VATA: A Library for Efficient Manipulation of Non-Deterministic Tree Automata*

Ondřej Lengál[1], Jiří Šimáček[1,2], and Tomáš Vojnar[1]

[1] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
[2] VERIMAG, UJF/CNRS/INPG, Gières, France

Abstract. In this paper, we present VATA, a versatile and efficient open-source tree automata library applicable, e.g., in formal verification. The library supports both explicit and semi-symbolic encoding of non-deterministic finite tree automata and provides efficient implementation of standard operations on both. The semi-symbolic encoding is intended for tree automata with large alphabets. For storing their transition functions, a newly implemented MTBDD library is used. In order to enable the widest possible range of applications of the library even for the semi-symbolic encoding, we provide both bottom-up and top-down semi-symbolic representations. The library implements several highly optimised reduction algorithms based on downward and upward simulations as well as algorithms for testing automata inclusion based on upward and downward antichains and simulations. We compare the performance of the algorithms on a set of test cases and we also compare the performance of VATA with our previous implementations of tree automata.

1 Introduction

Several current formal verification techniques are based on *finite tree automata* (TA). Some of these techniques are: (abstract) regular tree model checking [3,5] applied, e.g., for verification of programs with complex dynamic data structures [6,11], implementation of decision procedures of several logics, such as MSO or WSkS [17], or verification of programs manipulating heap structures with data [18]. The success of these techniques often depends on the performance of the underlying implementation of TA.

Currently, there exist several available tree automata libraries, they are, however, mostly written in OCaml (e.g., Timbuk/Taml [10]) or Java (e.g., LETHAL [9]) and they do not always use the most advanced algorithms known to date. Therefore, they are not suitable for tasks which require the available processing power be utilised as efficiently as possible. An exception from these libraries is MONA [17] implementing decision procedures over WS1S/WS2S, which contains a highly optimised TA package written in C, but, alas, it supports only binary deterministic tree automata. At the same time, it turns out that determinisation is often a very significant bottleneck of using TA, and a lot

* This work was supported by the Czech Science Foundation within projects No. P103/10/0306 and 102/09/H042, the Czech Ministry of Education within projects COST OC10009 and MSM 0021630528, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

of effort has therefore been invested into developing efficient algorithms for handling non-deterministic tree automata without a need to ever determinise them.

In order to allow researchers focus on developing verification techniques rather than reimplementing and optimising a TA package, we provide VATA[3], an easy-to-use open-source library for efficient manipulation of non-deterministic TA. VATA supports many of the operations commonly used in automata-based formal verification techniques over two complementary encodings: explicit and semi-symbolic. The *explicit* encoding is suitable for most applications that do not need to use alphabets with a large number of symbols. However, some formal verification approaches make use of such alphabets, e.g., the approach for verification of programs with complex dynamic data structures [5] or decision procedures of the MSO or WSkS logics [17]. Therefore, in order to address this issue, we also provide the *semi-symbolic* encoding of TA, which uses *multi-terminal binary decision diagrams* [8] (MTBDDs), an extension of reduced ordered binary decision diagrams [7] (BDDs), to store the transition function of a TA. In order to enable the widest possible range of applications of the library even for the semi-symbolic encoding, we provide both bottom-up and top-down semi-symbolic representations.

At the present time, the main application of the structures and algorithms implemented in VATA for handling explicitly encoded TA is the Forester tool for verification of programs with complex dynamic data structures [11]. The semi-symbolic encoding of TA has so far been used mainly for experiments with various newly proposed algorithms for handling TA.

In this paper, we do not present all exact details of the algorithms implemented in the library as they can be found in the referenced literature. Rather, we give an overview of the algorithms available, while mentioning various interesting optimisations that we used when implementing them. Based on experimental evidence, we argue that these optimisations are crucial for the performance of the library.

2 Preliminaries

A *ranked alphabet* Σ is a finite set of symbols together with a ranking function $\# : \Sigma \to \mathbb{N}$. For $a \in \Sigma$, the value $\#a$ is called the *rank* of a. For any $n \geq 0$, we denote by Σ_n the set of all symbols of rank n from Σ. Let ε denote the empty sequence. A *tree* t over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ that satisfies the following conditions: (1) the domain of t, $dom(t)$, is a finite prefix-closed subset of $\mathbb{N}^*$ and (2) for each $v \in dom(t)$, if $\#t(v) = n \geq 0$, then $\{i \mid vi \in dom(t)\} = \{1, \ldots, n\}$. Each sequence $v \in dom(t)$ is called a *node* of t. For a node v, we define the i^{th} *child* of v to be the node vi, and the i^{th} *subtree* of v to be the tree t' such that $t'(v') = t(viv')$ for all $v' \in \mathbb{N}^*$. A *leaf* of t is a node v which does not have any children, i.e., there is no $i \in \mathbb{N}$ with $vi \in dom(t)$. We denote by T_Σ the set of all trees over the alphabet Σ.

A (finite, non-deterministic) *tree automaton* (abbreviated sometimes as TA in the following) is a quadruple $\mathcal{A} = (Q, \Sigma, \Delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is a ranked alphabet, and Δ is a set of transition rules. Each transition rule is a triple of the form $((q_1, \ldots, q_n), a, q)$ where $q_1, \ldots, q_n, q \in Q, a \in \Sigma$,

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

and $\#a = n$. We use equivalently $(q_1,\ldots,q_n) \xrightarrow{a} q$ and $q \xrightarrow{a} (q_1,\ldots,q_n)$ to denote that $((q_1,\ldots,q_n),a,q) \in \Delta$. The two notations correspond to the *bottom-up* and *top-down* representation of tree automata, respectively. Note that we can afford to work interchangeably with both of them since we work with non-deterministic tree automata, which are known to have an equal expressive power in their bottom-up and top-down representations. In the special case when $n = 0$, we speak about the so-called *leaf rules*, which we sometimes abbreviate as $\xrightarrow{a} q$ or $q \xrightarrow{a}$.

Let $\mathcal{A} = (Q,\Sigma,\Delta,F)$ be a TA. A *run* of $\mathcal{A}$ over a tree $t \in T_\Sigma$ is a mapping $\pi : dom(t) \to Q$ such that, for each node $v \in dom(t)$ of rank $\#t(v) = n$ where $q = \pi(v)$, if $q_i = \pi(vi)$ for $1 \leq i \leq n$, then Δ has a rule $(q_1,\ldots,q_n) \xrightarrow{t(v)} q$. We write $t \xRightarrow{\pi} q$ to denote that π is a run of $\mathcal{A}$ over t such that $\pi(\varepsilon) = q$. We use $t \Longrightarrow q$ to denote that $t \xRightarrow{\pi} q$ for some run π. The *language* accepted by a state q is defined by $\mathcal{L}_\mathcal{A}(q) = \{t \mid t \Longrightarrow q\}$, while the language of a set of states $S \subseteq Q$ is defined as $\mathcal{L}_\mathcal{A}(S) = \bigcup_{q\in S} \mathcal{L}_\mathcal{A}(q)$. When it is clear which TA $\mathcal{A}$ we refer to, we only write $\mathcal{L}(q)$ or $\mathcal{L}(S)$. The language of $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_\mathcal{A}(F)$.

A *downward simulation* on TA $\mathcal{A} = (Q,\Sigma,\Delta,F)$ is a preorder relation $\preceq_D \subseteq Q \times Q$ such that if $q \preceq_D p$ and $(q_1,\ldots,q_n) \xrightarrow{a} q$, then there are states $p_1,\ldots,p_n$ such that $(p_1,\ldots,p_n) \xrightarrow{a} p$ and $q_i \preceq_D p_i$ for each $1 \leq i \leq n$. Given a TA $\mathcal{A} = (Q,\Sigma,\Delta,F)$ and a downward simulation $\preceq_D$, an *upward simulation* $\preceq_U \subseteq Q \times Q$ induced by $\preceq_D$ is a relation such that if $q \preceq_U p$ and $(q_1,\ldots,q_n) \xrightarrow{a} q'$ with $q_i = q$ for some $1 \leq i \leq n$, then there are states $p_1,\ldots,p_n,p'$ such that $(p_1,\ldots,p_n) \xrightarrow{a} p'$ where $p_i = p$, $q' \preceq_U p'$, and $q_j \preceq_D p_j$ for each j such that $1 \leq j \neq i \leq n$.

3 Design of the Library

The library is designed in a modular way (see Fig. 1). The user can choose a module encapsulating her preferred automata encoding and its corresponding operations. Various encodings share the same general interface so it is easy to swap one encoding for another, unless encoding-specific functions or operations are taken advantage of.

Thanks to the modular design of the library, it is easy to provide an own encoding of tree (or word) automata and effectively exploit the remaining parts of the infrastructure, such as parsers and serializers from/to different formats, the unit testing framework, performance tests, etc.

The VATA library is implemented in C++ using the Boost C++ libraries. In order to avoid expensive look-ups of entry points of virtual methods in the *virtual-method table* of an object and to fully exploit compiler's capabilities of code inlining and optimisation of code according to static analysis, the library heavily exploits polymorphism using C++ function templates instead of using virtual methods for core functions. We are convinced that this is the main reason why the performance of the optimised code (the `-O3` flag of `gcc`) is up to 10 times better than the performance of the non-optimised code (the `-O0` flag of `gcc`).

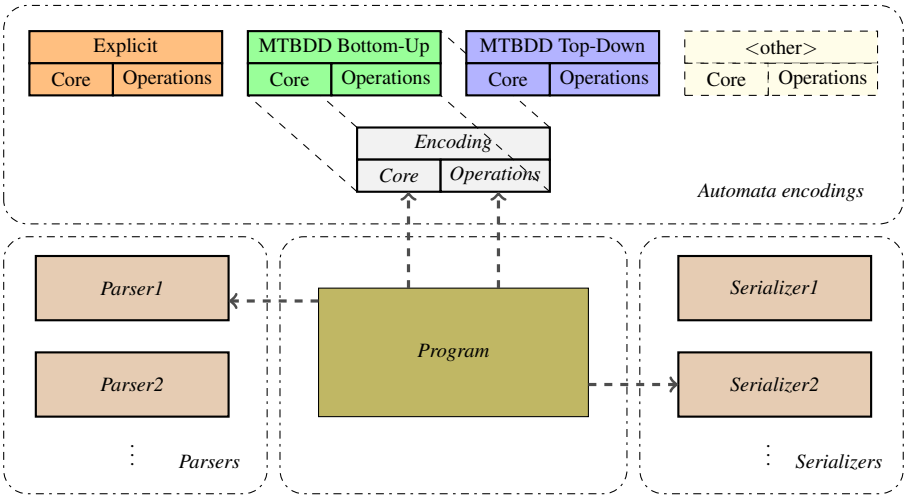

Fig. 1. The architecture of the VATA library

3.1 Explicit Encoding

In the explicit representation of TA used in VATA, top-down transitions having the form $q \xrightarrow{a} (q_1, \ldots, q_n)$ are stored in a *hierarchical data structure similar to a hash table*. More precisely, the top-level lookup table maps states to *transition clusters*. Each such cluster is itself a lookup table that maps alphabet symbols to a set of pointers to tuples of states. The set of pointers to tuples of states is represented using a red-black tree. The tuples of states are stored in a designated hash table to further reduce the required amount of space (by not storing the same tuples of states multiple times). An example of the encoding is depicted in Fig. 2.

Hence, in order to insert the transition $q \xrightarrow{a} (q_1, \ldots, q_n)$ into the transition table, one proceeds using the following algorithm:

1. Find a transition cluster which corresponds to the state q in the top-level lookup table. If such a cluster does not exist, create one.
2. In the given cluster, find a set of pointers to tuples of states reachable from q over a. If the set does not exist, create one.
3. Obtain the pointer to the tuple $(q_1, \ldots, q_n)$ from the tuple lookup table and insert it into the set of pointers.

If one ignores the worst-case time complexity of the underlying data structures (which, according to our experience, has usually a negligible real impact only), then inserting a single transition into the transition table requires a constant number of steps only. Yet the representation provides a more efficient encoding than a plain list of transitions because some transitions share the space required to store the parent states (e.g., state q in the transition $q \xrightarrow{a} (q_1, \ldots, q_n)$). Moreover, some transitions also share the alphabet symbol and each tuple of states appearing in the set of transitions is stored only

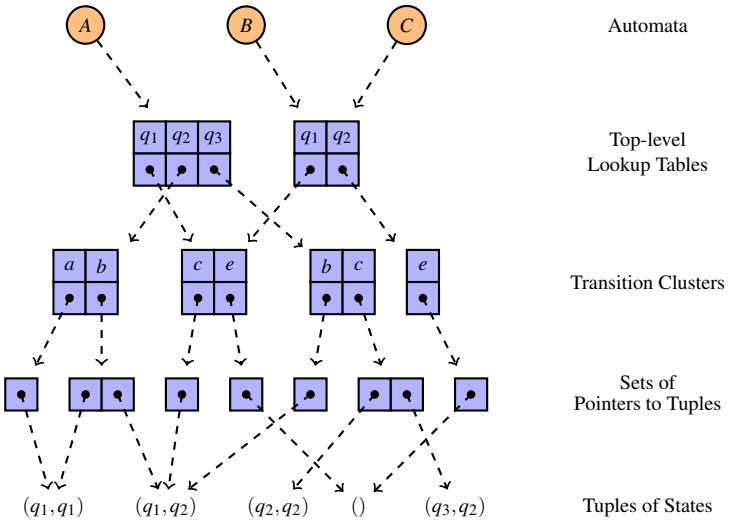

Fig. 2. An example of the VATA's explicit encoding of transition functions of three automata A, B, C. In particular, one can see that A contains a transition $q_1 \xrightarrow{c} (q_1, q_2)$: it suffices to follow the corresponding arrows. Moreover, B also contains the same transition (and the corresponding part of the transition table is shared with A). Finally, C has the same transitions as B.

once. Additionally, the encoding allows us to easily perform certain critical operations, such as finding a set of transitions $q \xrightarrow{a} (q_1, \ldots, q_n)$ for a given state q. This is useful, e.g., during the elimination of (top-down) unreachable states or during the top-down inclusion checking.

In some situations, one needs to manipulate many tree automata at the same time. As an example, we can mention the method for verifying programs with dynamic linked data structures introduced in [11] where (in theory) one needs to store one automaton representing a content of the heap for each reachable state of the program. To improve the performance of our library in such scenarios, we adapt the *copy-on-write* principle. Every time one needs to create a copy of an automaton A to be subsequently modified, it is enough to create a new automaton A' which obtains a pointer to the transition table of A (which requires constant time). Subsequently, as more transitions are inserted into A' (or A), only the part of the shared transition table which gets modified is copied (Fig. 2 provides an illustration of this feature).

3.2 Semi-Symbolic Encoding

The semi-symbolic encoding uses *multi-terminal binary decision diagrams* (MTBDDs) to encode transition functions of tree automata. MTBDDs are an extension of *binary decision diagrams* (BDDs), a popular data structure for compact encoding and manipulation with Boolean formulae. In contrast to BDDs that are used to represent a function

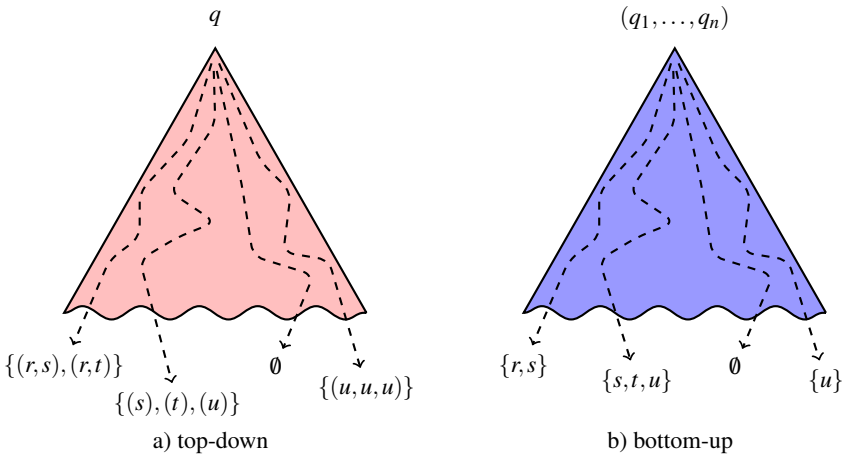

Fig. 3. The (a) top-down and (b) bottom-up semi-symbolic encodings of transition functions. Paths in the MTBDD correspond to symbols.

$b : \mathbb{B}^n \to \mathbb{B}$ for some $n \in \mathbb{N}$ and $\mathbb{B} = \{0,1\}$, MTBDDs extend the co-domain to an arbitrary set S, i.e., they represent a function $m : \mathbb{B}^n \to S$.

We support two representations of semi-symbolic automata: top-down and bottom-up. The *top-down* representation (see Fig. 3a) maintains for each state q of a tree automaton an MTBDD that maps the binary representation of each symbol f concatenated with the binary representation of its arity n onto a set of tuples of states $T = \{(q_1,\ldots,q_n),\ldots\}$ such that for all $(q_1,\ldots,q_n) \in T$ there exist the transition $q \xrightarrow{f} (q_1,\ldots,q_n)$ in the automaton. The arity is encoded in the MTBDD as a part of the symbol in order to be able to distinguish between several instances of the same symbol with different arity. The library thus supports a slight extension of tree automata in which a symbol does not have a fixed arity.

The *bottom-up* representation (see Fig. 3b), on the other hand, maintains for each tuple $(q_1,\ldots,q_n) \in Q^*$ an MTBDD that maps the binary representation of each symbol f onto a set of states $S = \{q,\ldots\}$ such that, for all $q \in S$, it holds that the transition $(q_1,\ldots,q_n) \xrightarrow{f} q$ is in the automaton. Note that the bottom-up representation does not need to encode the arity of the symbol f into the MTBDD as it is given by the arity of the tuple for which the MTBDD is maintained. It is easy to see that the two presented encodings are mutually convertible (see [13] for the algorithm).

MTBDD Package. Our previous implementation of semi-symbolically represented tree automata used a customisation of the CUDD [20] library for manipulating MTBDDs. The experiments in [12] and profiling of the code showed that the overhead of the customised library is too large. Moreover, the customisation of CUDD did not provide an easy and transparent way of manipulating MTBDDs. These two facts showed that VATA would greatly benefit from a major redesign of the MTBDD back-

end. Therefore, we created our own generic implementation of MTBDDs with a clean and simple-to-use interface.

The new MTBDD package uses *shared* MTBDDs for each domain, which means that all MTBDDs for the given domain are connected in a single *directed acyclic graph* (DAG), and an MTBDD corresponds to a pointer to a node in the DAG. In order to prevent memory leaks, each node of the MTBDD contains a reference counter of other nodes or variables pointing to it. In case the counter reaches zero, the node is deleted from the memory. Because of these implementation choices, copying an MTBDD can be easily done by simply copying the pointer to the root node of the copied MTBDD and incrementing its reference counter.

There are two types of nodes of the MTBDD: *internal nodes* and *leaf nodes*. A leaf node contains a value from the domain of the MTBDD, while an internal node contains a variable name and pointers to the *low* and *high* children of the node. In addition, nodes of both types also contain the aforementioned reference counter. The nodes are manipulated using pointers to them only, and the distinction between a leaf node and an internal node is done according to the least significant bit of the pointer (the compiler aligns these data structures to addresses which are multiples of 4, this bit can therefore be neglected and when accessing the value of a node pointer simply masked out).

For our use, we implemented unary, binary, and ternary *Apply* operations, which are operations that, given a unary, binary, or ternary function and one, two, or three MTBDDs, respectively, generate a new MTBDD the leaves of which correspond to the values of the given function applied to the provided MTBDDs. Note that the provided function does not need to be a pure function but my also have a side-effect. Further, we also provide *VoidApply* operations which are Apply operations that do not build a new MTBDD but that have a side-effect only. For operations that do not need to build new MTBDDs but rather, e.g., only collect data from the leaf nodes, using *VoidApply* saves a considerable and unnecessary overhead. During the execution of an *Apply* operation, both internal and leaf nodes are cached using hash tables.

The newly implemented MTBDD package does not support MTBDD reordering so far, yet the library performs better when compared to our original implementation of a semi-symbolic encoding that used customised CUDD.

4 Supported Operations

As we described in the previous section, the VATA library allows a user to choose one of three available encodings: the explicit top-down, the semi-symbolic top-down, and the semi-symbolic bottom-up. Depending on the choice, certain TA operations may or may not be available. The following operations are supported by at least one of the representations: union, intersection, elimination of (bottom-up, top-down) unreachable states, inclusion checking (bottom-up, top-down), computation of (maximum) simulation relations (downward, upward), and language preserving size reduction based on simulation equivalence. In some cases, multiple implementations of an operation are available, which is especially the case for language inclusion. This is because the different implementations are based on different heuristics that may work better for different applications as witnessed also by our experiments described in Section 5.

Below, we do not discuss the relatively straightforward implementation of the most basic operations on TA and we comment on the more advanced operations only.

4.1 Removing Unreachable States

As the performance of many operations on automata depends on the size of the automaton (in the sense of the size of the state set and the size of the transition table), it is often desirable to remove both bottom-up and top-down unreachable states. Indeed, such states are useless: bottom-up unreachable states cannot be used to generate a finite tree and although top-down unreachable states can generate a finite tree, this tree cannot be a subtree of any tree accepted by the automaton.

Removing both bottom-up *unreachable states* for the bottom-up representation and top-down unreachable states for the top-down representation can be easily done by a single traversal through the automaton. Nevertheless, sometimes, e.g., when checking language inclusion of the automata, it is useful to also remove states unreachable in the opposite direction.

The procedure for removing top-down unreachable states from a tree automaton represented in a bottom-up semi-symbolic way generates a directed graph (Q,E) where Q is the state set of the input automaton and $(q,r) \in E$ if $\exists a \in \Sigma : q \xrightarrow{a} (q_1,\ldots,q_n), \exists 1 \leq i \leq n : r = q_i$. When the graph is created, the states that are backward unreachable from the final states are removed from the automaton in a simple traversal.

Removing bottom-up unreachable states for the top-down semi-symbolic representation is more complex. First, the automaton is traversed in the top-down manner while creating an *And-Or* graph $(N_\forall, N_\exists, E)$ where $N_\forall = Q$, Q is the state set of the input automaton and represents the *And* nodes of the graph, and $N_\exists \subseteq Q^*$ represents the *Or* nodes. The set of edges E contains the edge $(q,(q_1,\ldots,q_n))$ if there exists the transition $q \xrightarrow{a} (q_1,\ldots,q_n)$ for some $a \in \Sigma$ in the automaton, and the edge $((q_1,\ldots,q_n),q)$ if $\exists 1 \leq i \leq n : q_i = q$. The algorithm starts by marking the node labelled by $()$ (which is an Or node) and proceeds by marking the nodes of the graph using the following rules: an Or node o is marked if there exists a marked node a such that $(o,a) \in E$, and an And node a is marked if all nodes o such that $(a,o) \in E$ are marked. When no new nodes can be marked, the states of the automaton are reduced to only those that correspond to marked And nodes in the graph.

4.2 Downward and Upward Simulation

Downward simulation relations can be computed over two tree automata representations in VATA: the explicit top-down and the semi-symbolic top-down encoding. The explicit variant first translates a tree automaton into a labelled transition system (LTS) as described in [1]. Then the simulation relation for this system is computed using an implementation of the state-of-the-art algorithms for computing simulations on LTSs [19,14] with some further optimisations mentioned in Section 4.6. Finally, the result is projected back to the set of states of the original automaton.

The semi-symbolic variant uses a simpler simulation algorithm based on a generalisation of [16] to trees.

Upward simulation can currently be computed over the explicit representation only. The computation is again performed via a translation to an LTS (the details are in [1]), and the relation is computed using the engine for computing simulation relations on LTSs as above.

4.3 Simulation-based Size Reduction

In a typical setting, one often wants to use a representation of tree automata that is as small as possible in order to reduce the memory consumption and/or speed up operations on the automata (especially the potentially costly ones, such as inclusion testing). To achieve that, the classical approach is to use determinisation and minimisation. However, the minimal deterministic tree automata can still be much bigger than the original non-deterministic ones. Therefore, VATA offers a possibility to reduce the size of tree automata without determinisation by their quotienting w.r.t. an equivalence relation—currently, only the downward simulation equivalence is supported.

The procedure works as follows: first, the downward simulation relation $\preceq_D$ is computed for the automaton. Then, the symmetric fragment of $\preceq_D$ (which is an equivalence) is extracted, and each state appearing within the transition function is replaced by a representative of the corresponding equivalence class. A further reduction is then based on the following observation: if an automaton contains a transition $q \xrightarrow{a} (q_1, \ldots, q_n)$, any additional transition $q \xrightarrow{a} (r_1, \ldots, r_n)$ where $r_i \preceq_D q_i$ can be omitted since it does not contribute to the language of the result (recall that, for the downward simulation preorder $\preceq_D$, it holds that $q \preceq_D r \Longrightarrow \mathcal{L}(q) \subseteq \mathcal{L}(r)$).

4.4 Bottom-up Inclusion

Bottom-up inclusion testing is implemented for the explicit top-down and the semi-symbolic bottom-up representation in VATA. As its name suggests, the algorithm naturally proceeds in the bottom-up way, therefore the top-down encoding is not very suitable here. In the case of the explicit representation, however, one can afford to build a temporary bottom-up encoding since the overhead of such a translation is negligible compared to the complexity of following operations.

Both the explicit and semi-symbolic version of the bottom-up inclusion algorithm are based on the approach introduced in [4]. Here, the main principle used for checking whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ is to search for a tree which is accepted by $\mathcal{A}$ and not by $\mathcal{B}$ (thus being a witness for $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$). This is done by simultaneously traversing both $\mathcal{A}$ and $\mathcal{B}$ from their leaf rules while generating pairs $(p_\mathcal{A}, P_\mathcal{B}) \in Q_\mathcal{A} \times 2^{Q_\mathcal{B}}$ where $p_\mathcal{A}$ represents a state into which $\mathcal{A}$ can get on some input tree and $P_\mathcal{B}$ is the set of *all* states into which $\mathcal{B}$ can get over the same tree. The inclusion then does clearly not hold iff it is possible to generate a pair consisting of an accepting state of $\mathcal{A}$ and of exclusively non-accepting states of $\mathcal{B}$.

The algorithm collects the so far generated pairs $(p_\mathcal{A}, P_\mathcal{B})$ in a set called *Visited*. Another set called *Next* is used to store the generated pairs whose successors are still to be explored. One can then observe that whenever one can reach a counterexample to inclusion from $(p_\mathcal{A}, P_\mathcal{B})$, one can also reach a counterexample from any $(p_\mathcal{A}, P'_\mathcal{B} \subseteq P_\mathcal{B})$

as $P'_{\mathcal{B}}$ allows less runs than $P_{\mathcal{B}}$. Using this observation, both mentioned sets can be represented using antichains. In particular, one does not need to store and further explore any two elements comparable w.r.t. $(=, \subseteq)$, i.e., by equality on the first component and inclusion on the other component.

Clearly, the running time of the above algorithm strongly depends on the total number of pairs $(p_{\mathcal{A}}, P_{\mathcal{B}})$ taken from *Next* for further processing. Indeed, this is one of the reasons why the antichain-based optimisations helps. According to our experience, the number of pairs which needs to be processed can further be reduced when processing the pairs stored in *Next* in a suitable order. Our experimental results have shown that we can achieve a very good improvement by preferring those pairs $(p_{\mathcal{A}}, P_{\mathcal{B}})$ which have smaller (w.r.t. the size of the set) second component.

Yet another way that we found useful when improving the above algorithm is to optimise the way the algorithm computes the successors of a pair from *Next*. The original algorithm picks a pair $(p_{\mathcal{A}}, P_{\mathcal{B}})$ from *Next* and puts it into *Visited*. Then, it finds all transitions of the form $(p_{\mathcal{A},1}, \ldots, p_{\mathcal{A},n}) \xrightarrow{a} p$ in $\mathcal{A}$ such that $(p_{\mathcal{A},i}, P_{\mathcal{B},i}) \in \mathit{Visited}$ for all $1 \leq i \leq n$ and $(p_{\mathcal{A},j}, P_{\mathcal{B},j}) = (p_{\mathcal{A}}, P_{\mathcal{B}})$ for some $1 \leq j \leq n$. For each such transition, it finds all transitions of the form $(q_1, \ldots, q_n) \xrightarrow{a} q$ in $\mathcal{B}$ such that $q_i \in P_{\mathcal{B},i}$ for all $1 \leq i \leq n$. Here, the process of finding the needed $\mathcal{B}$ transitions is especially costly. In order to speed it up, we cache for each alphabet symbol a, each position i, and each set $P_{\mathcal{B},i}$, the set of transitions $\{(q_1, \ldots, q_n) \xrightarrow{a} q \in \Delta_{\mathcal{B}} : q_i \in P_{\mathcal{B},i}\}$ at the first time it is used in the computation of successors. Then, whenever we need to find all transitions of the form $(q_1, \ldots, q_n) \xrightarrow{a} q$ in $\mathcal{B}$ such that $q_i \in P_{\mathcal{B},i}$ for all $1 \leq i \leq n$, we find them simply by intersecting the sets of transitions cached for each $(P_{\mathcal{B},i}, i, a)$.

Next, we propose another modification of the algorithm which aims to improve the performance especially in those cases where finding a counterexample to inclusion requires us to build representatives of trees with higher depths or in the cases where the inclusion holds. Unlike the original approach which moves only one pair $(p_{\mathcal{A}}, P_{\mathcal{B}})$ from *Next* to *Visited* at the beginning of each iteration of the main loop, we add the newly created pairs $(p_{\mathcal{A}}, P_{\mathcal{B}})$ into *Next* and *Visited* at the same time (immediately after they are generated). This, according to our experiments, allows *Visited* to converge faster towards the fixpoint.

Finally, another optimisation of the algorithm presented in [4] appeared in [2]. This optimisation maintains the sets *Visited* and *Next* as antichains w.r.t. $(\preceq_U, \succeq_U^{\exists\forall})$[4]. Hence, more pairs can be discarded from these sets. Moreover, for pairs that cannot be discarded, one can at least reduce the sets on their right-hand side by removing states that are simulated by some other state in these sets (this is based on the observation that any tree accepted from an upward-simulation-smaller state is accepted from an upward-simulation-bigger state too). Finally, one can also use upward simulations between states of the two automata being compared. Then, one can discard any pair $(p_{\mathcal{A}}, P_{\mathcal{B}})$ such that there is some $p_{\mathcal{B}} \in P_{\mathcal{B}}$ that upward-simulates p_A because it is then clear that no tree can be accepted from $p_{\mathcal{A}}$ that could not be accepted from $p_{\mathcal{B}}$. All these opti-

[4] One says that $P \preceq_U^{\exists\forall} Q$ holds iff $\forall p \in P\ \exists q \in Q :\ p \preceq_U q$. Note also that the upward simulation must be parameterised by the identity in this case [2].

misations are also available in VATA and can optionally be used—they are not used by default since the computation of the upward simulation can be quite costly.

4.5 Top-down Inclusion

Top-down inclusion checking is supported by the explicit top-down and semi-symbolic top-down representations in VATA. Note that when one tries to solve inclusion of TA languages top-down in a naïve way, using a plain subset-construction-like approach, one immediately hits a problem due to the top-down successors of particular states are *tuples* of states. Hence, after one step of the construction, one needs to check inclusion on tuples of states, then tuples of tuples of states, etc. However, there is a way how to get out of this trap as shown in [15,12]. Very roughly said, the main idea of the approach resembles a conversion from the *disjunctive normal form* (DNF) to the *conjunctive normal form* (CNF) taking into account that top-down transitions of tree automata form a kind of and-or graphs (the disjunctions are between top-down transitions and conjunctions among the successors within particular transitions).

VATA contains an implementation of the top-down inclusion checking algorithm of [12]. This algorithm uses several optimisations, e.g., caching of results of auxiliary language inclusion queries between states of the automata whose languages are being compared. More precisely, when checking whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ holds for two tree automata $\mathcal{A}$ and $\mathcal{B}$, the algorithm stores a set of pairs $(p_{\mathcal{A}}, P_{\mathcal{B}}) \in Q_{\mathcal{A}} \times 2^{Q_{\mathcal{B}}}$ for which the language inclusion $\mathcal{L}(p_{\mathcal{A}}) \subseteq \mathcal{L}(P_{\mathcal{B}})$ has been shown *not* to hold. As a further optimisation, the set is stored as an antichain based on comparing the states w.r.t. the downward simulation preorder. The use of the downward simulation is one of the main advantages of this approach compared with the bottom-up inclusion checking since this preorder is cheaper to compute and usually richer than the upward simulation. Indeed, [12] shows that top-down inclusion checking is often—though not always—superior to bottom-up inclusion checking.

Moreover, VATA has recently been extended by a new version of the top-down inclusion checking algorithm that extends the original version by caching even the pairs $(p_{\mathcal{A}}, P_{\mathcal{B}}) \in Q_{\mathcal{A}} \times 2^{Q_{\mathcal{B}}}$ for which the language inclusion $\mathcal{L}(p_{\mathcal{A}}) \subseteq \mathcal{L}(P_{\mathcal{B}})$ has been shown to hold. This extension is far from trivial since the caching must be done very carefully in order to avoid a sort of circular reasoning when answering the various auxiliary language inclusion queries. A precise description of this rather involved algorithm is beyond the scope of this article, and so we refer an interested reader to [13]. As our experiments show, the new kind of caching comes with some overhead, which does not allow it to always win over the previous algorithm, but there are still many cases in which it performs significantly better.

4.6 Computing Simulation over LTS

The explicit part of VATA uses a highly optimised LTS simulation algorithm proposed in [19] and greatly improved in [14]. The main idea of the algorithm is to start with an overapproximation of the simulation preorder (a possible initial approximation is the relation $Q \times Q$) which is then iteratively pruned whenever it is discovered that the simulation relation cannot hold for certain pairs of states. For a better efficiency, the

algorithm represents the current approximation R of the simulation being computed using a so-called *partition-relation pair*. The partition splits the set of states into subsets (called *blocks*) whose elements are equivalent w.r.t. R, and a relation obtained by lifting R to blocks.

In order to be able to deal with the partition-relation pair efficiently, the algorithm needs to record for each block a matrix of counters of size $|Q||\Sigma|$ where, for the given LTS, Q is the set of states and Σ is the set of labels. The counters are used to count how many transitions going from the given state via a given symbol a lead to states in the given block (or blocks currently considered to be bigger w.r.t. the simulation). This information is then used to optimise re-computation of the partition-relation pair when pruning the current approximation of the simulation relation being computed (for details see, e.g., [19]). Since the number of blocks can (and often does) reach the number of states, the naïve solution requires $|Q|^2|\Sigma|$ counters in the worst case. It turns out that this is one of the main barriers which prevents the algorithm from scaling to systems with large alphabets and/or large sets of states.

Working towards a remedy for the above problem, one can observe that the mentioned algorithm actually works in several phases. At the beginning, it creates an initial estimation of the partition-relation pair which typically contains large equivalence classes. Then it initialises the counters for each element of the partition. Finally, it starts the iterative partition splitting. During this last phase, the counters are only decremented or copied to the newly created blocks. Moreover, the splitting of some block is itself triggered by decrementing some set of counters to 0. In practice, late phases of the iteration typically witness a lot of small equivalence classes having very sparsely populated counters with 0 being the most abundant value.

This suggests that one could use sparse matrices containing only the non-zero elements. Unfortunately, according to our experience, this turns out to be the worst possible solution which strongly degrades the performance. The reason is that the algorithm accesses the counters very frequently (it either increments them by one or decrements them by one), hence any data structure with non-constant time access causes the computation to stall. A somewhat better solution is to record the non-zero counters using a hash table, but the memory requirements of such representation are not yet reasonable.

Instead, we are currently experimenting with storing the counters in blocks, using a copy-on-write approach and a zeroed-block deallocation. In short, we divide the matrix of counters into a list of blocks of some fixed size. Each block contains an additional counter (a block-level counter) which sums up all the elements within the block. As soon as a block contains a single non-zero counter only, it can safely be deallocated—the content of the non-zero counter is then recorded in the block-level counter.

Our initial experiments show that, using the above approach, one can easily reduce the memory consumption by the factor of 5 for very large instances of the problem compared to the array-based representation used in [14]. The best value to be used as the size of blocks of counters is still to be studied—after some initial experiments, we are currently using blocks of size $\sqrt{|Q|}$.

Table 1. Experiments with inclusion for the explicit encoding

	expldown	expldown+s	expldown-opt	expldown-opt+s	explup	explup+s
Winner	36.35 %	4.15 %	32.20 %	3.15 %	24.14 %	0.00 %
Timeouts	32.51 %	18.27 %	32.51 %	18.27 %	0.00 %	0.00 %

Table 2. Experiments with the explicit encoding for cases when inclusion does not hold

	expldown	expldown+s	expldown-opt	expldown-opt+s	explup	explup+s
Winner	39.85 %	0.00 %	35.30 %	0.00 %	24.84 %	0.00 %
Timeouts	26.01 %	20.31 %	26.01 %	20.31 %	0.00 %	0.00 %

Table 3. Experiments with the explicit encoding for cases when inclusion holds

	expldown	expldown+s	expldown-opt	expldown-opt+s	explup	explup+s
Winner	0.00 %	47.28 %	0.00 %	35.87 %	16.85 %	0.00 %
Timeouts	90.80 %	0.00 %	90.80 %	0.00 %	0.00 %	0.00 %

5 Experimental Evaluation of VATA

In order to illustrate the level of optimisation that has been achieved in VATA and that can be exploited in its applications (like the Forester tool [11]), we compared its performance against Timbuk and the prototype library considered in [12], which—despite its prototype status—already contained a quite efficient TA implementation.

The comparison of performance of VATA (using the explicit encoding) and Timbuk was done for union and intersection of more than 3,000 pairs of TA. On average, VATA was over 20,000 times faster on union and over 100,000 times faster on intersection.

When comparing VATA with the prototype library of [12], we concentrated on language inclusion testing which is one of the most costly operations on non-deterministic TA. In particular, we conducted a set of experiments evaluating the performance of the VATA's optimised TA language inclusion algorithms on pairs of TA obtained from *abstract regular tree model checking* of the algorithm for rebalancing red-black trees after insertion or deletion of a leaf node (which is the same test set that was used in [12]).

5.1 Experiments with the Explicit Encoding

For the explicit encoding, we measured for each inclusion method the fraction of cases in which the method was the fastest among the evaluated methods on the set of almost 2000 tree automata pairs. The results of this experiment are given in Table 1. The columns are labelled as follows: column `expldown` is for pure downward inclusion checking, column `expldown+s` is for downward inclusion using downward simulation, `expldown-opt` is a column for pure downward inclusion checking with the optimisation proposed in Section 4.5, and column `expldown-opt+s` is downward inclusion checking with simulation using the same optimisation. Columns `explup` and `explup+s` give the results for pure upward inclusion checking and upward inclusion checking with simulation respectively. The timeout was set to 30 s.

We also checked the performance of the algorithms for cases when inclusion either *does* or *does not* hold in order to explore the ability of the algorithms to either find a counterexample in the case when inclusion does not hold, or prove the inclusion in case it does. These results are given in Table 2 and Table 3.

Table 4. Experiments with inclusion for the semi-symbolic encoding

	symdown	symdown+s	symdown-opt	symdown-opt+s	symup
Winner	44.02 %	0.00 %	31.73 %	0.00 %	24.25 %
Timeouts	5.87 %	77.93 %	5.87 %	78.00 %	22.26 %

Table 5. Experiments with the semi-symbolic encoding for cases when inclusion does not hold

	symdown	symdown+s	symdown-opt	symdown-opt+s	symup
Winner	45.03 %	0.00 %	33.06 %	0.00 %	21.91 %
Timeouts	2.48 %	80.03 %	2.48 %	80.09 %	23.39 %

Table 6. Experiments with the semi-symbolic encoding for cases when inclusion holds

	symdown	symdown+s	symdown-opt	symdown-opt+s	symup
Winner	19.74 %	0.00 %	0.00 %	0.00 %	80.26 %
Timeouts	72.37 %	36.84 %	72.37 %	36.84 %	0.00 %

When compared to our previous implementation, VATA performed almost always better. The average speed-up was even as high as 200 times for pure downward inclusion checking. The old implementation was faster in about 2.5 % of the cases, and the difference was not significant.

5.2 Experiments with the Semi-Symbolic Encoding

We performed a set of similar experiments for the semi-symbolic encoding, the results of which are given in Table 4. The columns are labelled as follows: column `symdown` is for pure downward inclusion checking, column `symdown+s` is for downward inclusion using downward simulation, `symdown-opt` is a column for pure downward inclusion checking with the optimisation proposed in Section 4.5 and column `symdown-opt+s` is downward inclusion checking with simulation using the same optimisation. Column `symup` gives the results for pure upward inclusion checking. The timeout was again set to 30 s.

As in the experiments for the explicit encoding, we also checked the performance of the algorithms for cases when inclusion either *does* or *does not* hold. These results are given in Table 5 and Table 6.

When compared to our previous implementation, VATA again performs significantly better, with the pure upward inclusion being on average over 300 times faster and the pure downward inclusion being even over 3000 times faster.

6 Conclusion

This paper introduced and described a new efficient and open-source non-deterministic tree automata library that supports both explicit and semi-symbolic encoding of the tree automata transition function. The semi-symbolic encoding makes use of our own MTBDD package instead of the previously used customisation of the CUDD library.

We wish to continue in this work by attempting to implement a simulation-aware symbolic encoding of antichains using BDDs. Further, we wish to implement other operations, such as determinisation (which, however, is generally desired to be avoided),

or complementation (which we so far do not know how to compute without first determinising the automaton).

Finally, we hope that a public release of our library will attract more people to use it and even better contribute to the code base. Indeed, we believe that the library is written in a clean and understandable way that should make such contributions possible.

References

1. P. A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. Computing Simulations over Tree Automata: Efficient Techniques for Reducing Tree Automata. In *Proc. of TACAS'08*, LNCS 5148, Springer, 2008.
2. P. A. Abdulla, L. Holík, Y.-F. Chen, R. Mayr, and T. Vojnar. When Simulation Meets Antichains (On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata). In *Proc. of TACAS'10*, LNCS 6015, Springer, 2010.
3. P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, LNCS 2404, Springer, 2002.
4. A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In *Proc. of CIAA'08*, LNCS 5148, Springer, 2008.
5. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149, Elsevier, 2006.
6. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, LNCS 4134, Springer, 2006.
7. R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 1986.
8. E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. *FMSD*, 10, Springer, 1997.
9. P. Claves, D. Jansen, S.J. Holtrup, M. Mohr, A. Reis, M. Schatz, and I. Thesing. The LETHAL Library, 2009. URL: `http://lethal.sourceforge.net/`.
10. T. Genet. Timbuk/Taml: A Tree Automata Library, 2003. URL: `http://www.irisa.fr/lande/genet/timbuk`.
11. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV'11*, LNCS 6806, Springer, 2011
12. L. Holík, O. Lengál, J. Šimáček, T. Vojnar. Efficient Inclusion Checking on Explicit and Semi-Symbolic Tree Automata. To appear in *Proc. of ATVA'11*, LNCS 6996, Springer, 2011
13. L. Holík, O. Lengál, J. Šimáček, T. Vojnar. Efficient Inclusion Checking on Explicit and Semi-Symbolic Tree Automata. Tech. rep. FIT-TR-2011-04, FIT BUT, Czech Rep., 2011.
14. L. Holík, J. Šimáček. Optimizing an LTS-Simulation Algorithm. In: *Proc of MEMICS'09*, Znojmo, CZ, FI MU, 2009, p. 93-101, ISBN 978-80-87342-04-6
15. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.*, 27, 2005.
16. L. Ilie, G. Navarro, and S. Yu. On NFA Reductions. In *Proc. of Theory is Forever*, LNCS 3113, Springer, 2004.
17. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA Implementation Secrets. *International Journal of Foundations of Computer Science*, 13(4), 2002.
18. P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. *SIGPLAN Not.*, 46, 2011.
19. F. Ranzato and F. Tapparo. A New Efficient Simulation Equivalence Algorithm. In *Proc. of LICS'07*. IEEE CS, 2007.
20. F. Somenzi. CUDD: CU Decision Diagram Package Release 2.4.2, May 2011.