On the Implementation of Tree Automata: Limitations of the Naive Approach

Hendrik Maryns

Universität Tübingen
Sonderforschungsbereich 441
E-mail: hendrik@sfs.uni-tuebingen.de

Abstract

Monadic Second Order logic (MSO) has been used as a formalism to describe Government and Binding rules. It also lends itself to be used as a query language. Main arguments are its expressive power and linear data complexity. Particularly in the domain of tree banks, where the size of the data is much bigger than the size of the query, the data complexity is an influential factor.

It is known since the late 1960s that an MSO formula can be translated into a tree automaton. These results also predict an more than exponential blowup of the number of states of the tree automaton corresponding to the number of quantifier alternations in the formula. If one refrains from writing complicated formulae and thus avoids the theoretical blowup, one could hope the result becomes practically usable to write a query tool for tree banks. It is shown here that another problem arises: even with few quantifier alternations the transition tables get too big very soon.

This is illustrated by a straightforward implementation of tree automata: by taking the definition in its mathematical sense, and translating every element into an equivalent computer language construct. This was done in a prototype Java version of the tool described above. I describe why this approach reaches its limits all too soon, due to the above problem.

1 Introduction

MSO lends itself to querying tree banks. One of its interesting strengths is its ability to express the transitive closure of any binary relation which is expressible in the language: there is a formula in which a binary relation $R(x,y)$ can be entered,

Hajič, J. and Nivre, J. (eds.): Proceedings of the TLT 2006, pp. 235–246.

which then expresses the transitive closure of this relation. Since MSO is an extension of first-order logic, any first-order relation can be expressed in it, and thus also the transitive closure of any first-order relation.

Transitive closures appeaar quite often and very naturally in linguistics. The need for them has most of the time been met by introducing atomic formulae which directly express the transitive relation. E.g. for the ancestor relation, an atomic formula $x \prec^* y$ is introduced, next to the more basic parent relation $x \prec y$. The limitation of this approach is that those relations are hard-coded in the signature. There is no way to define new ones apart from extending the language. In the case of a corpus query tool, this would mean the extra relation has to be implemented by the programmer himself. In MSO, this can be done dynamically. As it is hard to foresee the need for such relations, this can come in handy.

For example, one could imagine the need to express that two nodes are connected by a path containing only possessive nodes, for example to find instances of chained possessives as in *The man's sister's daughter's cat was killed on the road*. (Of course one could just have said: *the man's niece*, but that would make the example much less interesting...)

The basic relation could be expressed as follows, with x the parent node and y the daughter:

$$R(x,y) :\equiv x \prec y \wedge \mathrm{POSS}(x),$$

where $\mathrm{POSS}(x)$ is supposed to express x is a node which is a possessive, which might either be marked as a node label or be expressed by another appropriate formula. Then this can be inserted in the formula[1] which computes the transitive closure of a binary relation to obtain the desired relation, which again has x and y as free variables:

$$\forall X(\forall z(R(x,z) \rightarrow z \in X) \wedge (\forall z,w(z \in X \wedge R(z,w) \rightarrow w \in X)) \rightarrow y \in X).$$

Here X is a second-order variable which stands for a set.

A formula containing free variables can be interpreted as a query for nodes, or sets of nodes, satisfying the formula. What is needed is a procedure for finding those nodes. This procedure is given by the process of translating an MSO formula into an equivalent finite tree automaton, as described by Thatcher and Wright (1968), Doner (1970). A tree automaton can be evaluated on a tree in linear time in the size of the tree. This sounds promising in the light of the ever-growing modern tree banks, making a low data complexity a precondition for a successful query language.

[1] Due to Courcelle (1990).

Another property of MSO which comes in nicely is that it is decidable over trees[2]. This assures that, when a query gives no results, this effectively means no tree in the tree bank satisfies the formula.

Unfortunately, the translation process itself is non-elementary in the length of the formula: the number of states of the resulting automaton is a tower of exponentials, whose height is linear in the number of quantifier alternations in the formula. That is, the number of states is $2^{n^{\cdot^{\cdot^{n}}}\}p}$, where p, the number of quantifier alternations, is the number of times an existential (universal) quantifier is followed by a universal (resp. existential) quantifier. For example, for the formula $\exists x \forall y\, x \prec^* y$, expressing there is a root node, the number of quantifier alternations is 1. Note that multiple quantifiers of the same type may follow each other without incrementing this number.

Reinhardt (2002) explains that, when one wants to convert formulae in automata, the non-elementary blowup in the number of states is unavoidable, independent of the way one attempts it. At the same time, Frick and Grohe (2002) show that, under a generally assumed precondition[3], there is no model-checking process to evaluate an MSO formula, which does better than the automaton approach. This justifies the focus on automata and is a motivation for looking for optimisations of this approach.

As it is rather difficult to understand a formula with a lot of quantifier alternations, it is to be expected that the potential users of the program will not construct such formulae. This gives hope that the theoretically predicted blowup will not occur, or at least be restricted enough to remain manageable.

Unfortunately, another problem shows up in the implementation of such a tool, which makes a simple implementation of tree automata impossible. I describe this simple, naive implementation, and why it doesn't work, even for relatively simple formulae.

2 Automata and Logic

There are several definitions of tree automata to be found in the literature. Notable differences are in whether initial states are used or not, and in whether the transition function is defined as a partial function or a (complete) algebra.

The use of initial states is closely related to the definition of terms or trees: if one wants to restrict automata to trees of a fixed branching degree, then variables at the leaves are needed, and an initial state or initial assignment in the automaton

[2] More precisely, over graphs with bounded tree width.

[3] $P \neq NP$

definition.[4] If one allows trees of mixed branching, then it is possible to omit them. Thus automata with an initial assignment can be regarded as a bit more expressive than those without.[5]

On the other hand, whether to use a partial function for the transitions or a total function or an algebra over the states is merely a matter of convenience.

The implementation of the tool mentioned above requires an initial state, as the application works on binary trees[6]. It is not necessary to insist on complete transition functions, but they will have to be completed for certain operations. The following definitions are a combination of the definitions in Comon et al. (2002) and Gécseg and Steinby (1984).

2.1 Automata and their properties

A **ranked alphabet** is a couple, $(\mathcal{F}, Arity)$, where $\mathcal{F}$ is a finite set and *Arity* is a mapping from $\mathcal{F}$ into $\mathbb{N}$. The **arity** of a symbol $f \in \mathcal{F}$ is $Arity(f)$. The set of symbols of arity n is denoted by $\mathcal{F}_n$.

Let X be a set of **variables**. Assume X and $\mathcal{F}_0$ are disjoint. The set $T(\mathcal{F}, X)$ of **terms** over the ranked alphabet $\mathcal{F}$ and the set of variables X is the smallest set defined by:

- $\mathcal{F}_0 \subseteq T(\mathcal{F}, X)$;
- $X \subseteq T(\mathcal{F}, X)$;
- if $n \geq 1$, $f \in \mathcal{F}_n$ and $t_1, \ldots, t_n \in T(\mathcal{F}, X)$, then $f(t_1, \ldots, t_n) \in T(\mathcal{F}, X)$.

Terms can be regarded as labelled trees, where each symbol of $f \in \mathcal{F}$ represents a node with label f, and the arguments are its children; hence the name tree automaton.

Definition (Tree Automaton)
*A **finite bottom-up tree automaton** is a quintuple $\mathcal{A} = (\mathcal{F}, Q, Q_f, \alpha, \Delta)$, where $\mathcal{F}$ is a ranked alphabet, also called the signature, Q is the (finite) set of states, $Q_f \subseteq Q$ is the set of final states, $\alpha : X \to Q$ is the initial assignment, and $\Delta : \bigcup_{n \in \mathbb{N}} \mathcal{F}_n \times Q^n \to Q$ is a partial function. X is also called its frontier alphabet.*

Automata following this definition are **deterministic**. If Δ maps into sets of states: $\Delta : \bigcup_{n \in \mathbb{N}} \mathcal{F}_n \times Q^n \to 2^Q$, the automaton is called **nondeterministic**.

An automaton is called **complete** if Δ is a complete function.

[4] Alternatively, one leaves out the arity of symbols, and allows arbitrary symbols at leafs. A corresponding altering of the transition function is needed too.

[5] An intermediary approach is to define only one initial state; however, an initial assignment is more general.

[6] See section 2.3 for the reason why.

A **run** of an automaton on a term is defined as follows: Variables are mapped to states by the initial assignment α. Now given a node labelled with $f \in \mathcal{F}_n$, suppose its children have been processed into states $q_1, \ldots, q_n$, then this node gets mapped to $\Delta(f, q_1, \ldots, q_n)$ if it is defined, or to an arbitrary element of the set if Δ is nondeterministic.

A term is **accepted** by an automaton, if there is a run on the term that assigns a final state to its root.

The language $L(\mathcal{A})$ **recognised** by $\mathcal{A}$ is the set of terms accepted by $\mathcal{A}$. A set L of terms is **recognisable** if there exists an automaton for which $L(\mathcal{A}) = L$. Two automata are **equivalent** if they recognise the same language.

A deterministic automaton is **minimal** if it is the automaton with the smallest number of states in its equivalence class. It was proved that there is a unique minimal deterministic automaton, but there can be several nondeterministic automata with a minimal number of states.

A state is **accessible** if there is a term such that the automaton run on that term results in that state. An automaton is **reduced** if all its states are accessible.

2.2 Manipulations of automata

I give a short sketch of the manipulations that can be done on automata, either unary or binary. For an extensive description of these operations I refer to Comon et al. (2002).

By considering every subset of the set of states as a state, and accordingly recomputing the transition function, it is possible to construct an equivalent deterministic automaton to every nondeterministic automaton. One sees that the **determinisation** construction is exponential in the number of states. Often the blowup can be avoided by considering only accessible states. The worst case, however, does occur.

Completion of an automaton is straightforward: add a new 'sink' state and let every transition that is not yet defined point to it. This operation is linear in the number of transitions of the complete automaton, which is equal to

$$\sum_{n \in \mathbb{N}} |\mathcal{F}_n| \cdot |Q^n|. \tag{1}$$

Note that this number is finite, since we restrict ourselves to finite signatures.

Reduction of an automaton can be implemented straightforwardly by starting at the constants of the signature and the states reached by the initial assignment, and incrementally computing all reachable states. There exists a more efficient algorithm, linear in the number of states.

One can **minimise** a deterministic automaton due to the algorithm following from the Myhill-Nerode Theorem for tree languages. It has to be reduced and complete first. Intuitively, states are equivalent if transitions containing them have the same right-hand side if the states are exchanged. One can compute the equivalence classes of states, and considering those equivalence classes to be the new states gives the minimal automaton. There are a lot of minimisation constructions, the fastest being $O(|Q| \cdot \log |Q|)$.

For a given automaton $\mathcal{A}$, an automaton recognising the **complement** of its language can be constructed. This process is very easy, *assumed $\mathcal{A}$ is deterministic and complete*: simply complement the set of final states. If the automaton is nondeterministic, it has to be determinised first.

For given automata $\mathcal{A}_1$ and $\mathcal{A}_2$, an automaton recognising the **union** of their languages can be constructed. The straightforward construction is to simply take the union of the states, final states, and transition functions. However, this does not preserve determinism and completeness. When both automata are complete, there is a more efficient construction which preserves determinism. Basically, it amounts to taking the cross product of the states and a fitting cross product of the functions. Thus its complexity is linear in the product of the sizes of the transition functions, given in (1).

From the above it follows that **intersection** of automata is also possible. However, there is a construction that does not require determinisation, almost identical to that for union, only the new final states are computed differently.

2.3 The Logic Connection

Monadic Second Order logic is an extension of first-order logic by variables representing sets. Quantification is possible over both first- and second-order variables. We will only be concerned with weak MSO, i.e., the second-order variables will only range over finite sets.

Free variables are encoded in the signature of the automaton. An ordering of the variables is presumed. Each signature symbol is a vector of length the number of free variable symbols in the formula being translated. A 1 in position i means that a node labeled with that symbol is in the set denoted by the ith variable, a 0 means it isn't. A third symbol, $\perp$, is introduced, meaning no element of the corresponding set has been encountered, yet. This entails that every variable is percieved as a set, i.e. a second-order variable. First-order variables are additionally restricted to be singletons. This can be done since the singleton property can be expressed using only second-order variables.

A fixed arity is chosen for the signature symbols, corresponding to the maximal branching degree for the trees to be searched. Arnborg et al. (1991) show it is

possible to convert every formula on trees (or even graphs with bounded tree width) into a formula on binary trees, thus it is safe to simply use 2.

It can be seen that by this encoding, the size of the signature is 3^n, where n is the number of free variables in the formula.

The encoding gives rise to the need for two additional operations, one for adding and one for removing free variables. These are cylindrification and projection, respectively. **Projection** to i does what can be expected from the name: it removes a variable from the signature by dropping the ith element of the symbol vector, also reducing the number of arguments, if necessary. It can easily be seen that this process potentially introduces nondeterminism.

Cylindrification to i is the inverse operation of projection: it adds a variable to the signature symbol encoding at place i, by inserting a slot in the symbol vector, which is filled up with a 0, a 1 or a $\perp$.

With all the above, the translation of an MSO formula into an automaton is easily explained: atomic formulae are translated into simple automata that are calculated by hand; logical connectives correspond immediately to automaton manipulations: negation to complementation, conjunction to intersection, disjunction to union and existential quantification to projection. Other connectives have to be eliminated first by logical manipulations of the formula. Special care must be taken for first-order variables by intersecting automata containing them with an automaton expressing the singleton property.

Note that it follows from this incremental process that binding the free variables by prefixing the formula with quantifiers does not help to reduce the memory usage: the automaton for the quantified subformula, in which the variables *are* free, has to be built first, and it has the larger signature size. A good heuristic is to expect the maximal signature size to be 3^N with N the total number of variables in the formula, though this is not necessarily the case.

The high theoretical time complexity arises because projection most of the times introduces nondeterminism and complementation asks for determinism, so an exponential determinisation step is performed for every quantifier alternation (remember that $\forall \equiv \neg\exists\neg$).

3 Naive Implementation

When trying to implement tree automata, the first and easiest approach is to literally implement the mathematical concepts as data structures.

In short, one takes an implementation of the set construct (i.e. a container data structure that checks for uniqueness of its elements) and uses it to model the state set, final states and signature of the automaton. Analogously, one takes an imple-

mentation of a map (i.e. a data structure that maps keys to values) and uses it to model the transition function and the initial assignment. The first takes a signature symbol with a list of states as keys, the latter a variable symbol. Both take sets of states as values (sets rather than single states in order to cope with nondeterminism).

An implementation of this sort was made in Java, using the standard library interfaces `java.util.Set` and `java.util.Map` together with their implementations `java.util.HashSet` and `java.util.HashMap`. Additional classes were written for the smaller units, such as states and symbols, and to fit all the pieces together.

4 Why It Doesn't Work

Consider the following formula[7],

$$\exists x(\mathrm{SIMPX}(x) \wedge \forall y(x \prec^* y \rightarrow \neg \mathrm{ON}(y))),$$

expressing there is a node labelled SIMPX which has no descendant labelled ON. Here SIMPX and ON are free second order variables, representing the nodes labelled with that symbol and $\prec^*$ denotes domination of nodes.

This formula leads to an automaton with a signature size of 9, with 7 states. Formula (1) tells us this automaton has 441 transitions.

In terms of memory, a transition takes 3 references for the argument (symbol + 2 states), and at least one for the value. In Java, a reference generally takes 4 bytes (see Wilson and Kesselman 2001). Thus the transition function of the above automaton alone takes at least 7056 bytes, or almost 7 kB. The surrounding data structure and memory used for book-keeping is neglected in this computation. All of the objects that are referenced have to be in memory too, which each take at least 8 bytes (see the reference above). Together they account for enough to make this automaton a little more than 7 kB.

Soon though the automata get too big. For the still rather simple conjunctive formula expressing there is a node labelled SIMPX, which has a descendant also labelled SIMPX, which on its turn has no descendants labelled ON:

$$\exists x \exists y(\mathrm{SIMPX}(x) \wedge \mathrm{SIMPX}(y) \wedge x \prec^* y \wedge \neg(x = y) \wedge \forall z(y \prec^* z \rightarrow \neg \mathrm{ON}(z))), \quad (2)$$

intermediary automata are created with a signature size of 81 symbols and 432 states, which leads to 241 864 704 transitions taking 230 MB. The full automaton representing this formula could not be constructed on a machine with 7 GB of memory.

[7] The formulae are examples from Kepser (2003), see there for the rationale behind the formulae.

Considering the above automaton is not judged very big in comparison to what can be expected for formulae with a higher number of quantifier alternations (the above formula has 1 quantifier alternation), it seems the naive approach of explicitly storing each transition is doomed to fail. One could argue that it is possible to use more memory efficient data structures to store states, for example by simple identifying integers, but there will always be a point where the sheer number of transitions is too big to fit in an average computer's main memory, if represented this way.

5 Optimisations

5.1 Simple Optimisations

5.1.1 Signature Encoding

In the light of the space problems described above, it is a good idea to leave the $\perp$ symbol out. This reduces the signature size to 2^n, but makes projection a bit trickier: care must be taken to prune branches consisting only of 0s. This pruning is similar to the process of reduction. As usual, space is traded for time. It should be investigated further whether a more optimal encoding of the variables in the signature is possible.

Although after equation (1) the number of states is much more influential on the space complexity than the signature size, this optimisation should not be neglected, considering there is no way to influence the number of states, being directly connected to the complexity of the formula.

5.1.2 Default State

It is observed that often a large portion of all transitions all lead to the same state. Most of the time this is also a sink state, i.e. once the automaton has got in that state, it cannot get out of it again. A first optimisation of the above implementation is not to store the transitions that lead into this default state and return the default state if no explicit transition is present. At first sight, this approach saves a lot of space. But once union or intersection is performed, it loses its advantages: states in the new automaton represent pairs of states of both old automata, and only the state representing the pair $(\texttt{default}, \texttt{default})$ is an equivalent default state for the new automaton. There are, however, a lot of states which represent pairs in which only one of the two is the default state, and those still need to be explicitly represented, such that the memory savings after union or intersection is relatively low.

5.2 Future Work

Both of the abovementioned simple optimisations have been applied to my program, without much success: it still doesn't succeed at compiling formula (2), although it can be observed that it gets a bit further in the incremental process before space runs out.

This indicates more radical methods are to be considered to reduce the size of the automaton representation in memory. For string automata, a successful optimisation is to use binary decision diagrams (Bryant 1986; 1992) for the transition function, as is discussed in Klarlund et al. (2002). A promising path is thus to investigate whether and how this concept can be generalised to tree automata.

Another problem is the fact that the free variables of the formula are strongly interweaved with the signature of the automaton. The signature is used as an encoding of those variables. Not only does this make the connection between the automaton and the trees it is to be evaluated on less intuitive, it also makes the size of the automaton dependent on the number of (free) variables in the formula. That is an unwanted side-effect. Remember the initial hope that one could avoid the theoretical complexity of the problem by having formulae with few quantifier alternations; this is now complicated by this connection. However, this encoding seems to be crucial for the translation process, so it is not to be expected that substantial improvement can be achieved. Nevertheless, it is worth investigating whether optimisations can be found in this area.

6 Conclusion

It was illustrated that the straightforward way to implement tree automata sketched above, i.e. by explicitly storing each and every transition, is usable only for very restricted applications. As soon as the combination of signature and states gets bigger, the amount of memory needed to represent the transition function gets unreasonably big. When trying to convert monadic second order formulae into automata, the approach is not feasible.

Another factor that greatly complicates the process is the strong connection between the signature of the automata and the free variables of the formulae. This makes the size of the automaton dependant on the number of variables in the formula. Once again, this is an incitement to look for more efficient data structures for the automata.

References

Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991.

Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, Sep 1992.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, October 2002.

Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 5, pages 193–242. Elsevier, 1990.

John E. Doner. Tree acceptors and some of their applications. *Journal of Computer and System Science*, 4:406–451, 1970.

Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 215–224, Washington, 2002. IEEE Computer Society. ISBN 0-7695-1483-9.

Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

Stephan Kepser. Finite structure query: A tool for querying syntactically annotated corpora. In *Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, volume 1, pages 179 – 186, Budapest, 2003.

Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4): 571–586, 2002.

Klaus Reinhardt. The complexity of translating logic to finite automata. In *Automata logics, and infinite games: a guide to current research*, pages 231–238. Springer-Verlag, New York, 2002. ISBN 3-540-00388-6.

James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

Steve Wilson and Jeff Kesselman. *Java™ Platform Performance Strategies and Tactics*. The Java Series... from the Source. Addison Wesley Professional, 2001. URL `http://java.sun.com/docs/books/performance/1st_edition/html/JPTitle.fm.html`.