An interpreter for Vaucanson

Louis-Noel Pouchet
<louis-noel.pouchet@lrde.epita.fr>

Technical Report n^o0413, revision 0.4, 06 2004

Vaucanson is a generic framework for finite state machine manipulation. There is a need for that kind of library to be able to work in a dynamic environment. Two previous solutions were developed to provide an interpreter for Vaucanson. This report propose another attempt to realize an interpreter, using Swig and OCaml. Also, a syntax proposal is done for automata manipulation.

Keywords
Vaucanson, interpreter, Swig, OCaml

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
lrde@epita.fr – http://www.lrde.epita.fr

Copying this document

Copyright © 2004 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just "Copying this document", no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Contents

Chapter 1

Introduction

When using generic programming on scientific libraries, performances and abstraction easiness are achieved. But in order to do small computations, for example, the user is waiting for a dynamic working environment, using the library.

Many solutions are possible, and this report recall the two previous ones realized for Vaucanson. But none of them are totally satisfying, and there is a need for another solution, combining advantages of the existing and without its drawbacks.

The solution presented here tries to combine flexibility of the interpreter, with the ability to use types dynamically defined in a static library, and syntax extension for automata manipulation. A syntax proposal is done, using OCaml.

Chapter 2

Context of this research

2.1 Vaucanson

Vaucanson is a finite state machine framework, written in C++. Its objective is to offer a generic framework containing algorithms working on automata. Since it manipulates algebraic structures, the best solution was to manipulate abstractions in order to write algorithms once, and to be able to specialize them regarding their abstraction [11]. The design pattern is described in section 5.3.1.

Since generic programming (GP) is used, we have to do the assumption of the closed world i.e. every type need to be known at compile time. The first problem encountered is when you want to dynamically use your static library. Either you want to work in a dynamic environment, like an interpreter, and be able to use interpreter types, or either you want to be able to use new type definitions on the fly, like in Just In Time compilation [3].

Using Vaucanson as a developer is quite easy, even if it can sometimes be counter-intuitive since GP is heavily used. But if a regular user wants to use Vaucanson for small computations, he needs to create a C++ program and deal with compilation times.

2.2 Possible solutions

Confronting this problem leads to two major classes of solutions:

- interpreter creation,
- JIT compilation.

The point of this report is to discuss the first solution: the creation of an interpreter, in order to work in a dynamic environment.

The JIT is a huge subject, and Fosse [8] delivered a shot for Vaucanson.

2.3 Existing solutions

The interpreter problem in Vaucanson is not new. Two previous solutions were developed by Regis-Gianas and Poss for the first one, and Poss for the second. They both have drawbacks, which lead us to search for a better solution.

2.3.1 The first interpreter for Vaucanson

Simultaneously to the initiation of Vaucanson, Regis-Gianas and Poss wrote an interpreter for the library. Since it manipulates algebraic complex structures, they oriented their work on the full design of the interpreter, including:

- syntax definition,
- typing system,
- interface with Vaucanson.

This method gives the advantage of a full control of all the elements of the interpreter (syntax especially). But the major drawback is the amount of work needed to design from scratch an interpreter. Furthermore, the actual interface with Vaucanson was designed in a way that all types from Vaucanson needed to be compiled. So it was quite hard to use a new type defined in the interpreter with all Vaucanson functionalities.

But the major idea of this interpreter is the ability to use its own OCaml like syntax, well suited for automata manipulation.

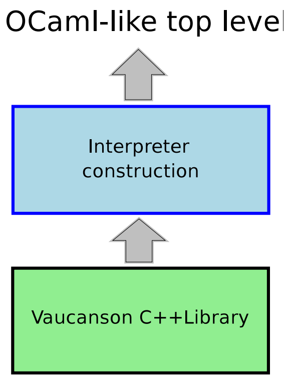

Figure 2.1: First interpreter build scheme

2.3.2 Vaucanswig

In the case of Vaucanswig [10], unlike the first interpreter, Poss tried to build up on existing solutions i.e. use Python as the support language and use Swig to create interfaces between Python and Vaucanson.

Mainly his work is built on Swig ability to manipulate complex interface files, handling objects [1] and data structure (see section 3.1.1). The error prone and heavy work needed with the first solution is delegated to Swig, and the whole language structure proposed is delegated to Python.

This solution comes with great advantages:

- very few code,
- non intrusive modifications in Vaucanson,
- use an existing and well known language,

- many of Vaucanson functionalities are mapped easily.

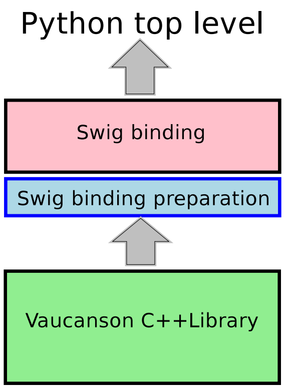

Figure 2.2: Vaucanswig build scheme

Of course, the solution is not totally satisfying. Using Python syntax is not always well suited for automata manipulations, and we may want to dispose of some specific syntactic tools. But the major drawback of this solution is highlited with the enormous compilation time needed to compile the whole interface: you must know every type statically to be able to compile a module. This drawback comes from GP, and lead to the necessity of compiling all possible combination of types that we may need in the interpreter. For example, all the following points must be compiled:

- standard_of algorithm for boolean semiring,
- standard_of algorithm for numerical semiring,
- standard_of algorithm for tropical max semiring,
- standard_of algorithm for tropical min semiring.

Of course, you also need to combine these with every wanted alphabet implementations (char, pair of char, int, . . .).

But under this rebarbative compilation time, which finally is not a point because the interpreter needs to be compiled only once, appears the fact that it is impossible to use Vaucanson with a type defined in the interpreter, because of course this type wouldn't be known statically and the world would not be closed anymore.

2.4 Another interpreter

Regarding the two existing solutions, two major issues need to be solved:

- use data types defined in the interpreter,
- have some syntactic extensions for automata manipulation.

In Vaucanson's case, the point is to get genericity over automata. That can be almost achieved with genericity over alphabets and semirings.

Here the idea is simple: use Swig to build the interface, so that the amount of work will be limited to create the interface. The target language chosen is OCaml because it suites very well our purpose. The reasons of this choice are fully described in section 3.1.2.

But in order to get genericity over alphabets and semiring, we have to figure a solution to be able to define a type in the interpreter, and use it in Vaucanson. The solution is very simple: offer a "generic" adapter for letters and numbers in Vaucanson, and ensure that every method call on these objects calls a method defined in the interpreter. A complete description of the solution is given in section 3.2 for the alphabets and section 3.3 for semiring, and a generalization attempt is done in section 5.3.

Of course, we also want to have a syntactic extension like the one proposed by Fosse [7]. This can be achieved by using OCaml pre-processor: Camlp4 [6]. The grammar extension is fully described in section 4.

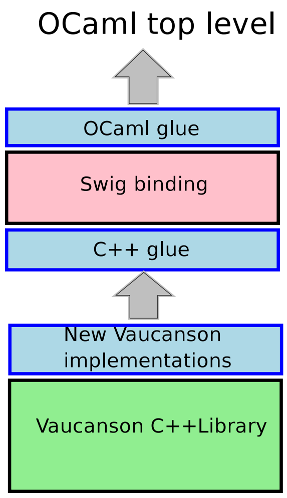

Figure 2.3: New build scheme

Chapter 3

Interpreter building

As said below, building an interpreter for Vaucanson is not a new problem. The two previous attempts were useful to focus potential issues they implies. It is now possible to take the best of Vaucanswig and to try to get as close as possible to the first interpreter.

3.1 Used tools

3.1.1 Swig

Swig is a software development tool for building scripting languages interfaces to C and C++ programs [1]. Its purpose is to make the job easier for developers who want their program to be used in a scripting language.

It was obvious since Vaucanswig that Swig is an excellent tool for interface generation. For Vaucanson, the development scheme could have been like:

- for all wanted classes, create an interface in Swig,
- for all wanted algorithms (i.e. functions), create an interface,
- run swig and compile output files.

Seen like that, this development scheme looks very simple. In a matter of fact, that can become much more complicated when manipulating C++ GP, with an intensive use of the STL [2].

First of all, there are some parser limitations in Swig. You should not run Swig on source code files, but only on interfaces files. For example, overloading methods of a class will not be supported. This is in fact quite normal, since you are defining in the interface what you are exporting in the target scripting language, and not all of them support overload.

In C++, template code is instantiated automatically by the compiler when needed in the program. In Swig, you have to explicitly tell what templates are instantiated. For example, if you have a module working on `Foo<T>`, you will have to tell Swig to instantiate `Foo<int>` and `Foo<char>`, if you need to export those two classes in the target language. Since GP is used in Vaucanson, an heavy use of template is done to get genericity and reproducing this genericity in the interpreter can only be done by explicitly instantiate all combination of templates that may be used.

Another problem is the use of the STL. This problem is dependant of the target language chosen, but almost recurrent everywhere. When you use the STL, you may want to map STL data types like list or vector to data types in the target language. Unfortunately, this is not always well done, and for example for the OCaml binding which is still experimental, the map between `std::list<T>` and the `'a list` is not done since `T` is not mapped to `'a`. Swig provides predefined typemaps for some STL containers, but writing your own ones can be very fastidious. Since you may expect your C++ code to use data structures available in the target language chosen, this is a potential issue.

3.1.2 OCaml

When designing an interpreter with Swig, the major decision is to choose an appropriate target language. Here OCaml was chosen, because of its various advantages:

- excellent interoperability with C,
- interpreted and compiled,
- functional approach,
- flexible thanks to Camlp4,
- ...

Initially the choice was oriented to OCaml because of the easiness of the grammar extension. Camlp4 (section 4) makes grammar extension so simple that anyone can alter the proposed syntax and use his own.

But the main advantage of OCaml is how the language interact with C / C++. As said above, the objective is to provide generic adapters for numbers and letters, and ensure that method calls which applies to these objects are in fact calls to dynamically defined functions in OCaml. Thanks to the internal mechanisms of the language, a closure can be registered in the OCaml heap and be identified with a name, which can be used to call the closure from the C code.

Also, data types in OCaml can be explored in C code very easily, with a bunch of macros. So, OCaml variable exploration or creation can be done in C code with full safety.

Another useful point is the C representation used for OCaml data: they can all be accessed and manipulated from a special type `caml_value_t`, which is a typedef for `long`. Your data can be either a sum type, a closure or an array, the unique C representation is a `caml_value_t` variable, which can easily be assimilated to a pointer. In fact, this variable is absolutely not a pointer, it is an OCaml structured type containing special informations like garbage collecting, etc. The structure is fully described in [5].

OCaml is not a scripting language. Although, the code can be interpreted in a top-level interaction loop. This top-level can be customized, with the grammar extension and the object files needed for an interpreter for Vaucanson.

Using OCaml leads us to have both an interpreter and a programming language very powerful and well known of students or research community.

3.2 Solving the alphabet problem

One of the main issue in automata manipulation is to be able to work on every kind of alphabet. The theory says that the alphabet is a monoid (in fact it is often a free monoid). The point here is to be able to use every type for letters that we can imagine.

The previous approaches were to offer predefined alphabets to the user. Usually, `char`, `int`, `char * int` and `char * char` gives a lot of computation possibilities. But in order to be generic, it is better to define the letter type in the interpreter, and use it in Vaucanson.

In GP, all types must be statically known. To achieve this, the solution is to provide a generic adapter for OCaml values. The generic adapter will always be the same, so the compiler will be able to instantiate every templates asking for a letter with this adapter.

```
class GenericLetter {
  public:
  GenericLetter();
  ...

  private:
  caml_value_t  val;
}
```

Here the C representation of all OCaml variables with `caml_value_t` is used, and is helpful to keep the adapter as simple as possible.

But the major point of this adapter is to emulate the behavior of a "normal" letter. All expected methods must be added to the class, and all of them must call a dynamic OCaml closure.

```
class GenericLetter {
  ...
  operator +(GenericLetter& l);
  ...
}
```

For a letter, the operators are the classical arithmetic operators (+, -, *, /) expected from a char, and comparison operators. The comparison operators can all be emulated from equality and less operators. The class also needs a printing operator, useful in Vaucanson.

The implementation of this class in Vaucanson is done shortly, and has a well suited place in `algebra/implementation/letters`.

But solving the letter problem does not fully solve the alphabet problem. A letter container must be defined, in order to make alphabets or words over this alphabet. Till now, a Vaucanson alphabet container is used (`AlphabetSet`) and `std::basic_string<GenericLetter>` is used for words. Since the STL is used, the concatenation operator of the monoid is the concatenation operator on `basic_string<T>`.

3.3 solving the semiring problem

Genericity over semiring is the most suitable genericity for multiplicity automata. People often work on Boolean semiring, but it is convenient to dispose of numerical and tropical ones, or to be able to define a special semiring for transducers with multiplicity in the second alphabet [12].

The first point is to get generic over numbers. It seems now obvious that the same method will be used for numbers than for letters. A generic adapter is provided for numbers, providing all needed methods which are expected from a number. They are similar to the letter ones: +, -, *, / and in place equivalent operators, and all comparison operators. As above, a printing function is needed.

The generic adapter is called `GenericNumber`, and takes place as an implementation in `algebra/implementation/semiring`.

But like for letters, solving the number problem doesn't solve the semiring problem. A semiring is an algebraic set, defining zero and identity. The best solution is to provide a generic semiring, with special methods for zero and identity, calling a dynamic closure defined in OCaml.

This semiring takes place below `SemiringBase` in the Vaucanson hierarchy, since our generic semiring can be numerical, tropical, . . .

There are also special functions expected from a semiring element in the automata theory. Numbers must support the Kleene star [12] for some algorithms. It is not always necessary, but this operation must be statically defined. An implementation method is discussed in section 3.5, which ensure safety when methods are not defined in OCaml context.

3.4 Interface creation in Swig

All classes and functions available in the interpreter must be specified in the interface file of Swig. In Vaucanswig, Poss chose to work with multiple interface files, for each type of automata (boolean, numerical, tropical, . . .). Since we now use one single semiring type and one single letter type, a single interface file is used.

This file contains an automaton class, with all needed methods. Here we don't map every method from the Vaucanson automaton class, but only the ones needed for computation. Till now, this seems to be sufficient, but a full map of all Vaucanson methods is studied. The point is to not overload the interpreter with useless methods used only in Vaucanson internal computations.

A class for K-Rational expressions is also provided, since special methods are expected from that kind of structure.

Finally, all algorithms are mapped in the interface file, so they will be available in the interpreter.

3.5 Some needed glue in C++

The solution presented in section 3.2 comes with some special functions in C++. In fact, this C++ glue comes from two issues:

- the experimental status of the Swig binding to OCaml,
- the will to get safe in the interpreter.

In a matter of fact, the actual binding in OCaml is quite poor. This binding supports Caml, so it doesn't translate C++ objects into OCaml objects. Till now, it doesn't handle most of the STL containers. If the output of a function is a `std::set<T>`, you have to build you own output set.

Three main points were needed:

- encapsulate the calls to the OCaml closures for class methods, in order to be safe,
- encapsulate special functions which outputs structured types,
- provide an automaton constructor, which translate a "list" of letters (i.e. an alphabet) in a Vaucanson alphabet.

Generic adapters methods are said to call OCaml closure. In fact, this is false. They delegate the call to some externalized functions, called `caml_wrappers`. The methods for a generic adapter are divided in three parts : those which return the same type as the parameters (for example arithmetic operators), those which return a boolean (for example comparison operators) and those returning a string (the printing functions). For all of them, the structure of the wrapper is the same:

```
...
value * closure_f = NULL;
closure_f = caml_named_value(method_name);
if (closure_f != NULL)
    // the closure is defined. Call it.
    result = callback2(*closure_f, arg1, arg2);
else
   // the closure is not defined in the heap, handle safely the error.
...
```

But regarding their output type, a special treatment is applied to the result using OCaml special macros for value manipulation. For example, for a Boolean:

```
...
return Bool_val(result);
...
```

These wrappers allow to have a very simple code in the generic adapter, and to ensure a maximum safety in the case of undefined methods. You can prompt a message on the standard error with the name of the undefined function and its signature, throw an exception, ... In fact it is possible to start with an interpreter with very few defined functions, and wait for error messages to just add needed ones.

The second point is to encapsulate functions that output structured data types. Till now, the automaton structure used is Vaucanson automaton so for example if you want to retrieve the alphabet defined in OCaml, you have to convert your `AlphabetSet` to a `'a list`.
This is also needed for some specific STL containers used in Vaucanson. For example, delta functions output a `std::set<hedges_t>`. It is needed, even if hedges can be manipulated as int, to convert this container to an OCaml one.

The last point is to provide a constructor for automata. When you are manipulating an alphabet in OCaml (a list of letters), you need to convert that list in a Vaucanson compliant alphabet, and then you can build the automaton.

In addition of these points, a context was created in Vaucanson to make generic adapters accessible in an automaton context. This context is called `generic_automaton`.

3.6 Some needed glue in OCaml

The OCaml swig module is in fact a Caml module. So objects are not handled like OCaml objects, and a grammar extension is provided by Swig to enable syntax like `object->method(args)`.

It was necessary to add few functions in OCaml, in order to handle properly these lack of functionalities in the module. Also, the system used to emulate objects is quite heavy, and was bypassed as much as possible by another new system based only on values. That leads in the interpreter to handle values and not objects for automata and K-Rational expressions.

Chapter 4

Grammar extensions

One of the reason of the choice of OCaml was its easiness in grammar extension. Camlp4 is the new standard OCaml pre-processor. It acts like a parser, with the possibility of rule extension, rule removal, lexer extension, . . .

All files using the extension must be pre-processed with the grammar extension files.In the interpreter, three different syntax are needed:

- standard syntax,
- Swig extension,
- Vaucanson extension.

The Swig OCaml extension is described in [1]. The Vaucanson extension is presented in this chapter.

4.1 Prepare the extension

Syntax extensions can be seen as a different way to call a known construction in the syntax. In order to ease the extension, some constructors are written in OCaml, as closures:

- automaton constructor as a six-tuple,
- K-Rational expression set constructor,
- semiring constructor,
- set operators.

These constructors can be used as is, just have a look in the code to do so. But they are suited to ease grammar extension, and not to be manipulated directly.

4.2 What we want to write

Regarding the disadvantage of Vaucanswig syntax, the following points are needed:

- use OCaml types everywhere,

- get genericity over alphabets (i.e. have only one constructor for automata),
- define new semirings easily.
- manipulate sets easily,

In the interpreter, all sets are now handled with list. So you write:

```
let states = [0; 1; 2]
let alphabet = ['a'; 'b']
let alphabet_pair = [('a', 1); ('b', 1)]
```

4.3 Automata construction

Let an automaton be a six-tuple [12]: automaton = $< Q, A, \mathbb{K}, E, I, T >$

The main syntax extension is to provide a constructor as similar as possible to the mathematical one:

```
let states = [0; 1; 2]
let alphabet = ['a'; 'b']
let edges = [(0, 'a', 1); (1, 'b', 2)]
let initials = [0]
let finals = [2]
let automaton = < states, alphabet, boolean_semiring,
                  edges, initials, finals >
```

The format kept for edges is a set of three-uple: `int * letter * int` where the state is represented by an `int`. `letter` must be the type of the alphabet list element.

For many of the general uses, this construct is sufficient. But you may want to add multiplicity in edges. This is possible with the keyword `mult`:

```
let automaton =
< states,
  alphabet,
  numerical_semiring,
  mult [(0, ('a', 1), 1);
        (1, ('b', 4), 2)],
  initials,
  finals >
```

Here the edges set is of type `int * (letter * multiplicity) * int`, where multiplicity must be the type of a semiring element.

It is also necessary to be able to build spontaneous transition. This is possible thanks to the keyword `eps`, which turns the six-tuple to a seven-tuple. (Another solution is studied, but this one is the most relevant at this time.)

```
let automaton =
< states,
  alphabet,
  numerical_semiring,
  edges,
  eps [(0, 1); (0, 0)],
  initials,
  finals >
```

You can of course use any kind of alphabet with exactly the same constructor:

```
let alphabet = ['a'; 'b']
let alphabet_pair = [('a', 1); ('b', 1)]
let automaton = < states, alphabet, boolean_semiring,
                  edges, initials, finals >
let automaton_pair = < states, alphabet_pair, boolean_semiring,
                       edges, initials, finals >
```

Another constructor is studied, designed especially for Boolean automata. This constructor would take only five parameters, ommiting the semiring and set it to `boolean_semiring` by default. This constructor is not present in the interpreter yet.

4.4 Semiring construction

In automata construction, the semiring is needed. A method to use generic semiring was described in section 3.3, in Vaucanson point of view.

A semiring is an algebraic set, with special addition and multiplication operations, both having their neutral element. Fosse proposed a similar syntax in [7]. To declare a semiring:

```
let n_semiring = < (+):0, (*):1 >;;
...
let add a b = a || b;;
let mul a b = a && b;;
let boolean_semiring = < add:true, mul:false >;;
```

The syntax is simple: a semiring is defined by its calculus properties. So if you define the two operations and their respective neutral elements, you defined the whole semiring. This syntax differs from Fosse in the lack of the set of elements. In OCaml, type inference permits to omit the working set, since it is deduced from the type of the arguments of the two operations. And if you want to work on set of odd numbers, for example, you can do so within the operations.

But it is also needed to support the Kleene star, even if it is not always necessary. That's why an alternative constructor is proposed:

```
let is_starable x = x == 0;;
let star x = 0;
let n_semiring = < (+):0, (*):1, is_starable, star >;;
```

4.5 Set construction

Automata work on sets. It is quite obvious that a bunch of functions for algebraic sets manipulation would be welcome:

```
let sum = [0; 1; 2] |/ set2;;
let diff = set1 /| set2;;
let x = 4;;
let is_included = x <! set1;;
let not_included = x <+ set1;;
```

The main set operations are provided : union, intersection, element membership and non-membership. Till now, sets are implemented with OCaml lists, but another method is studied, enabling intention lists construct.

Since Camlp4 is a very powerful an simple tool, all extensions proposed here can very easily be changed by the user, especially the one for sets which are still experimental at the moment.

4.6 Other possible extensions

When working on automata, you work on special algebraic sets like monoid or semiring. But you also work on series, for example. Fosse introduced a lot of very interesting syntactic points which will be integrated in the syntax extension as much as possible.

Chapter 5

Review and generalization of the solution

The solution presented in the previous chapters is a case study for Vaucanson. It is actually implemented, as a proof of concept. The glue in C++ is about 300 lines, which can be brought to less than 100 lines if the Swig OCaml module was improved; the glue in OCaml is about 350 lines including 50 lines of grammar extension, and the addition done in Vaucanson is three concrete class, with very few methods.

5.1 Advantages and drawbacks

The initial objectives were:

- be able to define a type for alphabet or semiring in the interpreter and use it with Vaucanson,
- dispose of syntactic extensions for automata manipulation.

All these goals were achieved. In addition to them, the solution brings:

- fast compilation of the interpreter,
- user code can be interpreted or compiled,
- syntax can be easily extended / modified,
- few performances loss.

Actually performance is not an issue when designing an interpreter. The user is aware that it is not the point of this kind of interface, he use the library when he needs high performance. But as said above, the code can be interpreted and compiled. A little bench (figure 5.1) proved that the loss of performance was quite reduced, and unexpectedly almost linear (factor 1.8 to 1.5 in the bench) and tends to decrease with the complexity of the treatment !

The following bench was realized with g++ 3.3 against ocamlopt, without optimization flags, on product and minimization algorithms:

g++	0.196	0.970	4.440	4.948	21.583
ocamlopt	0.358	1.785	7.305	8.119	33.591

Figure 5.1: Bench Vaucanson C++ source against OCaml source

But even if this solution comes with great advantages, there are drawbacks:

- works only with one kind of automata,
- works only on a subset of Vaucanson's functionalities,
- loss of Vaucanson's typing force.

The fact that the interpreter works only with one automaton representation is not a global restriction, but a wish to limit the complexity of the interpreter. Till now, the interpreter is not automatically build. A method is studied to mechanize the process, and to provide in the interpreter the three big kinds of automata of Vaucanson: "standard" ones, generalized ones, and transducers.

The major drawback of this method is the loss of Vaucanson's typing force. When you use Vaucanson, you are almost certified that if the code compile, it is algebraically correct. And if you want to do so in the interpreter, you have to provide the same predefined structures than in Vaucanson and that leads to code duplication.

Furthermore, the interpreter must have an improved typing system, since the user is lead to handle values generated in C. Actually these issues are almost solved, and will totally be solved soon.

5.2 Possible improvements

Most of the drawbacks initially raised with this method were solved easily, especially issues about typing restriction in OCaml. Lot of improvements are studied, but lot of the work implied by the interpreter construction was in fact to provide correct support of C++ objects in OCaml. This issue comes from the poverty of the Swig OCaml module. The major improvement to do is to write a new Swig module, working only with OCaml (and not backward compatible with Caml) that will handle properly objects. This module should also be able to hold more STL containers, like `std::set`.

5.3 Toward generalization

The method proposed in this report is based on the fact that abstractions and implementations are separated in the design pattern of Vaucanson. But this method can easily be generalized to other design patterns doing the same separation.

5.3.1 Recall on element and generic bridge design pattern

In the Element design pattern, the abstraction is kept separated from the implementation, as shown in figure 5.2.

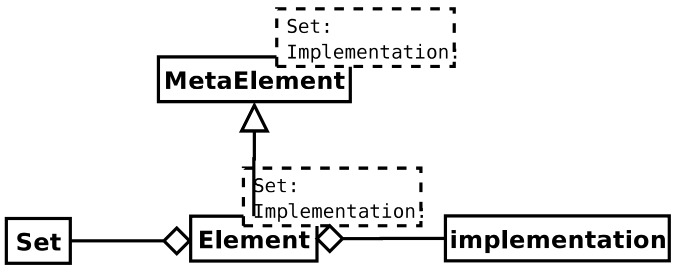

Figure 5.2: The element design pattern

This design pattern, fully described in [11], is perfectly suited to handle concrete classes needed by the interpreter.

The generic bridge do a similar distinction of abstraction and implementations, as shown in figure 5.3.

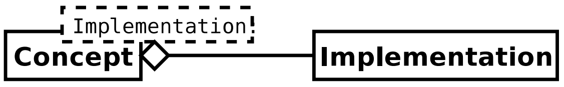

Figure 5.3: The generic bridge design pattern

Both of these patterns have the property to allow multiple specialization of an algorithm, regarding the concept or the implementation.

5.3.2 Generalization of the method

The principle is simple: when a concept aggregates several implementations, they all are expected to provide a same minimal interface. For example, in SCOOP [4]:

```
template <class A_impl>
class A {
  ...
  void foo() {
    return this->exact().foo_impl();
  }
  ...
}
class A_impl {
  void foo_impl();
}
void algorithm(A& abs) {
  abs.foo();
}
```

The work is to provide an implementation `A_gen` for `A` where all methods and attributes of `A_gen` are in fact calls or values from OCaml, using the method described in section 3.2.

```
class A_gen {
  ...
  void foo_impl() {
    return caml_wrapper("foo_impl");
  }
  ...
}
```

The `caml_wrapper` is described in section 3.5.

So, for each wanted abstraction where the implementation can vary, a generic implementation must be provided, with the same minimal interface as the abstraction class.
But there is a huge distinction to do between implementation as seen in these patterns and implementation that must be redefined. The best example is shown in Vaucanson, where getting genericity on alphabets is in fact getting genericity over letters, since the alphabet structure and methods don't change. A good way to isolate what must be defined is to see where attributes type of concrete class must vary and to do an adapter for these types, following the method of section 3.2.

Although, two important issues must be considered:

- all algorithms written for abstraction `A` must have a general implementation, and not only specialization for every possible implementation `A_impl`,
- the interface of `A` should not be too complex, or there will be an important work of method redefinition in the interpreter.

The major issue of GP is the absolute necessity for all types to be known at compile time. This leads the compiler to be able to optimize the code, and gives all its advantage to GP. But it is often pleasant to manipulate the statically written library in a dynamic environment, where performances are not an issue.
To do so, we let the compiler think the types are known statically, with a generic adapter for implementation classes where needed. This adapter fully substitute a regular implementation.
Of course, the compiler is unable to optimize the code as well as it would do with a regular implementation, and specializations for particular implementations are not available.

Chapter 6

Conclusion

The method presented in this report fulfilled our requirements, since it is possible to have an interpreter for Vaucanson, a statically typed library. The use of Swig let us to have very few development time of the interpreter, and the OCaml language fits very well the objectives of syntax extension and easiness in automata manipulation.

Furthermore, in order to use types dynamically defined in Vaucanson, it was necessary to define some mew implementations inside the library, and this lead to a possible generic method for interpreter creation in statically typed context.

Although, even if the method has its advantages, the solution needs to be polished before being able to do final conclusion and some implementation details must be generalized to get a global method.

Chapter 7

Bibliography

[1] Swig 1.3 reference manual.

[2] Matthew H. Austern. *Generic programming and the STL*. Addison-Wesley, 1998.

[3] John Aycock. A brief history of just-in-time. In *ACM computing surveys*, volume 35.

[4] Nicolas Burrus, Alexandre Duret-Lutz, Thierry Geraud, David Lesage, and Raphael Poss. A static c++ object-oriented programming (scoop) paradigm mixing benefits of traditional oop and generic programming. MPOOL'03, 2003.

[5] Emmanuel CHAILLOUX, Pascal MANOURY, and Bruno PAGANO. *Objective Caml*. O'Reilly, 2010.

[6] Daniel de Rauglaudre. Camlp4 - reference manual, September 2003.

[7] Loic Fosse. Domain specific language on automata. Technical report, Epita Research and Development Laboratory, 2003.

[8] Loic Fosse. Dynamic use of statically typed libraries. Technical report, Epita Research and Development Laboratory, 2004.

[9] Sylvain Lombardy, Yann Regis-Gianas, and Jacques Sakarovitch. Introducing vaucanson. 2004.

[10] Raphael Poss. Liaison de bibliothèques a généricité statique avec un langage interprèté. Technical report, Epita Research and Development Laboratory, 2002.

[11] Yann Regis-Gianas and Raphael Poss. On orthogonal specialization in c++: dealing with efficiency and algebraic abstraction in vaucanson. POOSC'2003, 2003.

[12] Jacques Sakarovitch. *Elements de théorie des automates*. 2003.