Available online at www.sciencedirect.com
SCIENCE DIRECT®

Theoretical Computer Science 328 (2004) 77–96

Theoretical Computer Science

www.elsevier.com/locate/tcs

Introducing VAUCANSON

Sylvain Lombardy[a], Yann Régis-Gianas[b], Jacques Sakarovitch[c,*]

[a]*LIAFA, Université Paris 7, 2 place Jussieu, F-75251 Paris, Cedex 05, France*
[b]*LRDE, EPITA, 14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre Cedex, France*
[c]*LTCI, UMR 5141, CNRS / ENST, 46 rue Barrault, F-75634 Paris, Cedex 13, France*

Abstract

This paper reports on a new software platform called VAUCANSON and dedicated to the computation with automata and transducers. Its main feature is the capacity of dealing with automata whose labels may belong to various algebraic structures.

The paper successively describes the main features of the VAUCANSON platform, including the fact that the very rich data structure used to implement automata does not weigh too much on the performance, shows how VAUCANSON allows to program algorithms on automata in a way which is very close to the mathematical expression of the algorithm and finally explains the main choices of the programming design that enable to achieve both genericity and efficiency.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Automata implementation; Automata with multiplicity; Generic programming

0. Introduction

This paper reports on the VAUCANSON[1] software platform dedicated to the computation with automata and transducers.[2]

* Corresponding author.
E-mail addresses: lombardy@liafa.jussieu.fr (S. Lombardy), yann.regis-gianas@lrde.epita.fr (Y. Régis-Gianas), sakarovitch@enst.fr (J. Sakarovitch).

[1]The VAUCANSON library can be downloaded from the URL: `http://vaucanson.lrde.epita.fr`.

[2]Two of the authors of the paper (S.L. and J.S.) have written a LaTeX macro package [14] that had also been coined VAUCANSON. This name has been changed into VAUCANSON–G in order to avoid confusion.

0304-3975/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2004.07.007

A 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 $\mathbb{Q}$ may compute probability of occurrences of words, those 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 deal with automata the labels of which are letters or words—with the notable exception of FSM [18] which can compute with transducers and automata with "numerical" multiplicity. [3]

The main idea in designing VAUCANSON has been to take advantage of the most recent techniques in generic 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 the one hand and the operations in the structure on the other hand.

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

1. Glimpses of the library

The purpose of this paper is not to be a user manual of VAUCANSON and even not to list all its functionalities. We give here only few hints on what is to be found in the library and on the way these functions have to be called in programmes. It will serve as an introduction to the design of VAUCANSON.

1.1. Description of automata

An automaton [4] is defined as a 5-uple $\langle Q, A, \delta, I, T\rangle$, where Q is a finite set of *states*, A a finite alphabet of *letters*, I and T the sets of initial and final states and $\delta\colon Q \times A \to \mathcal{P}(Q)$ the transition function.

[3] The FSA system [22] may also compute with such objects but as it is based on Prolog, the description of algorithms as well as the definition of automata is fairly different from the usage of the automata community.

[4] The reader is assumed to be familiar with the basic concepts and notations of automata theory, for which we essentially follow [10].

Let us consider the following family of automata $\mathcal{A}_n$ on the alphabet $A = \{a, b, c\}$: $\mathcal{A}_n = \langle \{0, 1, \ldots, n-1\}, A, \delta, \{0\}, \{0\} \rangle$, where the transition function δ is defined by

$$\delta(0, a) = \{1\}, \qquad \delta(0, b) = \delta(0, c) = \emptyset,$$

and, for every i different from 0,

$$\delta(i, a) = \{i + 1 \bmod n\}, \qquad \delta(i, b) = \{i\}, \qquad \delta(i, c) = \{0, i\}.$$

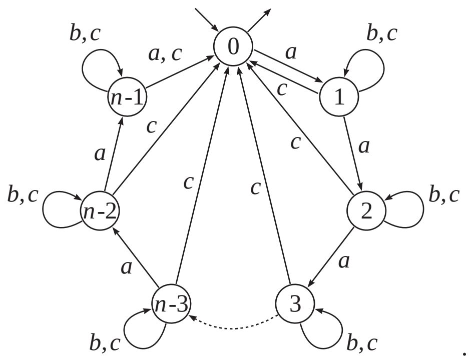

Fig. 1 gives a possible program for describing $\mathcal{A}_n$. Some knowledge about C++ is probably useful for the understanding of the sequel of this paragraph.

- l. 1: VAUCANSON provides a number of *classes*, that is *types* and *methods* attached to them, for dealing with objects involved in automata definition and computation. Every type is designed by a word ending by `_t`, like `automaton_t`, `hstate_t`,… These names are actually shortcuts as the types depend on a number of parameters such as the semiring of multiplicities. They are defined in files such as `vaucanson_boolean_automaton.hh` where the types are defined for automata with multiplicity in the Boolean semiring, that is the classical automata. We shall see in Section 3 how a type is defined in VAUCANSON.
- l. 2: The functions of the VAUCANSON library are contained in distinct modules. The `usual_algorithms.hh` header module allows to import many common functions such as `determinize`.
- l. 3: The VAUCANSON library is totally contained in the namespace `vcsn`. This allows easier access to the functions of the library in the program.
- l. 4: Indicates that the types that will be used are those that have been created by the macros in `vaucanson_boolean_automaton.hh`.
- l. 10: The class `alphabet_t` is equipped with the method `insert` that allows to build the alphabet `alpha`.
- l. 12: The automaton `an` is created as an automaton over the alphabet `alpha`. At this stage, `an` is “created” but is still empty.
- l. 14: The class `automaton_t` is equipped with the method `add_state` to define the states, …
- l. 18: …with the method `add_letter_edge` to define the transitions, …

```
 1 #include <vaucanson/vaucanson_boolean_automaton.hh>
 2 #include <vaucanson/usual_algorithms.hh>
 3 using namespace vcsn;
 4 using namespace vcsn::boolean_automaton;
 5 int main()
 6 {
 7 int n = 10;
 8 /* Definition of the alphabet */
 9   alphabet_t alpha;
10   alpha.insert('a'); alpha.insert('b'); alpha.insert('c');
11 /* Definition of the automaton */
12   automaton_t an= new_automaton(alpha);
13   hstate_t p,x,y;
14   p = an.add_state(); x = p;
15   for(int i=1;i<n;i++)
16   {
17     y=an.add_state();
18     an.add_letter_edge(x, y, 'a'); an.add_letter_edge(y, y, 'b');
19     an.add_letter_edge(y, y, 'c'); an.add_letter_edge(y, p, 'c');
20     x=y;
21   }
22   an.add_letter_edge(x, p, 'a');
23   an.set_initial(p); an.set_final(p);
24   automaton_t dn= determinize(an);
25 }
```

Fig. 1. Programming the automaton $\mathcal{A}_n$.

- l. 23: …and with the methods `set_initial` and `set_final` to define the initial and final states.
- l. 24: An example of a call of a VAUCANSON function over an automaton. The automaton `dn`, *of the same type as* `an`, is created, and then the determinized automaton of `an` is computed.

1.2. 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 and this is the case for the above example: the determinized automaton of (indeed the minimal deterministic automaton equivalent to) $\mathcal{A}_n$ has 2^n states. We have compared VAUCANSON with two other systems: AMoRe [16] and FSM [18]. [5]

[5] AMoRe is a software written in C, that allows to manipulate rational languages (given either through an automaton or a rational expression); it computes, for instance, the syntactic monoid or the minimal automaton of the language.
FSM is a C library that provides tools to manipulate (Boolean) automata as well as automata with multiplicity or transducers; these tools are basic commands (minimization, determinization, etc.) that can communicate by files or pipelines.

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

Fig. 2. Results for the determinization of the $\mathcal{A}_n$.

The determinization of the $\mathcal{A}_n$ has been run on a Xeon 2.4 Ghz, 256 Ko cache memory, 1 Go RAM. The results of this test are shown in Fig. 2.

1.3. A word on data structures and implementation

VAUCANSON default implementation for automata is a graph data structure. The design mainly focuses on providing fast structural and search operations.

First, the graph data structure is composed of many links between edges and states. Every state is a four-tuple of lists: two double-linked lists of states representing its successors and its predecessors, and two double-linked lists of edges representing incoming and outgoing edges. An edge is a triple formed by the source state, the destination state and a label. The label can be of any type: letter, polynom, abstract syntax tree denoting rational expression or user defined. This data structure is very redundant and this explains the quantity of memory used.

Second, VAUCANSON makes use of the data structure implementations provided by the C++ standard template library (STL). This allows to concentrate on the specific aspects of dealing with automata and avoids the error prone work of reimplementing usual data structures like double-linked lists, extensible arrays or balanced trees.

The efficiency—that is demonstrated in the above benchmark—is achieved not only by the versatility of the structure but also by a tight control by the VAUCANSON routines of the organization of this data structure in order to maximize the contiguity of the stored data in the memory. Thus, states and edges are handled by small integers which are offsets in one memory chunk. This yields fast graph operations and direct conversion to matrix representation.

Finally, and thanks to genericity, user-defined data structures closer to the requirements of a particular application can be transparently substituted.

1.4. 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 min-plus (or max-plus) semirings (for instance $(\mathbb{N}, \mathsf{min}, +)$ or $(\mathbb{Z}, \mathsf{max}, +)$).

The set of 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.

1.5. *From automata to expressions 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 literature. 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 [4].

1.6. *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. [21]). This is *not* a canonical automaton of the series realized by the $\mathbb{K}$-automaton. 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 Boolean DFAs (e.g. see [10]). VAUCANSON provides a generalized version of the Hopcroft algorithm that computes the minimal $\mathbb{K}$-covering of an automaton $\mathcal{A}$ with multiplicity in $\mathbb{K}$.

1.7. *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. Writing algorithms with VAUCANSON

Another characteristic feature of automata theory, when seen from a mathematical point of view is that most statements are *effective* and that proofs are indeed *algorithms*—and in many cases, “good” proofs yield algorithms of “optimal” complexity. An interesting feature of VAUCANSON is the possibility of writing programs for algorithms on automata in a language that is as close as possible to the mathematical description of the algorithm. We illustrate this ability by an example that is not too simple and that we treat completely.

2.1. Construction of the universal automaton

The universal automaton $\mathcal{U}_L$ of a rational (regular) language L is an automaton canonically attached to L. It has been (implicitly) introduced by Conway in [5] in order to solve some types of language equations. For sake of completeness, we give in Appendix A a brief account on the definition and the properties of $\mathcal{U}_L$.

The construction (implicitly) given by Conway takes place in the syntactic monoid of the language. We give here another construction (cf. [12,21]) that does not require the computation of the syntactic monoid and which is thus more efficient.

Let $\mathcal{D} = \langle Q, A, \delta, \{i\}, T\rangle$ be a deterministic automaton that accepts L (for instance, the minimal automaton of L); since $\mathcal{D}$ is deterministic, for every state p and every letter a, $\delta(p, a)$ is either the empty set or a singleton. The construction of $\mathcal{U}_L$ then goes as follow.

- Compute the co-determinized[6] 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 the family P. The result is a family 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:
  $J = \{X \in R \mid i \in X\}$: X is initial iff it contains the initial state of $\mathcal{D}$;
  $U = \{X \in R \mid X \subseteq T\}$: 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 labelled by a iff for *every* element of X, there is a transition labelled by a to some element of Y.
  This definition of $\eta(X, a)$ is equivalent to:

$$\eta(X, a) = \begin{cases} \emptyset & \text{if } \exists p \in X, \delta(p, a) = \emptyset, \\ \{Y \in R \mid \delta(X, a) \subseteq Y\} & \text{otherwise.} \end{cases}$$

This algorithm is written in pseudo-language in Fig. 3. It can be translated into a VAUCANSON function (Fig. 4), that is a C++ function written with primitives provided by the VAUCANSON library. Notice that the variables J, U and η, that represent initial states, final states and transitions in the pseudo-code, are useless in C++ because they are members of the automaton object. Opposite to the theoretical definition, these sets are built (both in the pseudo-language algorithm and in the VAUCANSON program) incrementally.

[6] An automaton is co-deterministic if its transposed automaton is deterministic; the co-determinized automaton is obtained by a subset construction, like the determinized automaton.

```
Universal ($\mathcal{D} = \langle Q, A, \delta, \{i\}, T\rangle$)
    $\mathcal{C}$ := co-determinized($\mathcal{D}$)
    $P$ := 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. 3. Construction of the universal automaton: the algorithm.

2.2. Comments on the code

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

- l. 3: `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. This pointer is dereferenced by the * and thus `i` is the initial state of `d`. The variable `i` is used at line 20 to decide whether a state of the automaton `u` is initial.
- l. 5: It holds co-determinize($\mathcal{D}$)=transposed(determinize(transposed($\mathcal{D}$))).
- l. 6: 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` (line 13) is a map from every state of `u` to a subset of states of `d`.
- l. 7: `pstate_t` is a shortcut for `std::set<std::set<hstate_t>>`, `c_states` and `u_states` are thus families of subsets of states of `d`.
- l. 10: Declaration of the variable `u` and creation of an automaton of the same type as automaton `d`, cf. Section 3.
- l. 13: `for_all_const` is a macro with 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 the elements of that container.
  This line is equivalent to:
  ```
  for ( pstate_t::const_iterator s = u_states.begin();
        s != u_states.end(); s++)
  ```
- l. 14–17: For every element of the closure `u_states`, a state is created and the link between the state and the corresponding subset is stored.
- l. 18: `for_each_state` is a macro; the first parameter `x` is an iterator of states—and thus a pointer—and the second one is an automaton. This line is equivalent to:
  ```
  for ( state_t::const_iterator x = u.states().begin();
        x != u.states().end(); x++)
  ```

```
 1 automaton_t universal(const automaton_t& d)
 2 {
 3   hstate_t i = *d.initial().begin();
 4   map_t subset_c_state;
 5   automaton_t t = transpose(d);
 6   automaton_t c = transpose(determinize(t, subset_c_state));
 7   pstate_t c_states = image(subset_c_state);
 8   pstate_t u_states = intersection_closure(c_states);
 9
10   automaton_t u(d.set);
11   map_t subset_u_state;
12
13   for_all_const(pstate_t, s, u_states)
14   {
15     hstate_t new_s = u.add_state();
16     subset_u_state[new_s] = *s;
17   }
18   for_each_state(x, u)
19   {
20     if (is_element(i, subset_u_state[*x]))
21       u.set_initial(*x);
22     if (is_subset(subset_u_state[*x], d.final()))
23       u.set_final(*x);
24     for_each_letter(a, u.series().monoid().alphabet())
25     {
26       std::set<hstate_t> delta_ret;
27       bool comp = delta_set(d, delta_ret, subset_u_state[*x], *a);
28       if (comp)
29         for_each_state(y, u)
30           if (is_subset(d_ret, subset_u_state[*y]))
31             u.add_letter_edge(*x, *y, *a);
32     }
33   }
34   return u;
35 }
```

Fig. 4. Construction of the universal automaton: the VAUCANSON code.

- l. 20–23: For every state, the property of being initial or terminal is set.
- l. 24: From the automaton `u`, one can access to the “series” of `u`, and then, to the monoid on which this series is build, and, at last, to the alphabet.
- l. 27: The result of `delta_set` is true if and only if, for every element p of `subset_u_state[*x]`, there exists a transition labelled by `*a`. In this case, the set of the aims of transitions labelled by `*a` whose origin is in `subset_u_state[*x]` is stored in `delta_ret`.
- l. 31: A transition from `*x` to `*y` is created, with label `*a`.

3. Design for genericity

The facilities exposed in the previous sections are not present in the standard C++. The kernel of VAUCANSON is a software layer that yields an abstraction level powerful enough for genericity. Then polymorphism has been implemented in a way such that this abstraction level does not spoil efficiency.

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 for this purpose is presented and constitutes the contribution in the generic programming field.

3.1. A unified generic framework

The VAUCANSON kernel consists in a typing system and a object-oriented layer. The design arguments are given in Section 3.2.

3.1.1. The VAUCANSON *typing system*

A typing system is meant to forbid the programmer to do invalid operations between incompatible values. In the object-oriented field, the point is to determine, if it exists, the most precise method to call w.r.t the types of object instances that receive a particular message [3]. Therefore, one of the goals of typing is to retrieve the most precise information about the variables manipulated by the programmer.

The VAUCANSON type system has been designed to manage moreover structures whose exact type depends on parameters that are known only at run-time. Let us consider for instance the definition of the scalar product between two *n-dimensional vectors*. In a static type system, if the dimension of the vectors is not known at compile-time, the programmer is forced to relax the input specification using a less precise type *vector* denoting any vector. From then on, the dimension of the vector is implemented as a *data* not as a *part of the type*. If nothing is provided by the system, the type checking is done manually by the programmer (or not done). Dependent type systems [23,2] are intended to overcome this defect and carry dynamic information (i.e. values) into types. By that way, types may depend on computed values. The VAUCANSON typing system is as an ad hoc implementation of a dependent type system.

In VAUCANSON, there are three categories of entity: pre-types, types and elements. A *pre-type* denotes a static information, that is a property known at compile-time. A *type* is a pre-type completed with dynamic information, that is values known at run-time. An *element* is a variable whose type is a VAUCANSON type. For instance, *vector of integers* is a pre-type, *n-dimensional vector of integers* is a type, a n-tuple of integers is an element of type *n-dimensional vector of integers*; *free monoid* is a pre-type, A^* is a type (free monoid over the alphabet A), a sequence of letters of A is an element of type A^*.

A very important feature of VAUCANSON typing system is that an element is not characterized only by its VAUCANSON type but also by the way it is implemented.

Section 3.1.2 describes how these different entities are emulated in C++. Sections 3.1.3 and 3.2 show how to take benefit of this type system to enhance genericity and algorithm input specification. Sections 3.1.4 and 3.1.5 discuss the implementation design.

VAUCANSON	C++	Examples
Pre-types	Classes	`FreeMonoid, Matrix`
Types	Pre-types instances.	`FreeMonoid Astar(A);` `Matrix matrix2(2);`
Elements	(s, t), Type : s Value : t	`Element<FreeMonoid, string> w(Astar);` `Element<Matrix, int**> m(matrix2);`

Fig. 5. The VAUCANSON type system.

3.1.2. Embedding in C++

As Pascal or C, C++ has a static type system. This means that types are not directly accessible at run-time. The specification of types during evaluation is made possible by using some C++-variables. VAUCANSON provides a specific hierarchy of C++-classes for that purpose; these classes are the VAUCANSON pre-types. A C++-*variable* of such class when instantiated at run-time by some dynamic information becomes a VAUCANSON *type*. This is summarized in the first two lines of Fig. 5.

The last line shows the two components that characterize a VAUCANSON element `x`. The C++-type of `x` is an instance of *the parametrized class* `Element`; the parameters are the static information on the element `x`, that is on one hand the VAUCANSON pre-type and on the other hand the C++-type of its implementation. The instantiation of `x` requires as a constructor parameter the value of the C++-variable that represent the VAUCANSON type of `x`.

Concretely, the programmer has to declare a C++ instance of a pre-type, to complete it with a dynamic value if necessary so as to obtain a type. It is then possible to declare some variables over this type. For instance, a natural declaration of two words w_1 and w_2 over the alphabet $\{a, b\}$ is:

"let A be the alphabet $\{a, b\}$ and let w_1, w_2 be two elements of type $A^\star$".

This statement becomes the following VAUCANSON program which defines two words of the free monoid over the alphabet $\{a, b\}$ implemented in two different ways:

```
// The variable 'A' is a value denoting {a, b}.
alphabet_t A;
A.insert('a');  A.insert('b');
// The variable Astar is a Vaucanson type denoting {a,b}*.
// FreeMonoid is a pre-type which must be
// completed by an alphabet value.
FreeMonoid Astar(A);
// The variables w_1 and w_2 are of Vaucanson type Astar.
// Their C++-type is the class Element instantiated with
// the FreeMonoid pre-type and two different implementations.
Element<FreeMonoid, std::string> w_1(Astar);
Element<FreeMonoid, const char*> w_2(Astar);
```

Once the variables `w_1` and `w_2` are declared, they can take a value; it can be noticed that there is an implicit conversion of a value of type T into an `Element` whose implementation type is T.

```
w_1 = "aa";  w_2 = "ab";
// The two variables have the same type: they can be
// compared.
return w1 == w2;
```

3.1.3. Writing generic algorithms within this framework

Given a VAUCANSON type `S`, an element of type `S` has a well-defined interface whatever its implementation: this is the basis of genericity. 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>);
```

Besides, the implementation parameter allows a choice between different algorithm versions depending on the underlying data structure. For example, an element of series can be implemented as a rational expression (i.e. a tree) or, if its support is finite, as a finite map. The constant term is computed differently according to the chosen implementation. More subtle specifications can be done and are described in Section 3.2.2.

3.1.4. Implementation definition

The data structure benefits are application dependant (from a time or space complexity point of view) and their choices should be done independantly from the algorithm that is used. Even if some algorithms may be *specialized* to take account of a particular feature of a data structure (see Section 3.2.2), general algorithms are written using general interfaces. This policy of encapsulation induces the reusability of code.

In VAUCANSON, an implementation is adapted to the VAUCANSON type requirements using binding operators. For example, the adaptation of the C++ integer type as an element of the $(\mathbb{Z}, \max, +)$ semiring consists in the definition of the binding operators `op_add`, `op_mul`, `op_identity` and `op_zero`. Most of the binding operators provide default behaviour based on assumption about implementation. Then, sometimes, the adaptation of an implementation can be done without any binding code, for instance, to define the semiring $(\mathbb{Z}, +, \times)$ implemented by C++ integers. Thanks to binding operators, implementation are not necessarily C++ classes. They can be C++ *builtins* or C structures from foreign libraries.

3.1.5. Object oriented layer

The previous sections have described the kernel of VAUCANSON. To simplify the basic usage of the library, a layer composed of shortcuts for object construction is provided. Moreover, the richness of the object services is as important as the generality of the type system to make the writing of algorithm easier. We illustrate what we call a "rich" service by describing how the δ function is declared in VAUCANSON.

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 VAUCANSON type. 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.

First, δ functions allow to handle either states (successors or predecessors) or transitions (outgoing or incoming ones).
Second, it is possible to choose the way the result is stored: containers, output iterator or read-only access begin/end iterator couple.
Finally, a criterion can be given to describe which outgoing (or incoming) transitions have to be considered: for instance, every outgoing transition, transitions labelled by a given letter, or any user condition.

```
// Store the output edges of ’s’ w.r.t the letter ’a’ in
// the list ’l’.
a.letter_deldac(l, s, ’a’, delta_kind::edges());
// Store the successors of ’s’ in the bitset ’b’.
a.deltac(b, s, delta_kind::states());
// Retrieve incoming transitions of ’s’ whose label is a
// monome.
// rdeltac is the reverse transition function.
a.rdeltac(l, s, is_a_monome, delta_kind::edges());
```

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

3.2. Polymorphism using C++ templates

Object-Oriented languages enable reusability based on contracts defined by abstract classes. Indeed, abstract classes define abstract services that can be expected from concrete classes instance. The choice of the concrete classes to instantiate is done at run-time and this implies that the abstract services calls are resolved at run-time too.

Yet, in practice, this late binding to abstract services is too expensive and leads too bad performance for intensive computing mainly because it breaks a potential code inlining. The generative power of C++ template allows the static resolution of abstract services by closing the object recursivity w.r.t. the self type. This ability, illustrated by the STL, allows to write high-level C++ programs whose speed is comparable to dedicated low-level C programs.

3.2.1. STL approach

As mentioned in [19], 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 parametrized by a free type variable `C`. The code is written assuming that an instance of `C` will be a container.

Yet, parametrization 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.

```
template <class BooleanAutomaton>
void minimization(const BooleanAutomaton& automaton);

template <class WeightedAutomaton>
void minimization(const WeightedAutomaton& automaton);

// BooleanAutomaton and WeightedAutomaton are mute variables,
// so the following function call is ambiguous:
automaton_t a;
minimization(a);
```

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 [8,6]. These programming methods entail a stronger typing, which enables a finer specialization power and solves the overloading problem.

3.2.2. Beyond classical use of templates

One classical object hierarchy is not enough to obtain extensibility and specialization power in our framework. The current section will describe a new design pattern we developed to allow a higher 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 two components to characterize an object. We notice that abstraction and implementation are quite orthogonal for at least two reasons. First, the writing of a general algorithm focusses on the mathematical concept (the general interface of any VAUCANSON variable of a particular VAUCANSON type). Implementation constraints are taken into account afterwards. Second, algorithm specialization should depend on implementation and on VAUCANSON type 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 [9] or more recently the GENERIC BRIDGE [7] (Figs. 6 and 7).

However, there remain two problems for us from a specialization point of view. First, these design patterns are asymmetric, privileging concept upon implementation. Then, we cannot

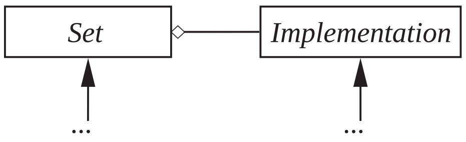

Fig. 6. UML diagram of the Bridge design pattern.

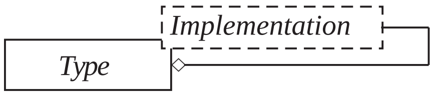

Fig. 7. UML diagram of the Generic Bridge design pattern.

define an algorithm that only works for a particular set of implementations whatever the concept. Moreover, a concept cannot be manipulated independantly from the implementation; therefore it is difficult to compare two concepts simply for equality or for subsumption.

Second, it does not allow subclassing w.r.t the two parameters because template arguments are invariant. In the following example, an element of series cannot be passed to the function `is_zero` even if the `Series` class inherits from the `Semiring` class.

```
template <class T>
bool is_zero(const Element<Semiring, T>& e)
{
  return e.set().zero() == e;
}

// e must be a weight.
is_zero(e);
```

3.2.3. *The* Element/MetaElement *design pattern*

To solve all these problems, the Vaucanson library uses a new design pattern which we have called Element/MetaElement [20]. The main idea is to enable de-construction of an object w.r.t its two components and to use them for typing purpose. For instance, the Vaucanson type of the object can be used as an argument to make the function signature more precise, this feature can be applied in the previous example

```
// Specialization of type 4.
template <class S, class T>
bool is_zero(const Semiring& s, const Element<S, T>& e)
{
  return e == s.zero();
}

// e can be a semiring weight but also a series.
is_zero(e.type(), e);
```

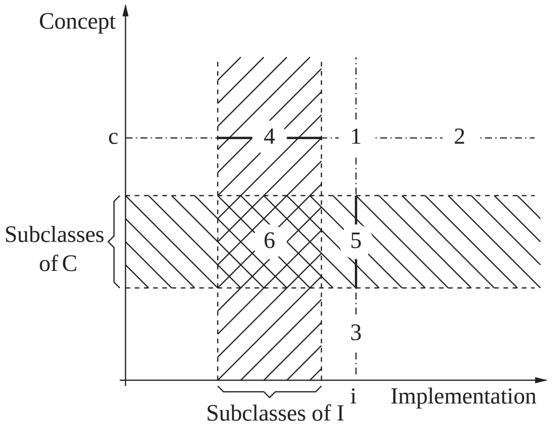

Fig. 8. Different type boundings of algorithm input. (1) i, c are fixed; (2) all implementations and c is fixed; (3) all concepts and i is fixed; (4) all sub-classes of I and c is fixed; (5) all sub-classes of C and i is fixed; (6) all sub-classes of C and I.

As another example, the following piece of code shows the procedure signature for the determinization algorithm specialized to any table-based automaton implementation:

```
// Specialization of type 6.
template <class S, class T>
Element<S, T> determinize(const Automata& s,
                          const Table& i,
                          const Element<S, T>& a)
{
  // ...
}

// The algorithm call takes the form:
determinize(a.type(), a.value(), a);
```

Fig. 8 sums up the different kinds of specialization that are useful in VAUCANSON. Each specialization kind corresponds to a boundary of the input types into the type domain.

More generally, the specifications of Fig. 8 are expressible:

```
// Type 1: the concept and value type are fixed.
void algorithm_impl(const S1& s, const T1& v,
                    const Element<S1, T1>& e);

// Type 2: concept fixed, generic implementation for any
// value type.
template <class T>
```

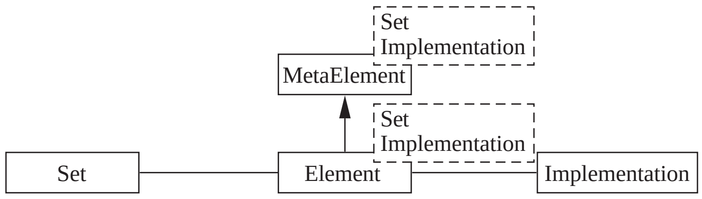

Fig. 9. UML diagram of the ELEMENT/METAELEMENT design pattern.

```
void algorithm_impl(const S1& s, const T& v,
                    const Element<S1, T>& e);

// Type 3: value type fixed, generic implementation for any
// concept.
template <class S>
void algorithm_impl(const S& s, const T1& v,
                    const Element<S, T1>& e);

// Type 4: generic implementation for any sub-concept of S1.
template <class S, class T>
void algorithm_impl(const S1& s, const T& v,
                    const Element<S, T>& e);

// Type 5: generic implementation for any value sub-class
// of T1.
template <class S, class T>
void algorithm_impl(const S& s, const T1& v,
                    const Element<S, T>& e);

// Type 6: generic implementation for any sub-class
// of (S1,T1).
template <class S, class T>
void algorithm_impl(const S1& s, const T1& v,
                    const Element<S, T>& e);
```

`Element` is a generic class associating a VAUCANSON type and an implementation. The role of MetaElement is to define the interaction between these two components that is, *how the data structure implements the* VAUCANSON *type*. A kind of multi-methods with static dispatch (the binding operators) is also used to allow default implementation and specialization of n-ary methods. The pattern is illustrated in Fig. 9 using the Unified Modelling Language. Its effective implementation involves some C++ meta-programming techniques which will not be explicited in this paper. For further technical information, the interested reader is referred to [20].

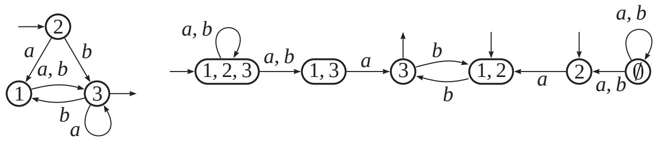

Fig. 10. The minimal automaton $\mathcal{A}_1$ and the co-determinized automaton of $\mathcal{A}_1$.

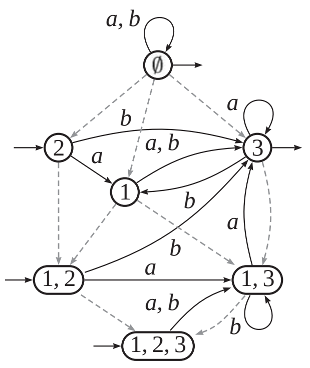

Fig. 11. The universal automaton of $L(\mathcal{A}_1)$. The dashed lines are ε-transitions. The universal automaton $\mathcal{U}_1$ of $L(\mathcal{A}_1)$ is given by the forward closure of this automaton: there is a transition (p, a, q) in $\mathcal{U}_1$ if, on the figure, there are a state r, a transition (p, a, r) and a path of ε-transitions from r to q.

Appendix A. On the universal automaton of a language

The universal automaton $\mathcal{U}_L$ of a rational (regular) language L is an automaton canonically attached to L. One can consider that it is a slight transformation of the "factor matrix" introduced by Conway in [5] in order to solve some types of language equations. This automaton $\mathcal{U}_L$ can also be used to find the smallest NFA that accepts L (cf. [1,17]), or—as was done by two of the authors—to study some properties of L (e.g. star height [15,13] or reversibility [12]) at least when L belongs to some subfamilies of the rational languages.

The automaton $\mathcal{U}_L$ is the smallest automaton such that there is a *morphism* from any automaton that accepts the language L into $\mathcal{U}_L$. This property characterizes it but is not constructive.

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

The automaton can be built without computing the syntactic monoid. Actually, every state p of the minimal automaton $\mathcal{A}$ corresponds to a (non necessarily maximal) factorization (H_p, K_p), where H_p is the set of words that label a path from the initial state of $\mathcal{A}$ to p

and K_p is the set of words that label a path from p to any terminal state of $\mathcal{A}$. It holds (cf. [12,21]) that every maximal factorization is a combination of these basic factorizations; more precisely, for every maximal factorization (H, K), there exists a subset Q' of states of $\mathcal{A}$ such that $H = \bigcup_{p \in Q'} H_p$ and $K = \bigcap_{p \in Q'} K_p$. More, the subsets Q' that give exactly all the maximal factorizations of the language are the intersections of the states of the co-determinized automaton of $\mathcal{A}$. The initial and terminal states and the transitions are then given by the following rules:
– P is initial if it contains the initial state of the minimal automaton $\mathcal{A}$,
– P is terminal if it is a subset of the set of terminal states of $\mathcal{A}$,
– (P, a, Q) is a transition if, for every p in P, there is a transition (p, a, q) in $\mathcal{A}$ such that q is in Q.
The co-determinized automaton of the automaton $\mathcal{A}_1$ (Fig. 10) is drawn on the same figure. The states of the co-determinized automaton are $\emptyset$, $\{2\}$, $\{3\}$, $\{1, 3\}$, $\{1, 2\}$ and $\{1, 2, 3\}$. The closure of this set under intersection contains one more set: $\{1\}$. Fig. 11 shows the resulting universal automaton.

References

[1] A. Arnold, A. Dicky, M. Nivat, A note about minimal non-deterministic automata, Bull. EATCS 47 (1992) 166–169.
[2] L. Augustsson, Cayenne a language with dependent types, Internat. Conf. on Functional Programming, 1998.
[3] G. Castagna, Object-Oriented Programming: A Unified Foundation, Progress in Theoretical Computer Science Series, Birkhauser, Basel, 1997.
[4] J.-M. Champarnaud, D. Ziadi, Canonical derivatives, partial derivatives and finite automaton constructions, Theoret. Comput. Sci. 289 (1) (2002) 137–163.
[5] J.H. Conway, Regular Algebra and Finite Machines, Chapman and Hall, London, 1971.
[6] J. Darbon, T. Géraud, A. Duret-Lutz, Generic implementation of morphological image operators, Internat. Symp. on Mathematical Morphology VI (ISMM’2002), April 2002, pp. 175–184.
[7] A. Duret-Lutz, T. Géraud, A. Demaille, Design patterns for generic programming in C++, Proc. Sixth USENIX Conf. on Object-Oriented Technologies and Systems (COOTS’01), USENIX Association, 2001, pp. 189–202.
[8] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, S. Schönherr, On the design of CGAL, the computational geometry algorithms library, Technical Report 3407, INRIA, April 1998.
[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Abstraction and Reuse of Object-Oriented Design, Lecture Notes in Computer Science, Vol. 707, 1993, pp. 406–431.
[10] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading, MA, 1979.
[11] V. Le Maout, Cursors, Proc. of CIAA 2000, Lecture Notes in Computer Science, Vol. 2088, 2001, pp. 195–207.
[12] S. Lombardy, On the construction of reversible automata for reversible languages, Proc. of ICALP’02, Lecture Notes in Computer Science, Vol. 2380, 2002, pp. 170–182.
[13] S. Lombardy, J. Sakarovitch, Star height of reversible languages and universal automata, Proc. of LATIN’02, Lecture Notes in Computer Science, Vol. 2286, 2002, pp. 76–89.
[14] S. Lombardy, J. Sakarovitch, Vaucanson–G, A package for drawing automata and graphs, `http://www.liafa.jussieu.fr/~lombardy/Vaucanson-G/`, 2002.
[15] S. Lombardy, J. Sakarovitch, On the star height of rational languages, a new presentation for two old results, in: M. Ito, T. Imaoka (Eds.), Proc. of Words, Languages & Combinatorics III, World Scientific, Singapore, 2003, pp. 266–285.
[16] O. Matz, A. Miller, A. Potthoff, W. Thomas, E. Valkema, The Program AMoRE, `http://www-i7.informatik.rwth-aachen.de/d/research/amore.html`, RWTH Aachen, 1995.

[17] O. Matz, A. Potthoff, Computing small nondeterministic finite automata, Proc. of TACAS'95, BRICS Notes Series, 1995, pp. 74–88.
[18] M. Mohri, F.C.N. Pereira, M. Riley, General-purpose Finite-State Machine Software Tools, `http://www.research.att.com/sw/tools/fsm/`, AT&T Labs—Research, 1997.
[19] D.R. Musser, A.A. Stepanov, Algorithm-oriented generic libraries, Software Pract. Exper. 24 (7) (1994) 623–642.
[20] Y. Régis-Gianas, R. Poss, On orthogonal specialization in C++: dealing with efficiency and algebraic abstraction in Vaucanson, Proc. of POOSC'2003, Darmstadt, July 2003.
[21] J. Sakarovitch, Éléments de théorie des automates, Vuibert, 2003. English translation: Element of Automata Theory, Cambridge University Press, to appear.
[22] G. van Noord, Finite State Automata Utilities, `http://odur.let.rug.nl/~vannoord/Fsa/`, 2000.
[23] H. Xi, F. Pfenning, Dependent types in practical programming, Proc. of POPL'1999, 1999, pp. 214–227.