Infinite transducers on terms denoting graphs

Bruno Courcelle
courcell@labri.fr

Irène A. Durand
idurand@labri.fr

LaBRI, CNRS, Université de Bordeaux, Talence, France

ABSTRACT

Starting from Courcelle's theorem which connects the problem of verifying properties of graphs of bounded clique-width with term automata, we have developed the Autograph Lisp library[1] which provides automata for verifying graph properties [2]. Because most of these automata are huge, fly automata have been introduced in the underlying library Autowrite and have become the default type of automata [11]. By default, the operations on automata are now performed on fly automata.

This article shows how fly automata can be generalized to *attributed fly automata* and finally to *transducers*.

We apply these concepts in the domain of graph theory. We present some computations performed by transducers on terms representing graphs. This transducer approach for computing graph values is an alternative to the classical algorithms of graph theory.

Categories and Subject Descriptors

D.1.1 [**Software**]: Programming Techniques, Applicative (Functional) Programming; F.1.1 [**Theory of Computation**]: Models of Computation, Automata; G.2.2 [**Mathematics of Computing**]: Graphs Theory, Graph Algorithms

Keywords

Term automata, Term transducers, Lisp, graphs

1. INTRODUCTION

At ELS2010 [2], we showed that bottom-up term automata could be used to verify monadic second order properties on graphs of bounded clique-width which can be represented by terms. Although finite, these automata are often so large that their transitions table cannot be built.

To solve this problem, we have introduced automata called *fly automata* which were first presented at ELS2011 [11]. In such automata, instead of the transition function being represented as a set of values, it is represented as the code of a computable function. In this setting, the states of an automaton need not be listed; a finite subset of the whole set of states is produced *on the fly* by the transition function during the run of the automaton on a term.

Fly automata solve the problem of representing huge finite automata but also yield new perspectives as they need not be finite; they may be infinite in two ways: they may have an infinite denumerable signature and an infinite denumerable set of states.

Fly automata are nicely implemented in Lisp, the core of the automaton being its transition function. Also Lisp conditions are used to detect early failure.

In the graph framework, we may obtain fly automata working for any clique-width with a signature containing an infinite number of constant symbols. Also we may use natural integers as states to count for instance the number of vertices of a graph.

Usually, a term automaton has a set of transitions and a set of states, a subset of which are the final states. A fly automaton has a transition function and a final state predicate which says whether a state is final or not. When running an automaton on a term, one gets a *target* which may be just one state if the automaton is deterministic or a set of states otherwise; the target is final if it is a final state (deterministic case) or contains a final state (non deterministic case). The term is recognized when the target is final.

As we have no limit on the number of states, we may extend our fly automata to compute *attributed states* which are states associated with an attribute which is computed along the run.

Attributes may be of many different types: they are often integers, sets of positions, symbols, terms and tuples, sets, and multisets of the previous types.

In the framework of graphs, attributes may be color assignments (colorings) or subset assignments, number of colorings, number of subset assignment, etc. An *attributed automaton* is just an automaton whose transition function is completed with an *attribute function* which synthetizes the attribute of the states. The attribute function computes the new attribute from the attribute of the arguments and the function symbol. When the automaton is non deterministic, the same state may be obtained by several computations but with different attributes; in that case, we must provide a function to *combine* the attributes for this state. We shall refer to this function as the `combine-fun`.

[1] itself based on the Autowrite Lisp library

In this paper, we extend the concept of fly automaton to *attributed fly automaton* and finally to *fly transducer*. A fly transducer has an output function instead of a final state predicate. As for an automaton, the run of a transducer on a term produces a target. The output function is applied to the final target for the final result. In our case, a transducer will be an attributed automaton with an output function that will be applied to the attribute of the final target.

A fly automaton may be seen as a particular case of a transducer where the output function is the final state predicate.

The `Autowrite` software [13] entirely written in Common Lisp was first designed to check call-by-need properties of term rewriting systems [9]. For this purpose, it implements classical finite term (tree) automata. In the first implementation, just the emptiness problem (does the automaton recognize the empty language) was used and implemented.

In subsequent versions [10], the implementation was continued in order to provide a substantial library of operations on term automata. The next natural step was to try to solve concrete problems using this library and to test its limits.

Starting from Courcelle's theorem [7] which connects the problem of verifying properties of graphs of bounded clique-width with term automata, we have developed the `Autograph` library [14] (based on `Autowrite`) which provides automata for verifying graph properties [2].

Because most of these automata are huge, we introduced fly automata into `Autowrite` and made them the default type of automata [11]. By default, the operations on automata are performed on the fly automata. The traditional table-automata are just compiled versions of finite fly automata.

The purpose of this article is:

- to show how fly automata can be generalized to attributed fly automata and finally to transducers,
- to describe part of the implementation,
- and to present some computations performed by such transducers on terms and terms representing graphs.

This transducer approach for computing graph values is an alternative to the classical algorithms of graph theory.

One advantage of the use of automata or transducers is that, using inverse-homomorphisms, we can easily get algorithms working on induced subgraphs from the ones working on the whole graph which is most often not feasible with classical algorithms of graph theory.

Some graph coloring problems will be used as examples throughout the paper.

2. PRELIMINARIES

We recall some basic definitions concerning terms and how terms may be used to represent graphs of bounded clique-width.

2.1 Signature and terms

We consider a signature $\mathcal{F}$ (set of symbols with fixed arity).

EXAMPLE 2.1. *Let $\mathcal{F}$ be a signature containing the symbols* {`a, b, add_a_b, ren_a_b, ren_b_a, oplus`} *with*

$$\mathsf{arity}(\texttt{a}) = \mathsf{arity}(\texttt{b}) = 0 \quad \mathsf{arity}(\texttt{oplus}) = 2$$
$$\mathsf{arity}(\texttt{add\_a\_b}) = \mathsf{arity}(\texttt{ren\_a\_b}) = \mathsf{arity}(\texttt{ren\_b\_a}) = 1$$

In Section 3, we show that this signature is suitable for writing terms representing graphs of clique-width at most 2*.*

We denote the subset of symbols of $\mathcal{F}$ with arity n by $\mathcal{F}_n$. So $\mathcal{F} = \bigcup_n \mathcal{F}_n$. By $\mathcal{T}(\mathcal{F})$, we denote the set of (ground) terms built upon the signature $\mathcal{F}$.

EXAMPLE 2.2. *t_1, t_2, t_3 and t_4 are terms built with the signature $\mathcal{F}$ of Example 2.1.*

```
t1 = oplus(a,b)
t2 = add_a_b(oplus(a,oplus(a,b)))
t3 = add_a_b(
       oplus(a,oplus(a,oplus(b,b))))
t4 = add_a_b(
       oplus(a,
         ren_a_b(add_a_b(oplus(a,b)))))
```

In Table 1, we see their associated graphs. The connection between terms and graphs will be described in Section 3.2.

3. APPLICATION DOMAIN

Part of this work will be illustrated in the framework of graphs of bounded clique-width. In this section, we present the connection between graphs and terms. First we define the graphs.

3.1 Graphs as a logical structure

We consider finite, simple, loop-free undirected graphs (extensions are easy)[2]. 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. Monadic Second order Logic is suitable for expressing many graph properties like k-colorability, acyclicity (no cycle), k-acyclic-colorability, ...

3.2 Term representation of graphs of bounded clique-width

DEFINITION 1. *Let $\mathcal{L}$ be a finite set of vertex labels also called* ports *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`*: the union of disjoint graphs,*

for every pair of distinct vertex labels $(a, b) \in \mathcal{L} \times \mathcal{L}$:

[2]We consider such graphs for simplicity of the presentation but we can also work with directed graphs, loops, labeled vertices and edges. A loop is an edge connecting one single vertex.

- *unary edge addition operations* `add_a_b`[3] *that add the missing edges between every vertex labeled* a *to every vertex labeled* b*,*
- *unary relabeling operations* `ren_a_b` *that rename* a *to* b*, and*

for every vertex label $a \in \mathcal{L}$*,*

- *constants* `a` *such that the 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 constant symbols.*

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 and that several terms may represent the same graph up to isomorphism.*

A graph has clique-width *at most* k *if it is defined by some* $t \in \mathcal{T}(\mathcal{F}_\mathcal{L})$ *with* $|\mathcal{L}| \leq k$*. The clique-width of a graph is the minimal such* k*. We shall abbreviate clique-width by* cwd*.*

Examples of terms with their associated graph are given in Table 1.

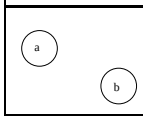
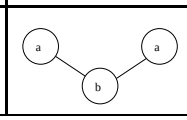
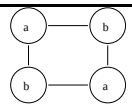
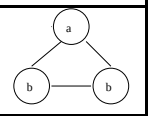
t_1	t_2	t_3	t_4
			

Table 1: The graphs corresponding to the terms of Example 2.2

3.3 Clique-width of some well-known graphs

The problem of finding a *decomposition* of a graph (*i.e.* a term representing the graph) with a minimal number of labels (so the clique-width) is NP-complete [15].

However, an approximation can always be found, the worst approximation being the one using as many labels as vertices in the graph and as many `add_a_b` operations as edges in the graph.

For instance, the graph corresponding to t_3 of Table 1 can be decomposed as

```
add_a_d(
 add_c_d(
  add_b_c(
   add_a_b(
    oplus(a,oplus(b,oplus(c,d))))))
```

which uses 4 ports labels.

However, the clique-width parameter being crucial in our algorithm, it is important to minimize the number of port labels. So term t_3 which uses 2 labels would be preferable to the term above.

For some classical family of graphs, the clique-width is known. For instance, cliques K_n have clique-width 2, for all $n > 1$. A term k_n representing K_n is recursively given by:

$$\begin{cases} k_1 = \texttt{a} \\ k_n = \texttt{ren\_b\_a(add\_a\_b(oplus(}k_{n-1}\texttt{,b)))} \end{cases}$$

P_n graphs (chains of n nodes) have clique-width 3 for $n > 3$. A term p_n representing a graph P_n is recursively given by:

$$\begin{cases} p_1 = \texttt{b} \\ p_n = \texttt{ren\_c\_b(ren\_b\_a(add\_b\_c(oplus(}p_{n-1}\texttt{,c)))} \end{cases}$$

Rectangular grids $n \times m$ with $m < n$ have clique-width $m + 2$. Square grids $n \times n$ have clique-width $n + 1$; the latest decomposition is a bit tricky [17].

3.4 Representation of colored graphs

To deal with colored graphs, we use a modified constant signature. If we are dealing with k colors then every constant `c` yields k constants `c~1`, ..., `c~k`. In a term, the constant `c~i` means that the corresponding vertex is colored with color i.

EXAMPLE 3.1. *For instance, the term* `add_a_b(oplus(a~1,oplus(b~2,oplus(a~1,b~2))))` *represents a proper* 2*-colored version of term* t_3 *of Example 2.2.*

3.5 Representation of sets of vertices

To deal with graphs with identified subsets of vertices, we also use a modified constant signature. If we are dealing with m subsets $V_1, \ldots, V_m$ then every constant `c` yields 2^m constants of the form `c^w` where w is a bit vector $b_1 \ldots b_m$ such that $b_i = 1$ if the corresponding vertex belongs to V_i, $b_i = 0$ otherwise.

4. TERM AUTOMATA

4.1 Finite term automata

We recall the definition of finite term automaton. Much more information can be found in the on-line book [1].

DEFINITION 2. *A (finite bottom-up)* term automaton[4] *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* Δ*. Every transition is of the form* $f(q_1, \ldots, q_n) \to q$ *with* $f \in \mathcal{F}$*,* $\mathsf{arity}(f) = n$ *and* $q_1, \ldots, q_n, q \in Q$*.*

Term automata recognize *regular* term languages[20]. The class of regular term languages is closed under the Boolean operations (union, intersection, complementation) on languages which have their counterpart on automata.

EXAMPLE 4.1. *A graph is* stable *if it has no edges. The automaton* `2-STABLE` *of Figure 1 recognizes stable graphs of clique-width* 2*. The states* `<a>`, `<b>`, `<ab>` *mean that the graph contains no edge and respectively at least a vertex labeled* `a`*, at least a vertex labeled* `b`*, at least a vertex labeled* `a` *and a vertex labeled* `b`*; they are all final states. The state* `error` *is the only non final*

[3] for the oriented case both `add_a_b` and `add_b_a` are used; for the unoriented case, we may assume a total order on the port labels and use the `add_a_b` such that `a` < `b`.

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

```
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
Final 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 q
```

Figure 1: Automaton recognizing stable graphs

state; it means that an edge has been found. The rule which triggers the first `error` *state is the* `add_a_b(<ab>) -> error` *rule; such operation adds an edge between vertices labeled* `a` *and vertices labeled* `b`*. We shall see later that this automaton is in fact the compiled version of a finite fly automaton.*

From this automaton, we can derive another automaton for deciding whether a subgraph induced by a subset of vertices V_1 is stable. We have seen that membership of a vertex to a subset of vertices V_1 is expressed with a bit added to the constants. `a^1` *represents a vertex labeled* `a` *belonging to V_1 while* `a^0` *represents a vertex labeled* `a` *not belonging to V_1. The term* `add_a_b(oplus(a^1,oplus(b^0,oplus(a^1,b^0))))` *represents the same graph as t_3 in Example 2.2 but with the two vertices labeled* `a` *in V_1.*

To the previous automaton, we add the symbol `@` *for representing an empty graph (so a stable graph), the state* `#f` *for representing a neutral final state (which will be used as long as no vertex in V_1 has been found) and the rules*

```
@ -> #f
zzz_x_y(#f) -> #f
oplus(q,#f) -> q
```

for every `zzz` $\in \{$`add`, `ren`$\}$*, every* `x`,`y` $\in \{$`a`, `b`$\}$ *such that* `x` $\neq$ `y` *and every* `q` *state of* `2-STABLE` *and consider the homomorphism h such that*

$$h(\texttt{a\^{}1})=\texttt{a}$$
$$h(\texttt{a\^{}0})=\texttt{@}$$
$$h(\texttt{b\^{}1})=\texttt{b}$$
$$h(\texttt{b\^{}0})=\texttt{@}$$

and $h(f) = f$ for every non constant symbol.

Applying h^{-1} to the automaton `2-STABLE` *yields an automaton which recognizes graphs such that V_1 is stable. Only the constant rules differ*

```
a^1 -> <a>
b^1 -> <b>
a^0 -> #f
b^0 -> #f
```

To distinguish these finite automata from the fly automata defined in Subsection 4.2 and as we only deal with terms in this paper we shall refer to the term automata defined in Definition 2 as *table-automata*.

4.2 Fly term automata

DEFINITION 3. *A* fly term automaton *(*fly automaton *for short) is a triple $\mathcal{A} = (\mathcal{F}, \delta, \mathsf{fs})$ where*

- *$\mathcal{F}$ is a countable signature of symbols with a fixed arity,*
- *δ is a computable transition function,*

$$\begin{array}{rccl} \delta: & \bigcup_n \mathcal{F}_n \times Q^n & \rightarrow & Q \\ & f q_1 \ldots q_n & \mapsto & q \end{array}$$

  where Q is a countable set of states, disjoint from $\mathcal{F}$,
- fs *is the final state predicate*

$$\mathsf{fs}: \quad Q \quad \rightarrow \quad Boolean$$

  which indicates whether a state is final or not.

Note that, both the signature $\mathcal{F}$ and the set of states Q may be infinite. A fly automaton is *finite* if both its signature and its set of states are finite.

Operations on term languages like Boolean operations, homomorphisms and inverse-homomorphisms have their counterpart on fly automata [3, 4]. For instance, the union of two fly automata recognizes the union of the two languages recognized by the automata.

We use the term *basic* for fly automata 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 the above theorem. We call the latter *composed* fly automata.

The *run* of an automaton on a term labels the nodes of the term with the state(s) reached at the corresponding subterm. The run goes from bottom to top starting at the leaves.

In `Autowrite`, this is implemented via the
`compute-target(term automaton)`
operation which, given a term and an automaton, returns the target (a single state if the automaton is deterministic or a container of states otherwise).

A term is *recognized* by the automaton when after the run of the automaton on the term, a final state is obtained at the root.

```
(defmethod stable-transitions-fun
    ((root constant-symbol) (arg (eql nil)))
  (let ((port (port-of root)))
    (when (or (not *ports*)
      (member port *ports*))
     port
     (make-stable-state
      (make-ports-from-port port)))))

(defmethod stable-transitions-fun
    ((root abstract-symbol) (arg list))
  (common-transitions-fun root arg))
```

Figure 2: Transition function for constants

In `Autowrite`, this is implemented by the `recognized-p(term automaton)` operation which returns true if at least one state in the target is final according to the final state predicate of the automaton.

In fact, we have an intermediate operation `compute-final-target(term automaton)` which returns the *final target* that is the target without non final states. Then the `recognized-p(term automaton)` operation is implemented by checking whether the final target is empty.

4.3 Examples with the stability property

We can create an infinite fly automaton that verifies that a graph is stable for *any* clique-width.

We create it as a basic automaton (in the sense given in Section 4.2). This means that we must define the structure of the states for this automaton and the transition function that computes the states. This automaton is deterministic.

Its states are of uniform type; the state computed at the root of a term represents the set of port labels encountered so far.

```
(defclass stable-state (graph-state)
  ((ports :type ports
          :initarg :ports
          :reader ports)))
```

The transition function is `stable-transitions-fun`.

For a constant symbol a, a `stable-state` is created with `ports` being the singleton $\{a\}$. This is shown in Figure 2.

For the non constant symbols, the transition function calls `common-transitions-fun` which switches to a call to `graph-oplus-target`, `graph-ren-target` or `graph-add-target` according to the symbol (see Figure 3).

If we fixed the clique-width cwd, then we could compile this fly automaton to a minimal table-automaton with 2^{cwd} states. The automaton `2-STABLE` of Figure 1 is in fact the compiled version of the fly version with $cwd = 2$.

Figure 4 shows that the graph corresponding to the term t_3 is not stable. Figure 5 that the subgraph induced by the vertices with port `a` is stable. The automaton used is obtained by inverse homomor-

```
(defclass stable-state (state)
  ((ports :type port-state :initarg :ports :reader

(defmethod make-stable-state ((ports port-state))
  (make-instance 'stable-state :ports ports))

(defmethod state-final-p ((so stable-state)) t)

(defun fly-stable-automaton (&optional (cwd 0))
  (make-fly-automaton
   (setup-signature cwd)
   (lambda (root states)
     (let ((*ports* (iota cwd)))
       (stable-transitions-fun  root states)))))

(defmethod stable-transitions-fun ;; a -> <a>
    ((root constant-symbol) (arg (eql nil)))
  (when (or (not *ports*) (member port *ports*))
    (make-stable-state (make-port-state (port root)
(defmethod graph-ren-target
    (a b (so stable-state))
  (make-stable-state
   (ports-subst
    b a
    (ports so))))

(defmethod graph-add-target
    (a b (so stable-state))
  (let ((ports (ports so)))
    (unless (and
     (ports-member a ports)
     (ports-member b ports))
      so)))

(defmethod graph-oplus-target
    ((s1 stable-state) (s2 stable-state))
  (make-stable-state
   (ports-union (ports s1) (ports s2))))
```

Figure 3: Transition function for stability

```
AUTOGRAPH> (recognized-p
    *t3*
    (stable-automaton))
NIL
NIL
AUTOGRAPH> *t3*
add_a_b(oplus(a,oplus(a,oplus(b,b))))
AUTOGRAPH> (recognized-p
    *t3*
    (stable-automaton))
NIL
NIL
```

Figure 4: Examples with stability

```
AUTOGRAPH> *s3*
add_a_b(oplus(a^1,oplus(a^1,oplus(b^0,b^0))))
AUTOGRAPH> (recognized-p
    *s3*
    (nothing-to-x1
     (stable-automaton)))
T
!<{a}>
```

Figure 5: Stability of induced subgraph

phism as described in Example 4.1.

4.4 Examples with graph-colorings

In graph theory, a graph is *k-colored* if its vertices are colored with k colors. The coloring is *proper* if two adjacent vertices do not have the same color. The graph represented by the term given in Example 3.1 has a proper 2-coloring.

4.4.1 Graph coloring verification

For a fixed number of colors k, we can create an infinite fly automaton which verifies that a graph has a proper k-coloring for any clique-width.

The constants have colors which are positive integers in $[1, k]$; the constant `a~i` means that this vertex has color `i` for $i \in [1, k]$.

We create it as a basic automaton (in the sense given in Section 4.2). This means that we must define the structure of the states for this automaton and the transition function that computes the states. This automaton is deterministic.

Its states are of uniform type; the state computed at the root of a term represents a function which, given a constant name `c`, gives the set of color numbers appearing on leaves `c~i` in the term.

```
(defclass colors-state (graph-state)
  ((color-fun :initarg :color-fun
              :reader color-fun)))
```

For instance, the `color-fun` of the state reached at the root of term `oplus(a~1,oplus(b~1,a~2))` should return `{1,2}` when applied to `a`, `{1}` when applied to `b`, and the empty set when applied to any other constant label.

The transition function of the automaton is described a little further.

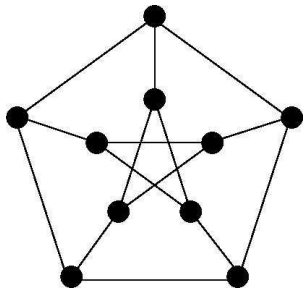


Figure 6: Petersen's Graph

If we fixed the clique-width cwd, then we could compile this fly automaton to a table-automaton with $2^{2cwd} - 1$ states. That would give $2^{2\times 6} - 1$ states in order to get an automaton able to work on Petersen's graph (see Figure 6) for which our best decomposition has $cwd = 6$. The term representing Petersen's graph has 37 nodes and depth 28.

We can use that automaton, to verify that some graphs are properly k-colored. We shall see later that by doing a color-projection (erasing the colors) of this automaton, we obtain a non deterministic automaton which recognizes graphs that are k-colorable. The two rules `a~1 -> q1`, `a~2 -> q2` would become the non deterministic rule `a -> o{q1,q2}` via the color-projection.

In Figure 7, we show the function `colored-automaton` which returns such an infinite automaton. When the optional `cwd` (clique-width) parameter is omitted, the resulting automaton has an infinite signature and an infinite number of states. The crucial part to implement is the operation `colored-transition-fun` which corresponds to the transition function of the automaton. When the optional parameter cwd is zero, then the automaton has an infinite signature and works on terms of any clique-width. Otherwise ($cwd > 0$), there is a finite number of port labels so that the signature has a finite number of constants and the automaton should not recognize constants with a port $>= cwd$. The `*ports*` special variable records the list of autorized labels when finite and is `NIL` otherwise; it is used when the transition function is applied to a constant.

```
(defun colored-automaton
    (k &optional (cwd 0))
  (make-fly-automaton
   (setup-color-signature cwd k)
   (lambda (root states)
     (let ((*ports* (port-iota cwd))
           (*colors* (color-iota k)))
       (colored-transitions-fun
        root states)))
   :name (format
          nil
          "~A-COLORED-~A" cwd k)))
```

Figure 7: Automaton for verifying the coloring of a graph

For a colored constant `c~i`, the transition function returns a `colors-state` whose `color-fun` gives the singleton $\{$`i`$\}$ for `c` and the empty set for every other constant.

```
(defmethod colored-transitions-fun
    ((root color-constant-symbol)
     (arg (eql nil)))
  (let ((port (port-of root)))
    (when (or (endp *ports*)
              (member port *ports*))
      (let* ((color (symbol-color root))
             (color-fun
               (add-color-to-port
                 color
                 port
                 (make-empty-color-fun))))
        (make-colors-state color-fun)))))
```

For the non constant symbols, as for the stable case, the transition function calls the method `graph-oplus-target`, `graph-ren-target` or `graph-add-target` according to the symbol.

For the disjoint union operation `oplus`, the `color-fun` of the new state returns the union of the `color-fun` of the children. There may be no failure. The function `graph-oplus-target` implements this operation.

```
(defmethod graph-oplus-target
    ((s1 colors-state) (s2 colors-state))
  (make-colors-state
   (merge-color-fun
    (color-fun s1)
    (color-fun s2))))
```

The only operations which may lead to a failure are the `add_a_b` operations, because they may connect two vertices which have the same color; in that case, the transition function returns `NIL`. This failure should be transmitted directly to the root of the term via `Lisp` conditions.
The function `graph-add-target` implements this operation.

```
(defmethod graph-add-target
    (a b (colors-state colors-state))
  (let ((color-fun
         (color-fun colors-state)))
    (unless (intersection
             (get-colors a color-fun)
             (get-colors b color-fun))
      colors-state)))
```

In Figure 8, we call the function to obtain an infinite automaton that verifies whether a graph of any clique-width has a proper 2-coloring.

```
AUTOGRAPH> (setf *2-colored*
             (colored-automaton 2))
0-COLORED-2 ;; deterministic
AUTOGRAPH> (compute-target
             (input-term "a~1")
             *2-colored*)
<a:1> ;; one state
```

Figure 8: Automaton for coloring verification

Finally, Figure 10 shows the use of this automaton on a 2-colored graph. We use term t_3 of Table1 (which corresponds to a cycle of size 4) with two different colorings:
one is proper (`*t3_1*` see Figure 9), the other one is not (`*t3_2*`).

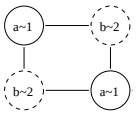


Figure 9: A proper 2-coloring of t_3

```
AUTOGRAPH> *t3_1*
add_a_b(
 oplus(a~1,oplus(a~1,oplus(b~2,b~2))))
AUTOGRAPH> (recognized-p
             *t3_1*
             *2-colored*)
T
<a:1 b:2>
AUTOGRAPH> *t3_2*
add_a_b(
 oplus(a~1,oplus(a~2,oplus(b~2,b~1))))
AUTOGRAPH> (recognized-p
             *t3_2*
             *2-colored*)
NIL
NIL
```

Figure 10: Verification of the coloring of a graph

4.4.2 Graph k-colorability

To obtain an automaton for deciding whether an uncolored graph is k-colorable, one must apply a projection (inverse homomorphism) which removes the colors from the constants to the previous automaton. The result is a non deterministic automaton.

```
AUTOGRAPH> (setf
             *2-colorability*
             (color-projection-automaton
              *2-colored* 2))
fly-asm(0-COLORED-2) ;; non deterministic
```

Now, we run the automaton on some terms.

```
AUTOGRAPH> (compute-target
             (input-term "a")
             *2-colorability*)
o{<a:1> <a:2>} ;; 2states
```

We verify that a clique of size 3 is not 2-colorable:

```
AUTOGRAPH> (recognized-p
             (graph-kn 3) ;; clique of size 3
             *2-colorability*)
NIL
NIL
```

but that the graph corresponding to t_3 is 2-colorable:

```
AUTOGRAPH> *t3*
add_a_b(oplus(a,oplus(a,oplus(b,b))))
AUTOGRAPH> (recognized-p
            *t3* *2-colorability*)
T
o{<a:1 b:2> <a:2 b:1>} ;; 2 states
```

It is nice to know that a graph is k-colorable but it would be even nicer to effectively find a proper coloring (or all proper colorings) of a graph. A simple fly automaton is not enough for that, as it just gives a boolean answer. In the next section, we shall show how a fly automaton may be enhanced in order to compute more interesting answers than boolean values. In particular, we shall be able to compute or enumerate the proper colorings of a graph.

5. FLY TRANSDUCERS

Because, the number of states of a fly automaton may be infinite, we may associate *attributes* to the states of the fly automata in order to compute more complicated information than just states.

5.1 Attributed fly automata

An *attributed* fly automaton $\mathcal{B}$ is a fly automaton which is based on another automaton $\mathcal{A}$.

The transition function of $\mathcal{B}$ is the one of $\mathcal{A}$ enhanced in order to compute an attribute associated with each state. So the automaton computes attributed states instead of states. Attributed states are a particular kind of state. They are states that contain states. In `Autowrite` we already had states that contain a state; for instance *indexed-states* for computing disjoint unions of automata. The `in-state` class captures that behaviour.

```
(defclass in-state-mixin ()
  ((in-state :initform nil
     :initarg :in-state
              :accessor in-state)))

(defclass in-state
    (in-state-mixin abstract-state) ())
```

Then the class for attributed-states is derived from the `in-state` class.

```
(defclass attributed-state (in-state)
  ((state-attribute
    :initarg :state-attribute
    :accessor state-attribute)
   (combine-fun :initarg :combine-fun
                :reader combine-fun)))
```

In the deterministic case, instead of computing just the state q, it computes an *attributed-target* which is just an *attributed-state* $[q, a]$ where q is the state computed by $\mathcal{A}$ and a the attribute.

In the non-deterministic case, instead of computing a set of states $\{q_1, \ldots, q_p\}$, it computes an attributed target which is a set of attributed states

$\{[q_1, a_1] \ldots, [q_p, a_p]\}$.

The final state predicate must be extended to work on attributed states: an attributed state $[q, a]$ is final if the state q is.

At each node of the term, the attribute (or the attributes in the non deterministic case) are computed from the ones obtained at the child nodes.

In the deterministic case, just one function must be provided which for each symbol $f \in \mathcal{F}_n$ returns a function of n arguments to be applied to the n attributes computed at the child node. We refer to it as the *symbol-fun*.

Suppose we have a term $t = f(t_1, \ldots, t_n)$ and already recursively computed the attributed state $[q_i, a_i]$ for each child t_i. Let $g =$ `symbol-fun`(f). The attribute for t is given by $g(a_1, \ldots, a_b)$.

In the non deterministic case, we may obtain the same state using different applicable rules. In that case, we need a function in order to combine the attributes into a single one. We refer to it as the *combine-fun*.

An instance of the class `afuns` contains all what is needed to attribute an automaton.

```
(defclass afuns ()
  ((symbol-fun :reader symbol-fun
               :initarg :symbol-fun)
   (combine-fun :reader combine-fun
                :initarg :combine-fun)))

(defun make-afuns (symbol-fun combine-fun)
  (make-instance 'afuns
                 :symbol-fun symbol-fun
                 :combine-fun combine-fun))
```

We can for instance count the number of runs (which is interesting for the non deterministic case). Here is the attribution mechanism for counting runs.

```
(defgeneric count-run-symbol-fun (symbol))
(defmethod count-run-symbol-fun
    ((s abstract-symbol))
  #'*)
(defparameter
    *count-afun*
  (make-afuns #'count-run-symbol-fun #'+))
```

The following `attribute-transitions-fun` operation transforms the transitions of a non attributed fly automaton into attributed transitions according to the attribution mechanism `afun`.

```
(defmethod attribute-transitions-fun
    ((transitions abstract-transitions) afun)
  (lambda (root attributed-states)
    (compute-attributed-target
     root
     (apply-transition-function
      root
      (mapcar #'in-state attributed-states)
      transitions)
     afun
     attributed-states)))
```

An attributed state $[q, a]$ is final for the attributed automaton if q was final for the non attributed automaton.

```
(defmethod attribute-final-state-fun
    ((automaton abstract-automaton))
  (lambda (attributed-state)
    (final-state-p
     (in-state attributed-state)
     automaton)))
```

With the two previous operations, we can define the operation which transforms a non attributed automaton into an attributed one according to the attribution mechanism `afun`.

```
(defmethod attribute-automaton
    ((automaton abstract-automaton) afun)
  (let ((transitions
         (transitions-of automaton)))
    (make-fly-automaton-with-transitions
     (make-fly-transitions
      (attribute-transitions-fun
       transitions afun)
      (deterministic-p automaton)
      (complete-p automaton)
      (completion-state-final transitions)
      :transitions-type
      'fly-casted-transitions)
     (signature automaton)
     (attribute-final-state-fun automaton)
     (format nil "~A-att" (name automaton)))))
```

5.2 Fly transducers

A fly transducer is just a fly automaton with an output function which may be applied to the targets.

If the fly automaton is not attributed, by default, the output function is the final state predicate which returns a Boolean value.

If the fly automaton is attributed, by default, the output function returns the attribute of the final target: in the deterministic case, the target is just a state $[q, a]$ and the result is just the attribute a; in the non deterministic case, the target is a container of attributed states $[q_1, a_1], \ldots, [q_p, a_p]$ and the result is computed by applying the `combine-fun` of the attribution mechanism to the attributes $a_1, \ldots, a_p$.

The main operations applicable to a term and a fly transducer are `compute-value (term fly transducer)` and `compute-final-value (term fly transducer)`.

5.2.1 Counting graph-colorings

We would like a fly transducer for counting the number of k-colorings of a graph. Note that the problem is $\#P$-complete for $k = 3$.

We start with the automaton `*2-colored*` computed previously (see Figure 8) and recognizing graphs having a proper k-coloring, then we attribute it with `*count-afun*`.

```
(setf *2-colored-counting*
      (attribute-automaton
       *2-colored*
       *count-afun*))
```

Then we do the color-projection as before.

```
(setf *count-2-colorings*
      (color-projection
       *2-colored-counting*
       2))
```

The resulting automaton computes attributed states each one containing the number of runs leading to the state.

```
AUTOGRAPH> (compute-final-target
            *t3*
            *count-2-colorings*)
o{[!<a:1 b:2>,1] [!<a:2 b:1>,1]}
```

Used as a transducer, it computes the number of k-colorings of the graph:

```
AUTOGRAPH> (compute-final-value
            *t3*
            *count-2-colorings*)
2
T
```

For some well-known graphs of graph theory like Petersen's graph, the chromatic polynomial has already been computed: it gives the number of colorings for each k. We could verify experimentally that our method gives the same values as the chromatic polynomial.

The chromatic polynomial for Petersen's graph is

$$k(k-1)(k-2)$$
$$(k^7 - 12k^6 + 67k^5 - 230k^4 + 529k^3 - 814k^2 + 775k - 352)$$

For instance, we may verify that Petersen's graph has 12960 4-colorings. Note that the problem of deciding whether a graph is k-colorable is NP-complete for $k > 2$.

```
AUTOGRAPH> (compute-final-value
    (petersen)
    (color-projection-automaton
     (attribute-automaton
      (coloring-automaton 4)
      *count-afun*)
     4))
12960
T
AUTOGRAPH> (petersen-chromatic-polynomial 4)
12960
```

5.2.2 Computing graph-colorings

The coloring of a graph described by a term may be given by a function which, given a constant position in the term, gives the assigned color number. The positions are denoted by Dewey's words which are words in $[0, m[^*$ where m is the maximal arity of a symbol in the signature.

For representing graphs, the maximal arity is 2 (`oplus`), so the positions will contain only zeros or ones.

The root position is denoted by the empty word. In outputs, it will be denoted by `E`.

For instance, the set of positions of the term `a` is { `E` } and the set

of positions of `add_a_b(oplus(a,b))` is
{ `E, 0, 00, 01` }.

It is not diffcult to compute such colorings as an attribute on the automaton which verifies that a graph has a proper k-coloring.

This attribution mechanism is accessible via the variable `*assignment-afun*`.
The code implementing this mecanism is presented in Figure 12 at the end of the paper. It works both for color assigment and subset assignment. The `combine-fun` is just a union; The `symbol-fun` is the `assignment-symbol-fun` function; for every non constant operation it returns the function for computing the attribute: it extends the positions computed so far with the correct child number; in the case of colored constant symbols, if returns a zeroary function that will initiate the attribute as a list containing the empty position associated with the color of the constant.

```
AUTOGRAPH> (setf *af2*
             (attribute-automaton
              *2-colored*
              *assignment-afun*))
0-COLORED-2-att
```

The following example shows how to obtain all the possible proper colorings for the graph t_3.

```
AUTOGRAPH> (setf *f2*
                 (color-projection-automaton
                  *af2* 2))
fly-asm(0-COLORED-2-att)
AUTOGRAPH> (compute-final-value *t3* *f2*)
(([00:1] [010:1] [0110:2] [0111:2])
 ([00:2] [010:2] [0110:1] [0111:1]))
T
```

Although being finite, the set of possible proper colorings of a graph may be of exponential size. We do not necessarily need all proper colorings. If that is the case, then the enumeration mechanism described in ELS2012 [12] is just what we need.

We construct an enumerator of the final values of the fly transducer whose values are colorings. Then we just enumerate these values in order to obtain as many proper colorings as we need.

```
AUTOGRAPH> (defparameter *e*
             (final-value-enumerator
              (petersen)
              *compute-4-colorings*))
*E*
```

Then we call the enumerator to get the colorings one by one.

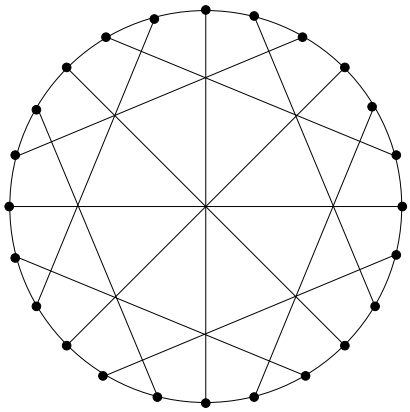


Figure 11: McGee's Graph

```
AUTOGRAPH> (call-enumerator *e*)
(([0000:3]
  [000100000:2]
  [0001000010000:3]
  [00010000100010000000000000:2]
  [00010000100010000000000001:1]
  [0001000010001000000000001:1]
  [000100001000100000000001:1]
  [00010000100010000000001:2]
  [0001000010001000000001:3]
  [000100001000100000001:2]))
T
```

Given the size and complexity of the value computed by our transducers, we obtain different classes of complexity (FPT, XP) [16, 8]. This has been studied and submitted to CAI2013 [5].

6. EXPERIMENTS

We have worked on many graph properties, many of which are described in [11, 4, 5] in particular on acyclic-colorings [18].

In graph theory, an acyclic-coloring is a (proper) vertex coloring in which every 2-chromatic subgraph is acyclic. The acyclic chromatic number $A(G)$ of a graph G is the least number of colors needed in any acyclic-coloring of G. It is NP-complete to determine whether $A(G) \leq 3$ (Kostochka 1978). So acyclic-colorability is not a trivial matter.

McGee's graph is shown in Figure 11; it is regular (degree 3); it has 24 vertices and 36 edges; we have a decomposition yielding a term of clique-width 10, size 76 and depth 99. This graph is 3-acyclic-colorable (but not 2-acyclic-colorable). We may verify this last fact in less than three hours and compute the number of 3-acyclic-colorings (57024) in less than six hours.

7. CONCLUSION AND PERSPECTIVES

We have defined fly transducers on terms which can compute information about terms. When terms represent graphs, we can compute information about graphs.

One advantage of fly automata and fly transducers are their flexibility and the possibility to transform or combine them in order to obtain new ones. This can be elegantly done through the functional

paradigm of Lisp. The CLOS layer is also heavily used both in `Autowrite` and `Autograph`.

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 [15]. [19] gives polynomial approximated solutions to solve this problem. More can be found in [6].

In some applications, the graphs may be given by their decomposition. Some methods exist for specific graphs like cliques, grids, square grids, trees, ... For other graphs like Petersen's or McGee's, we had previously done hand decompositions. In most cases, we do not know whether we have the best decomposition (the smallest clique-width).

Recently, we have started developping a system `DecompGraph` for approximated clique-width decomposition of graphs. *Approximated* means, that it will not necessarily give the smallest possible clique-width.

The `DecompGraph` is independent from `Autograph`.
With `DecompGraph`, we can at least decompose the graphs we had already worked on and were pleased to improve the hand decomposition of Petersen's graph from $cwd = 7$ to $cwd = 6$. For McGee's graph however, we did not find a better decomposition ($cwd = 10$) with `DecompGraph`.

The decomposition of a graph is a kind of preprocessing phase. Once the graph is decomposed into a term, we may keep the decomposition and apply as many fly automata or fly transducers as we want on the term.

The fact that we can now decompose graphs (although may be not very big ones) means that we are able to effectively prove graph properties and compute graph properties starting from the graph itself which was not possible before.

Any domain using terms for representing objects (language processing, protocol verification, ...) could benefit from fly transducers. We are looking for applications in graphs or any other domain using terms.

Acknowledgements

The authors thank the referee for his very accurate and constructive remarks.

8. REFERENCES

[1] 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`.

[2] B. Courcelle and I. Durand. Verifying monadic second order graph properties with tree automata. In *Proceedings of the 3rd European Lisp Symposium*, pages 7–21, May 2010.

[3] B. Courcelle and I. Durand. Fly-automata, their properties and applications. In B. B.-M. et al., editor, *Proceedings of the 16th International Conference on Implementation and Application of Automata*, volume 6807 of *Lecture Notes in Computer Science*, pages 264–272, Blois, France, July 2011. Springer Verlag.

[4] B. Courcelle and I. Durand. Automata for the verification of monadic second-order graph properties. *Journal of Applied Logic*, 10(4):368 – 409, 2012.

[5] B. Courcelle and I. Durand. Automata for monadic second-order model-checking. In *Proceedings of the 5th International Conference on Algebraic Programming*, Porquerolles Island, Aix-Marseille University, France, 2013. To appear in September 2013.

[6] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic, a Language Theoretic Approach*. Cambridge University Press, 2012.

[7] B. Courcelle, J. A. Makowsky, and U. Rotics. On the fixed parameter complexity of graph enumeration problems definable in monadic second-order logic. *Discrete Applied Mathematics*, 108(1-2):23 – 52, 2001.

[8] R. G. Downey, M. R. Fellows, and U. Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, volume 49, pages 49–99. AMS-DIMACS Proceedings Series, 1999.

[9] I. 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.

[10] I. Durand. Autowrite: A tool for term rewrite systems and tree automata. *Electronics Notes in Theorical Computer Science*, 124:29–49, 2005.

[11] I. Durand. Implementing huge term automata. In *Proceedings of the 4th European Lisp Symposium*, pages 17–27, Hamburg, Germany, March 2011.

[12] I. Durand. Object enumeration. In *Proceedings of the 5th European Lisp Symposium*, pages 43–57, Zadar, Croatia, May 2012.

[13] I. Durand. Autowrite. Software, since 2002.

[14] I. Durand. Autograph. Software, since 2010.

[15] 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.

[16] M. R. Fellows, F. V. Fomin, D. Lokshtanov, F. Rosamond, S. Saurabh, S. Szeider, and C. Thomassen. On the complexity of some colorful problems parameterized by treewidth. *Information and Computation*, 209(2):143 – 153, 2011.

[17] M. C. GOLUMBIC and U. ROTICS. On the clique-width of some perfect graph classes. *International Journal of Foundations of Computer Science*, 11(03):423–443, 2000.

[18] B. Grünbaum. Acyclic colorings of planar graphs. *Israel Journal of Mathematics*, 14:390–408, 1973.

[19] S.-I. Oum. Approximating rank-width and clique-width quickly. *ACM Trans. Algorithms*, 5(1):1–20, 2008.

[20] J. Thatcher and J. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.

```
(defun union-fun (&key (fun #'union) (test #'equalp))
  (lambda (&rest attributes)
    (reduce (lambda (a1 a2) (funcall fun a1 a2 :test test))
            attributes :initial-value '())))

(defgeneric position (position-assignment))
(defgeneric assignment (position-assignment))

(defclass position-assignment ()
  ((position :initarg :position :reader position)
   (assignment :initarg :assignment :reader assignment))
  (:documentation "position with color or subset assignment"))

(defun make-position-assignment (position assignment)
  (make-instance 'position-assignment
                 :position position
                 :assignment assignment))

(defgeneric left-extend-position-assignment (position-assignment i))
(defmethod left-extend-position-assignment
    ((position-assignment position-assignment) (i integer))
  (make-position-assignment
   (left-extend-position (position position-assignment) i)
   (assignment position-assignment)))

(defgeneric assignment-fun (attributes1 attributes2))
(defmethod assignment-fun ((attributes1 list) (attributes2 list))
  (loop
     with attributes = '()
     for a1 in attributes1
     do (loop for a2 in attributes2
              do (push (append a1 a2) attributes))
     finally (return attributes)))

(defgeneric assignment-symbol-fun (s))
(defmethod assignment-symbol-fun ((s vbits-constant-symbol))
  (lambda ()
    (list (list (make-position-assignment (make-position '()) (vbits s))))))

(defmethod assignment-symbol-fun ((s color-constant-symbol))
  (lambda ()
    (list (list (make-position-assignment
                 (make-position '())
                 (symbol-color s))))))

(defmethod assignment-symbol-fun ((s abstract-parity-symbol))
  (lambda (&rest attributes)
    (setf attributes
          (loop
             for attribute in attributes
             for i from 0
             collect
               (loop
                  for position-assignments in attribute
                  collect
                    (loop
                       for position-assignment in position-assignments
                       collect (left-extend-position-assignment
                                  position-assignment i)))))
    (if (endp (cdr attributes))
        (car attributes)
        (reduce #'assignment-fun attributes))))

(defparameter *assignment-afun* (make-afuns #'assignment-symbol-fun (union-fun)))
```

Figure 12: The attribute mecanism for computing position assignment