# On orthogonal specialization in C++

## Dealing with efficiency and algebraic abstraction in Vaucanson

Yann Régis-Gianas, yann.regis-gianas@lrde.epita.fr,
Raphaël Poss, raphael.poss@lrde.epita.fr

LRDE: EPITA Research and Development Laboratory
14-16, rue Voltaire - F-94276 Le Kremlin-Bicêtre Cedex - France
http://www.lrde.epita.fr/

**Abstract.** The Vaucanson library works on weighted finite state machines in an algebraic framework. As computing tools, FSMs must provide efficient services. Yet, abstraction is needed to obtain genericity but also to define properly what objects we are working on.
Even if parameterized classes are a known solution to this problem, the different kinds of algorithm specializations are limited when using usual template techniques.
This paper describes a new design pattern called ELEMENT which enables the orthogonal specialization of generic algorithms w.r.t. the algebraic concept and w.r.t. the implementation. The idea is to make concept and implementation explicitly usable as object instances.
First, we show how it solves the specialization problem. Then, we detail its implementation and how we deal with some technical pitfalls.

Vaucanson is a C++ generic library for weighted finite state machine manipulation. For the sake of generality, in Vaucanson FSMs are defined using algebraic structures such as alphabet (for the letters), free monoid (for the words), semiring (for the weights) and series (mapping from words to weights) [5]. As usual, the challenge is to maintain efficiency while providing a high-level layer for the writing of generic algorithms. One of the particularities of FSM manipulation is the need for a fine grained specialization power on an object which is both an algebraic concept and an intensive computing machine.

Vaucanson is the core of a project initiated in 2001 by Jacques Sakarovitch of the École Nationale Supérieure des Télécommunications (ENST, Paris). The project is now a collaborative work between the ENST and the École Pour l'Informatique et les Techniques Avancées (EPITA, Paris).

## 1 Algorithms for weighted finite state machines

### 1.1 Two points of view

On the one hand, the mathematical aspect of automata requires the definition of a precise context. Indeed, an algorithm must specify on what kind of semiring or alphabet it works. A hierarchy of algebraic concepts is necessary to make their context explicit. Such a hierarchy can be found in any book about algebraic structures.

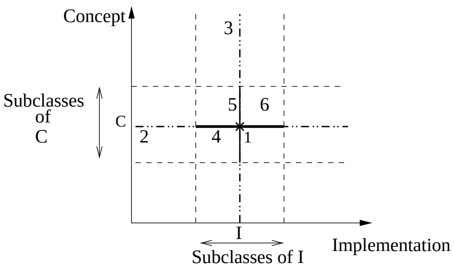
On the other hand, weighted finite state machines are used to process large amount of data. In addition, algorithms on automata can have exponential complexity, so primitive operations should be as fast as possible. Efficiency cannot be sacrificed to gain the convenience of abstraction. Choosing the most relevant data structure is essential. However, many data structures exist to represent letters, alphabets, words, weights, series and automata. Furthermore, they are highly correlated since an automaton is built with series, a series is defined by words and weights, and a word by letters. Then each implementation is parameterized by some other implementations leading to something like a nest of dolls. Such implementations cannot be easily mixed in a monolithic hierarchy. Also, we want to reuse data structures from external libraries.

Thus, the design problem is to unify these two points of view into the same object to enable both implementation-driven and algebraic-driven writing of algorithms.

### 1.2 Generic algorithms and specialization power

Abstraction has lead to many algorithms with a general formulation. Generic programming is relevant, because general algorithms should be written once. More precisely, an algorithm can be generic w.r.t the mathematical concept and w.r.t the underlying data structure used as implementation of that concept.

However, some theoretical results are restricted to a precise algebraic context. Thus, we must be able to bound the algorithm input to a particular family of concepts. Likewise, algorithms can be written using the properties of a particular implementation, so restriction facilities over implementation parameters must be available. The figure 1 sums up some desirable specializations.




1. $I, C$ are fixed ;
2. all $I$ and $C$ is fixed ;
3. all $C$ and $I$ is fixed ;
4. all sub-classes of $I$ and $C$ is fixed ;
5. all sub-classes of $C$ and $I$ is fixed ;
6. all sub-classes of $C$ and $I$.

**Fig. 1.** Different type boundings of algorithm input

To be transparent to the final user and to improve genericity, these different specializations must be compatible with overloading.

### 1.3 Plan

The sequel of this paper is organized as follows. The section 2 shows the lack of answer in the well known C++ template techniques. Then, the section 3.1 presents our design pattern. In section 4, we apply it and observe that it fills our requirements. Finally, in section 5, the implementation techniques are presented to explain precisely how we deal with some pitfalls.

## 2 Confronting our desires with C++ template techniques

In practice, polymorphism implemented with late binding is too expensive for intensive computing. The generative power of C++ template mechanisms is known to enable abstraction with limited efficiency loss. The Standard Template Library (STL) has shown the workability of such polymorphism [4].

Yet, parameterization *à la* STL is unbounded. We cannot define two generic functions with the same name and the same arity because type variables are free. The Barton and Nackman trick [6] and other works [2] tend to reproduce the object oriented programming. The idea is to compel the open recursion to be static, *ie* the static type system knows exactly the subclasses that are used as instantiations of a particular abstract class. The following code attempts to illustrate this idea:

```
// This version is valid for any sub-class of A.
template <class C>
void algorithm(const A<C>& i);

// This version is valid for any sub-class of B.
template <class C>
void algorithm(const B<C>& i);
```

However, the one-dimensional discrimination of a single object hierarchy is not enough to design both the mathematical concept and the implementation. At first sight, the BRIDGE design pattern [3], or more precisely the GENERIC BRIDGE [1] design pattern could be suitable. Yet, the GENERIC BRIDGE is asymmetric, it is centered on the concept. Consequently, if the object is a concept parameterized by its implementation, specialization of type 4 is forbidden because of the invariance of the template argument. The following code illustrates this:

```
struct Matrix {};
struct CompressedMatrix : public Matrix {};

// This cannot be called with arguments of type A<CompressedMatrix>.
void algorithm(const A<Matrix>& i);
```

## 3 The ELEMENT design pattern

### 3.1 Presentation of the design pattern

Traditionally, concepts and implementations are separated by using abstract and concrete classes in hierarchies. Doing so, the chosen implementation for member function calls only depends on the actual class type. Then, a concept can be denoted by an abstract class whose subclasses are its implementations.

Our idea is to make explicit the separation between the concept and the implementation at the object level. We compose our entities with an instance of a concept class and an instance of an implementation.

By separating concepts and implementations in different hierarchies, we allow separate refinements of concepts and implementation algorithms. Moreover, we allow to use the same data type to implement distinct concepts, without the hassle of defining whole new concrete classes.

For example, elements of a tropical semiring are distinguished by their association with an instance of the concept $(\mathbb{Z}, max, +)$, and can be implemented by several basic C++ integer types. Conversely, basic C++ integer types can either represent elements of a tropical semiring or elements of a "classical" semiring $(\mathbb{Z}, +, \times)$, depending on the concept instance they are linked to. In either case, a single class $Element$ is responsible for the composition.

As demonstrated later, this design entails more freedom and specialization facilities.

For the sake of simplicity, we denote the abstract concept associated to an entity, instance of class $Element$, its *structure* and the corresponding instance the *structural element*.

### 3.2 A two-component generic object

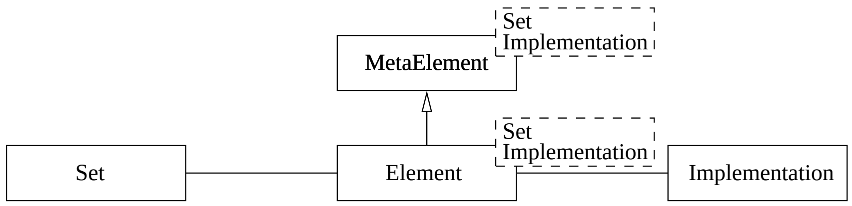



**Fig. 2.** Class diagram of the ELEMENT design pattern

The main item in this pattern is the $Element$ class, a generic class which acts as a glue between a concept and an implementation. Indeed, the pattern can be read as $Element{<}S, T{>}$ is *the type of an element of set $S$ implemented by $T$*, or in other words $Element{<}S, T{>}$ *structures the value type $T$ by $S$*.

All services except construction and assignment are provided by decoration through the parent class $MetaElement$. Class $MetaElement{<}S, T{>}$ serves three roles:

– *to specify the interface* for the class $Element{<}S, T{>}$ viewed as an implementation of concept $S$ denoting *how any instance of $T$ must fulfill the requirement of the concept $S$*,
– to offer additional abstract services implemented using only services defined in the specified interfaces,
– to *link* services to their external implementations.

Of course, this class must be specialized over $S$ and $T$. For additional genericity, the hierarchy between concepts should be mapped to a hierarchy between their specializations of $MetaElement$, so that the final $Element{<}S, T{>}$ is decorated in correlation to the inheritance graph of concepts.

Figure 2 describes the design pattern in UML, while figure 3 details the decoration mechanism.

### 3.3 External functions as an adaptation layer

To reduce the number of *MetaElement* specializations, default code for concept requirement implementation is needed. We could define it directly in the specialization of $MetaElement\langle S, T\rangle$ for $S$ fixed and $T$ free, assuming the presence of methods and inner types (such as `begin()` / `end()`, `iterator`, *etc*). Yet, this solution inhibits implementation with partial default behavior. Moreover, this forces implementations to be C++ classes, whereas C++ builtin types or externally-defined structures can also be wanted as implementation types.

We decided to use external functions to define how the implementation fulfills concept requirements. Therefore, any *MetaElement* specialization is just a way to choose what external functions are to be used. Doing so, we introduce a fine grained specification of implementation services. At the same time, we also solve some binary method problems.

## 4 Applying the pattern: decomposition for specialization control

Given class *Element*, we can decompose any entity, or *Element* instance, for typing purpose. The following listing illustrates generic way to implement algorithms over *Element*:

```
// Generic wrapper
template <typename S, typename T>
void algorithm(const Element<S, T>& e)
{
  // Call the implementation, decomposing
  // the Element instance along the way.
  return algorithm_impl(e.set(), e.value(), e);
}
```

Once this framework is set up, implementations of `algorithms` can be specialized in any of the directions illustrated in figure 1. This is done by the following constructions:

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

// Type 2: concept fixed, generic implementation for any value type.
template <class T>
void algorithm_impl(const S1& s, const T& v, const Element<S1, T>& e);

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

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

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

// Type 6: generic implementation for any sub-class of (S1,T1).
```

```
template <class S, class T>
void algorithm_impl(const S1& s, const T1& v, const Element<S, T>& e);
```

# 5 Implementation of class *Element*

## 5.1 Design considerations

The implementation of class template *Element*, and therefore the whole structure of the design pattern, was subject to the following guidelines:

– object instances of class *Element* should behave as "naturally" as possible w.r.t. the user. Especially, a user who has no experience with the library should be able to infer most of the use cases of *Element* from simple examples.
– the behavior and the set of available services in class *Element* can change depending on its static parameters. For example, instances of *Element* intended to represent values in an algebraic semiring have a `star()` method. Similarly, instances intended to represent automata have an `add_state()` method.
– at any time, a reference to the structural element of an *Element* instance can be retrieved with no computation cost. For example, it is possible from an instance of *Element* intended to represent a word to retrieve the whole alphabet over which it is defined.
– singleton structures should induce no memory footprint in *Element* instances. For example, there is no run-time data associated with the canonical semiring structural element surrounding the basic C++ types (`int`, `short`, ...). Therefore corresponding *Element* instances should be as small (from the C++ compiler's point of view) as the basic C++ types used as value types, for optimization purposes.

There are three facets in the current implementation of class *Element*, closely related to the requirements presented above.

## 5.2 $Element\langle S, T\rangle$ as a wrapper around $T$

Because $Element\langle S, T\rangle$ is actually a wrapper around type $T$, its main role is to aggregate a value of type $T$. Therefore, a number of basic services to handle the value data are provided by class *Element*, presented in table 1. Their use is valid iff the corresponding requirements over type $T$ are met.

These basic services are trivially implemented using only the properties of type $T$. They are therefore distinct from all additional services presented below, which also depend on type $S$ and on the availability of related operators.

## 5.3 $Element\langle S, T\rangle$ as an element of a set

The power of our design pattern is that the same data type $T$ can be structured by several distinct structural elements.

However, parameterization of class *Element* by the static type of its structure $S$ is usually not sufficient. Indeed, a structure type $S$ may denote several different structural elements with distinct behavioral influences on $Element\langle S, T\rangle$. For instance, this can be observed in Vaucanson when using tropical semirings where the special infinity value is

| Description | Example use | Requirements |
| --- | --- | --- |
| Referencing | `Element<S,T>& e;` `const Element<S,T>& ce;` | (none) |
| Access to the aggregated value | `T& v = e.value();` `const T& cv = ce.value();` | |
| Default construction | `Element<S,T> ev;` | $T$ default-constructible |
| Copy construction | `Element<S,T> ev(ce);` | $T$ copy-constructible |
| Construction from a value | `Element<S,T> ev(cv);` | |
| Assignment | `ev = e;` | $T$ assignable |
| Destruction | | $T$ destructible |

**Table 1.** Services of class $Element{<}S, T{>}$

dynamically defined: the static type information ($TropicalSemiring$) is not sufficient to express the correct computation of addition and multiplication in the semiring, because the actual, dynamic value for infinity must be tested.

Because of this, we chose to hold a reference to the structural element in each object instances of $Element{<}S, T{>}$. For this purpose, $Element{<}S, T{>}$ aggregates a reference to an instance of $S$ *via* a mechanism presented in section 5.5. This reference can be retrieved with the `set()` method. For consistency purposes, the following properties must hold:

- once defined, the structural element of an $Element$ instance cannot be changed nor modified; this is virtually ensured by `set()` returning a "const" reference.
- structural elements must be classifiable by means of `operator==`; this helps keeping[1] global instances of structures, using unique references to designate unique structural elements, for efficient by-reference comparisons.

Linking $Element$ instances to structural elements is done at instantiation time, using the following construct:

```
Element<S,T> e(/* structural element */ s, /* value */ v);
```

Take note of the additional argument `s` given to the constructor of class $Element$. This construction does not invalidate the construction style presented in table 1; in fact, $Element$ instances that have been constructed without giving a reference to the structural element are in a state called "transitional", during which only the basic operations are valid. Passing to the normal state is done by post-construction binding to a structural element with the `attach()` method:

```
Element<S,T> e(v);
// Here e is in the transitional, incomplete state.
e.attach(s);
// Now e is fully defined.
```

### 5.4 Subjecting the behavior of values to structures

The design of class $Element$ targeted maximum extensibility via template specialization and method overloading, as presented in section 1.2. It was achieved by delegating computation for *all* services offered by class $Element$ to global functions with special names (of

[1] The uniqueness is ensured by a global type table discussed in section 5.6.

the form `op_X`, for each operation `X`). These can then be refined *via* template specialization and function overloading (as in section 4).

By default, this delegation is set up for all standard C++ operations; table 2 shows how delegations are expressed and table 3 shows the mapping between standard C++ operations and special function names.

| Description | Operation | Function call |
|---|---|---|
| binary operations | `e1 Op e2` | `op_##OpName(e1.set(), e2.set(), e1.value(), e2.value())` |
| | `e1 Op v1` | `op_##OpName(e1.set(), e1.value(), v1)` |
| | `v1 Op e1` | `op_##OpName(e1.set(), v1, e1.value())` |
| | `e1 Op v2` | `op_##OpName(e1.set(), e1.value(), op_convert(e1.set(), SELECT(T1), v2))` |
| | `v1 Op e2` | `op_##OpName(e2.set(), op_convert(e2.set(), SELECT(T2), v1), e2.value())` |
| difference | `e1 != e2` | `!(e1 == e2)` |
| comparison | `e1 > e2` | `e2 < e1` |
| | `e1 >= e2` | `!(e1 < e2)` |
| | `e1 <= e2` | `!(e2 < e1)` |
| negation | `- e1` | `op_neg(e1.set(), e1.value())` |
| prefix incr. and decr. | `Op e1` | `op_in_##OpName(e1.set(), e1.value())` |
| postfix incr. and decr. | `e1 Op` | `Element<S1,T1> copy(e1); Op copy` |

`e1`: $Element<S_1, T_1>$, `e2`: $Element<S_2, T_2>$, `v1`: $T_1$, `v2`: $T_2$

**Table 2.** Delegation of standard C++ operations to function calls

| | | | | | |
|---|---|---|---|---|---|
| | | | | `operator=()` | `op_assign` |
| `operator+()` | `op_add` | `operator+=()` | `op_in_add` | `operator==()` | `op_eq` |
| binary `operator-()` | `op_sub` | `operator-=()` | `op_in_sub` | `operator<()` | `op_lt` |
| `operator*()` | `op_mul` | `operator*=()` | `op_in_mul` | prefix `operator++()` | `op_in_inc` |
| `operator/()` | `op_div` | `operator/=()` | `op_in_div` | prefix `operator--()` | `op_in_dec` |
| `operator%()` | `op_mod` | `operator%=()` | `op_in_mod` | unary `operator-()` | `op_neg` |
| | | | | `swap()` | `op_swap` |

**Table 3.** Mapping between C++ operator names and function names

Distinction between sets of delegations is made by parameterized inheritance of class *Element*. Indeed, $Element<S, T>$ inherits from $MetaElement<S, T>$, which is by default empty but can be specialized to provide additional methods. For example, in Vaucanson delegations such as star() (op_star) for semiring elements or add_state (op_add_state) for automata, have been added.

As a matter of fact, all the standard delegations are set up in *Element*'s root parent class, *SyntacticDecorator*, from which each specialization of *MetaElement* must inherit directly or indirectly.

Figure 3 shows a UML description of the model.

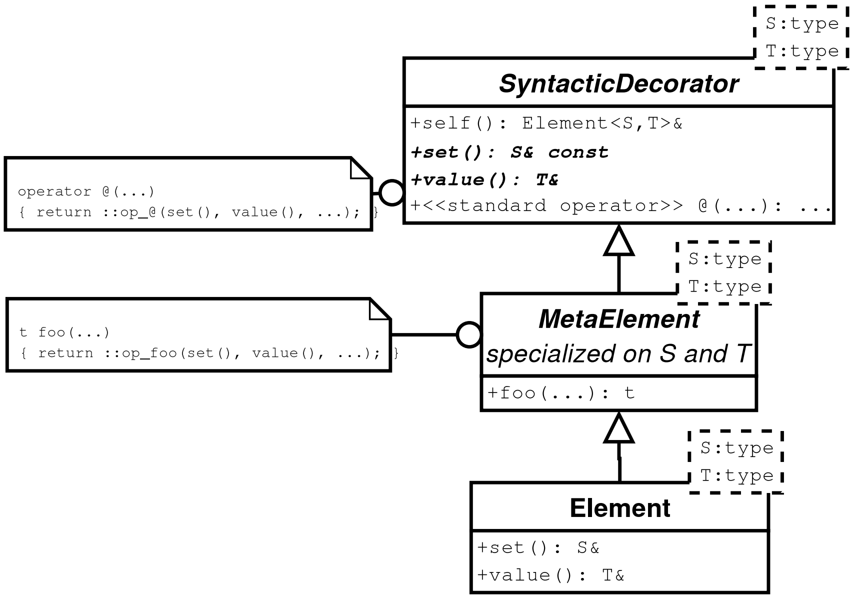



**Fig. 3.** Model for *Element*

### 5.5 Eliding references to structural elements

As presented in section 5.3, $Element<S, T>$ holds a reference to its structural element, an instance of type $S$. However in many cases a structural element is entirely defined by its static type $S$, i.e., there is no useful dynamic data associated to instances of $S$.

In these cases, a simple aggregation of a C++ reference (pointer) in $Element<S, T>$ would be a waste of memory space and time (for allocation and copy of the unneeded reference).

We avoided this waste by the encapsulation of the aggregation through a dedicated class, *SetSlot*. *SetSlot* derives from class *SetSlotAttribute*, parameterized by $S$ and a Boolean value: the specialization of *SetSlotAttribute* for the Boolean true actually has a pointer attribute, whereas its default specialization has no such attribute but an accessor that returns a null reference.

When instantiating *SetSlot*, the Boolean attribute passed to the parent instance of *SetSlotAttribute* is taken from the value of dynamic_traits<S>::ret, *dynamic_traits* being a helper which defaults its attribute `ret` to `false` but can be specialized for any structure type $S$.

This mechanism is illustrated on figure 4.

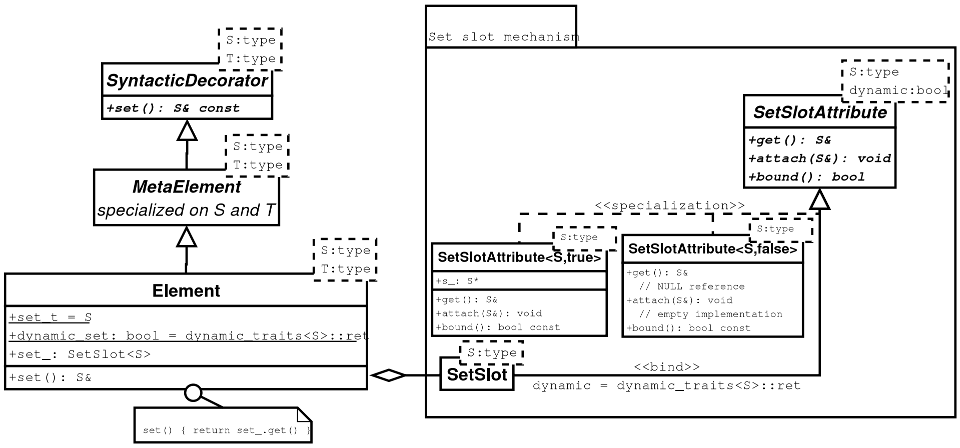



**Fig. 4.** Intelligent aggregation of references to structural elements

### 5.6 Ensuring unique instances of structural elements

To allow efficient by-reference comparison of structural elements, a mechanism was set up to ensure that all *Element* instances sharing the same *dynamic type* (structurally equal structural elements) share the same reference to a unique structural element.

Practically, it ensures that if any two distinct *Element*$<S, T>$ instances `e1` and `e2` were instantiated from distinct structural elements `s1` and `s2` verifying `s1 == s2`, then the property `&e1.set() == &e2.set()` always holds even if `&s1 != &s2`.

This was done by implementing an operator that keeps, for each static type $S$, a list of all distinct instances. It reports for any instance of $S$ the address of the equivalent instance in the list, adding it to the list if necessary.

This mechanism therefore implies two requirements over $S$ for *Element*$<S, T>$:

- $S$ must possess an equivalence `operator==`,
- $S$ must be copy-constructible, and values created by copy construction must be equal by means of `operator==`.

It is important to notice that this implementation is only efficient when there are few distinct instances of any structure. When a structure has many instances, containers such as `std::set` (requiring `operator<` over structural elements) or hash maps (requiring a hash function) could replace the list.

### 5.7 Return types for operators

Most operations over *Element* instances return values whose types are independent from their arguments. That is, the return type can either be the same *Element* type or another basic or compound C++ type. However, some operators, especially arithmetical operators, should return a type computed from the types of its arguments. In Vaucanson this is shown, for instance, in the multiplication of a polynom by a weight or the lazy transposition of an automaton, which returns its argument encapsulated in a dedicated *TransposeView* adapter.

For this purpose, most operators are associated with a dedicated trait structure which computes the return type from both the structure type and value type; the generalized form for operators is thus:

```
template<typename S1, typename S2, typename T1, typename T2>
typename op_##OpName##_traits<S1, S2, T1, T2>::ret_t
operator Op(const Element<S1,T1>&, const Element<S2,T2>&)
```

with `op_##OpName##_traits` being specialized as needed.

Of course, since it represents the most widely used case, the default return type for `op_##OpName##_traits<S, S, T, T>` is $Element<S, T>$.

## 6 Conclusion

The ELEMENT design pattern is relevant for the orthogonal algorithm specialization problem, and we are thus using it successfully in Vaucanson. The idea can be generalized to problems where objects are built with more than two orthogonal components. Therefore, we hope that this design pattern will be used in other fields.

Finally, we want to thanks Astrid Wang-Reboud, David Lesage, Nicolas Burrus, Niels Van-Vliet and Akim Demaille for their advises about both technical and writing issues.

## References

1. Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille. Design patterns for generic programming in C++. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, 2001.
2. Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, the computational geometry algorithms library. Technical Report 3407, INRIA, April 1998.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.
4. David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software - Practice and Experience*, 24(7):623–642, 1994.
5. Jacques Sakarovitch. *Elément de théorie des automates*. 2003.
6. Todd Veldhuizen. Techniques for scientific C++. Technical report, Indiana University Computer Science, 2000.