

# Verifying monadic second order graph properties with tree automata

**Bruno Courcelle**

courcell@labri.fr

**Irène A. Durand**

idurand@labri.fr

LaBRI, CNRS, Université de Bordeaux, Talence, France

**Abstract:** We address the concrete problem of verifying graph properties expressed in Monadic Second Order (MSO) logic. It is well-known that the model-checking problem for MSO logic on graphs is fixed-parameter tractable (FPT) [Cou09, Chap 6] with respect to tree-width and clique-width. The proof uses tree-decompositions (for tree-width as parameter) and clique-decompositions (for clique-width as parameter), and the construction of a finite tree automaton from an MSO sentence, expressing the property to check. However, this construction may fail because either the intermediate automata are too big even though the final automaton has a reasonable size or the final automaton itself is too big to be constructed: the sizes of automata depend, exponentially in most cases, on the tree-width or the clique-width of the graphs to be verified. We present ideas to overcome these two causes of failure. The first idea is to give a direct construction of the automaton in order to avoid explosion in the intermediate steps of the general algorithm. When the final automaton is still too big, the second idea is to represent the transition function by a function instead of computing explicitly the set of transitions; this entirely solves the space problem. All these ideas have been implemented in Common Lisp.

**Key Words:** Tree automata, Monadic second order logic, Graphs, Lisp

## 1 Introduction

It is well-known from [DF99], [FG06],[CMR01] that the model-checking problem for MSO logic on graphs is fixed-parameter tractable (FPT) with respect to tree-width and clique-width (*cwd*).

The standard proof is to construct a finite bottom-up tree automaton that recognizes a tree (or clique) decomposition of the graph. However, the size of the automaton can become extremely large and cannot be bounded by a fixed elementary function of the size of the formula unless  $P=NP$  [FG04]. This makes the problem hard to tackle in practice, because it is just impossible to construct the tree automaton.

Systematic approaches have been proposed for subclasses of MSO formulas with limited quantifications in [KL09]. Our approach is not systematic; we consider specific problems which we want to solve in practice, for large classes of graphs.

In the general algorithm, the combinatorial explosion may occur each time we encounter an alternation of quantifiers which induces a determinization of the current automaton. We want to avoid determinizations as much as possible. Initial ideas to achieve this goal were first presented in [CD10].

We do not capture all MSO graph properties, but we can formalize in this way coloring and partitioning problems to take a few examples. In this article, we only discuss graphs of bounded clique-width, but the ideas work as well for graphs of bounded tree-width, in particular because if a graph has a tree-width  $tw \leq k$ , it has a clique-width  $cw \leq 2^{k+1}$ . There is however an exponential blow-up.

The Autowrite<sup>1</sup> software written in Common Lisp was first designed to check call-by-need properties of term rewriting systems [Dur02]. For this purpose, it implements tree (term) automata. In the first implementation, just the emptiness problem (does the automaton recognize the empty language) was used and implemented.

In subsequent versions [Dur05], the implementation was developed in order to provide a complete library of operations on term automata. The next natural step is to solve concrete problems using this library and to test the limits of the implementation. Checking graph properties is a perfect challenge for Autowrite.

Given a property expressed by a MSO formula, we have experimented the three following techniques.

1. compute the automaton from the MSO formula and using the general algorithm,
2. compute directly the final automaton,
3. define the automaton with implicit transition function instead of computing its set of transitions.

The first technique is the only one which is completely general in theory. The two first techniques have the advantage that once the final automaton is computed (and minimized), it can be memorized for further use. The minimal automaton obtained in both cases is unique: it depends only on the property and not on its logical description. This can be helpful to verify that the two constructions are correct.

The limits are soon reached using the first technique. The second technique allows to go somewhat further. With the third technique there is almost no more limitation (at least not the same ones) because the whole automaton is never constructed.

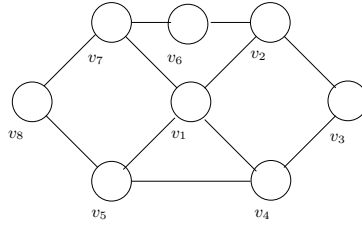
In this paper, we do not address the problem of finding terms representing a graph, that is, to find a clique-width decomposition of the graph. In some cases, the graph of interest may come with a “natural decomposition” from which the clique decomposition of bounded clique-width is easy to obtain but for the general case the known algorithms are not practically usable.

To illustrate our approach, we shall stick to a unique example along the paper although we have made experiments with many more graph properties.

**Path Property:** Let  $Path(X_1, X_2)$  be the monadic second-order formula expressing that, for an undirected graph  $G$  and sets  $X_1$  and  $X_2$  of vertices this graph, we have  $X_1 \subseteq X_2$ ,  $|X_1| = 2$  and there is a path in  $G[X_2]$  linking the two vertices of  $X_1$ <sup>2</sup>.

<sup>1</sup> <http://dept-info.labri.fr/~idurand/autowrite/>

<sup>2</sup> To simplify the presentation, we confuse somewhat syntax and semantics. We note in the same way a variable  $X_i$  and its values (sets of vertices)



**Figure 1:** A graph to test the property  $Path(X_1, X_2)$

Consider the graph of Figure 1. If  $X_1 = \{v_3, v_8\}$  and  $X_2 = \{v_1, v_3, v_4, v_7, v_8\}$  the property  $Path(X_1, X_2)$  holds for  $G$ :  $|X_1| = 2$  and there is a path  $v_8 - v_7 - v_1 - v_4 - v_3$  from  $v_8$  to  $v_3$  with vertices in  $X_2$ . The property does not hold if  $X_1 = \{v_3, v_8\}$  and  $X_2 = \{v_1, v_3, v_4, v_8\}$ .

For  $cwd = 2$ , we were able to obtain the term automaton (see below how terms describe graphs) directly from the MSO formula starting from the automata representing the basic operations, transforming and combining them with boolean operations, determinization, complementation, projection, cylindrification. But it runs out of memory for  $cwd = 3$ .

We were successful in constructing the direct automaton for  $cwd$  up to 4. But for  $cwd = 5$ , the program runs out of memory because the constructed automaton is simply too big.

For higher clique-width, there is no way of representing explicitly the transitions. This is when the third method comes on stage. The really new idea here is to represent the transition function precisely by a function. Consequently, there is no more need to store the transitions. Transitions are computed on the fly when the automaton is running on a given term (representing a graph). A graph of clique-width  $k$  having  $n$  vertices is represented by a term  $t$  of size  $|t| \leq f(k).n$ . Hence, only  $|t|$  transitions are needed. This number is in practice much less than the number of transitions of an automaton able to process all possible terms denoting graphs of clique-width  $\leq k$ .

After recalling how graphs of bounded clique-width are represented by terms and how properties on such graphs can be expressed in MSO, we shall describe our experiments using Autowrite trying to construct automata verifying properties on graphs.

## 2 Preliminary

### 2.1 Term automata

We recall some basic definitions concerning terms and term automata. Much more information can be found in the on-line book [CDG<sup>+</sup>02]. We consider a finite signature

$\mathcal{F}$  (set of symbols with fixed arity) and  $\mathcal{T}(\mathcal{F})$  the set of (ground) terms built from a signature  $\mathcal{F}$ .

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

$$\begin{array}{l} \text{arity}(a) = \text{arity}(b) = 0 \quad \text{arity}(\oplus) = 2 \\ \text{arity}(add_{a,b}) = \text{arity}(rel_{a,b}) = \text{arity}(rel_{b,a}) = 1 \end{array}$$

We shall see in Section 2.3 that this signature is suitable to write terms representing graphs of clique-width at most 2.

*Example 2.*  $t_1, t_2, t_3$  and  $t_4$  are terms built with the signature  $\mathcal{F}$  of Example 1.

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

We shall see in Table 1 their associated graphs.

**Definition 1.** A (finite bottom-up) *term automaton*<sup>3</sup> is a quadruple  $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$  consisting of a finite signature  $\mathcal{F}$ , a finite set  $Q$  of states, disjoint from  $\mathcal{F}$ , a subset  $Q_f \subseteq Q$  of final states, and a set of transitions rules  $\Delta$ . Every transition is of the form  $f(q_1, \dots, q_n) \rightarrow q$  with  $f \in \mathcal{F}$ ,  $\text{arity}(f) = n$  and  $q_1, \dots, q_n, q \in Q$ .

Term automata recognize *regular* term languages[TW68]. The class of regular term languages is closed by the boolean operations (union, intersection, complementation) on languages which have their counterpart on automata. For all details on terms, term languages and term automata, the reader should refer to [CDG<sup>+</sup>02].

## 2.2 Graphs as a logical structure

We consider finite, simple, loop-free, undirected graphs (extensions are easy)<sup>4</sup>. 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  $y$ .

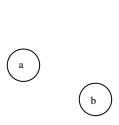
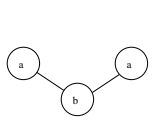
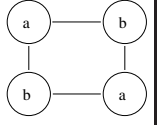
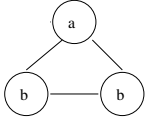
Properties of a graph  $G$  can be expressed by sentences of relevant logical languages. For instance, “ $G$  is complete” can be expressed by

$$\forall x, \forall y, edg_G(x, y)$$

Monadic Second order Logic is suitable for expressing many graph properties.

<sup>3</sup> Term automata are frequently called tree automata, but it is not a good idea to identify trees, which are particular graphs, with terms.

<sup>4</sup> We consider such graphs for simplicity of the presentation but we can work as well with directed graphs, loops, labeled vertices and edges

$t_1$	$t_2$	$t_3$	$t_4$
			

**Table 1:** Graphs corresponding to the terms of Example 2

### 2.3 Term representation of graphs of bounded clique-width

**Definition 2.** Let  $\mathcal{L}$  be a finite set of vertex labels and we 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  $\oplus$ , the union of disjoint graphs, the unary edge addition  $add_{a,b}$  that adds the missing edges between every vertex labeled  $a$  to 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* at most  $k$  if it is defined by some  $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}})$  with  $|\mathcal{L}| \leq k$ .

Note also that if the term  $t$  describing a graph  $G$  does not use redundancies like  $add_{a,b}(add_{a,b}(\dots))$ , then  $|t| = \Theta(|\mathcal{V}_G|)$ .

*Example 3.* For  $\mathcal{L} = \{a, b\}$ , the corresponding signature has already been presented in Example 1. The graphs corresponding to the terms defined in Example 2 are depicted in Table 1.

*Example 4.* The graph of Figure 1 is of clique-width  $\leq 5$ . It can be represented with the term built with  $\mathcal{L} = \{a, b, c, d, e\}$  and shown on the left of Figure 2.

Let  $X_1, \dots, X_m$  be sets of vertices of a graph  $G$ . We can define properties of  $(X_1, \dots, X_m)$ . For example,

- $E(X_1, X_2)$  : there is an edge between some  $x_1 \in X_1$  and some  $x_2 \in X_2$ ;
- $Sgl(X_2)$  :  $X_2$  is a singleton set;
- $X_1 \subseteq X_2$  :  $X_1$  is a subset of  $X_2$ .

**Definition 3.** Let  $P(X_1, \dots, X_m)$  be a property of sets of vertices  $X_1, \dots, X_m$  graphs  $G$  denoted by terms  $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}})$ . Let  $\mathcal{F}_{\mathcal{L}}^m$  be obtained from  $\mathcal{F}_{\mathcal{L}}$  by replacing each constant  $a$  by the constants  $a^w$  where  $w \in \{0, 1\}^m$ . For fixed  $\mathcal{L}$ , let  $L_{P, (X_1, \dots, X_m), \mathcal{L}}$

be the set of terms  $t$  in  $\mathcal{T}(\mathcal{F}_{\mathcal{L}}^m)$  such that  $P(X_1, \dots, X_m)$  is true in  $G(t)$ , where  $X_i$  is the set of vertices which corresponds to the leaves labeled by  $a^{\wedge}w$  where the  $i$ -th bit of  $w$  is 1. Hence  $t \in \mathcal{T}(\mathcal{F}_{\mathcal{L}}^m)$  defines a graph  $G(t)$  and an assignment of sets of vertices to the set variables  $X_1, \dots, X_m$ .

*Example 5.* The graph of Figure 1 with vertex assignment  $X_1 = \{v_3, v_8\}$  and  $X_2 = \{v_1, v_3, v_4, v_7, v_8\}$  can be represented<sup>5</sup> by the term at the right of Figure 2; it satisfies the path property. With vertex assignment  $X_1 = \{v_3, v_8\}$  and  $X_2 = \{v_1, v_3, v_4, v_8\}$ , it can be represented by almost the same term but with  $b^{\wedge}00[v7]$  instead of  $b^{\wedge}01[v7]$  but it does not satisfy the path property anymore.

<pre> add_c_d(   add_b_d(     oplus(       d[v1],       rel_d_b(         add_a_d(           oplus(             d[v2],             add_c_e(               oplus(                 add_a_b(                   add_b_c(                     oplus(                       a[v3],                       oplus(                         b[v4],                         c[v5])))),                 add_a_b(                   add_b_e(                     oplus(                       a[v6],                       oplus(                         b[v7],                         e[v8]))))))))))) </pre>	<pre> add_c_d(   add_b_d(     oplus(       d^01[v1],       rel_d_b(         add_a_d(           oplus(             d^00[v2],             add_c_e(               oplus(                 add_a_b(                   add_b_c(                     oplus(                       a^11[v3],                       oplus(                         b^01[v4],                         c^00[v5])))),                 add_a_b(                   add_b_e(                     oplus(                       a^00[v6],                       oplus(                         b^01[v7],                         e^11[v8]))))))))))) </pre>
--	--

**Figure 2:** Terms representing the graph of Figure 1

*Example 6.* The property  $Path(X_1, X_2)$  can be expressed by the following MSO formula:

$$\begin{aligned}
& \forall x[x \in X_1 \Rightarrow x \in X_2] \wedge \\
& \exists x, y[x \in X_1 \wedge y \in X_1 \wedge x \neq y \wedge \forall z(z \in X_1 \Rightarrow x = z \vee y = z) \wedge \\
& \forall X_3[x \in X_3 \wedge \forall u, v(u \in X_3 \wedge u \in X_2 \wedge v \in X_2 \wedge \text{edg}(u, v) \Rightarrow v \in X_3) \Rightarrow y \in X_3]]
\end{aligned}$$

of quantifier-height 5. Uppercase variables denote sets of vertices, and lowercase variables denote individual vertices.

### 3 Implementation of term automata

The part of Autowrite which is of interest for this work is the implementation of term automata together with some operations on these automata.

The main operations that are implemented are:

<sup>5</sup> Note that the vertex number inside brackets is not part of the signature; it is there to help the reader make the correspondence between the leaves of the term and the vertices of the graph.

- Reduction (removal of inaccessible states), decision of emptiness; they have been implemented in the very first version of Autowrite.
- Determinization, Complementation, Minimization, Union, Intersection which have been added in subsequent versions of Autowrite.
- Signature transformation, Projection and Cylindrification which have been added to deal with changes of signatures typically from  $\mathcal{F}_{\mathcal{L}}^m$  to  $\mathcal{F}_{\mathcal{L}}^{m'}$ .

The object at the core of this library is the term automaton. The efficiency of many operations depends heavily on the data structures chosen to represent the states and transitions of the automata. Since the first version of Autowrite [Dur02], much care has been devoted to improve the representation of automata and the performances have improved significantly. However, this work, which leads us to the limits of what is computable in a human's life, has also shown limits in our implementation, in terms of space and time. In particular, we have realized that representing the set of transitions is a crucial point. Since, we use binary terms, the number of transitions is  $O(s^2)$  where  $s$  is the number of states.

From the start, we have represented an automaton as a signed object, (an object with a signature), a list of references to its states, a list of references to its final states and its set of transitions.

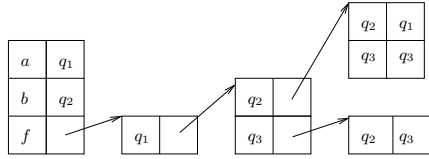
### 3.1 Representation of states

The principle that each state of an automaton is represented by a unique Common Lisp object has been in effect since the beginning of Autowrite. It is then very fast to compare objects: just compare the references. This is achieved using hash-consing techniques. On the contrary to systems like MONA [KM01], a state is not just represented by a number, it can also have constituting elements. The first reason for this choice is that each state has a meaning which can be better expressed by any Lisp object than by a simple number. The second reason is that states can themselves contain states from other automata when building an intersection automaton for example. The third reason will be made clear in Section 6 when we define the transition function as a function instead of defining it as a set of transitions.

Often we need to represent *sets* of states of an automaton. We have two ways of representing sets of states, *bit vectors* or *containers* of ordered states.

Bit vectors are faster, but tend to use more space; containers are slower but can be used when bit vectors lack of space.

Each state has an internal unique number which allows us to order states in the containers. Operations on containers (equality, union, intersection, addition of a state, ...) can then use algorithms on sorted lists which are faster.



**Figure 3:** Dag representation of the transitions

### 3.2 Transitions

The definition itself of an automaton suggests that the transition function should be represented by a set of transitions. And it is indeed the only solution that we had in mind when we started writing Autowrite. Whatever representation is chosen to store the transitions, it must offer a function  $\delta(f, states)$  which according to a symbol  $f$  of arity  $n$  and a list of states  $q_1, \dots, q_n$  returns the target state (or a set of target states in a non-deterministic case)  $q$  of the transition  $f(q_1, \dots, q_n) \rightarrow q$  stored in the data structure.

However, the transition function is really a function: if the states have a meaning as specified in Section 3.1, then in some cases,  $\delta(f, states)$  can be written as a function which computes the target state  $q$  according to  $f$  and the contents of the states  $q_1, \dots, q_n$  without the transition being stored in any data structure. We shall explain this novel implementation in Section 6.

The first representation chosen to represent a set of transitions is a hash-table: the key is the list  $(f q_1 \dots q_n)$  (where  $q_i$  is in fact the reference to the object representing the state  $q_i$ ) and the value is the target state  $q$  of the transition  $f(q_1, \dots, q_n)$ .

For instance, the following set of transitions:

$$\begin{array}{ll}
 a \rightarrow q_1 & f(q_1, q_2, q_3) \rightarrow q_1 \\
 b \rightarrow q_2 & f(q_1, q_3, q_2) \rightarrow q_3 \\
 & f(q_1, q_2, q_2) \rightarrow q_2
 \end{array}$$

yields a hash-table with 5 entries corresponding to the 5 left-hand-sides of the transitions. The advantage of this representation is that the left-hand-sides are kept together and that we can easily take into account commutative symbols. However, when the symbols have arity  $n \geq 2$  the table may become of size  $|Q|^n$ . In order, to reduce the size of the data structure representing the set of transitions, we have also considered a dag representation which is illustrated by Figure 3.

We now turn our attention to the problem of computing an automaton accepting the terms over  $\mathcal{F}_{\mathcal{L}}$  for fixed  $\mathcal{L}$  representing graphs verifying an MSO property.



#### 4 The general method (first method)

The first technique consists in applying the general algorithm which transforms a MSO formula into an automaton. The algorithm can be applied recursively until an atomic formula is reached. In order to process a MSO formula, we must translate it into a formula without first-order variables (which has the same quantifier-height) and which uses only boolean operations (and, or, negation) and simple atomic properties like  $X = \emptyset$ ,  $Sgl(X)$  (denoting that  $X$  is a singleton set),  $X_i \subseteq X_j$  for which an automaton is easily computable.

Some standardization on the names of set variables is then necessary in order to apply our operations.

The formula given in Example 6 is thus translated as shown below. Note that this translation is done by hand but could be automated as this is in MONA [KM01].

*Example 7.*

$$\begin{aligned}
Path(X_1, X_2) &= X_1 \subseteq X_2 \wedge P_1(X_1, X_2) \\
P_1(X_1, X_2) &= \exists X_3, X_4, P_2(X_1, X_2, X_3, X_4) \\
P_2(X_1, X_2, X_3, X_4) &= Sgl(X_3) \wedge Sgl(X_4) \wedge X_3 \subseteq X_1 \wedge X_4 \subseteq X_1 \wedge X_3 \neq X_4 \\
&\quad \wedge |X_1| = 2 \wedge P_4(X_2, X_3, X_4) \\
P_4(X_2, X_3, X_4) &= \neg P_5(X_2, X_3, X_4) \\
P_5(X_2, X_3, X_4) &= \exists X'_1, P_6(X'_1, X_2, X_3, X_4) \\
P_6(X'_1, X_2, X_3, X_4) &= X_3 \subseteq X_5 \wedge \neg X_4 \subseteq X_5 \wedge P_7(X'_1, X_2) \\
P_7(X'_1, X_2) &= \neg P_8(X'_1, X_2) \\
P_8(X'_1, X_2) &= \exists X_3, X_4, P_9(X'_1, X_2, X_3, X_4) \\
P_9(X'_1, X_2, X_3, X_4) &= Sgl(X_3) \wedge Sgl(X_4) \wedge X_3 \subseteq X'_1 \wedge X_3 \subseteq X_2 \wedge X_4 \subseteq X_2 \wedge \\
&\quad Edge(X_3, X_4) \wedge \neg X_4 \subset X'_1
\end{aligned}$$

##### 4.1 Basic automata for graph properties

We have implemented constructions parametrized by  $\mathcal{L}$  of the basic automata which may appear as atomic formulas in our MSO sentences (the leaves of our MSO formulas), among them:

setup-singleton-automaton (cwd m j)	$Sgl(X_j)$
setup-edge-automaton (cwd m i j)	$Edge(X_i, X_j)$
setup-subset-automaton (cwd m j1 j2)	$X_{j1} \subseteq X_{j2}$
setup-inequality-automaton (cwd m j1 j2)	$X_{j1} \neq X_{j2}$
setup-equality-automaton (cwd m j1 j2)	$X_{j1} = X_{j2}$
setup-inequality-automaton (cwd m j1 j2)	$Sgl(X_{j1}) \wedge Sgl(X_{j2}) \wedge X_{j1} \neq X_{j2}$
setup-cardinality-automaton (cwd m j1 i)	$card(X_{j1}) = i$

For example, a call to setup-singleton-automaton(2, 2, 1) returns an automaton working on terms representing graphs of clique-width at most 2 (with  $\mathcal{L} = \{a, b\}$ ) with two

```

NAUTOWRITE> (setf *a* (setup-singleton-automaton 2 2 1))
Singleton-X1 2 states 17 rules
NAUTOWRITE> (show *a*)
Automaton Singleton-X1
States q0 q1
Final States q1
Transitions
a^00 -> q0    b^00 -> q0    rel_a_b(q0) -> q0    rel_b_a(q0) -> q0
a^01 -> q0    b^01 -> q0    rel_a_b(q1) -> q1    rel_b_a(q1) -> q1
a^10 -> q1    b^10 -> q1    add_a_b(q0) -> q0    oplus(q0,q1) -> q1
a^11 -> q1    b^11 -> q1    add_a_b(q1) -> q1    oplus(q1,q0) -> q1
oplus(q0,q0) -> q0
NIL
NAUTOWRITE> (setf *t* (input-term "add_a_b(oplus(a^10,b^00))"))
add_a_b(oplus(a^10,b^00))
NAUTOWRITE> (recognized-p *t* *a*)
!q1
NAUTOWRITE> (recognized-p *t* *a*)
q1
NAUTOWRITE> (setf *nt* (input-term "add_a_b(oplus(a^10,b^10))"))
add_a_b(oplus(a^10,b^10))
NAUTOWRITE> (recognized-p *nt* *a*)
NIL

```

**Table 2:** Automaton for  $Sgl(X_1)$  with  $m = 2$  and  $cwd = 2$

sets of vertices  $X_1$  and  $X_2$  and recognizing terms such that  $X_1$  is a singleton, for instance the term  $add\_a\_b(oplus(a^10, b^00))$ . An example of such call is shown in Table 2.

## 4.2 The recursive algorithm

Given a formula  $\phi = P(X_1, \dots, X_m)$ , we want to compute the associated automaton  $\mathcal{A}(\phi)$ .

- If the formula is atomic then we call the function which computes the automaton. For instance, in  $P_9(X'_1, X_2, X_3, X_4)$ ,  $Sgl(X_3)$  is computed by

```
setup-singleton-automaton (cwd, 4, 4).
```
- If the formula is a disjunction  $\phi = \phi_1 \vee \phi_2$ , we compute the union of  $\mathcal{A}_{\phi_1}$  and  $\mathcal{A}_{\phi_2}$ .
- If the formula is a conjunction  $\phi = \phi_1 \wedge \phi_2$ , we compute the intersection of  $\mathcal{A}_{\phi_1}$  and  $\mathcal{A}_{\phi_2}$ .
- If the formula is a negation  $\phi = \neg(\phi')$ , we complement the automaton  $\mathcal{A}_{\phi'}$ . To be complemented  $\mathcal{A}_{\phi'}$  must be determinized.
- If the formula is an existential formula of the form  $\exists X_j, P(X_1, \dots, X_m)$ , we do a projection of  $\mathcal{A}_{P(X_1, \dots, X_m)}$  on  $(1, \dots, i-1, i+1, m)$  which implies a shift in the indices of variables  $X_{i+1}, \dots, X_m$ .
- If the formula  $\phi = P(X_1, \dots, X_m)$  does mention  $X_j$ , we can obtain  $\mathcal{A}_\phi$  by a cylindrification of the automaton  $\mathcal{A}_{P(X'_1, \dots, X'_{m-1})}$  (with  $X'_i = X_i$  for  $1 \leq i < j$  and  $X'_i = X_{i+1}$  for  $j \leq i < m$ ) on the  $j$ -th components.

Intersection which is handled by saturation (producing a reduced automaton) preserves determinism. The bottleneck of this general algorithm is the necessity of determinizing an automaton in order to complement it. Each determinization can increase exponentially the number of states.

Most properties that we tried could not be tested for graphs of clique-width strictly higher than 2 with this method. It is nevertheless interesting to implement it because it is completely general and for small clique-width we can use the computed automaton for a comparison with the automaton that we obtain using the second method that we are presenting now. The automaton can also be compared with the automaton computed by MONA (see Section 7).

## 5 The second method: direct construction of the final automaton

The last remark motivates the following development. For some graph properties expressible in MSO, the corresponding automaton can be described directly by a set of states and a description of the transition function on these states. Once a proof has been made that the description is correct (it produces an automaton which recognizes the terms satisfying the property), one can directly compute the automaton without using the MSO sentence. Chapter 6 of the book in progress [Cou09], gives such descriptions for several properties among them  $Path(X_1, X_2)$ . As said in the introduction, we shall stick to the path property although we can handle many others.

We shall not go into all the details of the construction of the automaton for  $Path(X_1, X_2)$ , but we shall present at least a description of its states and how the transitions function works.

$$\text{Let } \alpha(G, x) = \{label_G(y) \mid y \in \mathcal{V}_G \text{ and } x \overset{*}{-}_G y\} \subseteq \mathcal{L}.$$

$$\text{Let } \beta(G) = \{(label_G(x), label_G(y)) \mid x, y \in \mathcal{V}_G \text{ and } x \overset{*}{-}_G y\} \subseteq \mathcal{L} \times \mathcal{L}.$$

$$\begin{aligned} Q = & \{Ok, Error\} \cup \{(0, B) \mid B \subseteq \mathcal{L} \times \mathcal{L}\} \cup \\ & \{[1, A, B] \mid \emptyset \neq A \subseteq \mathcal{L}, B \subseteq \mathcal{L} \times \mathcal{L}\} \cup \\ & \{[2, \{A, A'\}, B] \mid A, A' \subseteq \mathcal{L}, A \neq \emptyset, A' \neq \emptyset, B \subseteq \mathcal{L} \times \mathcal{L}\} \end{aligned}$$

The meaning of these states is described in Table 3. We have  $2^{cwd^2/2} < |Q| < 2^{cwd^2+2}$  where  $cwd = |\mathcal{L}| \geq 2$ .

The transition rules are shown in Table 4. In this table, we use the auxiliary functions  $(\otimes, f, g)$  which can be found in [Cou09].

With the direct construction, we were first able compare the obtained automaton with the automaton obtained with the general method for  $cwd = 2$ . Then we solved the problem for  $cwd \in \{3, 4\}$ .

cwd	2	3	4	5
A/min(A)	25 / 12	214 / 127	3443 / 2197	out

State $q$	Property $P_q$
$[0, B]$	$X_1 = \emptyset, B = \beta(G(t, X_2)), X_1 = \{v\} \subseteq X_2, A = \alpha(G(t, X_2), v)$
$[1, A, B]$	$B = \beta(G(t, X_2)), X_1 = \{v, v'\} \subseteq X_2, v = v', A = \alpha(G(t, X_2), v),$
$[2, \{A, A'\}, B]$	$A = \alpha(G(t, X_2), d), B = \beta(G(t, X_2))$ there is no path between $v$ and $v'$ in $G(t, X_2)$
$Ok$	$P(X_1, X_2)$ holds
$Error$	All other cases

**Table 3:** Meaning of states for the path property  $Path(X_1, X_2)$

Transition rules	Conditions
$c^{\wedge}00 \rightarrow [0, \emptyset]$ $c^{\wedge}00 \rightarrow [0, \{(a, a)\}]$ $c^{\wedge}11 \rightarrow [1, \{a\}, \{(a, a)\}]$	$c \in \mathcal{L}$
$rel_{a,b}(Ok) \rightarrow Ok$ $rel_{a,b}([0, B]) \rightarrow [0, h_{a,b}(B)]$ $rel_{a,b}([1, A, B]) \rightarrow [1, h_{a,b}(A), h_{a,b}(B)]$ $rel_{a,b}([2, \{A, A'\}, B]) \rightarrow [2, \{h_{a,b}(A), h_{a,b}(A')\}, h_{a,b}(B)]$	where $h_{a,b}$ replaces $a$ by $b$
$add_{a,b}(Ok) \rightarrow Ok$ $add_{a,b}([0, B]) \rightarrow [0, B']$ $add_{a,b}([1, A, B]) \rightarrow [1, D, B']$ $add_{a,b}([2, \{A, A'\}, B]) \rightarrow [2, \{D, D'\}, B']$	$B' = f(B, a, b)$ $D = g(A, B, a, b)$ $D' = g(A', B, a, b)$ $(A \odot ((a \otimes b) \circ B)) \cap A' = \emptyset$
$add_{a,b}([2, \{A, A'\}, B]) \rightarrow Ok$	$(A \odot ((a \otimes b) \circ B)) \cap A' \neq \emptyset$
$\oplus(Ok, [0, B]) \rightarrow Ok$ $\oplus([0, B], Ok) \rightarrow Ok$ $\oplus([0, B], [0, B']) \rightarrow [0, B'']$ $\oplus([0, B], [1, A, B']) \rightarrow [1, A, B'']$ $\oplus([1, A, B], [0, B']) \rightarrow [1, A, B'']$ $\oplus([1, A, B], [1, A', B']) \rightarrow [2, \{A, A'\}, B'']$ $\oplus([0, B], [2, \{A, A'\}, B']) \rightarrow [2, \{A, A'\}, B'']$ $\oplus([2, \{A, A'\}, B'], [0, B]) \rightarrow [2, \{A, A'\}, B'']$	$B'' = B \cup B'$

**Table 4:** Transition rules of the automaton for  $Path(X_1, X_2)$

However, with higher values of clique-width ( $cwd \geq 5$ ), we are confronted to a memory space problem. And indeed the number of states is at least  $2^{5^2/2} = 2^{12} \leq |Q|$  which gives at least  $2^{25}$  transitions (see [Cou09], Chapter 6).

We have presented experiments only with the path property. But we have tried several other properties <sup>6</sup> like connectivity, existence of a cycle,  $k$ -colorability, ... Most of the time, the limit is around  $cwd = 3$ . The conclusion is that for greater values of clique-width, it is not possible to compute in extenso the transitions of the automata because its number of states is simply too big (exponential in  $cwd$  or more). In a few cases, we do not run out of memory but the program runs “for ever” (3-colorability with  $cwd = 3$ ).

## 6 The third method: fly-automata

The problems of space (for most properties) or time (coloring property) disappear if we represent transitions with a function. Defining such transitions (which we call *fly-transitions*) consists in defining a lisp function which applies to a symbol  $f$  and a list of states  $(q_1, \dots, q_n)$  and returns the target state  $q$  of the transitions  $f(q_1, \dots, q_n) \rightarrow q$ .

This is easily done from the description of the direct construction of the automaton as the one given in Section 5. Actually, the code that is written to define a concrete transition can be directly called in the fly-transitions function.

States that will be accessed when running the automaton on a particular term are initially not known. In most cases, we do not even want to compute the list of accessible states of the automaton because, this list is simply too big to be computed. The states are formally described in a compact way; the ones that are useless will never be computed. The situation is the same for the list of final states. The easiest way to represent final states is also to use a predicate which tells whether a state is final or not.

So a fly-automaton is just a signed object which has a transition function and a final state predicate. Of course Common Lisp is very suitable to represent objects containing functions since functions are first-class objects. Defining a fly-automaton reduces to defining the transition function and final state predicate.

```
(defun fly-path-automaton (cwd)
  (make-fly-automaton-automaton
   (setup-vbits-signature cwd 2)
   (lambda (root states)
     (make-state
      (path-transitions-fun root (mapcar #'state-contents states))))
   (lambda (state)
     (and (ok-p (state-contents state)) state))
   :name (format nil "~A-PATH-X1-X2-fly-automaton" cwd)))
```

The transition function of union and intersection automata is an anonymous function which calls the respective functions of the combined automata. Note that a concrete automaton can be transformed into a fly automaton: the transition function simply looks

<sup>6</sup> See some results at <http://dept-info.labri.fr/~idurand/autowrite/Graphs/Graphs.pdf>

for the transition in the stored transitions. But the converse may fail for space and time reasons. We did not reach any limitation using fly-automata which we tried up to  $cwd = 18$ . We could run the automata on terms representing terms on any graph we had a term representation for. Our problem right now is to find big graphs with their clique-decomposition in order to perform tests.

In this paper we did not address the difficult problem of finding a clique-width decomposition of a graph (so the clique-width) of a graph.

This problem was shown to be NP-complete in [FRRS06]. [Oum08] gives polynomial approximated solutions to solve this problem. More can be found in [Cou09].

Often, when automata are used (in compilation for instance), the automaton is “small” and the input is much much larger. In the present case, it is the opposite. In particular, because we do not know how to decompose very large graphs, we are only in position of using our tools for relatively small graphs (say 100 vertices). Consequently, there is no overhead in using fly-automata. Also, it is not important that the terms representing graphs be optimal because the computation “on the fly” of transitions does not depend much on the total number ( $|\mathcal{L}|$ ) of vertex labels.

## 7 Related work

Monadic second-order logic on finite and infinite words and binary terms is implemented in the software MONA [KM01] developed by Klarlund and others. Its use for checking graph properties is considered by Soguet in [Sog08]. MONA, with some technical adaptations, is usable for the first technique: it is able to automatically compute the automaton corresponding to an MSO formula; in that it seems quicker than Autowrite. States are represented by an integer. MONA works with binary terms only which is ok for graphs represented with a signature with a maximum arity of 2 ( $\oplus$ ). The symbols with higher arity are simply transformed into binary symbols which have fake children when used in terms. The transitions are represented by a two dimensional array. The cell  $(i, j)$  contains a binary decision diagram (BDD) which leads for every symbol  $f^w$  to the target state  $k$  such that  $f^w(i, j) \rightarrow k$ . MONA has deterministic transitions only. When a projection is performed, the determinization is done at the same time. Autowrite can deal with symbols of any fixed arity. An important point is that Autowrite has both deterministic and non deterministic automata. This is very useful when the deterministic automaton corresponding to the desired property cannot be computed by lack of space. In that case, Autowrite will be able to check the property with the non deterministic automaton. See also [Cou09] about this last point.

## 8 Perspectives

We have still many more properties of graph to experiment among them connectivity. For the automata for which we could compute the set of transitions, it would be nice

to create an on-line library of automata corresponding to properties available to the community of researchers. There is still a lot to be done for improving the efficiency of Autowrite. We have maintained several data structures for representing the automata transitions but have not yet conducted systematic tests to evaluate their performances. In order to do more experiments with our fly-automata, we are currently working on a program for generating automatically random or particular graphs (with their decompositions) of arbitrary clique-width.

## References

- [CD10] Bruno Courcelle and Irène Durand. Tractable constructions of finite automata from monadic second-order formula. In *Workshop on Logical Approaches to Barriers in Computing and Complexity*, Greifswald, Germany, February 2010.
- [CDG<sup>+</sup>02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2002. Draft, available from <http://tata.gforge.inria.fr>.
- [CMR01] B. Courcelle, J. A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Appl. Math.*, 108(1):23–52, 2001.
- [Cou09] Bruno Courcelle. Graph structure and monadic second-order logic. Available at <http://www.labri.fr/perso/courcell/Book/CourGGBook.pdf>. To be published by Cambridge University Press, 2009.
- [DF99] Rod G. Downey and Michael R. Fellows, editors. *Parameterized Complexity*. Springer-verlag, 1999.
- [Dur02] Irène Durand. Autowrite: A tool for checking properties of term rewriting systems. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 371–375, Copenhagen, 2002. Springer-Verlag.
- [Dur05] Irène Durand. Autowrite: A tool for term rewrite systems and tree automata. *Electronics Notes in Theoretical Computer Science*, 124:29–49, 2005.
- [FG04] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of Pure and Applied Logic*, 130:3–31, 2004.
- [FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- [FRRS06] M. Fellows, F. Rosamond, U. Rotics, and S. Szeider. Clique-width minimization is np-hard. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, pages 354–362, Seattle, 2006.
- [KL09] Joachim Kneis and Alexander Langer. A practical approach to Courcelle’s theorem. *Electron. Notes Theor. Comput. Sci.*, 251:65–81, 2009.
- [KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from <http://www.brics.dk/mona/>. Revision of BRICS NS-98-3.
- [Oum08] Sang-Il Oum. Approximating rank-width and clique-width quickly. *ACM Trans. Algorithms*, 5(1):1–20, 2008.
- [Sog08] David Soguet. Génération automatique d’algorithmes linéaires. Doctoral dissertation (in French), Spécialit: Informatique, July 2008.
- [TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.