Introducing VAUCANSON

Sylvain Lombardy[1], Raphaël Poss[2], Yann Régis-Gianas[2], and Jacques Sakarovitch[3]

[1] LIAFA, Université Paris 7, lombardy@liafa.jussieu.fr
[2] LRDE, EPITA, {yann.regis-gianas,raphael.poss}@lrde.epita.fr
[3] LTCI, CNRS / ENST, sakarovitch@enst.fr

Abstract. This paper reports on a new software platform dedicated to the computation with automata and transducers, called VAUCANSON, the main feature of which is the capacity of dealing with automata whose labels may belong to various algebraic structures.
The paper successively shows how VAUCANSON allows to program algorithms on automata in a way which is very close to the mathematical expression of the algorithm, describes some features of the VAUCANSON platform, including the fact that the very rich data structure used to implement automata does not weight too much on the performance and finally explains the main issues of the programming design that allow to achieve both genericity and efficiency.

This paper reports on a new software platform dedicated to the computation with automata and transducers, which we have named VAUCANSON [4]. The present status of VAUCANSON is still fairly experimental and many of its functions are under active development. However, the purposes, the design policy, and the implementation issues of the platform are well-established by now, so that it has appeared that VAUCANSON could be introduced to the community[5], with the undisguised aim that this software will be tried and tested and that the authors will get some feedback.

The striking feature of automata is the versatility of the concept — a labelled oriented graph — and its ability to modelize so many different kinds of machines simply by varying the domain where the *labels* are taken. In the most general setting, these labels are *polynomials* (or even *rational series* indeed) over a monoid M with multiplicity in a semiring $\mathbb{K}$. "Classical" automata are obtained when M is a free monoid A^*, when the multiplicity semiring is the Boolean semiring $\mathbb{B}$ and when every label is a letter in A; transducers can be seen as automata over a monoid $A^* \times B^*$ with multiplicity in $\mathbb{B}$ as well as automata over A^* with multiplicity in $\mathcal{P}(B^*)$; automata over A^* with multiplicity in $(\mathbb{N}, \mathsf{min}, +)$ have been used in order to represent jobshop problems, *etc.*

Many systems already exist which manipulate automata and related structures (expressions, grammars, ...) but almost all these systems deal with automata the labels of which are letters or words — with the notable exception of FSM which can compute with transducers and automata with "numerical" multiplicity. Most of these systems deal with automata the label of which are letters, or words — with the notable exception of the FSM system which can compute with transducers and automata with multiplicity. The main idea in designing VAUCANSON has been to take advantage of the most recent techniques in generic

[4] Jacques Vaucanson (1709–1782), a French automaton maker, built the famous "flute player" and "duck" automata.

[5] Two of the authors of the paper (S. L. and J. S.) have written a LaTeX macro package that has also been coined VAUCANSON. The latter name will soon be changed in order to avoid confusion.

programming in order to deal with automata the labels of which may be freely chosen in any algebraic structure, with the capacity of writing independently (as far as they are independent) the algorithms on the automata on one hand and the operations in the structure on the other hand.

In the brief presentation that follows, we shall first show how the functions implemented in VAUCANSON allow to program algorithms on automata in a way which is very close to the mathematical expression of the algorithm. The second part will describe some features of the VAUCANSON platform, including the fact that the very rich data structure used to implement automata does not weight too much on the performance. The third part explains the main issues of the programming design of the platform that allow to achieve both genericity and efficiency.

1 Writing algorithms with VAUCANSON

Another characteristic feature automata theory, when seen from a mathematical point of view is that most statements are *effective* and that proofs are indeed *algorithms* — and moreover, in many cases "good" proofs yield algorithms of "optimal" complexity. A first goal that is aimed with VAUCANSON is to give the possibility of writing programs for algorithms on automata in a language that is as close as possible of the mathematical description of the algorithm. We illustrate this capacity on an example that is not too well-known and that we treat completely.

1.1 The universal automaton of a language

The universal automaton of a rational language L is a canonical automaton. It has been introduced by Conway in 1971 [3]. This automaton can be used to find the smallest NFA that accepts the language (*cf.* [1, 12]), or, to study properties of some regular languages (*e.g.* star height [10, 9] or reversibility [8]).

The main property of this automaton $\mathcal{U}_L$ is that there is a *morphism* of any automaton that accepts the language L into $\mathcal{U}_L$.

The states of this automaton are the (maximal) factorizations of the language, *i.e.* the maximal pairs (X, Y) of languages such that $X.Y$ is a subset of L. A state (X, Y) is initial (*resp.* final) iff the empty word belongs to X (*resp.* to Y). There is a transition labeled by a between (X, Y) and (X', Y') iff $X.a.Y'$ is a subset of L. These factorizations can be computed in the syntactic monoid, hence the universal automaton is finite and effectively computable.

1.2 Construction of the universal automaton

We give here another construction (*cf.* [8, 14]), that does not require the computation of the syntactic monoid.

Let $\mathcal{D} = \langle Q, A, \delta, \{i\}, T \rangle$ be a deterministic automaton that accepts L (for instance, the minimal automaton), where Q is its set of states, A is the alphabet, δ the transition function ($p \in \delta(p, a)$ means that there is a transition from p to q with label a), i is the initial state and T is the set of final states.

- Compute the co-determinized automaton $\mathcal{C}$ of the automaton $\mathcal{D}$. Let P be the set of states of $\mathcal{C}$. Every element of P is a subset of Q.
- Compute the closure under intersection of P. The result is a set R: every element of R is a subset of Q.
- The universal automaton is $\mathcal{U}_L = \langle R, A, \eta, J, U \rangle$, where:
  - $J = \{X \in R \mid i \in X\}$: a state X is initial iff it contains the initial state of $\mathcal{D}$;
  - $U = \{X \in R \mid X \subseteq T\}$: a state X is final iff every element of X is final in $\mathcal{D}$;
  - $\eta(X, a) = \{Y \in R \mid \forall p \in X, \delta(p, a) \cap Y \neq \emptyset\}$: there is a transition from X to Y labeled by a iff for every element of X, there is a transition labeled by a to some element of Y.

This algorithm is written in pseudo-language on Figure 1. J, U and $\eta(X, a)$ (for every state X and every letter a) are sets. In the pseudo-code and in the C++ program, they are built incrementally.

This algorithm can be translated into a C++ function written with VAUCANSON. The code is given on Figure 2. In C++, there is no need for variables to deal with initial states, final states or transition function of automata, since they are fields or methods of the automaton object.

1.3 Comments on the code

A good understanding of this paragraph may require some knowledge about C++.

- VAUCANSON provides a lot of new types. Every type is designed by a word ending by `_t`, like `usual_automaton_t`, `hstate_t`,. . .
- l. 3: the alias `AUTOMATON_TYPES_EXACT` describes the frame in which the function will be defined and used. It fixes particulary some types. For instance, the automata we deal with are here Boolean automata (without any multiplicity) on free monoid. This implies definitions of particular names for types. For instance, `automaton_t` is an alias for `usual_automaton_t`. This is the reason why `usual_automaton_t` must be used in the declaration of function whereas, after line 3, one can declare `t`, `c` or `u` as `automaton_t`. Likewise, `alphabet_t`, `states_t` are defined, what is used in some macros like `for_each_state` that we will explain further.
- l. 4: *`d`* is an automaton, *`d`*`.initial()` is the set of its initial states (which has one element, because *`d`* is deterministic). *`d`*`.initial().begin()` is a pointer on the first element of this set and thus *`i`* is the initial state of *`d`*.
- l. 6: It holds co-determinized($\mathcal{D}$)=transposed(determinized(transposed($\mathcal{D}$)). the name of the function is `auto_transpose` to avoid any confusion with the transposition (or mirror image) of words.
- l. 7: Every state of $\mathcal{C}$ is a subset of states of $\mathcal{D}$. This relation must be made explicit: this is done with *`subset_c_state`*, which is a map from every state of *`c`* to a subset of states of *`d`*. This map is an optional parameter of `determinize`. Likewise, *`subset_u_state`* is a map from every state of *`u`* to a subset of states of *`d`*.
- l. 11: The declaration of the variable *`u`* induces the creation of the automaton.
- l. 12: Some parameters of automata are dynamically defined. In general, it can be the monoid of labels and the semiring where multiplicities stand. This line means that these parameters are the same for `u` as for `d`. This assignement is more an assignement of types than an usual assignement of values. In this particulary case, we only have to define the alphabet since the other characteristics are induced by the type `automaton_t` (defined line 3).

```
Universel ($\mathcal{A} = \langle Q, A, \delta, \{i\}, T \rangle$)
    $\mathcal{C}$ := co-determinized($\mathcal{A}$)
    $P$ := states-of($\mathcal{C}$) (* $\subseteq \mathcal{P}(Q)$ *)
    $R$ := intersection-closure($P$)
    $J := \emptyset$     $U := \emptyset$
    $\forall X \in R$, $\forall a \in A$, $\eta(X, a) := \emptyset$

    $\forall\ X \in R$ (* $\Leftrightarrow$ $X$ state of $\mathcal{U}$ *)
        if ($i \in X$) then $J := J \cup \{X\}$
        if ($X \subseteq T$) then $U := U \cup \{X\}$
        $\forall\ a \in A$
            if ($\forall\ p \in X$, $\delta(p, a) \neq \emptyset$) then
                $\forall\ Y \in R$
                    if ($\delta(X, a) \subseteq Y$) then
                        $\eta(X, a) := \eta(X, a) \cup Y$
    return $\mathcal{U} = \langle R, A, \eta, J, U \rangle$
```

Fig. 1. Construction of the universal automaton: the algorithm

```
usual_automaton_t universal(const usual_automaton_t& d)
{
   AUTOMATON_TYPES_EXACT(usual_automaton_t);
   hstate_t i = *d.initial().begin();
   map_t subset_c_state;
   automaton_t t = auto_transpose(t);
   automaton_t c = auto_transpose(determinize(t, subset_c_state));
   pstate_t c_states = image(subset_c_state);
   pstate_t u_states(intersection_closure(c_states));

   automaton_t u;
   u.series() = d.series();
   map_t subset_u_state;
   for_each_const(pstate_t, s, u_states)
   {
      hstate_t new_s = u.add_state();
      subset_u_state[new_s] = *s;
   }
   for_each_state(x, u)
   {
      if (is_element(i, subset_u_state[x]))
         u.set_initial(x);
      if (is_subset(subset_u_state[x], d.final()))
         u.set_final(x);
      for_each_letter(a, u.series().monoid().alphabet())
      {
         std::set<hstate_t> delta_ret;
         bool comp = delta_set(d, delta_ret, subset_u_state[*x], *a);
         if (comp)
            for_each_state(y, u)
               if (is_subset(d_ret, subset_u_state[*y]))
                  u.add_letter_edge(*x, *y, *a);
      }
   }
      return u;
}
```

Fig. 2. Construction of the universal automaton: VAUCANSON code

- l. 14: For every element of the closure *u_states*, a state is created and one store the link between the state and the corresponding subset.
  `for_each_const` is a macro that takes three parameters, the first one is a type, the third one is a container of this type and the second one is an iterator that handles every element of that container. This line is equivalent to:

```
for ( pstate_t::iterator s = u_states.begin();
      s != u_state.end();
      s++)
```

- l. 19: `for_each_state` is a macro; the first parameter is an iterator of states and the second is an automaton. This line is equivalent to:

```
for ( state_t::iterator x = u.states().begin();
      x != u.states().end();
      x++)
```

- l. 21-24: For every state, the initial and final properties are set.
- l. 25: From the automaton `u`, one can access to the “series” of `u`, and then, to the monoid on which this series is build. Once the monoid is known, one can get the generators of this monoid, *i.e.* the alphabet.
- l. 28: The result of `delta_set` is true if and only if, for every element p of `subset_u_state[*x]`, there exists a transition labeled by `*a`. In this case, the set of the aims of transitions labeled by `*a` whose origin is in `subset_u_state[*x]` is stored in *delta_ret*.
- l. 32: A transition from *x* to *y* is created, with label `a`; actually, `x`, `y` and `a` are iterators, and this is the reason why there is a star at the front of each of them.

2 Glimpses of the library

The purpose of this communication is not to be a user manual of Vaucanson and even not list all its functionalities. We give here only few hints on what is to be found in the library.

2.1 Determinization for benchmarking

The determinization of automata (over A^*) is a basic algorithm found in every system. It is known that this algorithm may imply a combinatorial explosion.

We consider the following family of automata: $\mathcal{A}_n$ is an automaton with n states: $0, 1, ..., n-1$ such that 0 is the only initial and the only final state, the alphabet is $\{a, b, c\}$ and the transition function δ (on the alphabet $\{a, b, c\}$) is defined by:

$$\delta(0, q) = \{1\}, \qquad \delta(0, b) = \delta(0, c) = \emptyset,$$
$$\forall i \neq 0, \quad \delta(i, a) = \{i + 1 \mod n\}, \qquad \delta(i, b) = \{i\}, \qquad \delta(i, c) = \{0, i\}.$$

The following test has been run on a Xeon 2.4Ghz, 256Ko cache memory, 1Go RAM.

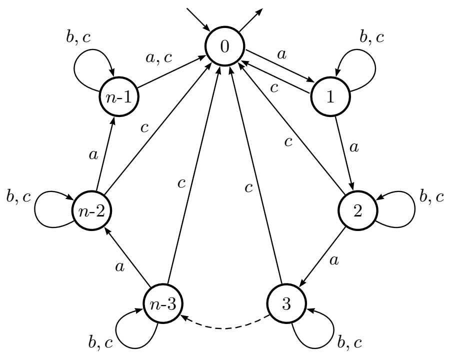

Fig. 3. The automaton $\mathcal{A}_n$

n		3	5	7	9	11	13	15	17	19
Time (s)	FSM	0.01	0.01	0.02	0.02	0.05	0.21	1.04	5.74	35.7
	AMoRE	0.02	0.02	0.03	0.13	0.55	2.62	12.0	57.4	$*$
	VAUCANSON	0.00	0.00	0.00	0.01	0.08	0.39	1.89	9.08	43.0
Space (MB)	FSM	0.006	0.01	0.03	0.1	0.4	1.7	7.3	30.5	128
	AMoRE	0.6	0.6	0.6	0.6	0.6	0.7	2.1	8.1	$*$
	VAUCANSON	0.015	0.04	0.1	0.4	1.7	7	29	116	437

2.2 Minimization of $\mathbb{K}$-automata

In many semirings of multiplicities, it can be hard and sometimes even impossible to find a smallest automaton that realizes a series. Yet, there exist some local conditions on the states of an automaton that allow to merge some of them. The result of this process is an equivalent $\mathbb{K}$-automaton called the minimal $\mathbb{K}$-covering (*cf.* [14]). This is *not* a canonical automaton of the language. Two $\mathbb{K}$-automata are bisimulation equivalent iff they have the same minimal $\mathbb{K}$-covering. This is a generalization of the well-known Nerode equivalence involved in the minimization of DFAs. VAUCANSON provides a generalized version of the Hopcroft algorithm that computes this equivalence for an automaton $\mathcal{A}$ with multiplicity in $\mathbb{K}$ and the corresponding minimal $\mathbb{K}$-covering.

2.3 From automata to expression and back

Almost all systems computing with automata implement Kleene's Theorem, that is compute a rational (regular) expression equivalent to a given automaton and conversely. VAUCANSON library implements the so-called *state elimination method*. This method relies (as the other methods indeed) on an ordering of the states of the automaton and the expression obtained as the result depends on that ordering. A feature of the VAUCANSON implementation is that the ordering is a parameter of the algorithm and can also be computed via heuristics.

The transformation of an expression into an automaton has given rise to a very rich litterature. VAUCANSON implements three methods: the Thompson construction, the standard automaton of an expression (also called *position automaton* or *Glushkov automaton*) and the automaton of derived terms of an expression (also called *partial derivatives* or *Antimirov automaton*). For the latter, VAUCANSON implements the algorithm due to Champarnaud and Ziadi [2].

2.4 Transducer computation

VAUCANSON implements the two central theorems: the evaluation theorem and the composition theorem, with algorithms that correspond to the two mains proof methods: the morphism realization and the representation realization and that are used according to the type of the transducers (normalized, letter-to-letter, real-time).

2.5 Programming the algebraic structures

The definition of an automaton requires the definition of a semiring of multiplicities (or weights) and a monoid of labels. VAUCANSON allows the definition of any of these structures – and every generic algorithm can be applied on the resulting automata. A few of them are provided *e.g.* free monoids over any finite alphabet or product of monoids; this gives access to transducers that can be considered as automata over a monoid $A^* \times B^*$. Some semirings are pre-defined too: the Boolean semiring, the usual numerical semirings (integers, floating numbers) and tropical semirings (for instance $(\mathbb{N}, \mathsf{min}, +)$ or $(\mathbb{Z}, \mathsf{max}, +)$).

A series over a monoid with multiplicity in a semiring is itself a semiring and can be used as such. For instance, $\mathsf{Rat}(B^*)$ (the rational series over B^* with Boolean coefficients) is a semiring and automata over A^* with multiplicity in this semiring are another representation of transducers.

3 Design for genericity

The facilities exposed in the previous sections are not directly present in C++. A software layer is necessary to yield an abstraction level powerful enough for genericity. Yet, abstraction should not imply poor efficiency so the way of implementing polymorphism has to be carefully chosen.

This section points out the design issues involved in the development of the Vaucanson library and its position confronted with the current known solutions of generic programming. First, we describe what helps the writing of algorithms in the framework. Then, we explain how we deal with the usual trade-off between genericity and efficiency. A new Design Pattern (*cf* [7]) for this purpose is presented and constitutes the contribution in the generic programming field.

3.1 A unified generic framework

We describe the framework for the writing of algorithm. It relies on how the object are typed, how the types of algorithm inputs are specified and how VAUCANSON can be adapted to foreign environments.

Every VAUCANSON object is an element of a set As in Java where every entity has the `Object` type, every VAUCANSON entity (automaton, series ...) is an instance of the `Element<S, T>` class. `Element<S, T>` can be read like *an element of a set* `S` *implemented by the type* `T`. An instance of the `Element<S, T>` class is always linked with an instance of `S` and an instance of `T`. The `S` instance denotes the dynamic features of the set and the `T` instance represents the value of the element.

As a set, the `S` attribute represents the concept handled by the element. Because it is always linked to its set, an element can retrieve all the algebraic information it was built from. For example, in the algorithm (Figure 2), `u.set().series().monoid().alphabet()` returns the alphabet on which the automaton `u` is defined. This encapsulation enables shorter function prototypes and then, better readability.

Given a set `S`, an element of `S` has a well defined interface whatever its implementation. Therefore, an algorithm can mix elements implemented by different ways transparently, just by specifying that the implementations can be different. For instance, a generic algorithm which computes the product of two automata could be prototyped by:

```
template <class T1, class T2>
Element<Automata, T1>
product(Element<Automata, T1>, Element<Automata, T2>);
```

Finally, the implementation parameter allows a choice between different algorithm versions depending on the underlying data structure. For example, a serie can be implemented as a finite map or as a rational expression. The constant term is computed differently according to the chosen implementation.

Focus on weighted automata services Thanks to our generic design, the design issues are centered on the general algebraic concepts (weighted automaton, general series ...). Although algebraic objects (alphabet, monoid, semiring and series) do not involve particular problems, the design decisions about the automaton object are essential according to an algorithmic and an ergonomic point of view.

As emphasized in [11], the δ function (*the successor function*) is a crucial primitive because it is a general mechanism with a real algorithmic effect and which depends both on the implementation and on the concept. The δ function must act as a glue between algorithms and data structures conveying only necessary information. Indeed, too rich a δ can lead to inefficiency whereas too poor a δ implies clumsy use. As a consequence, the VAUCANSON library provides a large variety of δ functions depending on algorithm needs.

The user can choose between states or edges as results. In order to obtain them, she also has the choice between container, output iterator or read-only access begin/end iterator couple. Finally, a criterion defines what kind of successors has to be retrieved. One can choose to return all output transitions, transitions whose labels match a particular letter or a user condition passed as an function-object.

Extending VAUCANSON with a new automaton implementation does not necessarily imply the definition of all of these δ. Indeed, many default implementations are automatically deduced from the others.

Interaction with external libraries Initiated by the Standard Template Library (STL), the iterator concept is a common ground between C++ libraries. It is an abstract way of defining the generic traversal of data structures. VAUCANSON implements it to manage interoperability with STL. VAUCANSON algorithms and data structures are highly based on STL, so, a lot of development effort is saved, avoiding the development of well known structures such as list, bit vector or red-black tree.

Furthermore, importing new data structure from an external library to use it as implementation of some element can be done easily. The user has just to specify the foreign C or

C++ type as implementation. Next, only a small set of external functions must be written to explain how the foreign implementation will fill the concept requirements. Then, linking with C/C++ external libraries is made natural and simple.

3.2 Polymorphism using C++ templates

Let us now introduce the considerations about genericity which have led to our framework.

Object-Oriented languages enable reusability based on contracts defined by abstract classes. Yet, in practice, the late binding to abstract services is too expensive and leads too bad performance for intensive computing. The generative power of C++ template allows the static resolution of abstract services. This results in high-level C++ programs whose speed is comparable to dedicated low-level C programs. The STL has initiated this trend and demonstrates its relevancy by its popularity.

STL approach As mentioned in [13], the writing of generic algorithms is made easier by using primitive services common to all library data structures. For example, the iterator concept uses the presence of a `begin()/end()` method couple in every container to abstract its traversal. An algorithm which is generic *w.r.t.* the container concept is parameterized by a free type variable `C`. The code is written assuming that an instance of `C` will be a container.

Yet, parameterization *à la* STL does not provide any constraints to ensure that parameters really fill the requirement. Moreover, this general typing leads to overloading problems, like prototyping two algorithms with the same name and arity. As a consequence, fine grained specialization is unavailable. Concretely, this means that writing a generic algorithm for a particular class of automata is not allowed.

The main explanation is that STL lost the subclassing relation between objects because of a non constrained universal type quantification. The VAUCANSON design solved this problem by making a step further in generic programming that consists in implementing a generic object framework with static dispatch using C++-templates [6, 4]. These programming methods entail a stronger typing, which enables a finer specialization power and solves the overloading problem.

Beyond classical use of templates One classical object hierarchy is not enough to obtain extensibility in our framework. The current section will describe a new design pattern we developped to allow a richer genericity.

One more time, the main issue is to bound as precisely as possible the domain of an algorithm. Using only one object hierarchy would yield a one dimensional discrimination. Yet, a fine grained specialization would require the hierarchy to be a directed acyclic graph (with multiple inheritance).

To simplify the object organization, we define more components to characterize an object. We notice that abstraction and implementation are quite orthogonal for at least two reasons. Firstly, when writing a general algorithm, people should only focus on the mathematical concepts. Implementation constraints are taken into account afterwards. Second, algorithm specialization should depend on implementation and concept symmetrically.

Because of this orthogonality, it is easier to design the implementation and the concept separately. Design patterns for this purpose are the classical BRIDGE [7] or more recently the GENERIC BRIDGE [5]. However, there remain two problems for us: first, it is asymmetric, privileging concept upon implementation; second, it does not allow subclassing *w.r.t* the two parameters because template arguments are invariant.

To solve all these problems, the VAUCANSON library uses a new design pattern called ELEMENT/METAELEMENT. The main idea is to enable de-construction of an object *w.r.t* its two components and to use them for typing purpose. Element is a generic class associating a concept class and an implementation one. The role of MetaElement is to define the interaction between these two components that is, *how the data structure implements the concept.* A kind of multi-methods with static dispatch is also used to allow default implementation and specialization of n-ary methods. The pattern is illustrated in the figure 4 using the Unified Modelling Language. Its effective implementation involves some C++ meta-programming techniques which will not be explicited in this paper.

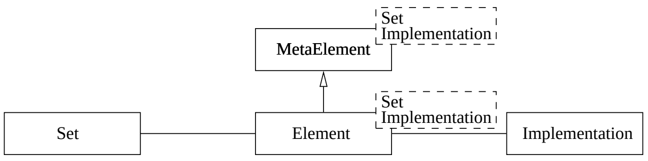

Fig. 4. UML diagram of the ELEMENT/METAELEMENT design pattern.

References

1. A. Arnold, A. Dicky, and M. Nivat, *A note about minimal non-deterministic automata*, Bull ot the EATCS **47** (1992), 166–169.
2. J.-M. Champarnaud and D. Ziadi, *Canonical derivatives, partial derivatives and finite automaton constructions*, Theor. Comp. Sci. **289** (2002), no. 1, 137–163.
3. J.H. Conway, *Regular algebra and finite machines*, Chapman and Hall, 1971.
4. J. Darbon, T. Géraud, and A. Duret-Lutz, *Generic implementation of morphological image operators*, Int. Symp. on Mathematical Morphology VI (ISMM'2002), April 2002, pp. 175–184.
5. A. Duret-Lutz, T. Géraud, and A. Demaille, *Design patterns for generic programming in C++*, Proc. of the 6th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'01), USENIX Association, 2001, pp. 189–202.
6. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, *On the design of CGAL, the computational geometry algorithms library*, Tech. Report 3407, INRIA, April 1998.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Abstraction and reuse of object-oriented design*, LNCS **707** (1993), 406–431.
8. S. Lombardy, *On the construction of reversible automata for reversible languages*, Proc. of ICALP'02, LNCS **2380** (2002), 170–182.
9. S. Lombardy and J. Sakarovitch, *Star height of reversible languages and universal automata*, Proc. of LATIN'02, LNCS **2286** (2002), 76–89.
10. ______, *On the star height of rational languages, a new presentation for two old results*, World Scientific (to appear).
11. V. Le Maout, *Cursors*, Proc. of CIAA 2000, LNCS **2088** (2001), 195–207.
12. O. Matz and A. Potthoff, *Computing small nondeterministic finite automata*, proc. of TACAS'95, BRICS Notes Series, 1995, pp. 74–88.
13. D.R. Musser and A.A. Stepanov, *Algorithm-oriented generic libraries*, Software - Practice and Experience **24** (1994), no. 7, 623–642.
14. J. Sakarovitch, *Éléments de théorie des automates*, Vuibert, 2003.