(T_1', \ldots, T_n') \in L \end{array}$$

be the quotient of L by L', where if T has n leaves (canonically ordered according to some principle), then $T(T_1', \ldots, T_n')$ is the tree that is gotten by inserting T_i' at leaf i in T. Also, let L_i consist of all trees that are labeled with 0 in components different from i. Then, it can be seen that $L_{nat}(\psi) = \pi_i(L_{nat}(\phi)\backslash L_i)$ [10].

In practice, the quotient operation $L(\phi)\backslash L_i$ is quite easy handled. It suffices to replace the initial subset states $\{\overline{q}_d\}$ with I_d, where I_d are states that are reachable along paths labeled with letters that are 0 everywhere except in component i, see Figure 5.

```
procedure $zero\_path\_states$ $(\omega, j)$
return $\{\omega(0^m), \omega(0^{i-1}10^{m-i})\}$

procedure $quotient((\{Q\}_D, \Sigma, \{\delta\}_D, \{\bar{q}\}, F), i)$
$unprocessed_d, I_d \ = \ \{\bar{q}_d\}$ for all $d \in D$
while $unprocessed_{\hat{d}}$ is not empty for some $\hat{d}$ do
    pick (and remove) a state $\hat{q}$ from $unprocessed_{\hat{d}}$
    let $d$ be such that $\hat{d} \times d'' \to d$
    for $q'' \in I_{d''}$ do
        $unprocessed_d \ = \ unprocessed_d \cup (\text{zero\_path\_states}(\delta(\hat{q}, q''), i) \setminus I_d)$
        $I_d \ = \ I_d \cup \ zero\_path\_states \ (\delta(\hat{q}, q''), i)$
    od
    let $d$ be such that $d' \times \hat{d} \to d$
    for $q' \in I_{d'}$ do
        $unprocessed_d \ = \ unprocessed_d \cup (\text{zero\_path\_states}(\delta(q', \hat{q}), i) \setminus I_d)$
        $I_d \ = \ I_d \cup \ zero\_path\_states \ (\delta(q', \hat{q}), i)$
    od
od
return $(\{Q\}_D, \Sigma, \{\delta\}_D, \{I\}_D, F)$
```

Figure 7. Algorithm for the quotient operation.

6 Minimization

Minimizing guided tree automata is a rather complex task compared to the minimization of ordinary tree automata (which is already a non-trivial affair that as far as we know has not been described in the literature from an algorithmic point of view; but see [8] for an elegant proof that a minimum automaton exists). Before discussing the minimization process, we extend the notation provided by [9]:

Notation A *partition* $\mathcal{P}$ of a finite set U is a set of disjoint subsets of U such that the union of these sets is all of U. The elements of a partition are called its *blocks*. A *refinement* $\mathcal{Q}$ of $\mathcal{P}$ is a partition such that any block of $\mathcal{Q}$ is a subset of a block of $\mathcal{P}$. We let $[q]_{\mathcal{P}}$ denote the block of the partition $\mathcal{P}$ containing the element q, and when no confusion arises, we drop the subscript.

Let $M_G = (\{Q\}_D, \Sigma, \{\delta\}_D, \{\bar{q}\}_D, F)$ be a GTA guided by $G = (D, \mu, d_0)$, and let $\{\mathcal{P}_d\}_{d \in D}$ be a family of partitions such that $\mathcal{P}_d$ is a partition of Q_d. We extend the shorthand notation introduced in the previous sections, and we write $\mathcal{P}$ for the partition $\mathcal{P}_d$, $\mathcal{P}'$ for $\mathcal{P}_{d'}$ etc. when no confusion occurs. Let $\{\mathcal{Q}\}_D$ be a refinement of $\{\mathcal{P}\}_D$, i.e. $\mathcal{Q}_d$ is a refinement of $\mathcal{P}_d$ for all $d \in D$. Let $\delta_d \in \{\delta\}_D$ be a transition function of type $d' \times d'' \to d$.

A block B' of $\mathcal{Q}'$ *δ_d-respects* $\mathcal{P}_d$ if

$$\forall q_1', q_2' \in B', \forall q'' \in Q'', \forall \alpha \in \Sigma : [\delta_d(q_1', q'')(\alpha)]_{\mathcal{P}_d} = [\delta_d(q_2', q'')(\alpha)]_{\mathcal{P}_d}$$

Similarly a block B'' of $\mathcal{Q}''$ *δ_d-respects* $\mathcal{P}_d$ if

$$\forall q_1'', q_2'' \in B'', \forall q' \in Q', \forall \alpha \in \Sigma : [\delta_d(q', q_1'')(\alpha)]_{\mathcal{P}_d} = [\delta_d(q', q_2'')(\alpha)]_{\mathcal{P}_d}$$

Thus B' δ_d-respects $\mathcal{P}_d$ if δ_d cannot distinguish between the elements in B' relative to $\mathcal{P}_d$. A partition $\mathcal{Q}'$ δ_d-respects $\mathcal{P}_d$ if every block of $\mathcal{Q}'$ δ_d-respects $\mathcal{P}_d$, and a family of partitions $\{\mathcal{Q}\}_D$ δ_d-respects $\mathcal{P}_d$ if $\mathcal{Q}'$ and $\mathcal{Q}''$ δ_d-respects $\mathcal{P}_d$. A family of partitions $\{\mathcal{Q}\}_D$ respects the family of partitions $\{\mathcal{P}\}_D$ if $\{\mathcal{Q}\}_D$ δ_d-respects $\mathcal{P}_d$ for all transition functions $\delta_d \in \{\delta\}_D$, where δ_d is of type $d' \times d'' \to d$. A family of partitions is *stable* if it respects itself. The *coarsest, stable family of partitions $\mathcal{Q}_D$ respecting $\mathcal{P}_D$* is a unique family of partitions such that any other stable family of partitions respecting $\mathcal{P}_D$ is a refinement of $\mathcal{Q}_D$.

The minimization algorithm works by gradually refining a current family of partitions so that each step of the algorithm ensures that the refinement δ_d-respects the current family of partitions for some transition function δ_d. We first show how to split a current family of partitions with respect to a single transition function, and later how this is used to minimize a guided tree automata. We assume for the rest of this section that our representation is symmetric, that this, the sets $\mathit{explicit}_d$ and the default behaviors $\mathit{default}_d$ are also present for right states. It is straightforward to precompute these values from the $\mathit{explicit}_d$ and $\mathit{default}_d$ information of the left states (this is the information calculated by the product and project algorithms).

Splitting with respect to δ_d of type $d' \times d'' \to d$ Let $\mathcal{Q}$, $\mathcal{Q}'$ and $\mathcal{Q}''$ denote the current partition of Q, Q' and Q'' respectively and assume that the current family of partitions does not δ_d-respect $\mathcal{Q}$. We now show how to compute the coarsest partition which δ_d-respects the current partition.

1. Replace the leaf-values in the Σ-BDD by canonical representatives according to $\mathcal{Q}$ and reduce it. This induces a partition of the nodes in the Σ-BDD denoted $\mathcal{S}$.
2. Refine $\mathcal{Q}'$ to $\mathcal{P}'$ such that $q_1' \equiv_{\mathcal{P}'} q_2'$ iff $q_1' \equiv_{\mathcal{Q}'} q_2'$ and $\forall q'' \in Q''$, $\delta_d(q_1', q'') \equiv_{\mathcal{S}} \delta_d(q_2', q'')$
3. Refine $\mathcal{Q}''$ to $\mathcal{P}''$ such that $q_1'' \equiv_{\mathcal{P}''} q_2''$ iff $q_1'' \equiv_{\mathcal{Q}''} q_2''$ and $\forall q' \in Q'$, $\delta_d(q', q_1'') \equiv_{\mathcal{S}} \delta_d(q' q_2'')$

Step 1 ensures $\omega \equiv_{\mathcal{S}} \omega'$ iff $\forall \alpha \in \Sigma\ \omega(\alpha) \equiv_{\mathcal{Q}} \omega'(\alpha)$. For the partition calculated in step 2 we have $q_1' \equiv_{\mathcal{P}'} q_2'$ iff $\delta_d(q_1', q'') \equiv_{\mathcal{S}} \delta_d(q_2', q'')$ for all $q'' \in Q''$, i.e. $\forall q'' \in Q''$, $\forall \alpha \in \Sigma$, $\delta_d(q_1', q'')(\alpha) \equiv_{\mathcal{Q}} \delta_d(q_2', q'')(\alpha)$. Thus all blocks of $\mathcal{P}'$ δ_d-respect $\mathcal{Q}$. Similarly, step 3 ensures that all blocks of $\mathcal{P}''$ δ_d-respect $\mathcal{Q}$.

The refinement operation in step 2 is performed by assigning to each element $q' \in Q'$ a canonical representative for its block in the new partition $\mathcal{P}'$ respecting $\mathcal{Q}$. Similar representatives are calculated for $q'' \in Q''$ in step 3. For a Σ-BDD node ω, we denote its canonical representative with respect to $\mathcal{S}$ by $\hat{\omega}$.

We now address the problem of calculating the canonical representatives in step 2. (Step 3 is symmetric.) Consider a state $q' \in Q'$. The problem is to calculate in linear time a unique characterization of the function $q'' \mapsto [\delta_d(q', q'')]$. We must deal with the default representation while making sure that the characterization remains unique. The following techniques allow a default based representation, where the default case is used only when its uniqueness can be assured.

Let $\nu_{q'} = q'.\mathit{default}_d$ and let $\hat{\nu}_{q'}$ be its representative according to $\mathcal{S}$. By traversing the states in $q'.\mathit{explicit}_d$, we calculate the set:

$$M_{q'} = \{(q'', \hat{\omega}) \mid (q'', \omega) \in q'.\mathit{explicit}_d \text{ and } \hat{\omega} \neq \hat{\nu}_{q'}\}.$$

If $|M_{q'}| < \frac{1}{2}|Q''|$, then $\hat{\nu}_{q'}$ is a default behavior that applies to more than half the states in Q''. Otherwise, $|M_{q'}| \geq \frac{1}{2}|Q''|$ and we find a $\hat{\nu}_{q'}'$ minimizing the size of

$$\{(q'', \hat{\omega}) \mid (q'', \omega) \in Q'' \times \Omega,\ \omega = \delta_d(q', q'') \text{ and } \hat{\omega} \neq \hat{\nu}_{q'}'\},$$

by another linear traversal. Redefine $M_{q'}$ to be this set, and $\hat{\nu}_{q'}$ to be $\hat{\nu}_{q'}'$. If the size of $|M_{q'}|$ still is larger than $\frac{1}{2}|Q''|$, then there is no way of characterizing the default behavior uniquely by means of the class that contains more than half the states. Thus in this case, we redefine $M_{q'}$ once more to be:

$$\{(q'', \hat{\omega}) \mid (q'', \omega) \in Q'' \times \Omega,\ \omega = \delta_d(q', q'')\},$$

by utilizing all states in Q'' and redefine $\hat{\nu}_{q'}$ to be a fixed value – different from any representative $\hat{\omega}$. We remember the old values of $M_{q'}$ and $\hat{\nu}_{q'}$ and denote these as $M^{-}_{q'}$ and $\hat{\nu}^{-}_{q'}$ respectively. It is now not difficult to show that the tuple $(M_{q'}, \hat{\nu}_{q'})$ is a canonical representative for the block in $\mathcal{P}'$ containing q', i.e. $q'_1 \equiv_{\mathcal{P}'} q'_2$ iff $(M_{q'_1}, \hat{\nu}_{q_1}) = (M_{q'_2}, \hat{\nu}_{q_2})$. Also, it can be seen that the calculation of the representative is linear in the size of $q'.explicit_d$. We note that in practice we would additionally need to calculate a canonical index (an integer) from the canonical representative using some hashing approach.

In total, calculating the canonical representative for a state q' in step 2 takes time $O(|q'.explicit_d|)$ given that the representatives with respect to $\mathcal{S}$ have been calculated. Hence in total step 2 and step 3 take time proportional to the representation of the transition relation δ_d.

Minimizing a guided tree automaton With the splitting operation of the previous section, minimization of a guided tree automaton is now an easy task. Consider a GTA $M_G = (\{Q\}_D, \Sigma, \Delta, \{\bar{q}\}_D, F)$ guided by $G = (D, \mu, d_0)$ and let $N_G = (\{P\}_D, \Sigma, \{\phi\}_D, \{\bar{r}\}_D, H)$ denote the resulting automaton. Now let d be a state space ID with $\mu(d) = (d', d'')$ and assume we have just done a δ_d-split. If the left partition became strictly finer, then we say that a *left-split* occurred. In that case, we must also carry out a $\delta_{d'}$-split operation. Similar considerations apply for a *right-split* and a subsequent $\delta_{d''}$-split operation. This process is repeated until no more split operations need to be done, i.e. until a fixed point has been found.

The algorithm is specified in some more detail in Figure 8.

In the first phase, it performs the split operations according to a set called *candidates*, where ID $d \in$ *candidates* if a δ_d-split must be carried out. The set *candidates* is updated with respect to the left-splits and right-splits that occur. The first phase of the algorithm terminates when *candidates* $= \emptyset$. The function *split* called with parameter δ performs a split operation with respect to δ as described in the previous section. It returns a pair of Booleans (*lsplit,rsplit*) that indicates whether a left-split (right-split respectively) occurred. The resulting family of partitions, $\mathcal{Q}_D$, is the coarsest, stable family of partitions respecting the initial family of partitions.

In the final phase, the algorithm builds the minimized guided tree automaton from the family of partitions $\mathcal{Q}_D$.

Analysis Since each split operation is linear in the total size of the GTA representation and since each operation (except for the last) results in a finer partition, the total running time is $O(n \cdot m)$, where n is the total number of states and m is the total representation size.

Note that the selection of the next transition function to use for a split operation is arbitrary. It would be interesting to study whether more judicious choices could entail asymptotic gains.

It is possible to minimize BDD-represented automata on finite strings in time $O(m \cdot \log m)$ [5], but it is an open question whether this result can be extended to tree automata.

```
Initial family of partitions :$\mathcal{Q}_d = \begin{cases} \{F, Q_{d_0} \setminus F\} & \text{if } d = d_0 \\ \{Q_d\} & \text{otherwise} \end{cases}$
$candidates = \{d_0\}$
while $candidates \neq \emptyset$ do
    remove a state space ID $d$ from $candidates$
    $(d', d'') = \mu(d)$
    $(lsplit, rsplit) = split(\delta_d)$
    if $lsplit$ then
        add $d'$ to $candidates$
    fi
    if $rsplit$ then
        add $d''$ to $candidates$
    fi
od

Replace the values in the leaves of the $\Sigma$-BDD by canonical representatives
according to $\mathcal{Q}_D$ and reduce it. The induced partition of
$\Sigma$-BDD nodes is denoted $\mathcal{S}$.

for each $\delta_d$ of type $d' \times d'' \to d$ do
    for each $[q'] \in \mathcal{Q}'$ with canonical representative $(M_{q'}, \hat{\nu}_{q'})$ do
        add $[q']$ as a state to $P'$
        if $\hat{\nu}_{q'} = \perp$ then
            $[q'].default_d = \hat{\nu}_{q'}^{-}$
            $[q'].explicit_d = M_{q'}^{-}$
        else
            $[q'].default_d = \hat{\nu}_{q'}$
            $[q'].explicit_d = M_{q'}$
        fi
    od
    for each $[q''] \in \mathcal{Q}''$ with canonical representative $(M_{q''}, \hat{\nu}_{q''})$ do
        add $[q'']$ as a state to $P''$
        if $\hat{\nu}_{q''} = \perp$ then
            $[q''].default_d = \hat{\nu}_{q''}^{-}$
            $[q''].explicit_d = M_{q''}^{-}$
        else
            $[q''].default_d = \hat{\nu}_{q''}$
            $[q''].explicit_d = M_{q''}$
        fi
    od
od
```

Figure 8. Algorithm for minimizing GTAs.

7 Experimental results

The current Mona tool supports *guided tree automata*, but does not yet use the representation of the transitions functions presented in this paper. Instead the implementation uses the BDD encoding of the state spaces mentioned in Section 3. Nevertheless, we have had some successful experimental results with this implementation.

A major goal of the implementation was to provide the means for making FIDO [7] a tractable programming language for expressing regular constraints on parse trees. From a FIDO program, a M2L formula is generated. By processing this formula, MONA calculates an automaton, which can be viewed as an attribute grammar for the specified grammar satisfying the syntactic side constraints. The grammar example from [7] and an additional HTML grammar example are processed by FIDO and MONA in approximately half a minute on a Sparc Station 1000. In both examples, the M2l formulas generated by FIDO are several (dense) pages long. Our current tool was also used to compute the architectural software constraints in [6].

Our experience is that for most of these examples, the intermediate and final automatons exhibit the property of sparse transition functions. Thus, we expect that the proposed algorithms together with successful attempts of speeding up the current BDD-package will give rise to a significant speed-up in future implementations of the GTA operations.

We have experimented with the guide to determine its practical importance. For the HTML example, we experienced that with a guide with three state spaces, MONA could process the example in 40 seconds, with intermediate automata reaching at most 70 states. With a one-state guide (i.e. with an ordinary DFTA), MONA generates an intermediate automaton with a state space of more than 7000 states—which the subsequent project operation is unable to handle.

References

1. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.
2. A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *TCS*, 19:85–98, 1982.
3. Rowan Davis. Personal communication. 1995.
4. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996. Also available through http://www.brics.dk/~klarlund/MonaFido/papers.html.
5. N. Klarlund. An $n \log n$ algorithm for online BDD refinement. Technical report, BRICS, Aarhus, Denmark. http://www.brics.dk/~klarlund/MonaFido/papers, 1996.
6. N. Klarlund, J. Koistinen, and M. Schwartzbach. Formal design constraints. In *Proc. OOPSLA '96*, 1996. to appear.
7. N. Klarlund and M. Schwartzbach. Regularity = logic + recursive data types. Technical report, BRICS, 1997. To appear.
8. D. Kozen. On the Myhill-Nerode theorem for trees. *EATCS Bulletin*, 47, 1992.
9. R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal of Computing*, 16(6), 1987.

10. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2:57–82, 1968.
11. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.