GENERIC SOFTWARE ABSTRACTIONS FOR AUTOMATA IMPLEMENTATION

Vincent Le Maout

Institut Gaspard Monge, Université de Marne La Vallée, Champs sur Marne, France*

lemaout@univ-mlv.fr

ABSTRACT

This paper presents a set of software abstractions or concepts firstly designed to satisfy the natural language processing demand for highly efficient and reusable software components. Inspired by the good results of the generic programming techniques applied to algorithms on simple sequences, the concepts developed here broaden the existing abstractions to more complex data structures, that is automata. In particular, special care is taken in devising the automaton counterpart of the sequence accessor (the iterator) called cursor and allowing encapsulation and traversals of the underlying data structure. The result is general purpose C++ implementations adaptable to a variety of time and space constraints levels bundled in an open-source library called the Automaton Standard Template Library.

Keywords: finite-state machine, automaton, generic programming, C++, object-oriented design, template

1. Introduction

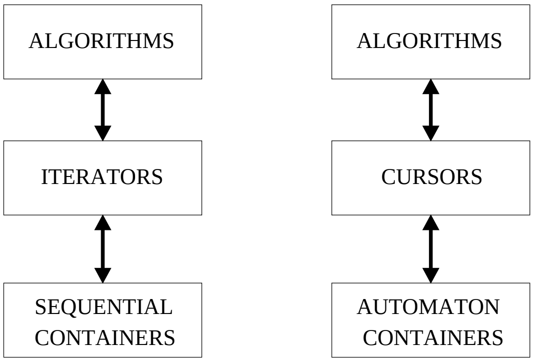
The interesting results of the application of the generic programming paradigm ([6]) to the design of the C++ standard template library (STL, [1], [9]) yielded a solid and consistent foundation for more ambitious projects tackling the issue of designing efficient and reusable software components for complex data structures as trees, graphs and finite-state machines ([5], [4]). The STL software abstractions address the need for reusable basic bricks that one can agglomerate and combine easily to build more sophisticated applications : sequential and associative containers (linked lists, hash tables, ...), accessors (iterators), memory allocators, object functions and frequently used sequential algorithms (sorts, searches, ...). Three of these concepts play a central part in the library design with the containers as first layer, managing the data and the collections of processed objects, the iterator as second layer granting the data access to the third one, the algorithms (see figure 1 where arrows denote interactions between components). By hiding the real involved data structure from the algorithm, the iterators allow for a higher component reusability since the lowest layer is totally uncoupled from the highest one. Moreover, the extensive use of C++ templates guarantees enhanced interchangeability and practical efficiency through static polymorphism.

It seems then natural to extend such concepts to automata programming where the needs are focused on generic traversal algorithms, operations on languages, grammars and flexible containers matching very different uses and constraints from the fast transition access of the matrix representation to the little memory consuming compact representation. Moreover, most of the time, the amount of processed data

*Supported by Lexiquest SA, http://www.lexiquest.com/

imposes on-the-fly and lazy computing versions of algorithms.
This paper introduces a new accessor concept, the cursor, equivalent for the automata to the iterator on sequences and presents several containers and generic algorithms using them, bundled in an open-source library called the Automaton Standard Template Library[a] (ASTL, [2]). We then show how to get three versions of an algorithm: the on-the-fly, lazy and by-copy implementations by simply writing the first one of them.

Figure 1: The three-layer designs of STL and ASTL



2. Definitions

2.1. Deterministic Finite Automaton

To make our concepts sufficiently generic to satisfy a wide range of algorithmic constraints, we add to the classical DFA definition a set of *tags*, that is any data associated to a state and needed to apply an algorithm. The set τ maps each state to a tag.
Let $A(\Sigma, Q, i, F, \Delta, T, \tau)$ be a 7-uple of finite sets defined as follow: Σ is the alphabet, Q a set of states, $i \in Q$ the initial state, $F \subseteq Q$ a set of final (accepting) states, $\Delta \subseteq Q \times \Sigma \times Q$ a set of transitions, T a set of tags, $\tau \subset Q \times T$ a relation from states to tags. We will omit tag-related information whenever it is unrelevant.
We distinguish one special state noted 0 and called the null or sink state. For every automaton $A(\Sigma, Q, i, F, \Delta, T, \tau)$ we have $0 \in Q$. The recognized language of a DFA is written $L(A)$.
We define $P(X)$ as the power-set of a set X and the right context of a state q : $\vec{c}(q) = \{\sigma \in \Sigma | \exists p \in Q, (q, \sigma, p) \in \Delta\}$, that is, the set of all letters of the outgoing transitions of a state.

2.2. Access Functions and Sink Transitions

To access Δ we define two transition functions δ_1 and δ_2 :

[a]ASTL is freely available at `http://www-igm.univ-mlv.fr/~lemaout`

$$\delta_1 : Q \times \Sigma \to Q$$
$$\forall q \in Q,\ \forall a \in \Sigma,\ \delta_1(q,a) = \begin{cases} p \text{ if } (q,a,p) \in \Delta \\ 0 \text{ otherwise} \end{cases}$$

δ_1 retrieves a transition target given the source state and a letter. In the case of undefined transitions the result is the null (sink) state.
A transition verifying the following property is called a *sink transition*:

$$(q,a) \in Q \times \Sigma,\ q = 0 \text{ or } \delta_1(q,a) = 0$$

δ_2 retrieves the set of all outgoing transitions of a source state allowing thus its traversal :

$$\delta_2 : Q \to P(\Sigma \times Q)$$
$$\forall q \in Q,\ \delta_2(q) = \{(a,p) \in \Sigma \times Q \text{ such that } (q,a,p) \in \Delta\}$$

2.3. Concepts and Models

To make sure software components interact properly and consistently it is necessary for them to satisfy requirements that one can rely on.
We call a *concept* a set of requirements on a type covering three aspects:

1. The interface (the methods signatures).
2. The methods semantics (the behavior).
3. The methods complexities (the computing time).

We call *model* of a concept a type of object conforming to this concept. Concepts are usually abstract classes and models are concrete types implementing the concept.
We will refer to the *algorithmic needs* or *constraints* as the requirements that a particular processing imposes to the manipulated objects. Thus, an algorithm may apply to any object that is a model of the required concept. For example a binary search function requires that the object designating the elements of the sequence be able to access the middle element in constant time. More generally, an object representing a position in a sequence that can read any element value instantly is called a *random access iterator*. A C pointer providing an operator `[ ]` that returns a reference on the i^{nth} element in constant time is a model of random access iterator and can then be used by the algorithm to process an array of objects.

2.4. Object Properties

1. An object has a *singular value* when it only guarantees assignment operation and results of most expressions are undefined. For example, an uninitialized pointer has a singular value.
2. We say that an object `x` is *assignable* iff it defines an operator `=` allowing assignment from an object `y` of the same type. The postcondition of the assignment "`x = y`" is "`x` is a copy of `y`".

3. An object is *default constructible* iff no values are needed to initialize it. In C++, it defines a default constructor.

4. An object is *equality-comparable* iff it provides a way to compare itself with other objects of the same type. In C++, such an object defines an operator `==` returning a boolean value.

5. An object is *less-than-comparable* iff there exist a partial order relation on the objects of its type. In C++, such an object provides an operator `<` returning a boolean value.

2.5. Adapters

An adapter is an object encapsulating another object giving it a different interface and/or behavior. It is responsible for implementing any extra functionality required. For example, a stack may be a linked list adapter and provides the pop and push functionalities.

2.6. Iterators

Quoting from SGI Standard Template Library reference documentation [9]:

> Iterators are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.

Iterators are:

1. default constructible and have by default a singular value.

2. assignable (infix operator `=`).

3. singular if none of the following properties is true. An iterator with a singular value only guarantees assignment operation.

4. incrementable if applying `++` operator leads to a well-defined position.

5. dereferenceable if pointed object can be safely retrieved (prefix operator `*`).

6. equality-comparable (infix operator `==`).

Iterators constitute the link between the algorithm and the underlying data structure: they provide the sufficient level of encapsulation to make processing undependant from the data.

2.7. Range

A valid *range* `[x,y)` where `x` and `y` are iterators represents a set of positions from `x` to `y` with the following properties :

1. `[x,y)` refers to all positions between `x` and `y` but not including `y` which is called the *end-of-range* iterator (also denoted as a *past-the-end* iterator).
2. All iterators in the range except `y` are incrementable and dereferenceable.
3. All iterators including `y` are equality comparable.
4. A finite sequence of incrementations of `x` leads to position `y`.

2.8. Iterator categories

There are different concepts of iterators depending on their inherent abilities: input iterators only provide for operations needed in a one-pass, read-only algorithm on sequences whereas output iterators are used in one-pass, write-only algorithm. Forward iterators provide both, plus the ability to iterate on the sequence more than once, but only in a one-way manner. Bidirectional iterators allow incrementations and decrementations and random access iterators allow constant time access to any element. The categories are inclusive and induced by the data structure the iterator operates on. Another characteristic of the forward, bidirectional and random access iterators is that they can be mutable or constant depending on whether the objects accessed through them can be modified or only read.
In the following example, the standard `copy` algorithm only imposes that the ranges it applies on be defined by the weakest iterator concepts : input and output iterator.

Example : Part of the STL copy algorithm

```
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result)
{
  for ( ; first != last; ++result, ++first)
    *result = *first;
  return result;
}
```

3. The ASTL Containers

ASTL provides several data structures implementing deterministic and non deterministic finite automata. Each class sports the same standardized interface allowing algorithms to work with any of them interchangeably. Thus, the decision to use a particular representation is taken at use time and not at design time which guarantees a better adaptability.
All automaton classes are *templates* which means they are parameterized by one or more types that the user must specify in the objects declarations (this is called *instanciation*) according to his needs. ASTL class templates take two types as instanciation parameters: the `Alphabet` type and the `Tag` type. Symmetrically, the automaton classes export two types, `State` and `Edges`, which means that they make them available to the external user by defining them in their public interface. The

type `State` is a *handler* to a internal representation of an element of Q, most of the time a simple unsigned integral type and `Edges` is a *proxy* granting access to a private object storing the outgoing transitions of a state q, that is $\delta_2(q)$, a set of couples in $\Sigma \times Q$ (for a complete description of what a proxy object is, refer to [3]). This proxy has a standard pair associative container interface that maps letters to transitions targets. It should be seen as a STL `map` object with letters as keys and states as values, allowing at worst logarithmic access to a particular transition (see [7]) and providing functionalities to iterate over $\delta_2(q)$.
Following is a list of the automaton container classes implemented so far with a brief description of their internal representations and their advantages and drawbacks:

- The matrix representation consists in storing a matrix $\Sigma \times Q$ of transition targets. It boasts the fastest transition retrieval function but memory space is proportional to the alphabet size.

- The `map` adjacency list representation associates to each state a STL `map` associative container[b] enabling logarithmic access to outgoing transitions but three pointers per transition are needed to maintain the structure.

- A representation by hash table consumes less memory space than a `map` and has a good access time. It should be used when the design of a hash function is easier than an order relation.

- The compact representation takes advantage of holes in the matrix transition lines to reduce them to a single array of transitions. This is the most economical representation in terms of memory space but it is a read-only structure (see [8] for a detailed description).

- The array adjacency list representation stores a vector of pairs in $\Sigma \times Q$ for each state.

  - When the array is maintained sorted, the lookup is logarithmic thanks to a binary search, as for the `map` representation but the insertion is linear. On the other hand, there is no need for the three extra pointers of the `map` structure.

  - For the last two representations, the transition lookup is linear but the use of two heuristics to speed up the process provides substantial enhancement. On the one hand, once the transition is found, it is swapped with its immediately preceding neighbor, implementing an incremental bubble sort at each access ("transpose" method).

  - On the other hand, once found, the transition is moved to the front of the array, speeding up the next access to this transition ("move-to-front" method).

[b] A `map` uses an order relation defined on its keys to maintain a balanced tree of values

The interface of the automaton class in ASTL is willingly limited to some very basic operations to minimize the amount of work when creating a new class. Here is the deterministic automaton interface:

```
State new_state();
void  set_trans(State source, Alphabet a, State target);
void  del_trans(State source, Alphabet a);
void  change_trans(State source, Alphabet a, State new_target);
void  del_state(State s);
void  copy_state(State from, State into);
State duplicate_state(State s);
State delta1(State s, Alphabet a) const;
Edges delta2(State s) const;
Tag&  tag(State s);
void  initial(State s);
State initial() const;
bool& final(State s);
bool  final(State s) const;
```

The methods `delta1` and `delta2` implements the access functions defined in section 2.2. The other methods semantics are rather self explanatory.

The figure 2 presents an abstract of all automata operations complexities for all memory representations including state creation and deletion, transition access, creation and deletion.
Notations: if X is a set, we use X as a shortcut for $|X|$ and $\vec{c}$ for $|\vec{c}(q)|$, $q \in Q$.

Figure 2: Time complexities of ASTL automata containers operations

	Matrix	Map	Compact	Hash Table[c]	Bin. Search	Arrays[d]
Ins/del state	1	1	No	1	1	1
Ins/del trans	1	$\log(\vec{c})$	No	≈ 1	$\vec{c}$	$\vec{c}$
Access trans	1	$\log(\vec{c})$	1	≈ 1	$\log(\vec{c})$	$\vec{c}$

We now need to figure out a proper automaton accessor concept equivalent to the iterator on sequences.

4. The Concept of Cursor

Likewise the iterator, the cursor represents a position in a container and enables the algorithm to access data in an implementation-undependant manner. The general cursor concept encompasses four major inclusive sub-concepts with their own

[c] The access time depends on the quality of the hash function. A proper hash function should minimize collisions and distribute values evenly.
[d] Transition access is optimized with the heuristics Move-To-Front and Transpose.

level of capabilities much like the iterators hierarchy defined in section 2.8. Depending on the algorithmic needs, a particular type of cursor conforming to one of these concepts is required:

- The weakest of them, the plain *cursor* represents a state $q \in Q$ in an automaton $A(\Sigma, Q, i, F, \Delta)$ and is able to move along transitions.
- The *forward cursor* that is a model of cursor too, points to transitions $(q, \sigma, p) \in \Delta$ and is able to step through the sequence of outgoing transitions of q in a forward iteration over $\delta_2(q)$.
- The *trajectory cursor* represents a path in the automaton, that is a sequence of transitions. This sequence can be seen as a "position" in a traversal algorithm such as the depth-first iteration: one can map each stage of the algorithm to a stack state and therefore apply the iteration over ranges defined by pairs of stacks.
- The *traversal cursor* manages everything that is related to the policy of iteration over the automaton. It often adapts a trajectory cursor storing the current position at each iteration step.

In the next section, we give a description of what cursors and forward cursors concepts are. We then present a particular concept of trajectory cursor, the stack cursor and an adapter making it a traversal cursor, the depth-first cursor.

4.1. The Cursor

The cursor is basically a pointer to an automaton state used by the algorithm to keep track of the current position. It should be used when trivial traversal is needed, that is, a path along transitions defined by a word. A cursor is assignable, default constructible, it has then a singular value, it is equality-comparable, incrementable and dereferenceable iff it does not point to the sink state.
The following table presents the valid expressions on cursors where `x` and `y` are object of a type that is a model of cursor, q is the pointed state and `a` is a letter.

Name	**Expression**	**Semantics**
source state	`x.src();`	return q
final source	`x.src_final();`	return `true` if $q \in F$
comparison	`(x == y)`	`true` if `x` points to the same state as `y`
forward with letter	`x.forward(a);`	move along transition labeled with `a`
defined transition	`x.exists(a);`	`true` if an outgoing transition of q labelled with `a` is defined
set position	`x = q;`	make `x` point to the state q
sink	`x.sink();`	return `true` if $q = 0$

Making a forward move along an undefined transition leads the cursor to the sink state and `forward` returns `false`. The method `exists` tests for the existence of an outgoing transition labelled with `a` and different from a sink transition, i.e. `exists`

returns `false` if $\delta_1(q, a) = 0$.
In the following example, the algorithm `is_in` tests if a word defined by a range `[first, last)` is in the recognized language of a DFA accessed through a cursor `c` :

```
bool is_in(Iterator first, Iterator last, ForwardCursor c) {
  while (first != last && c.forward(*first))
    ++first;
  return first == last && c.src_final();
}
```

4.2. The Forward Cursor

The forward cursor has the ability to iterate over the outgoing transitions of the pointed state q. It therefore represents a transition whereas a plain cursor represents a state. It points to a triple $(q, a, p) \in \Delta$, has the same properties as the cursor and implements all the cursor requirements plus:

Name	**Expression**	**Semantics**
target state	`x.aim();`	return p
letter	`x.letter();`	return a
final aim	`x.aim_final();`	return `true` if $p \in F$
comparison	`(x == y)`	`true` if `x` points to the same transition as `y`
forward	`x.forward();`	move along currently pointed transition
first transition	`x.first_transition();`	set `x` on the first transition of $\delta_2(q)$ return `true` if $\delta_2(q) \neq \emptyset$
next transition	`x.next_transition();`	move on to the next transition of set $\delta_2(q)$. return `false` if reached transition (q, a', p') is a sink transition
find	`x.find(a);`	set `x` on the transition labeled with letter `a`. return `true` if $(q, a, \delta_1(q, a))$ is not a sink transition

The following piece of code prints out all outgoing transitions of a source state pointed by a forward cursor `c` :

```
cout << "Source:" << c.src() << endl;
if (c.first_transition()) {
  do
    cout << "Label:" << c.letter() << " Target:" << c.aim() << endl;
  while (c.next_transition());
}
```

4.3. The Stack Cursor

The stack cursor, as the name implies, is simply a stack of forward cursors. It is mostly reused by the traversal cursors storing their path in the automaton. Its interface has only one method more than the forward cursor implementing the `pop` action. The other ones apply to the stack top cursor. A stack cursor is default constructible, it then represents the empty stack, it is equality-comparable, dereferenceable and incrementable iff it does not point to a sink transition. It its not assignable for efficiency reasons. Remark that the comparison operator compares whole stacks and not only stack tops because the underlying concept of a stack cursor is the path. The following table summarizes the specific behavior of a stack cursor:

Name	**Expression**	**Semantics**
forward	`x.forward();`	move along current stack top transition and push reached transition (p, a', p')
forward with letter	`x.forward(a);`	move along transition labeled with `a` push $(q, a, \delta_1(q, a))$ and return `true` if $\delta_1(q, a) \neq 0$
backward	`x.backward();`	pop. return `false` if resulting stack is empty
comparison	`(x == y)`	return `true` if `x` stack is a copy of `y` stack

4.4. The Depth-First Cursor

The depth-first cursor is a sub-concept of the traversal cursor. It must implement all functionalities related to the managing of transitions traversal in depth-first order. It is in some sense an iterator since the user has no ways to influence its route which comes down to iterate over a sequence by successive incrementations of the accessor.
A depth-first cursor is default constructible, it then represents the empty stack, it is equality-comparable, dereferenceable and incrementable iff it is not the empty stack. A depth-first cursor never points to sink transitions.
Here is the complete interface of a depth-first cursor:

Name	**Expression**	**Semantics**
source	`x.src();`	return q
aim	`x.aim();`	return p
letter	`x.letter();`	return a
final source	`x.src_final();`	return `true` if $q \in F$
final aim	`x.aim_final();`	return `true` if $p \in F$
forward	`x.forward();`	move on to the next transition in depth-first order. return `true` if `x` actually moved forward or `false` if `x` poped.
comparison	`(x == y)`	compare whole stacks.

Since the depth-first cursor can be seen as an iterator it can be used to materialize "positions" in an algorithm processing by defining ranges: an algorithm starts with initial conditions and stops when a certain condition becomes true. For depth-first iteration, the start condition is usually a stack holding the first transition of the initial state and the stop condition is reached when the stack is empty. That is why constructing by default a depth-first cursor y initialize it to the empty stack. On the other hand, constructing x with a forward cursor c initialize it to the first outgoing transition of the state pointed to by c. This pair of depth-first cursors defines a range [x, y) over the automaton that is passed to the algorithms:

```
DFA A;
language(cout, dfirstc(forwardc(A)), dfirstc());
```

This function write the language of A to the stream cout (the standard output). dfirstc and forwardc are *helper functions* building respectively a depth-first cursor from a forward cursor and a forward cursor from a DFA. They are meant to simplify the task of declaring the objects types since these function templates deduce them from their arguments types. Remark that the third argument of language appears for clarity purpose but is in fact not required as the usual use of depth-first iteration is to iterate over the entire automaton. Therefore, the language algorithm deduces from a missing end-of-range cursor that it should use the empty stack as stop condition. Thus, the code reduces to:

```
language(cout, dfirstc(forwardc(A)));
```

To give an insight of what an algorithm using traversal cursors looks like we present in the following piece of code the implementation of the language algorithm:

```
void language(ostream &out, DepthFirstCursor first,
              DepthFirstCursor last = DepthFirstCursor()) {
  vector<char> word;
  while (first != last) {               // until end of range
    word.push_back(first.letter());     // push current letter
    if (first.aim_final()) out << word; // if target is final
                                        //    output the word
    while (! first.forward())           // while moving backward
      word.pop_back();                  // pop the last word letter
  }
}
```

5. The Clone and Ccopy Algorithms

The algorithms ccopy (cursor copy) and clone are fundamental algorithms allowing to construct a DFA as the result of the application of algorithm. ccopy makes a copy of an automaton defined by a range of depth-first cursors in the ascending stage of the depth-first iteration to trim unneeded paths leading to non-final states. clone does the same thing but in the descending stage making an exact copy of the automaton:

```
DFA A, result;
ccopy(result, dfirstc(forwardc(A)));
```

What makes these algorithms particularly interesting is the possibility to call them with cursor adapters as argument as exposed in the following section. Another interesting application is to pass as first argument a stream encapsulated in a DFA interface to save the automaton to a file for example:

```
DFA A;
DFA_stream out(my_output_file);
ccopy(out, dfirstc(forwardc(A)));
```

6. The Cursor Adapters

The real power of the cursor concept lies in the possibility to modify and extend the default behavior through the design of adapters. The use of binary cursor adapters to implement the usual set operations on automata provides a good example of their ease of use. Let us see how to build the intersection cursor from two other cursors.
Let $A(\Sigma, Q, i, F, \Delta)$ and $A'(\Sigma, Q', i', F', \Delta')$ be two deterministic automata. We define the intersection automaton B of A and A' as:

$$B = (\Sigma, Q \times Q', (i, i'), F \times F', \Delta'')$$

with Δ'' defined by the following transition function :

$$\delta_1''((q, q'), \sigma) = (\delta_1(q, \sigma), \delta_1'(q', \sigma))$$

Let us call `i` the intersection cursor of two cursors `x` and `y`. `i` should implement the following behavior:

- `i.src()` returns a pair of states:

  ```
  State src() const {
    return make_pair(x.src(), y.src());
  }
  ```

- `i` points to a final state iff `x` and `y` point to final states:

  ```
  bool src_final() const {
    return x.src_final() && y.src_final();
  }
  ```

- `forward(a)` implements the δ_1'' transition function. `i` can move forward iff `x` and `y` can:

  ```
  bool forward(unsigned int a) {
    return x.forward(a) && y.forward(a);
  }
  ```

- A transition is defined in the intersection automaton iff it is defined in the other two:

  ```
  bool exists(unsigned int a) const {
    return x.exists(a) && y.exists(a);
  }
  ```

- A state in the intersection automaton is a sink state if one of the underlying state is a sink state:

  ```
  bool sink() const {
    return x.sink() || y.sink();
  }
  ```

Since `i` conforms to the cursor concepts, it is straightforward to test if a word is in the intersection of two DFA languages:

```
DFA A1, A2;
const char word[] = "word to check";
if (is_in(word, word + 13, intersectionc(cursor(A1), cursor(A2))))
  cout << "found";
else
  cout << "not found";
```

The function `intersectionc` is simply a helper function building an object of type `intersection_cursor` from the two cursors passed as arguments.
For simplification purpose, we will not present the code for the intersection forward cursor but the same apply to it when it comes to output the recognized language for instance:

```
DFA A1, A2;
language(cout, dfirstc(intersectionc(forwardc(A1), forwardc(A2))));
```

Here, the depth-first cursor is builded from a stack of intersection forward cursors. This way, the intersection is computed on-the-fly during the algorithm execution. To get the real resulting automaton of the intersection, simply call the `ccopy` algorithm:

```
DFA A1, A2, result;
ccopy(result, dfirstc(intersectionc(forwardc(A1), forwardc(A2))));
```

By writing only the on-the-fly version of the intersection algorithm we get the two implementations thanks to the copy function. By using the lazy cursor adapter we get the third one:

```
DFA A1, A2, result;
if (is_in(word, word + 13,
          lazyc(intersectionc(cursor(A1), cursor(A2)), result))
  cout << "found";
```

```
else
  cout << "not found";
```

The helper function `lazyc` builds a lazy forward cursor from the intersection cursor and the DFA `result`. If needed, this cursor will construct the resulting automaton during the traversal along the path defined by `word`. If this automaton part was to be created, the cursor will add the necessary states and transitions otherwise it will follow the already constructed path as if it was a simple iteration over a DFA. At each call to an algorithm on the intersection automaton, a piece of the overall resulting DFA will be added, speeding up the access to often used states and transitions. This is particularly useful when building the entire automaton requires a lot of time and memory space since only the interesting parts will be created.
Many algorithms can be implemented by adapters and on-the-fly processing, for example a cursor can simulate a virtual automaton of all permutations of a word in a very simple manner avoiding combinatorial explosion or give a string of characters a cursor interface making it look like a flat automaton recognizing one word. The applications are too numerous to make a comprehensive list. ASTL provides about 20 adapters that one can combine with the use of the `ccopy` function or the lazy cursor to get three implementations for the price of one.

7. The Cyclic Automata

In the case of cyclic automata, we must make sure that the traversal algorithms stop. To do so, we must devise generic and efficient mechanisms to keep track of the already visited states but this a non- trivial issue. First of all, who should be responsible for maintaining the structure ? We cannot make the algorithm responsible for it because that would result in the duplication of the same code, one for cyclic and one for acyclic automata and this contradicts generic programming purpose. On the other hand, the automaton containers can hardly detect the fact that they can be firstly acyclic, then become cyclic because of a particular operation and so on. The only solution lies in the third party of the model, the cursors. The user has to choose at use-time what traversal cursor to use depending on the automaton structure and operations applied to it. ASTL provides four implementations of depth-first cursor:

1. The plain depth-first cursor to be used on tree-like automata.
2. The tag depth-first cursor that uses each state tag to store a boolean value indicating if a state has already been visited.
3. The hash depth-first cursor uses an STL `hash_set` to store the set of visited states.
4. The set depth-first cursor uses an STL `set` to store the visited states. The access time is logarithmic. It relies on an order relation defined on states.

The `set` should be used when defining an order relation on states is easy. The `hash_set` should be used when the design of the hash function is more appropriated.

The tag version is the most efficient but should be used carefully since the flags consistency can be broken if a traversal algorithm limits itself to a sub-part of the automaton.

8. Conclusion

We concluded from these programming experiments that the three-layer model of STL should be regarded as fundamental and extended to as many as possible designs and development environments. The central layer uncoupling the algorithms from the data structure allows great reusability and true genericity through data hiding. Moreover, the static typing of C++ templates prevents efficiency lost thanks to function inlining and maximal optimizations. It resulted from our speed tests that these techniques induce no lack of efficiency compared with classical non-generic C implementations.
Future developments should be concerned by the modelization of mutative algorithms, that is algorithms directly modifying the data structure they operate on which is a long time non-trivial issue especially when it comes to handle accessors such as iterators or cursors that can be invalidated by the destruction or the reallocation of the underlying objects. There has been no satisfying solutions so far: STL and ASTL algorithms never suppress elements in a sequence or in an automaton. They read or change values, sometimes create a new sequence or automaton as computing result but solid concepts about true mutative operations remain to be invented.

References

1. *Information Technology - Programming Languages - C++*, international standard, ISO document number ISO/IEC 14882-1998, 1998.
2. Vincent Le Maout. *Tools to Implement Automata, a first step: ASTL*. Lecture Note in Computer Science 1436, Springer-Verlag, 1998, pp 104-108.
3. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns*, Addison-Wesley, 1995, pp 207–217.
4. L.Q. Lee, J.G. Siek and A. Lumsdaine. *The Generic Graph Component Library (GGCL) User Manual*, documentation, University of Notre Dame, `http://www.lsc.nd.edu/research/ggcl`, 2000.
5. M. Forster, A. Pick and M. Raitner. *The Graph Template Libray 0.3.1 (GTL) Manual*, documentation, University of Passau, `http://www.fmi.uni-passau.de/Graphlet/GTL/`, 1999.
6. D. Musser and A. Stepanov. *Generic Programming*, Lecture Notes in Computer Science 358. Spring-Verlag, 1989.
7. M. Nelson. *C++ Programmer's Guide to the Standard Template Library*, IDG Books Worldwide, 1995.
8. D. Revuz. *Dictionnaires et Lexiques, Méthodes et Algorithmes*, PhD dissertation, Université Paris VII, 1991, pp 38–39.
9. *Standard Template Library Programmer's Guide*, Silicon Graphics Computer Systems. `http://www.sgi.com/Technology/STL/`, 1999