Fly-automata, their properties and applications

Bruno Courcelle and Irène A. Durand

LaBRI, CNRS, Université de Bordeaux, Talence, France
{idurand,courcell}@labri.fr

Abstract. We address the concrete problem of implementing huge bottom-up term automata. Such automata arise from the verification of Monadic Second Order propositions on graphs of bounded tree-width or clique-width. This applies to graphs of bounded tree-width because bounded tree-width implies bounded clique-width. An automaton which has so many transitions that they cannot be stored in a transition table is represented be a fly-automaton in which the transition function is represented by a finite set of meta-rules.
Fly-automata have been implemented inside the `Autowrite` [1] software and experiments have been run in the domain of graph model checking[2].

1 Introduction

The following theorem connects the problem of verifying graph properties with term (tree) automata.

Theorem 1. *Monadic second-order model checking is* fixed-parameter tractable *for tree-width [Courcelle (1990)] and clique-width [Courcelle, Makowski, Rotics (2001)].*

Tree-width and *clique-width* are graph complexity measures based on graph decompositions. A *decomposition* produces a term representation of the graph. For a graph property expressed in monadic second order logic (MSO), the *algorithm* verifying the property takes the form of a term automaton which recognizes the terms denoting graphs satisfying the property.

In [2], we have given two methods for finding such an automaton given a graph property. The first one is totally general; it computes the automaton directly from the MSO formula; it starts with ad-hoc automata corresponding to atomic formulas and combines them with boolean operations, relabellings and inverse relabellings; however this method it is not practically usable because the intermediate automata that are computed along the construction can be very big even if the final one is not. The second method is very specific: it is a direct construction of the automaton; one must describe the states and the transitions of the automaton. Although the direct construction avoids the bigger intermediate automata, we are still faced with the hugeness of the automata. For instance, one can construct an automaton recognizing graphs which are acyclic has 3^{3^k} states where k is the clique-width of the graph. Even for $k = 2$, which yields the

[1] `http://dept-info.labri.fr/~idurand/autowrite/`
[2] `http://dept-info.labri.fr/~idurand/autograph/`

very restricted class of co-graphs, it is unlikely that we could store the transition table of such an automaton.

The solution to this last problem is to use *fly-automata*. In a fly-automaton, the transition function is represented, not by a table (that would use too much space), but by a finite set of meta-rules. Little space is then required to represent the transition function. In addition, fly-automata are more general than finite bottom-up term automata; they can be infinite in two ways: they can work on an infinite (countable) signature. they can have an infinite (countable) number of states. They are more powerful: a fly-automaton can recognize $\{t \in \mathcal{T}(\mathcal{F}) \mid t = f(t_1, t_2) \text{ and } |t_1| = |t_2|\}$ where $\mathcal{F}$ is a finite signature.

The purpose of this article is to present in detail the concept of fly-automaton and some experiments done with these automata for the verification of properties of graphs of bounded clique-width.

2 Preliminaries: terms

We recall some basic definitions concerning terms. The formal definitions can be found in the on-line book [1]. We call *signature* $\mathcal{F}$ a set of symbols equiped with a function $\mathsf{arity} : \mathcal{F} \to \mathbb{N}$. We denote by $\mathcal{F}_n$ the subset of symbols of $\mathcal{F}$ with arity n. So $\mathcal{F} = \bigcup_n \mathcal{F}_n$. $\mathcal{T}(\mathcal{F})$ denotes the set of (ground) *terms* built upon the signature $\mathcal{F}$. Given a term t, $\mathcal{P}\mathsf{os}(t)$ denotes the set of positions of the term. The position of the root of a term is denoted by ϵ. A term t can also be viewed as a map from its set of positions $\mathcal{P}\mathsf{os}(t)$ to $\mathcal{F}$.

Example 1. Let $\mathcal{F}$ be a signature containing the symbols $\{a, b, add_{a\_b}, rel_{a\_b}, rel_{b\_a}, \oplus\}$ with

$$\boxed{\begin{array}{l}\mathsf{arity}(a) = \mathsf{arity}(b) = 0 \quad \mathsf{arity}(\oplus) = 2\\ \mathsf{arity}(add_{a\_b}) = \mathsf{arity}(rel_{a\_b}) = \mathsf{arity}(rel_{b\_a}) = 1\end{array}}$$

We will see in Section 3 that this signature is suitable to build terms representing graphs of clique-width at most 2.

t_1, t_2, t_3 and t_4 are terms built upon the signature $\mathcal{F}$ of Example 1.

$$\boxed{\begin{array}{l}t_1 = \oplus(a, b)\\ t_2 = add_{a\_b}(\oplus(a, \oplus(a, b)))\\ t_3 = add_{a\_b}(\oplus(add_{a\_b}(\oplus(a, b)), add_{a\_b}(\oplus(a, b))))\\ t_4 = add_{a\_b}(\oplus(a, rel_{a\_b}(add_{a\_b}(\oplus(a, b)))))\end{array}}$$

We will see in Table 1 their associated graphs.

3 Application domain

All this work will be illustrated through the problem of verifying properties of graphs of bounded clique-width. We present here the connection between graphs and terms and the connection between graph properties and term automata.

3.1 Graphs as logical structures

We consider finite, simple, loop-free, undirected graphs (extensions are easy)[3]. Every graph can be identified with the relational structure $\langle \mathcal{V}_G, edg_G \rangle$ where $\mathcal{V}_G$ is the set of vertices and edg_G the binary symmetric relation that describes edges: $edg_G \subseteq \mathcal{V}_G \times \mathcal{V}_G$ and $(x, y) \in edg_G$ if and only if there exists an edge between x and y. Properties of a graph G can be expressed by sentences of relevant logical languages. For instance, G *is complete* can be expressed by $\forall x, \forall y, edg_G(x, y)$ or G *is stable* by $\forall x, \forall y, \neg edg_G(x, y)$ Monadic Second order Logic is suitable for expressing many graph properties like k-colorability, acyclicity, ...

3.2 Term representation of graphs of bounded clique-width

Let $\mathcal{L}$ be a finite set of vertex labels and let us consider graphs G such that each vertex $v \in \mathcal{V}_G$ has a label $label(v) \in \mathcal{L}$. The operations on graphs are $\oplus$[4], the union of disjoint graphs, the unary edge addition add_{a_b} that adds the missing edges between every vertex labeled a and every vertex labeled b, the unary relabeling rel_{a_b} that renames a to b (with $a \neq b$ in both cases). A constant term a denotes a graph with a single vertex labeled by a and no edge. Let $\mathcal{F}_\mathcal{L}$ be the set of these operations and constants. Every term $t \in \mathcal{T}(\mathcal{F}_\mathcal{L})$ defines a graph $G(t)$ whose vertices are the leaves of the term t. Note that, because of the relabeling operations, the labels of the vertices in the graph $G(t)$ may differ from the ones specified in the leaves of the term. A graph has *clique-width* (*cwd* for short) at most k if it is defined by some $t \in \mathcal{T}(\mathcal{F}_\mathcal{L})$ with $|\mathcal{L}| \leq k$.

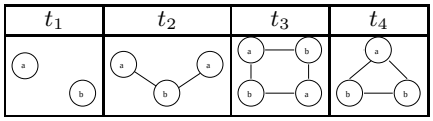

Table 1. The graphs corresponding to the terms of Example 1

We will express graph properties using MSO formulas that formalize coloring and partitioning problems to take a few examples.

4 Term automata

We recall some basic definitions concerning term automata. Again, much more information can be found in the on-line book [1].

4.1 Finite bottom-up term automata

Definition 1. *A finite (bottom-up)* term automaton[5] $\mathcal{A}$ *is a quadruple* $(\mathcal{F}, Q_\mathcal{A}, Q_\mathcal{A}^{Acc}, \Delta_\mathcal{A})$ *consisting of a finite signature* $\mathcal{F}$*, a finite set* $Q_\mathcal{A}$ *of states, disjoint from* $\mathcal{F}$*, a subset*

[3] We consider such graphs for simplicity of the presentation but we can work as well with directed graphs, loops, labeled vertices and edges

[4] `oplus` will be used instead of $\oplus$ inside the software `Autowrite`

[5] Term automata are frequently called tree automata, but it is not a good idea to identify trees, which are particular graphs, with terms.

$Q_{\mathcal{A}}^{Acc} \subseteq Q_{\mathcal{A}}$ *of accepting states, and a set of transitions rules* $\Delta_{\mathcal{A}}$*. Every transition is of the form* $f(q_1, \ldots, q_n) \to q$ *with* $f \in \mathcal{F}$, $\mathsf{arity}(f) = n$ *and* $q_1, \ldots, q_n, q \in Q_{\mathcal{A}}$.

Example 2. Figure 1 shows an example of such an automaton. It recognizes terms representing graphs of clique-width 2 which are stable (do not contain edges). State `<a>` (resp. `<b>`) means that we have found at least a vertex labeled `a` (resp. `b`). State `<ab>` means that we have at least a vertex labeled `a` and at least a vertex labeled `b` but no edge. State `error` means that we have found at least an edge so that the graph is not stable. Note that when we are in the state `<ab>`, an `add_a_b` operation creates at least an edge so we reach the `<error>` state.[6]

```
Automaton 2-STABLE
Signature: a b ren_a_b:1 ren_b_a:1 add_a_b:1 oplus:2*
States: <a> <b> <ab> <error>
Accepting States: <a> <b> <ab>
Transitions  a -> <a>           b -> <b>
  add_a_b(<a>) -> <a>           add_a_b(<b>) -> <b>
  ren_a_b(<a>) -> <b>           ren_b_a(<a>) -> <a>
  ren_a_b(<b>) -> <b>           ren_b_a(<b>) -> <a>
  ren_a_b(<ab>) -> <b>          ren_b_a(<ab>) -> <a>
  oplus*(<a>,<a>) -> <a>        oplus*(<b>,<b>) -> <b>
  oplus*(<a>,<b>) -> <ab>       oplus*(<b>,<ab>) -> <ab>
  oplus*(<a>,<ab>) -> <ab>      oplus*(<ab>,<ab>) -> <ab>
  add_a_b(<ab>) -> <error>      ren_a_b(<error>) -> <error>
  add_a_b(<error>) -> <error>   ren_b_a(<error>) -> <error>
  oplus*(<error>,q) -> <error> for all states q
```

Fig. 1. An automaton recognizing terms representing stable graphs

Finite term automata recognize *regular* term languages[7]. The class of regular term languages is closed under the Boolean operations (union, intersection, complementation) on languages which have their counterpart on automata. For all details on terms, term languages and finite term automata, the reader should refer to [1]. Figure 2 shows in a graphical way the run of the automaton `2-STABLE` on a term representing a graph of clique-width 2. Below we show a successful run of the automaton on a term representing a stable graph.

```
add_a_b(ren_a_b(oplus(a,b))) -> add_a_b(ren_a_b(oplus(<a>,b)))
-> add_a_b(ren_a_b(oplus(<a>,<b>)) -> add_a_b(ren_a_b(<ab>))
-> add_a_b(<b>) -> <b>
```

To distinguish these automata from the infinite automata defined in the next section (4.2) and as we only deal with terms in this paper we will refer to the previously defined term automata as *table-automata*.

[6] Our software Autowrite takes into account the notion of commutative symbols. The star in `oplus*` means that this symbol is commutative. When we have a rule like `oplus*(q1,q2) -> q` the rule `oplus*(q2,q1) -> q` is implicitly defined.

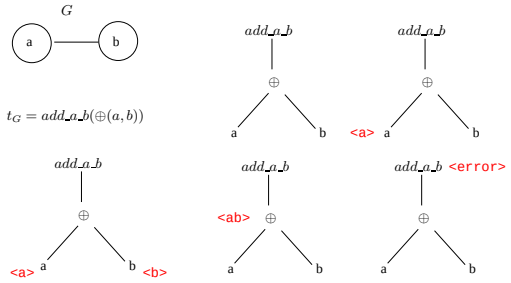

Fig. 2. Graphical representation of an (unsuccessful) run of the automaton on a term

4.2 Infinite (bottom-up) term automata

Definition 2. *From now on, a* term automaton $\mathcal{A}$ *will be given by* $(\mathcal{F}, \delta, \mathsf{acf})$ *or* $(\mathcal{F}, \delta)$ *where the signature* $\mathcal{F}$ *may be countably infinite,* δ *is the transition relation defined as a function*

$$\begin{array}{rl} \delta : \bigcup_n \mathcal{F}_n \times Q^n \to \mathcal{P}_f(Q) & \\ \quad f q_1 \dots q_n & \mapsto \{q \in Q \mid f(q_1, \dots, q_n) \to q\} \end{array}$$

where the set Q *of states accessible using* δ *may be countably infinite,* $\mathcal{P}_f(Q)$ *is the set of* **finite** *subsets of* Q *and* acf *is the accepting state function* $\mathsf{acf} : Q \to Boolean$ *which indicates whether a state is accepting or not. Note that the set of states* Q *is given implicitely by* δ*. The notions of a run, of an accepting run, the sets* $\mathcal{L}(\mathcal{A}, q)$ *and* $\mathcal{L}(\mathcal{A})$ *are the same. Term automata may be complete and/or deterministic in an obvious way. We will shortly consider effectivity conditions insuring that membership of a term to* $\mathcal{L}(\mathcal{A})$ *is decidable.*

Sometimes, in the case where the number of states is infinite, these automata will have no accepting state function. It is the case for instance, for counting automata as shown in the following example.

Example 3. The automaton `COUNTING` presented p.6 is an example of an infinite automaton. Given a term, it counts the number of vertices of the associated graph of any clique-width. State `<i>` means that we have found i vertices. The set of states $Q = \{\texttt{<i>} \mid i \in \mathbb{N}\}$. There is no accepting state function. However, if we want an automaton recognizing terms corresponding to graphs having an *prime* number (or a multiple of some integer) of vertices, we may add an accepting state function $\mathsf{acf} : \texttt{<i>} \mapsto T$ if i is prime, F otherwise. Note that as the automaton works on graphs of any clique-width, we need a countable set of labels, so we use numbers instead of letters in the finite examples.

4.3 Fly term automata

Definition 3. *A* fly-automaton *is an automaton* $(\mathcal{F}, \delta, \mathsf{acf})$ *such that* δ *and* acf *are computable functions.*

Theorem 2. *Let* $\mathcal{A}$ *be a fly-automaton. Membership to* $\mathcal{L}(\mathcal{A})$ *is decidable. The emptiness of* $\mathcal{L}(\mathcal{A})$ *is not decidable.*

```
Automaton COUNTING
Signature: 0 1 2 ...
  ren_0_1:1 ren_1_0:1 add_0_1:1 ren_0_2:1 ren_2_0:1 add_0_2:1 ...
  oplus:2*
States: <0> <1> <2> ...
Metarules:
  x -> <1> for all x
  add_x_y(<i>) -> <i> for all x,y s.t. x < y
  ren_x_y(<i>) -> <i> for all x,y
  oplus*(<i>,<j>) -> <i+j> for all i,j
```

Theorem 3. *Fly-automata are closed under Boolean operations, arity-preserving relabellings and inverse-relabellings.*

Proof. Let $\mathcal{A} = (\mathcal{F}, \delta, \mathsf{acf})$ be a deterministic and complete fly-automaton. The complement of $\mathcal{A}$ is $(\mathcal{F}, \delta, \mathsf{acf}^c)$ where $\mathsf{acf}^c(q) = \neg\mathsf{acf}(q)$ for every $q \in Q_\mathcal{A}$.
Given two fly-automata $\mathcal{A}_1 = (\mathcal{F}, \delta_1, \mathsf{acf}_1)$ and $\mathcal{A}_2 = (\mathcal{F}, \delta_2, \mathsf{acf}_2)$, one can easily define a computable transition function δ corresponding to the product of the two automata whose states are in $Q_\mathcal{A} \times Q_\mathcal{B}$. The following accepting state functions are suitable (and computable) for union and intersection respectively.

$$\begin{array}{lll} \mathsf{acf}^u : Q_\mathcal{A} \times Q_\mathcal{B} \to Boolean & & \mathsf{acf}^i : Q_\mathcal{A} \times Q_\mathcal{B} \to Boolean \\ \quad \{q_1, q_2\} \quad \mapsto \mathsf{acf}_1(q_1) \vee \mathsf{acf}_2(q_2) & & \quad \{q_1, q_2\} \quad \mapsto \mathsf{acf}_1(q_1) \wedge \mathsf{acf}_2(q_2) \end{array}$$

Then $\mathcal{A}_1 \cup \mathcal{A}_2 = (\mathcal{F}, \delta, \mathsf{acf}^u)$ and $\mathcal{A}_1 \cap \mathcal{A}_2 = (\mathcal{F}, \delta, \mathsf{acf}^i)$ are fly-automata. The proofs are similar for arity-preserving relabellings and inverse-relabellings.

In the same spirit, fly-automata may be determinized and completed. The determinized version of $\mathcal{A}$ is an automaton $d(\mathcal{A}) = (\mathcal{F}, \delta', \mathsf{acf}')$. If $Q_\mathcal{A}$ is the domain of δ (the set of states of $\mathcal{A}$), let $d(Q_\mathcal{A})$ denote the set of states of $d(\mathcal{A})$. Each subset $\{q_1, \ldots, q_p\}$ of $Q_\mathcal{A}$ yields a state $[q_1, \ldots, q_p]$ in $d(Q_\mathcal{A})$. δ' is defined by with

$$\begin{array}{c} \delta' : \bigcup_n \mathcal{F}_n \times d(Q_\mathcal{A})^n \to d(Q_\mathcal{A}) \\ f, S_1, \ldots, S_n \quad \mapsto S \end{array}$$

with $q \in S$ if and only if $\exists q_1, \ldots, q_b \in S_1 \times \ldots S_n$ such that $q \in \delta(f, q_1, \ldots, q_n)$.

When a fly-automaton $(\mathcal{F}, \delta, \mathsf{acf})$ is finite, it can be compiled into a table-automaton $(\mathcal{F}, Q, Q^{Acc}, \Delta)$, provided that the resulting table is not too big. The transition table Δ can be computed from δ starting from the constant transitions and then saturating the table with transitions involving new accessible states until no new state is computed. The set of (accessible) states Q is obtained during the construction of the transitions table. The set of accepting states Q^{Acc} is obtained by removing the non accepting states (according to the accepting state function acf) from the set of states. A table-automaton is a particular case of a fly-automaton. It can be seen as a compiled version of a fly-automaton whose transition function δ is described by the transitions table Δ and whose accepting state function acf corresponds to membership to Q^{Acc}. It follows that the automata operations defined for fly-automaton will work for table-automata. Table-automata are faster for recognizing a term but they use space for storing the transitions table and the access time may be important in case of a very large table.

Fly-automata use a much smaller space (the space corresponding to the code of the transition function) but are slower for term recognition when the transition function is complex. A table-automaton should be used when the transition table can be computed in reasonable space and a fly-automaton otherwise.

5 Implementation of fly-automata

We will call *basic* fly-automata the ones that are built from scratch in order to distinguish them from the ones that are obtained by combinations of existing automata using the operations cited in Theorem 3, determinization and completion. We call the later *composed* fly-automata. Fly-automata have been implemented inside the software Autowrite [5] (entirely written in Common Lisp) which already had table-automata. States are not stored in the representation. For basic fly-automata, they are created on the fly by calls to the transition function. For composed automata, the states returned by the transition function are constructed from the ones returned from the transition functions of the combined automata. For operations like determinization, inverse-relabellings, sets of states are involved. The implementation of fly-automata use intensively the functional paradigm to represent and combine transition and accepting states functions. More details about the implementation can be found in [6]. The main operations that are implemented on fly-automata are: run of an automaton $\mathcal{A}$ on a term t, recognition of a term t by an automaton $\mathcal{A}$, decision of emptiness for $\mathcal{L}(\mathcal{A})$ (when $\mathcal{A}$ is finite), completion, determinization, complementation of an automaton $\mathcal{A}$, union, intersection of two (or more) automata, relabellings and inverse-relabellings of constants.

For table-automata, we have also implemented reduction (removal of inaccessible states), minimization but this is not discussed in this paper. Because a table-automaton can always be transformed into a fly-automaton and a finite fly-automaton back to a table-automaton we get the corresponding operations for table-automata for free once we have implemented them for fly-automata. However, for efficiency reasons, it might be interesting to implement some of these operations at the level of table-automaton. For instance, the complementation which consists in inverting non accepting and accepting states is easily performed directly on a table-automaton. Implementing operations directly at the level of table-automaton has the drawback that it depends on the representation chosen for the transitions table. Whenever, we would want to change this representation we would have to re-implement these operations. The only advantage is a gain in efficiency. Some operations on table-automata may give a blow-up in terms of the size of the transition table (determinization, intersection). In this case, the solution is to omit to compile the resulting operation back to a table-automaton. It is though possible to deal uniformly with table and fly-automata.

6 Experiments

Most of our experiments have been run in the domain of verifying graph properties. Many construction of basic automata can be found in [4, 3] and have been implemented with `Autowrite`. In order to compare the running time of a fly-automaton and that of the corresponding table-automaton, we have chosen a property and a clique-width for

which the automaton is compilable. This is the case for the *connectedness property*. We have a direct construction of an automaton verifying whether a graph is connected. The corresponding table automaton has $2^{2^{cwd}-1} + 2^{cwd} - 2$ states. It is compilable up to $cwd = 3$. For $cwd = 4$, which gives $|Q| = 32782$, we run out of memory. It is possible to show that the number of states of the minimal automaton is $|Q| > 2^{2^{\lfloor cwd/2 \rfloor}}$. So there is no hope of having a table-automaton for this property and $cwd > 3$.

We have direct constructions of the automata for properties like Edge(X_1, X_2), k-Cardinality(), k-Coloring$(X_1, \ldots, X_k)$, Connectedness(), Acyclic() among others. With these properties and using relabellings and Boolean operations, we obtain automata for properties like k-Colorability(), k-Acyclic-Colorability(), k-Vertex-Cover() among others. The Vertex-Cover property can be expressed by a combination (intersection, homomorphisms) of already defined basic automata (stablility, k-cardinality).

Many problems that where unthinkable to solve with table-automata could be solved with fly-automata. For very difficult (NP-complete) problems we still reach time or space limitations.

7 Conclusion and perspectives

In the near future, we plan to implement more graph properties and to run tests on real and random graphs. We cannot hope to check arbitrary Monadic Second Order formulas because, even on words, the problem is intractable if the formula is part of the input. However, many interesting graph properties seem to be reachable. We did not address the problem of finding terms representing a graph, that is, to find a clique-width decomposition of the graph. In some cases, the graph of interest comes with a "natural decomposition" from which the clique decomposition of bounded clique-width is easy to obtain but for the general case the known algorithms are not practically usable.

The concept of fly-automaton is general and could be applied to other domains where big automata are needed.

References

1. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2002), draft, available from `http://tata.gforge.inria.fr`
2. Courcelle, B., Durand, I.: Verifying monadic second order graph properties with tree automata. In: Proceedings of the 3rd European Lisp Symposium. pp. 7–21 (May 2010)
3. Courcelle, B., Durand, I.: Automata for the verification of monadic second-order graph properties (2011), in preparation
4. Courcelle, B., Engelfriet, J.: Graph structure and monadic second-order logic, a language theoretic approach (2011), available at `http://www.labri.fr/perso/courcell/Book/CourGGBook.pdf` To be published by Cambridge University Press
5. Durand, I.: Autowrite: A tool for term rewrite systems and tree automata. Electronics Notes in Theorical Computer Science 124, 29–49 (2005)
6. Durand, I.: Implementing huge term automata. In: Proceedings of the 4th European Lisp Symposium. pp. 17–27 (March 2011)
7. Thatcher, J., Wright, J.: Generalized finite automata theory with an application to a decision problem of second-order logic. Mathematical Systems Theory 2, 57–81 (1968)