Inside Vaucanson

Thomas Claveirole[1], Sylvain Lombardy[2], Sarah 0'Connor[1], Louis-Noël Pouchet[1], and Jacques Sakarovitch[3]

[1] LRDE, EPITA, {claveirole,o-connor,pouchet}@lrde.epita.fr
[2] LIAFA, Université Paris 7, lombardy@liafa.jussieu.fr
[3] LTCI, CNRS / ENST , sakarovitch@enst.fr

Abstract. This paper presents some features of the VAUCANSON platform. We describe some original algorithms on weighted automata and transducers (computation of the quotient, conversion of a regular expression into a weighted automaton, and composition). We explain how complex declarations due to the generic programming are masked from the user and finally we present a proposal for an XML format that allows implicit descriptions for simple types of automata.

1 Introduction

At CIAA'03, we had announced our project VAUCANSON, a software platform for computing with automata and transducers (see [**?**]). We have made some demonstration of the possibilities of VAUCANSON at CIAA'04. We would like to report now on how some features of VAUCANSON have been implemented at the light of the first years of experiments. This applies to the algorithms as well as to the programming facilities that had to be incorporated within VAUCANSON.

We first describe three of the algorithms implemented in VAUCANSON: those which generalize to automata with multiplicity the Hopcroft algorithm of minimization, the construction of the derived term automaton of a regular expression and the composition of (sub-)normalized transducers.

We then explain how we have overcome the intrinsic difficulty of generic static programming. And we finally introduce the last version of an XML format to describe automata, implemented as input-output in VAUCANSON. In particular, VAUCANSON is complemented with a model of a graphical interface, which relies on the XML format to interact with the VAUCANSON library.

The description of the algorithms is complemented with the results of some benchmarks of the last version of VAUCANSON [1]. All the tests have been run on a server installed at ENST, a bi-Xeon 3.2 GHz with 4 Go of RAM.

2 On the algorithms

The VAUCANSON platform provides the most usual algorithms on automata: determinization, minimization, product, Thompson automaton of an expression,

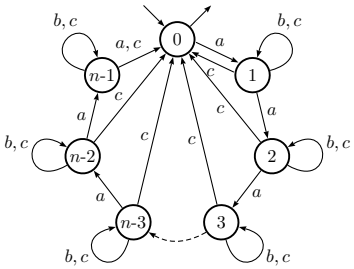
[1] Downloadable at http://vaucanson.lrde.epita.fr.

standard automaton, ε-transitions removal, *etc.* Each of these algorithms has been written such that it can be applied to the largest range of automata. For instance, product or ε-transitions removal are generic and can be applied to automata with any multiplicity.

As an example, let us mention the automaton $\mathcal{A}_n$, drawn below, that had been used in [**?**] for benchmarking the determinization and that will serve as the basis of other tests in this paper.

On the latest version of VAUCANSON the determinization test gives the following result for $\mathcal{A}_{20}$ (the minimal deterministic automaton equivalent to $\mathcal{A}_n$ has 2^n states).

Platform	time (seconds)	space (MB)
FSM	60	447
VAUCANSON	105	1709



We focus now on three algorithms that are extensions for automata with multiplicities of more or less well-known algorithms: a minimization algorithm adapted from Hopcroft algorithm, a rather sophisticated algorithm for building an expression, adapted from a method due to Champarnaud and Ziadi [**?**], and finally an algorithm for the composition of transducers with multiplicity.

2.1 Minimal quotient

Definition of the minimal quotient The notion of minimal quotient (or $\mathbb{K}$-covering) is the generalization to weighted automata of the minimal automaton for DFA's. It consists in computing a (smaller) automaton by merging states which have the "same" outgoing transitions. (*cf.* [**?**,**?**]).
More formally, this definition is equivalent to the following one (straightforward from [**?**]). Let $\mathcal{A} = (I, E, T)$ be an automaton characterized by its initial vector I, its transition matrix E and its final vector T. The minimal quotient of $\mathcal{B}$ is the (unique) smallest automaton $\mathcal{B} = (J, F, U)$ such that:

$$J = IK, \qquad KF = EK, \qquad \text{and} \qquad KU = T,$$

where K is an amalgamation matrix (*i.e.* with one and only one non-zero coefficient, equal to 1, on every row). It is quite obvious that the minimal quotient of $\mathcal{A}$ is equivalent to $\mathcal{A}$.

Algorithm for the minimal quotient The minimal quotient of a DFA can be computed either by the Moore algorithm or by the Hopcroft algorithm [**?**].[2] These both algorithms consist in refining a partition over states (initialized w.r.t. the terminal states) but they are different: given a class P and a letter a, the

[2] The Brzozowski algorithm is not a computation of quotient, even if it gives the same result on DFAs.

Moore algorithm consists in considering the classes of successors of P by a and splitting P, whereas the Hopcroft algorithm consists in considering the classes of predecessors of P by a and splitting them. Therefore, the Hopcroft algorithm can more directly be extended to NFAs or weighted automata:

1. Computation of a backward transition table: for every state q and every letter a, the list of pairs (p, w) is stored, for all transitions from p to q labelled by a with multiplicity w. The current implementation of automata in VAUCANSON already provides this table.
2. The algorithm is initialized by sorting states with respect to their terminal function. This provides the initial partition that has to be refined to be a right congruence. For every part P and every letter a, the pair (P, a) is inserted into a queue l.
3. While l is not empty, the front of l, a pair (P, a) is poped up; for every part Q which has successors by a in P, the part Q is splitted such that two states q and q' remain in the same part if and only if the sum of weights of their outgoing transitions labelled by a that arrive in P are equal. If new parts are created, they are inserted into l (paired with every letter).
4. An automaton whose states are the parts is then created. The terminal function of the part is the (common) terminal function of each of its states, the initial function is the sum, and the multiplicity of (P, a, Q) is the sum, for any state p of P of the multiplicities of transitions (p, a, q) for every q in Q.

Comparison with the classical Hopcroft algorithm The principle of the algorithm is the same as the principle of the minimization algorithm: a backward transition table is computed and a partition is refined by considering predecessors of each part. Nevertheless, the existence of multiplicities has a number of outcomes in every step of the algorithm.

First, the transition table does not contain lists of states, but lists of states paired with weights.

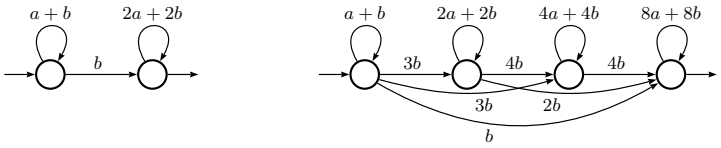
In step 2, the initial partition has as many parts as the terminal function has values. In step 3, it is not sufficient to know wether a state q has a successor in P but which is the multiplicity from q to P. Moreover, one add to l every subpart obtained from Q whereas in the Hopcroft algorithm the smallest among both subparts is inserted, which is crucial to reach the $n \log n$ complexity in the classical case. The complexity of that generalized Hopcroft algorithm is therefore more likely to be quadratic.

Minimizing deterministic automata Minimization of DFAs is a special case of computing the quotient and we have first tested implementations in this case to allow comparison with other platforms. It has been shown in [**?**] that the de Brujin graph, $\mathcal{B}_n$ is the worst case for Hopcroft minimization algorithm, since it can lead to $n2^n$ steps in the main loop of the algorithm: let $A = \{a, b\}$ be the alphabet; the states of $\mathcal{B}_n$ are labelled by words of length n over A, that gives 2^n states. The state a^n is initial, for every x, y in A, every word w in A^{n-1}

there is a transition from xw to wy labelled by y and xw is final if and only if $x = a$. We also test minimization on the the determinized automaton of $\mathcal{A}_n$. Four procedures are tested; the one proposed by FSM (unknown algorithm), and two algorithms proposed by VAUCANSON: Moore and Hopcroft.[3]

Input	$\mathcal{B}_{12}$	$\mathcal{B}_{17}$	$det(\mathcal{A}_{12})$	$det(\mathcal{A}_{17})$
FSM	0.048	1.791	0.065	2.829
Moore	0.271	37.11	0.470	146.77
Hop.	0.074	45.599	0.338	1752.33

Test of the generalized quotient Let C_1 be the $\mathbb{Z}$-automaton of Figure 1 that maps every word w on $\{a, b\}^2$, seen as a binary number, on its value $\bar{w}$. For every positive integer n, let C_{n+1} be the automaton recursively defined as the product of C_1 by C_n. C_n maps every words w on $\bar{w}^n$. In the following tests, we compute C_n, which has 2^n states and then the minimal quotient $\mathcal{V}_n$ of C_n which has $n + 1$ states.



(a) The automaton C_1 (b) The quotient of C_3

Fig. 1. The automata C_n

n	8	9	10	11	12
$\mathcal{C}_n$ edges	6817	20195	60073	179195	535537
$\mathcal{V}_n$ edges	45	55	66	78	91
Time	0.036	0.112	0.340	0.999	3.161

2.2 Automaton of derived terms

The VAUCANSON platform provides several algorithms to convert any regular expression (with multiplicity) into a (weighted) finite automaton. We present here the algorithm that constructs the derived term automaton $\mathcal{A}_{\mathsf{E}}$ of an expression E [?,?]. This automaton is rather small: it has been proven that $\mathcal{A}_{\mathsf{E}}$ is a quotient of the standard (or position, or Glushkov) automaton of E [?,?].

We have implemented the algorithm and the data structure proposed by Champarnaud and Ziadi [?] for the computation of derived terms, together with the necessary improvement in order to deal with multiplicity in expressions. The main point proven in [?] is that every derived term of a regular expression E is a product of subexpressions of E. Therefore, each derived term is represented by a list of nodes in the tree of the regular expression E. Moreover, this tree is equipped with some "links" that help to perform the derivation: for every $*$-node n, there is a link from the child of n to n itself, and for every $\cdot$-node, there is a link from the left child of n to its right child.

[3] The Brzozowski algorithm that consists in applying a co-determinization followed by a determinization does not succeed in reasonable time on these inputs.

Two basic functions are $\mathsf{c}(\mathsf{E})$ and $\mathit{first}(\mathsf{E}, a)$. The function $\mathsf{c}(\mathsf{E})$ gives the weight of the empty word in the series described by E and $\mathit{first}(\mathsf{E}, a)$ returns a set of pairs weight/position recursively defined by:

$$\mathit{first}(\mathsf{0}, a) = \emptyset\,, \quad \mathit{first}(\mathsf{1}, a) = \emptyset\,,$$

$$\forall b \in A \qquad \mathit{first}(b, a) = \begin{cases} (1_{\mathbb{K}}, y) & \text{with } y \text{ position of } a \text{ in } \mathsf{E}, \text{ if } a = b, \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{first}(k\,\mathsf{E}, a) = \bigcup\{(kx, y) \mid (x, y) \in \mathit{first}(\mathsf{E}, a)\}\,, \quad \mathit{first}(\mathsf{E}\,k, a) = \mathit{first}(\mathsf{E}, a)\,,$$

$$\mathit{first}(\mathsf{E} + \mathsf{F}, a) = \mathit{first}(\mathsf{E}, a) \cup \mathit{first}(\mathsf{F}, a)\,, \quad \mathit{first}(\mathsf{E} \cdot \mathsf{F}, a) = \mathit{first}(\mathsf{E}, a) \cup \mathit{first}(\mathsf{c}(\mathsf{E})\mathsf{F}, a)\,,$$

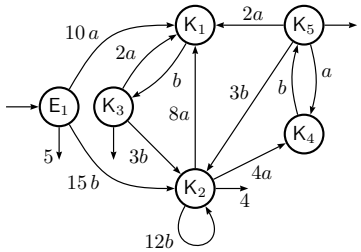
$$\mathit{first}(\mathsf{E}^*, a) = \mathit{first}(\mathsf{c}(\mathsf{E})^*\mathsf{E}, a)\,, \text{ if } \mathsf{c}(\mathsf{E})^* \text{ is defined in } \mathbb{K}.$$

These functions are easily computed on the tree of the regular expression. The computation of the derivatives of E with respect to a consists, for every position x in $\mathit{first}(\mathsf{E}, a)$, to go up to the root of the tree of E and collect the destinations of the links starting from the nodes on that path.

Example 1. Let $\mathsf{E}_1 = (5\,\mathsf{F}_1)$ with $\mathsf{F}_1 = ((2ab) + ((3b) \cdot (4(ab)^*)))^*$.

The derived terms of E_1 are:

$$\begin{aligned} \mathsf{K}_1 &= b \cdot \mathsf{F}_1, \\ \mathsf{K}_2 &= (4\,(ab)^*) \cdot \mathsf{F}_1, \\ \mathsf{K}_3 &= \mathsf{F}_1, \\ \mathsf{K}_4 &= (b \cdot (ab)^*) \cdot \mathsf{F}_1, \text{ and} \\ \mathsf{K}_5 &= (ab)^* \cdot \mathsf{F}_1. \end{aligned}$$



The automaton $\mathcal{A}_{\mathsf{E}_1}$.

For instance, $\frac{\partial}{\partial a}\,\mathsf{K}_2 = 8\mathsf{K}_1 \oplus 4\mathsf{K}_4$.

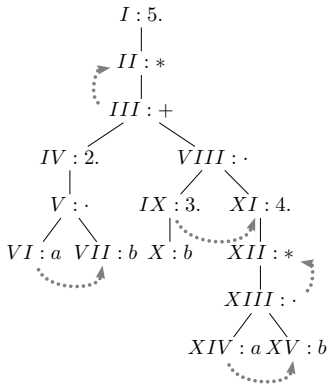
The tree of E_1 is equipped with links. The coding for E_1 itself is I.

One computes $\mathit{first}(I, a) = \{(10, VI)\}$ and going up from VI, one get:

$$\frac{\partial}{\partial a}\,\mathsf{E}_1 = 10[VII, II].$$

Finally, we get the same automaton with the following coding for derived terms:

$$\begin{array}{lll} \mathsf{K}_0 = [I] & \mathsf{K}_1 = [VII, II] & \mathsf{K}_2 = [XI, II] \\ \mathsf{K}_3 = [II] & \mathsf{K}_4 = [XV, XII, II] & \mathsf{K}_5 = [XII, II] \end{array}$$



For instance,
$\mathit{first}([XI, II], a) = \{(4, XIV), (8, VI)\}$
and $\frac{\partial}{\partial a}[XI, II] = 4[XV, XII, II] \oplus 8[VII, II]$.

Test on the derived terms A set of expressions is provided by the elimination algorithm applied on $\mathcal{A}_{15}$ with random orderings on states. The automaton of derived terms $\mathcal{A}_{\mathsf{E}}$ of every expression E is computed, and also the quotient $\mathcal{V}_{\mathsf{E}}$ of

the standard (Glushkov) automaton of the expression E. One thousand expressions are generated this way and classified w.r.t. their litteral length l_{E}– which is the size of $\mathcal{S}_{\mathsf{E}}$. We present here means for four significant classes.

Class	l_{E}	Derived term $\mathcal{A}_{\mathsf{E}}$		Standard $\mathcal{S}_{\mathsf{E}}$	
		$\mathcal{A}_{\mathsf{E}}$ states	time	$\mathcal{V}_{\mathsf{E}}$ states	time
1	110	24	0.123	24	0.012
7	410	53	0.470	51	0.050
14	1035	66	1.169	60	0.138
20	7821	90	13.412	78	1.418

2.3 Composition of transducers with multiplicity

A most fundamental result in the theory of transducers is Elgot and Mezei's Composition Theorem ([**?**]): *The composition of two finite transducers is realized by a finite transducer.* The same result holds true for weighted transducers — up to some definition problems which will not be considered here. The proof is, or can be translated into, an algorithm for the construction of the transducer that realizes the composition. And there are two main proofs for the Composition Theorem.

The first proof follows from Kleene-Schützenberger characterization of rational relations from A^* into B^* as *recognizable series* on A^* with multiplicity in $\mathrm{Rat}\, B^*$. Transducers are thus *representations* of A^* by matrices with entries in $\mathrm{Rat}\, B^*$ and representations can be *composed* in a natural way: this yields a representation for the composition of transducers [**?**]. This proof has the advantage that it generalizes directly to weighted transducers: they are representations by matrices with entries in $\mathbb{K}\mathrm{Rat}\, B^*$ if $\mathbb{K}$ is the multiplicity semiring. It is thus perfectly "generic" *i.e.* independent from the type of considered transducers and hence fits well with the architecture of VAUCANSON. It is the one we have first implemented. Besides its genericity, this algorithm has a serious drawback: as it deals with *real-time* transducers, the transition "outputs" may be regular expressions and the composition requires the computation of the image (by the second transducer) of all these expressions, a computation that may prove to be costly.

The other proof, certainly better known, relies on the realization of rational relations by projections and intersection with rational (regular) languages (see [**?**,**?**]). We have also implemented *another composition algorithm* which follows more closely this classical proof and which works directly on transducers seen as labeled graphs.

Let us first sketch quickly an algorithm that corresponds to that proof in the unweighted case. In spite of its simplicity, it has not been described so often; it can be seen as a simplified version of the algorithm for the weighted case of [**?**,**?**] which we shall mention again later. It can be also found in [**?**].

We consider two *normalized* transducers $\mathcal{T} = \langle Q, A^* \times B^*, E, I, T \rangle$ and $\mathcal{U} = \langle R, B^* \times C^*, F, J, U \rangle$, that is transitions of $\mathcal{T}$ are labeled in $A \times 1$ or in

$1\times B$ and those of $\mathcal{U}$ are labeled in $B\times 1$ or in $1\times C$. The proof of the Composition Theorem as presented in [**?**] is equivalent to the construction of the transducer

$$\mathcal{T} \bowtie \mathcal{U} = \langle Q \times R, A^* \times C^*, G, I \times J, T \times U \rangle$$

by the following rules.
(i) If $(p,(a,1),q) \in E$ then for all $r \in R$ $\big((p,r),(a,1),(q,r)\big) \in G$.
(ii) If $(r,(1,c),s) \in F$ then for all $q \in Q$ $\big((q,r),(1,c),(q,s)\big) \in G$.
(iii) If $(p,(1,b),q) \in E$ *and* $(r,(b,1),s) \in F$ then $\big((p,r),(1,1),(q,s)\big) \in G$.

A next possible step is to eliminate the transitions with label $(1,1)$ by means of a classical closure algorithm.

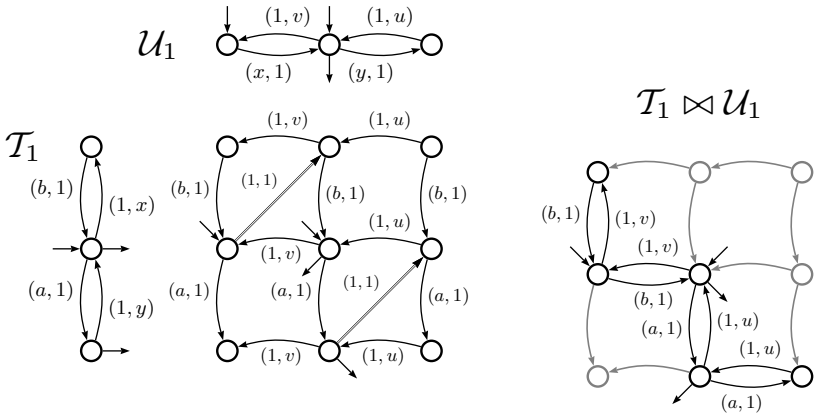


Fig. 2. Composition Theorem on Boolean transducers

This construction can easily be extended to transducers which we shall call *sub-normalized* and which are such that transitions are labeled in $\widehat{A}\times\widehat{B}\setminus(1,1)$ where $\widehat{A} = A \cup \{1\}$. It amounts to replace (iii) by
(iii') If $(p,(x,b),q) \in E$ with $x \in \widehat{A}$ *and* $(r,(b,y),s) \in F$ with $y \in \widehat{C}$ then $\big((p,r),(x,y),(q,s)\big) \in G$.

In this form, it contains as a particular case the composition of letter-to-letter transducers.

It is known that this construction is not correct if multiplicities are to be taken into account. Let us say that two paths in $\mathcal{T} \bowtie \mathcal{U}$ are *equivalent* if they correspond to the same pair of paths in $\mathcal{T}$ and $\mathcal{U}$. For instance, there is one path labeled (aa,y) in $\mathcal{T}_1$ and one path labeled (y,u) in $\mathcal{U}_1$; and there are *two* equivalent paths labeled (aa,u) in $\mathcal{T}_1 \bowtie \mathcal{U}_1$. Hence, $\mathcal{T} \bowtie \mathcal{U}$ does not realize the composition of the weighted relations realized by $\mathcal{T}$ and $\mathcal{U}$.

In [**?**], the Composition Theorem is proved for weighted transductions (at least for those with weights taken in a *complete positive and commutative semiring*, which allows to dispose of the question of definition). In this proof, the

multiplicity, that is the selection among the equivalent paths, is taken care of, so to speak, by the intersection with a certain local language T.

As we already mentioned, a construction of a weighted transducer that realizes the composition of two weighted transductions is given in [**?**,**?**]. It amounts first to *mark* the transitions which, in the above construction, have a label one component of which is the empty word, and then to choose a *filter*, that is a language on the alphabet of marks which retains *one* path in every set of equivalent paths. Besides implementing a proof of the Composition Theorem, this construction has the advantage of being well-suited to the lazy evaluation of the composition, that is the implementation of an algorithm that does not compute the composed transducer but the output of it on any input word (with the same number of steps as if the composed transducer had been computed). On the other hand, it is easy to verify that the language T in Eilenberg's proof plays the role of a filter.

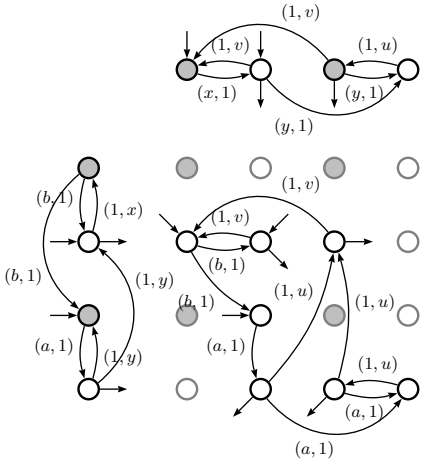


Fig. 3. A composition that preserves multiplicity

We have implemented a construction on transducers that corresponds to this filter T and as it is chosen beforehand we avoid the introduction of marked transductions. We replace them by a preliminary operation on the transducers and the intersection with T is then realized by the deletion of certain states in the product. The construction on $\mathcal{T}$ and $\mathcal{U}$ can be described as follows:
(a) Split the states of $\mathcal{T}$ and their *outgoing* transitions in such a way they are labeled either in $(A \times 1)$ — black states — or in $\widehat{A} \times B$ (or the state is final) — white states; the incoming transitions are duplicated on split states. This is transducer $\mathcal{T}'$.
(b) Split the states of $\mathcal{U}$ and their *incoming* transitions in such a way they are labeled either in $(1 \times C)$ — black states — or in $B \times \widehat{C}$ (or the state is initial) — white states; the outgoing transitions are duplicated on split states. This is transducer $\mathcal{U}'$.
(c) Apply the preceeding algorithm [steps (i), (ii) and (iii')] to $\mathcal{T}'$ and $\mathcal{U}'$ in order to build $\mathcal{T}' \bowtie \mathcal{U}'$.
(d) Delete the black-black states (every state in $\mathcal{T}' \bowtie \mathcal{U}'$ is a pair of states).

(e) Trim and eliminate the transitions with label $(1, 1)$ by classical closure.

Figure 3 shows the construction applied to $\mathcal{T}_1$ and $\mathcal{U}_1$.

Composition algorithm We consider the rewriting rule $ab^n \to ba^n$. This transformation is achieved by the composition of a left sequential transducer by a right sequential transducer, respectively performing rewriting from right to left and left to right. The composition has been implemented using both the composition of representations and the composition of sub-normalized transducers.

Algorithm	n	Nb. states	Nb. transitions	Time
Sub-normalized transducer	20	30084	40356	0.551
	40	232564	305506	4.849
Representation	20	441	882	2.042
	40	1681	3362	36.195

3 Coping with generic static programming

Genericity in Vaucanson In order to ensure maximal genericity of the functions and algorithms written in the VAUCANSON library, most of the objects that come into the definiton of automata are parameterizable. For instance and to quote a few, one can, but also one has to, define the type of the following entities:

- the alphabet, *i.e.* the type of "letters": characters, pairs of characters, etc.
- the multiplicity, which involves both the *domain* ($\mathbb{B}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ for instance) and also the semiring operations considered on these domains: usual $+$ and $\times$, or min and $+$, or max and $+$, etc.
- the transition label type such as letter, polynomial, (rational) series, etc.

As already advocated in [**?**], the use of C++ static genericity is one of the characteristic features of VAUCANSON. Algorithms are written once, and the assurance is given that they will work for all kinds of automata (concerning the above parameters). In order to achieve efficiency, the use of "classical" virtual methods and abstract classes is avoided. Instead, static mechanisms similar to those described by [**?**] and [**?**] are used. The combination of genericity for such a wide range of types and the use of such methods for static mechanism have a heavy counterpart: programming becomes pretty tough, even for most advanced users. The solution to this drawback which threatened the usability of VAUCANSON came through the writing of "context headers".

Context headers The VAUCANSON platform now provides a set of context headers, each of them contains all the needed declarations for a classical type of automata such as Boolean automata, automata with multiplicity in $\mathbb{Z}$, max-plus or min-plus automata, or transducers.

The objective is achieved to some extend. The wide range of functions implemented in the VAUCANSON library may be used with a minimal amount of declarations when applied to classical types of automata. On the other hand, advanced

users may also use their own definitions to take the most of the genericity in VAUCANSON. On the developer's side, genericity is kept and algorithms are written once and specializable in various ways (regarding the automaton type, a particular implementation, etc). By offering predefined types to the user, VAUCANSON provides services which are in fact context-sensitive, as the `new_rat_exp()` or `thompson_of()` functions for instance.

The future of context headers As explained in [**?**], the "type" of an entity in VAUCANSON does not refer only to the type of a variable but also to how this variable is implemented. The present headers refer to the general implementation of automata and do not thus insure the best possible efficiency.

Moreover, the writing of a context header is a tedious process, and every user's wish or need cannot be fulfilled by a library of headers: the possible combinations of types are potentially infinite.

A more elegant solution that we plan to implement in a near future will be to provide a kind of *parameterized context*, for which only the most usual parameters are fixed. As an example, an automaton with "numerical" multiplicity would be defined by a header `weighted_automaton` which will have as parameters the type of the letters of the alphabet and the type of the weight: `int`, `float`, etc.

4 The XML exchange format for automata

At CIAA'04, the VAUCANSON group presented an XML description format for automata. This format was elaborated both as a proposal for an exchange format within the community of automata users and as an input-output standard in order to allow communications between VAUCANSON and other softwares dealing with automata[4]. We shall present a new proposal at CIAA'05, and the XML format proposed will be described there. We describe here only the main features of this new format, their motivation, and the way VAUCANSON handles it.

4.1 The XML proposal

Quick review of the format The description of automata is structured in two parts. The `<type>` tag provides automaton type definition, like Boolean automaton, or weighted ones with the ability to specify weight type, alphabet specification, etc. The `<content>` tag provides the definition of the automaton "structure". The visual representation of automata involves a very large amount of informations. The `<geometry>` data corresponds to the embedding of the automaton in a plane (with informations such as state coordinates or edge type for a transition). The `<drawing>` data contains the definition of attributes that characterize the actual drawing of the graph (such as label position or state color for instance). Most of them are indeed implicit and provided by drawing programs; the format only provides the possibility to make them explicit at every level of the description.

[4] VAUCANSON supports as well the FSM format for loading and saving automata.

From DTD to XSD The most important difference with our previous proposal is the change from a DTD (Document Type Definition) describing the tags for automata representation to an XSD Schema.

This change is indeed a consequence of the same simplification policy which lead us to the definition of context headers: it is desirable to keep the description of automata simple when describing widely used structures, while giving the possibility to describe the most complex ones.

For XML, this simplification amounts to have default types, in order to omit `<type>` tag when describing common Boolean automata or transducers.

The problem then arises when describing an automaton or a transducer, the default values for the `<type>` tag must of course be different. This is not possible with a DTD description. The use of a XSD overcomes this difficulty, since it is possible to define different properties for a same element, according to the embracing context. Is is so possible to locally alter the behavior of a tag, and make it context-sensitive. With this feature, default values for the `<type>` tag are achieved, whether it is a child of `<transducer>` or of `<automaton>`.

It is of course possible to redefine only the tag where default values are inappropriate, inside the `<type>` tag. For instance, in order to define a weighted automaton on $\mathbb{Z}$, it is sufficient to write a `<semiring>` tag as a child of `<type>`, with `set` attribute set to `Z`.

4.2 Implementation in Vaucanson

In order to implement support of proposed XML format in VAUCANSON, two main objectives need to be achieved: maintenance easiness in case of format modification or extension and routines availability to access state geometric coordinates specified in the XML document.

Parsing the XML document To parse the XML document and create the associated tree, we use the Apache Xerces C++ parser [**?**]. Xerces is a validating XML parser, and handles well DTD document validation or XSD validation.

Building the automaton When reading and interpreting data, the program faces a totally dynamic content. It doesn't know, *a priori*, tag properties it will read. We face the problem of knowledge of the treatment type, not data type. In order to solve this problem, we use the Factory Method design pattern [**?**].

Factory Method is a creational pattern. It encourages the user to create a common interface for handled objects (in this case tags), while the exact type of the object is chosen by a subclass according to the context. The main routine deals with abstraction since it knows how to manipulate tags, but doesn't know about data it is dealing with.

Acknowledgments

The help and support of all members of the VAUCANSON Group is gratefully acknowledged: A. Demaille for the management of the group at LRDE, R. Poss and Y. Régis-Gianas for keeping an eye on the evolution of the platform, R. Bigaignon, M. Cadilhac, F. Terrones at LRDE and R. Souza at ENST for their participation to the writing of the platform and especially for the benchmarking.

The help of H. Assaoui and Ph. Martins for the installation of the `vaucanson` server at ENST is also gratefully acknowledged.

References

1. Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.
2. Jean Berstel. *Transductions and context-free languages*, volume 38 of *Leitfäden der Angewandten Mathematik und Mechanik [Guides to Applied Mathematics and Mechanics]*. B. G. Teubner, Stuttgart, 1979.
3. Jean Berstel and Olivier Carton. On the complexity of hopcroft's state minimization algorithm. In Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu, editors, *CIAA*, volume 3317 of *Lecture Notes in Computer Science*, pages 35–44. Springer, 2004.
4. N. Burrus, A. Duret-Lutz, T. Géraud, D. Lesage, and R. Poss. A static c++ object-oriented programming (scoop) paradigm mixing benefits of traditional oop and generic programming. In *Proc. of MPOOL'03, 18th SIGPLAN Conf.*, 2003.
5. Jean-Marc Champarnaud and Djelloul Ziadi. Canonical derivatives, partial derivatives and finite automaton constructions. *Theor. Comput. Sci.*, 289(1):137–163, 2002.
6. Samuel Eilenberg. *Automata, languages, and machines. Vol. A*. Academic Press [A subsidiary of Harcourt Brace Jovanovich, Publishers], New York, 1974. Pure and Applied Mathematics, Vol. 58.
7. C. C. Elgot and J. E. Mezei. On relations defined by generalized finite automata. *IBM J. Res. Develop*, 9:47–68, 1965.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
9. John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971)*, pages 189–196. Academic Press, New York, 1971.
10. W. Kuich and K. Walk. Block-stochastic matrices and associated finite-state languages. *Computing (Arch. Elektron. Rechnen)*, 1:50–61, 1966.
11. Sylvain Lombardy, Yann Régis-Gianas, and Jacques Sakarovitch. Introducing VAUCANSON. *Theoretical Computer Science*, 328(1-2):77–96, 2004.
12. Sylvain Lombardy and Jacques Sakarovitch. Derivatives of rational expressions with multiplicity. *Theoretical Computer Science*, 332:141–177, 2005.
13. M. Lothaire. *Algebraic combinatorics on words*, volume 90 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 2002.
14. Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. The design principles of a weighted finite-state transducer library. *Theor. Comput. Sci.*, 231(1):17–32, 2000.
15. F. Pereira and M. Riley. *Finite State Devices for Natural Language*, chapter Speech Recognition by Composition of Weighted Finite Automata. Proc. MIT Press, 1997.
16. Y. Régis-Gianas and R. Poss. On orthogonal specialization in c++: Dealing with efficiency and algebraic abstraction in vaucanson. In *Proc. of POOSC'2003*, 2003.
17. J. Sakarovitch. *Eléments de théorie des automates*. Vuibert, 2003. Translation: *Elements of Automata Theory*, Cambridge Universiy Press, to appear.
18. M. P. Schützenberger. On the definition of a family of automata. *Information and Control*, 4:245–270, 1961.
19. Xerces. `http://xml.apache.org/xerces-c/`.