Implementation Concepts in Vaucanson 2

Akim Demaille[1], Alexandre Duret-Lutz[1], Sylvain Lombardy[2], and Jacques Sakarovitch[3]

[1] LRDE, EPITA, akim@lrde.epita.fr, adl@lrde.epita.fr
[2] LaBRI, Université de Bordeaux, Sylvain.Lombardy@labri.fr
[3] LTCI, CNRS / Télécom-ParisTech, sakarovitch@telecom-paristech.fr

Abstract. Vaucanson is an open source C++ platform dedicated to the computation with finite *weighted* automata. It is *generic*: it allows to write algorithms that apply on a wide set of mathematical objects. Initiated ten years ago, several shortcomings were discovered along the years, especially problems related to code complexity and obfuscation as well as performance issues. This paper presents the concepts underlying Vaucanson 2, a complete rewrite of the platform that addresses these issues.

1 Introduction

Vaucanson[4] is a free-software[5] *platform* dedicated to the computation of and with *finite automata*. As a "platform" it is composed of a high-performance (somewhat unfriendly) low-level C++ library, on top of which more humane interfaces are provided: a comfortable high-level Application Program Interface (API), flexible formats for Input/Output, easy-to-use command-line tools, and eventually, a Graphical User Interface (GUI). Here, "automata" is to be understood in the broadest sense: weighted automata on a free monoid — that is automata that not only accept, or recognize, *words* but compute for every word a *multiplicity* which is taken in an arbitrary semiring — and even weighted automata on *non-free* monoids. As for now, are implemented in Vaucanson only the (weighted) automata on (direct) products of free monoids, machines that are often called *transducers* — automata that realize (weighted) relations between words.

Vaucanson was started about ten years ago [7] with two main goals and a constraint in mind: genericity (offering support for a wide set of automaton types) and a natural (to mathematicians) programming style, while keeping performances that compete with similar platforms. Genericity was obtained by extensive use of C++ overloading and template programming, which are well known means to avoid the costly dynamic polymorphism of traditional object-oriented programming. A novel design, dubbed Element/MetaElement, allowed a consistent pattern of implementation of mathematical concepts (such

[4] Work supported by ANR Project 10-INTB-0203 VAUCANSON 2.
[5] http://vaucanson-project.org, http://vaucanson.lrde.epita.fr

as monoids, semirings, series, rational expressions, automata and transducers). Annual releases have shown that genericity was indeed achieved: for instance TAF-Kit, the command-line tools exposed by Vaucanson 1, features about 70 commands on 18 different kinds of automata/transducers [9].

However performances did not keep on par. The causes for these poor performances are manifold. For instance "genericity" was sometimes implemented by "generality": instead of a carefully crafted specialized structure, a very general one was used. Most prominently, usual automata, labeled with letters, were implemented as automata whose transitions are labeled by polynomials. The aforementioned Element/MetaElement design also conducted to error-prone, obfuscated code, which eventually impeded the performance of the *developers* themselves.

The Vaucanson 2 effort aims at keeping the best of Vaucanson 1 (genericity, rich feature set, easy-to-use shell interface, etc.) while addressing its shortcomings.

This paper presents the design of Vaucanson 2, which is only partly implemented. Although much remains to be done, enough is already functional so that we can report on our experience in the redesign of Vaucanson. The remainder of this paper is structured as follows. Section 2 introduces the fundamental types used in Vaucanson 2, and their implementation is presented in Section 3. In Section 4 we explain how we provide flexible and easy-to-use interfaces on top of an efficient but low-level library. We conclude in Section 5.

2 Types Implementation

2.1 From Typing Automata to Typing Transition

The type of a weighted automaton is classically given by the *monoid* M of its labels and the *semiring* $\mathbb{K}$ of its weights. For instance, if M is a free monoid over a finite alphabet A, the weighted automaton is an automaton over finite words; if M is a product $A^* \times B^*$ of two free monoids (A and B are finite alphabets), then the weighted automaton is actually a transducer. The Boolean semiring $\mathbb{B}$ corresponds to usual NFAs, while $\mathbb{R}$ can be used for probabilistic automata and $\langle \mathbb{Z}, \min, + \rangle$ for distance automata.

Such a weighted automaton realizes an application from M into $\mathbb{K}$. The applications that can be realized this way are called ($\mathbb{K}$-)*rational series*. They are a subset of formal power series over M with coefficients in $\mathbb{K}$, that are usually denoted $\mathbb{K}\langle\langle M \rangle\rangle$, but seen as applications from M into $\mathbb{K}$, they can also be denoted $\mathbb{K}^M$, or, with a notation closer to type theory $M \to \mathbb{K}$.

Vaucanson 1 follows this characterization, and, on top of its implementation, the type of a weighted automaton is made of both the type of a monoid and the type of a semiring. The pair Monoid/Semiring is called the *context* of the automaton (in Vaucanson 1 vocabulary).

This definition of types, based on the algebraic characterization, has two main drawbacks. First, each implementation must support the most general

automata of a given type. Therefore, any element of M is accepted as a label of a transition; actually, in Vaucanson 1, labels and weights were handled at the same time, since each transition can be labeled by a polynomial (a linear combination of letters). Second, the application of many algorithms requires the label of the automaton to have a particular form. For instance, the product of two automata (that recognizes the intersection of languages in the case of NFAs) requires the labels of both automata to be letters, or the usual composition algorithm for transducers requires that the transducers are *subnormalized*. In Vaucanson 1, these prerequisites, crucial for the effective computation, are not enforced statically (i.e., by specific C++ types): they must be checked at runtime.

In the design of Vaucanson 2 we decided to refine our typing system to fit with the effective requirements of algorithms and implementations. It led to focus on typing the *transitions* rather than the automata themselves. Indeed, even if different automata on words have the same context in the sense of Vaucanson 1, depending whether they are labeled by letters, or letters and ε, or by words, their implementation and their behavior w.r.t. algorithms may be very different. Hence, in Vaucanson 2 the type of an automaton is characterized by the type of its transitions, which we name a *context*. Since the type of a transition depends on the type of its label and of its weight, the context of an automaton in Vaucanson 2 is a pair made from a *LabelSet* and a *WeightSet* (Section 2.3). OpenFst [2] preceded us, and our contexts are alike their "arctypes".

2.2 Bridge between Structures and Values

The implementation of mathematical objects in Vaucanson 2 follows a pattern, which we name Element/ElementSet: the C++ type of values is separated from the C++ type of their set of operations. For instance, weights of both semirings $\mathbb{Z}$ and $\langle \mathbb{Z}, \min, + \rangle$ are implemented by `int`, but their operations are defined by the corresponding WeightSets: `z` and `zmin`. Other Element/ElementSet pairs include `bool`/`b`, `double`/`r`, etc., but also the rational expressions and their set: `ratexp<`*`Context`*`>`/`ratexpset<`*`Context`*`>`. Thanks to this compliance with this Element/ElementSet duality, rational expressions can be used as weights. The behavior of labels (concatenation, neutral element...) is also provided by a LabelSet.

Whereas Vaucanson 1's Element/MetaElement is a rich and complex design in itself, our segregation between values and their operations is hardly novel. It is a return to N. Wirth's "Algorithms + Data Structures = Programs" equation [10], phased out by traditional object-oriented programming's "Programs = Objects", and resurrected by generic programming à la STL.

2.3 LabelSet and WeightSet

LabelSet: Different Kinds of Label The LabelSet contains the information on the type of the labels, that is the implementation of the labels themselves, the implementation of the elements of the monoid (that may be different: a letter is not implemented as a word). It also knows how to multiply (concatenate) labels.

Last, but not least, it provides a type called *Kind* that is used to choose the most appropriate algorithm that has to be applied to the automaton (or rational expression). There are two different kinds: LAW, LAL, and a third sort of automata that is treated as a kind in itself: LAU.

LAW, 'Labels Are Words'. The most general weighted automata have transitions labeled by the elements of a monoid (and weighted by the elements of a semiring). This class is very general, does not correspond to the usual definition of finite automata and is not suitable for many algorithms (determinization, product, evaluation, etc.). However it is the only kind so far that describes automata over direct product of monoids. The LabelSet for the free monoid is called `wordset`.

LAL, 'Labels Are Letters'. This class of automata corresponds to the usual way NFAs, or even WFAs, are defined: labels are restricted to being letters. It also corresponds to the description of automata as *linear representations*. Some algorithms require the automata to be in this class, such as the product (which implements the tensor product of the representations) or the reduction algorithm (which can be applied when the weights belong to a field). The LabelSet implementing this kind is `letterset`.

LAU, 'Labels Are Unit'. This kind corresponds to the case where there is no label, or, equivalently, where every label is the element of a trivial monoid. These automata are therefore weighted graphs (with initial and final values attached to vertices). They prove to be very useful since they allow to unify for instance the state elimination (which computes a rational expression from an automata) and the ε-removal algorithms. Its LabelSet is `unitset`.

WeightSet: Different Values The WeightSets define the nature of the weights, and the operations that apply. They must provide addition, multiplication and Kleene-star operators. Currently, basic WeightSets include (i) Booleans $\langle \mathbb{B}, \vee, \wedge \rangle$, implemented by the class `b`; (ii) integers, $\langle \mathbb{Z}, +, \times \rangle$ and $\langle \mathbb{Z}, \min, + \rangle$, implemented by `z` and `zmin`; (iii) double precision floats $\langle \mathbb{R}, +, * \rangle$, implemented by `r`.

Every object that offers the same set of methods, i.e., every object that satisfies the *concept* of WeightSet can be used. For instance, rational expressions can be used as weights, even though they do not form a semiring.

2.4 Implementation of Contexts

In Vaucanson 2, the LabelSets manage the labels. Most aspects of labels are *static*: their type (e.g., `char`, `std::string`, etc.) as well as the operations or services that are available (algebraic operations, iteration on generators, etc.). Others aspects may vary at runtime: an alphabet (a set of letters) is needed to fully define a LabelSet (except for LAU). Therefore, LabelSets are C++ objects (values), not just classes.

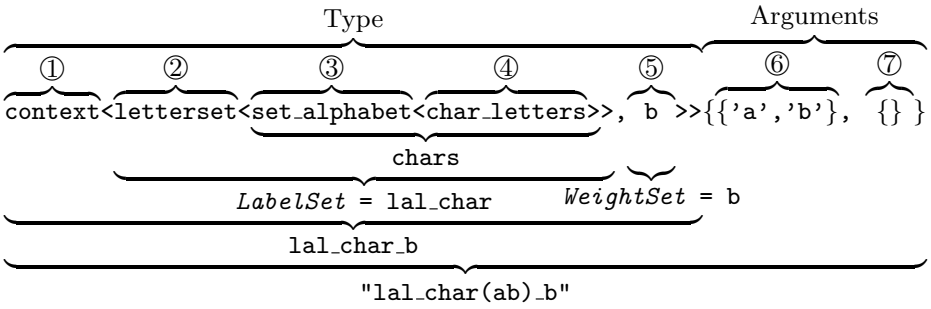

Fig. 1. Anatomy of `lal_char(ab)_b`, the C++11 implementation of $\{a, b\} \to \mathbb{B}$, the context for Boolean automata. *Class template* (named *generics* in C# and Java) generate classes once provided with parameters, in angle brackets. For instance, `set_alphabet` is a class template expecting a single parameter (a class), and `set_alphabet<char_letters>` is class.

Basic WeightSets (`b`, `zmin`, etc.) could be defined only as types, but more complicated weights, like rational expressions (cf. Section 3.2), require runtime values.

Because they depend on runtime values, *contexts* are *objects* (not just classes) which aggregate an instance of a LabelSet and an instance of a WeightSet. Contexts are the cornerstone on top of which the whole VAUCANSON 2 platform types its mathematical entities (words, weights, automata, rational expressions etc.). As such, they could be named "types", but it would lead to too much confusion with C++'s own types. However, these (VAUCANSON 2) "types" have *names* which we write with quotes (e.g., `"lal_char(ab)_b"`), which should not be confused with the C++ objects that implement them, even though C++ names match those of the types, written without quotes (e.g., `lal_char_b`).

The context of Boolean automata (see Fig. 1) will help understanding the C++ implementation of contexts. This context object is named `"lal_char(ab)_b"`. As any object, it is the instantiation of a *Type*, provided with *Arguments*.

We name the C++ Type `lal_char_b`. It is a context ①, which aggregates a LabelSet (`lal_char`) and a WeightSet (`b`). Since we define an LAL context, the LabelSet is `letterset` ②, which must be provided with the type of the letters. The alphabet ③ is a set of letters. We designed alphabets to work on different types of letters, `char`, but also `int`; in Fig. 1 we work with the traditional C++ type for characters, `char`, provided by the `char_letters` class ④. The effective set of these letters is given in ⑥. The WeightSet is `b`, i.e., $\langle \mathbb{B}, \vee, \wedge \rangle$. In this case, `b` needs no argument, so none is provided ⑦. Table 2 gives more examples of contexts.

Context	
C++ Type	C++ Initializers
VAUCANSON Static Name	VAUCANSON Dynamic Name
$\{a,b\} \to \mathbb{B}$	
`context<letterset<chars>, b>`	`{{'a', 'b'}, {}}`
`"lal_char_b"`	`"lal_char(ab)_b"`
$\{a,b\}^\star \to \mathbb{Z}$	
`context<wordset<chars>, z>`	`{{'a', 'b'}, {}}`
`"law_char_r"`	`"law_char(ab)_z"`
$\{1\} \to \{a,b\} \to \mathbb{Z}$	
`context<unitset,` `ratexpset<letterset<chars>, z>>`	`{{}, {'a', 'b'}}`
`"lau_ratexpset<lal_char_z>"`	`"lau_ratexpset<lal_char(ab)_z>"`
$\{a,b\} \to \{x,y\} \to \mathbb{Z}$	
`context<letterset<chars>,` `ratexpset<letterset<chars>, z>>`	`{{'a', 'b'}, {'x', 'y'}}`
`"lal_char_ratexpset<lal_char_z>"`	`"lal_char(ab)_ratexpset<lal_char(xy)_z>"`

Table 2. Examples of contexts. `chars` stands for `set_alphabet<char_letters>`, an alphabet whose letters and words are of type `char` and `std::string`.

3 Object Implementation

3.1 Automata

The design of automata in VAUCANSON 2 is driven by their API, which was designed after the experience of VAUCANSON 1 to allow easy and efficient implementation of algorithms. It offers a number of methods to access both in reading and writing to the automaton content. The automaton is the object that "knows" how to manipulate all sub-entities like states, transitions, labels or weights. Following the ELEMENT/ELEMENTSET principle, these are only values (that may be numbers, pointers or structures) with no methods; they can be handled directly by the automaton or in some cases by the ELEMENTSET objects (the LabelSet or the WeightSet) that can be retrieved from the automaton.

To make the use of the VAUCANSON library as intuitive as possible, all the algebraic information is confined in the context object. An automaton is therefore a class template that depends on the implementation of the automaton, where the template parameter is the context. For instance, the class template `mutable_automaton<`*`Context`*`>` is a linked list based implementation that supports all the API. VAUCANSON 2 provides also some wrappers that can be applied to a totally defined automaton (implementation and context). For instance, the class template `transpose_automaton<`*`Automaton`*`>` wraps an automaton to present a reverse interface, read and write.

The API of `mutable_automata` is totally symmetrical: the forward and backward transition functions are both accessible. Other implementations can be provided that partially implement the API, e.g., `forward_mutable_automaton` (and `backward_mutable_automaton` for symmetry), which does not track incoming transitions, should improve performances in most cases. More wrappers

can also be provided, for instance to change the alphabets, to change the weight sets and so on.

Our API is designed to support implementation of automata computed on-the-fly as well [6].

Pre and Post States Weighted automata feature not only weights on transitions but also on the initial and final states. An implementation keeping these initial and final weights on states is awkward, as we found out in VAUCANSON 1. The data structure is heavy, and the API error-prone.

In VAUCANSON 2, weights on initial states are modeled as weights on (plain) transitions from `pre`, a (unique) invisible "preinitial" state, to every initial state. These transitions are labeled by a fresh label, denoted by `$`, which can be interpreted as the empty word. As a consequence the initial weights are handled seamlessly, which simplifies significantly the implementation of several algorithms. Besides, since there is a single `pre` state, algorithms that start by loading a queue with the initial states (e.g., determinization) can now push only `pre` and let the loop iterate on the successors.

The final weights are handled in a similar fashion, with `post`, a (unique) invisible "postfinal" state.

3.2 Rational Expressions

Although the concept of "context" emerged to design our implementation of automata, it fits perfectly the same job to define the nature of the corresponding rational expressions. By virtue of the ELEMENT/ELEMENTSET principle, rational expressions are implemented by `ratexp<`*`Context`*`>` and manipulated by `ratexpset<`*`Context`*`>`.

Syntax VAUCANSON, 1 and 2, supports weighted rational expressions, such as $a + (2b)^\star + 3(c^\star)$. Because weights can be arbitrarily complex, we delimit them with braces: '`a+({2}b)*+{3}(c*)`'.

VAUCANSON 1 provides customizable syntax for rational expressions: one can define which symbols are used for the constants 0 and 1 (denoting the empty language and empty word), and so forth. In the predefined parser of VAUCANSON 2 '`\z`' and '`\e`' denote 0 and 1. Because braces are already used for weights they cannot be used for the generalized quantifiers as in POSIX extended notation (*a*`{`*min*`,`*max*`}`), we denote them *a*`(*`*min*`,`*max*`)`. Other parsers can be added to support alternative syntaxes.

Contexts Like automata, rational expressions support not only various types of weights, but also of labels — for simplicity and by symmetry with automata, we name "labels" the "atoms" of rational expressions. They are therefore parameterized by a context. Classical rational expressions such as '`(a+b)*a(a+b)`' use letters ('`a`' and '`b`') as atoms, and Booleans as weights ("true" is implicit): their context is `"lal_char(ab)_b"`.

Labels may also be words (LAW) or reduced to 1 (LAU). Weights may be integers (e.g., '`one+{2}(two)+{3}(three)`' from `"law_char(ehnortw)_z"`) or even rational expressions: '`{{1}x}a+{{2}y}b`' is a rational expression from `"lal_char(ab)_ratexpset<lal_char(xy)_z>"` (see Table 2).

Associativity, Commutativity We intend to experiment with different implementations of rational expressions, including relying on associatitivity and commutativity. To this end, our `ratexp` structure supports variadic sums and products. It is up to `ratexpset` to decide whether to exploit this feature or not. Currently sums and products are binary, and only straightforward rewritings are performed on rational expressions ("Trivial Identities", [9, Table 2.5]).

4 Dynamic Polymorphism Implementation

4.1 Dynamic I/O routines

Contexts Input/Output As its predecessor, VAUCANSON 2 is more than just a C++ library: it is a platform that provides a set of tools to manipulate automata and rational expressions, means to save and restore them, eventually a GUI, etc. I/O of automata and rational expressions is therefore crucial, which requires a means to save, and read, their type. In short, a file containing a Boolean automaton must specify that it is a Boolean automaton. This is achieved by I/O support for contexts.

VAUCANSON 2 can compute a name from a context, and conversely it can build a context from its name. Examples of (dynamic) context names are provided in Table 2; for instance the name for $\{a, b\} \to \mathbb{B}$ is `"lal_char(ab)_b"`. Static names are computed from the (dynamic) names by removing the alphabets: `"lal_char_b"`.

VAUCANSON 1 relies on FSMXML [4] for typed I/O: specific tags define the algebraic context of the stored automaton or rational expression. Context names play a similar role in VAUCANSON 2's I/Os.

Dynamic Automaton/Rational Expression Input VAUCANSON 2 aims at providing an even larger set of algorithms than VAUCANSON 1, yet in a more flexible way. TAF-KIT 2 consists of a unique binary that replaces the set of binaries of TAF-KIT 1. The following command lines for the computation of the product of two automata and for the construction of the standard (or position) automaton of an expression show the contrast between both interfaces.

```
# Vaucanson 1: use the context−specific tool.
$ vcsn-char-z product a.xml b.xml > c.xml  # product of automata a and b
$ vcsn-char-z  -a abc standard '{2}a+{3}c' > s.xml # automaton for 2a+3c

# Vaucanson 2: use the generic tool, the context is handled dynamically.
$ vcsn product a.xml b.xml > c.xml
$ vcsn -C 'lal_char(abc)_z' standard -e '{2}a+{3}c' > s.xml
```

In TAF-Kit 1, there is a command for every single context (here, `vcsn-char-z`), while in TAF-Kit 2 context is just data of a unique command (`vcsn`). The context must be either carried by the argument, for instance in the FSMXML description of an automaton, or provided with a simple command line option (`-C 'lal_char(abc)_z'`). In either case, the very profound difference between the static nature of the C++ library and the dynamic one of data must be resolved. This is addressed by our static/dynamic bridge.

4.2 Dynamic Calls to Static Algorithms

Static/Dynamic Polymorphisms *Polymorphism* is the ability for a single function name to denote several implementations that depend on the type of its argument(s). It provides a means to specialize algorithms: some can be run on automata of special kind, with special type of weights, or special type of labels, others may have different versions according to the type of weights or of labels. For instance, determinization applies to Boolean automata only, product to LAL automata only, etc. The elimination of ε-transitions gives an example of the second kind: any closure algorithm works on Boolean automata, for automata with weights in $\mathbb{Z}$ it boils down to an algorithm that tests for acyclicity, and for automata with weights in $\mathbb{R}$, it is again another algorithm [8].

The *dynamic* polymorphism (i.e., specialization selection at runtime via virtual tables) provides a flexible solution to these problems: roughly, when a method is called on an object, the exact type of this object is examined at run time and the corresponding implementation is then called. It is the essence of object-oriented programming. Unfortunately this elegant mechanism slows down the execution of the program when used in intensive computation loops.

C++ offers another mechanism, *templates*, that allows *static* polymorphism (i.e., specialization selection at compile time). Functions and classes can be parameterized by one or several types (or constants); some specific implementations can also be provided for particular types (see Fig. 1). When a method is called on an object, the type of this object is known at compile time and the method, if needed, is compiled especially for this type. Method invocation is then as efficient as a plain function call. There are some drawbacks to this mechanism. First, every object must be precisely typed: the type of an automaton shows its implementation and is moreover parameterized by the type of its labels and weights, which can themselves be parameterized. Hence, users of the library may handle quite complicated types. For instance the type for a classical Boolean automaton is `mutable_automaton<context<letterset<set_alphabet<char_letters >>, b>` (see Table 2 for a description of the context part). Second, to compile binaries that offer all the required algorithms for each type of automata, the corresponding functions must be compiled for each of these types. The compilation, as a consequence, is an extremely long process, and the introduction of a new "context" (that is, a new kind of weights, labels or implementation) needs a configuration to specify which algorithms need to be compiled.

This is our experience with Vaucanson 1; compiling the whole package and the 18 commands of TAF-Kit takes hours (annoying for users, crippling for

```
template <typename Context> void
static(const string& lhs, const string& rhs, const string& word)
{
  using automaton_t = vcsn::mutable_automaton<Context>;
  automaton_t l = vcsn::read_automaton_file<automaton_t>(lhs);
  automaton_t r = vcsn::read_automaton_file<automaton_t>(rhs);
  automaton_t prod = vcsn::product<automaton_t, automaton_t>(l, r);
  typename Context::weight_t w = vcsn::eval<automaton_t>(prod, word);
  prod.context().weightset()->print(std::cout, w);
}

void dynamic(const string& lhs, const string& rhs, const string& word)
{
  using namespace vcsn::dyn;
  automaton l = read_automaton_file(lhs);
  automaton r = read_automaton_file(rhs);
  automaton prod = product(l, r);
  weight w = eval(prod, word);
  print(w, std::cout);
}
```

Fig. 3. Static/Dynamic APIs: evaluation of the word `word` by a product of the automata stored in the files named `lhs` and `rhs`. The `dynamic` routine hides the complexity of templated programming, demonstrated by `static`. It is also more flexible, as it can be invoked with automata of equal algebraic type, but of different C++ types (e.g., a `transpose_automaton` and a `mutable_automaton`).

developers!). Beside, the resulting API was repelling, and very few people dared programming with the library, and preferred to use TAF-Kit.

To address these shortcomings while keeping the good properties of static polymorphism (efficiency and rigorous type checking), Vaucanson 2 provides a two-level API (see Fig. 3). The low-level API, named "static", is fully typed: the C++ types of the object are precise and heavily templated. When programming at this level, the user is in charge of every detail, including memory management. For sake of efficiency, all the algorithms of the library are written at the static level: dynamic polymorphism and its costs are avoided. The high-level API, named "dynamic", provides the user with the comfort of dynamic polymorphism, but at such a coarse grain that its cost is negligible (to select the appropriate version of an algorithm, not in the implementation of the algorithms themselves). This layer takes in charge details such as the exact type of the objects, and memory management.

Dynamic Access to Template Algorithm Fig. 4 demonstrates how the dynamic API invokes the low-level one. In the high-level API, an automaton `a1` is handled as a `dyn::automaton`, whatever its exact type. When the user invokes `dyn::determinize(a1)`, several steps must occur. First `a1` is asked for

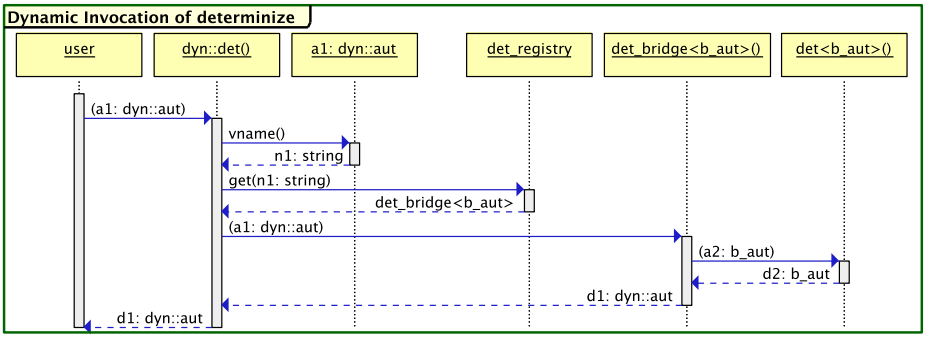

Fig. 4. Dynamic invocation on top of templated implementations. For conciseness, in this diagram `b_aut` denotes the type of Boolean automata, `mutable_automaton<context<letterset<set_alphabet<char_letters>>, b>`, the "exact" type of the automaton `a2`, contained by the dynamic automaton `a1`.

the name of its exact type (`n1` in Fig. 4). This string is used to query a registry of all the static *bridges* of `determinize`. These bridges all have the same type (`dyn::automaton -> dyn::automaton`), yet they are parameterized by the exact type. Once the specific `det_bridge<b_aut>` is obtained, it is invoked with `a1` as argument. The bridge extracts `a2`, the static automaton, from `a1`, and invokes the low-level `determinize`, whose type is `b_aut -> b_aut`. Its result, the static automaton `d2`, is returned to the bridge, that wraps it in the dynamic automaton `d1`, which is eventually returned to the user.

As far as we know, this two-layer technique to implement dynamic polymorphism on top of static polymorphism is unpublished. However several of its components have already been documented in the literature. For instance, dynamic objects such as `dyn::automaton` implement the External Polymorphism design pattern [3]. The implementation of algorithm registries is very similar to "Object Factories" [1, Chap 8.], an implementation of the Abstract Factories [5] tailored for C++.

5 Conclusion and Future Work

Vaucanson 2 is a complete rewrite of the Vaucanson platform. Its whole design relies on objects called "contexts" that behave as a custom typing-system for automata and rational expressions. They are also the cornerstone of the two-level API, which provides an easy-to-use and flexible dynamic interface on top of an efficient static C++ library. Although Vaucanson 2 is still in its early phase, it already features 30+ different dynamic algorithms (including I/O) and 15+ predefined contexts. The benchmarks show that the new concepts have a clear and positive effect on performances. For instance, as of mid 2013, Vaucanson 2 is about 2.5 times faster on determinization than its predecessor, and 50% slower than OpenFst.

Much remains to do. Importing the algorithms from VAUCANSON 1 will be a non-straightforward effort, as it will require adjusting to the new API, taking advantage of C++11 features, and binding with dynamic API. It is expected to improve both the readability of the algorithms, and their efficiency.

Adding more contexts should be reasonably simple. We are confident that adding new WeightSets is simple and straightforward, as there are already many very different implementations. Variety in the LabelSets is more challenging: supporting other concepts of generators (integers for instance), and different label kinds (tuples of letters and empty word, for instance, for implementing automata on multiple tapes, a generalization of VAUCANSON 1's support for transducers [7]).

References

1. A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
2. C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In J. Holub and J. Zdárek, editors, *Proceedings of Implementation and Application of Automata, 12th International Conference (CIAA'07)*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23, 2007, Springer. `http://www.openfst.org`.
3. C. Cleeland, D. C. Schmidt, and T. Harrison. External polymorphism — an object structural pattern for transparently extending C++ concrete data types. In *Proceedings of the 3rd Pattern Languages of Programming Conference*, 1997.
4. A. Demaille, A. Duret-Lutz, F. Lesaint, S. Lombardy, J. Sakarovitch, and F. Terrones. An XML format proposal for the description of weighted automata, transducers, and regular expressions. In J. Piskorski, B. W. Watson, and A. Yli-Jyrä, editors, *Post-proceedings of the seventh international workshop on Finite-State Methods and Natural Language Processing (FSMNLP'08)*, volume 19 of *Frontiers in Artificial Intelligence and Applications*, pages 199–206, 2009, IOS Press.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
6. F. Guingne and F. Nicart. Finite state lazy operations in NLP. In J.-M. Champarnaud and D.Maurel, editors, *Proceedings of Implementation and Application of Automata, 7th International Conference (CIAA'02)*, volume 2608 of *Lecture Notes in Computer Science*, pages 138–147, 2002. Springer.
7. S. Lombardy, R. Poss, Y. Régis-Gianas, and J. Sakarovitch. Introducing Vaucanson. In O. H. Ibarra and Z. Dang, editors, *Proceedings of Implementation and Application of Automata, 8th International Conference (CIAA'03)*, volume 2759 of *Lecture Notes in Computer Science*, pages 96–107, 2003, Springer.
8. S. Lombardy and J. Sakarovitch. The validity of weighted automata. *Int. J. of Algebra and Computation*, 2013. To appear. Available on `ArXiV`.
9. Vaucanson Group. *Vaucanson TAF-Kit Documentation*, 1.4.1 edition, sep 2011. `http://www.vaucanson-project.org`.
10. N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.