The Automaton Standard Template Library
ASTL version 2.0

Reference Documentation

Vincent Le Maout

May 15, 2008

Contents

1 Introduction

The Automaton Standard Template Library (ASTL) is a set of generic C++ components for efficient automata manipulation. As any library geared toward supporting the generic programming paradigm, it is made of two distinct parts : a collection of concepts specific to the automata domain which is described by this documentation and a set of software components (containers, accessors and algorithms) implementing the concepts.

1.1 License

1.2 Acknowledgments

I would like to thank all the people who have worked on and with the library especially Dominique Revuz for his support and advices on the code design, Xavier Daragon for writing the very first version of this documentation and Arnaud Adan for his review and extension to the weighted automata and transducers.

1.3 Availability

The package and this documentation are downloadable from `http://astl.sourceforge.net/` which is the official site for news, announcements and releases.

1.4 Supported Compilers

ASTL 2.0 has been successfully used with GNU g++ 2.96 and later.

1.5 Compiling

ASTL is made of header files, it does not require to compile any code prior to using it and therefore requires no specific file to link to.
There is a number of `#defines` modifying the compiler behavior :

- `ASTL_USES_NAMESPACE`

The only information needed is the `include` subdirectory location, for instance:

```
g++ -I/home/vince/astl/include main.cc
```

1.6 Files Hierarchy

ASTL 2.0 is made of five subdirectories :

1. `bin` : command line executables
2. `doc` : LaTeX documentation source, postcript and PDF documentation

3. `include` : headers
4. `src` : source code for the executables
5. `templates` : code templates
6. `ext` : some rather experimental extensions to the library

2 Definitions and Notations

2.1 Finite Automaton

To make our concepts sufficiently generic to satisfy a broad range of algorithmic constraints, we add to the classical automaton definition a set of *tags*, that is, any data associated to a state and needed to apply an algorithm. We will however omit tag-related considerations whenever they are not relevant.

Let $A(\Sigma, Q, I, F, \Delta, T, \tau)$ be a 7-tuple of finite sets defined as follows :

Σ	An alphabet
Q	A set of states
$I \subseteq Q$	A set of initial states
$F \subseteq Q$	A set of final states (also called terminal or accepting states)
$\Delta \subseteq (Q \times \Sigma \cup \{\epsilon\} \times Q)$	A set of transitions
T	A set of tags
$\tau \subset (Q \times T)$	A set mapping a state to its associated tag

We distinguish one special state noted 0 and called the *null* or *sink state*. The *label* of a transition $(q, \sigma, p) \in \Delta$ is the letter σ, q is the *source* state and p is the *destination* state or *aim*. When $\sigma = \epsilon$ (the empty word) the transition is said to be an ϵ-*transition*.

We call *incoming* transitions (respectively *outgoing* transitions) of a state s, the set of transitions $(q, \sigma, p) \in \Delta$ such that $p = s$ (respectively $q = s$). By default, the transitions of a state are its outgoing transitions.

We will write $P(X)$ for the powerset of a set X and $|X|$ for its number of elements.

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

$$\delta_1 \ : \ Q \times \Sigma \cup \{\epsilon\} \to P(Q)$$
$$\delta_2 \ : \ Q \to P(\Sigma \times P(Q))$$

δ_1 retrieves the set of transitions targets given the source state and a letter. δ_2 allows to access the set of all the outgoing transitions of a given state.

δ_1 can be naturally extended to words :

$$\begin{array}{rcl} \delta_1^* : Q \times \Sigma^* & \to & P(Q) \\ (q, \epsilon) & \mapsto & q \\ (q, w \cdot a) & \mapsto & \delta_1(\delta_1^*(q, w), a) \end{array}$$

The *right context* of a state q is the set of letters labelling the outgoing transitions of q : $\vec{c}(q) = \{\sigma \in \Sigma \mid \exists p \in Q, \ (q, \sigma, p) \in \Delta\}$.
A *path* is a sequence $c = t_1 \, t_2 \, ... \, t_n$ of transitions $t_i = (q_i, \sigma_i, p_i)$ such that $\forall i, t_i \in \Delta$ and for $i < n$, $q_{i+1} = p_i$. The path length n is noted $|c|$ and its label is the concatenation of the transitions letters : $w = \sigma_1 \, \sigma_2 \, ... \, \sigma_n$.

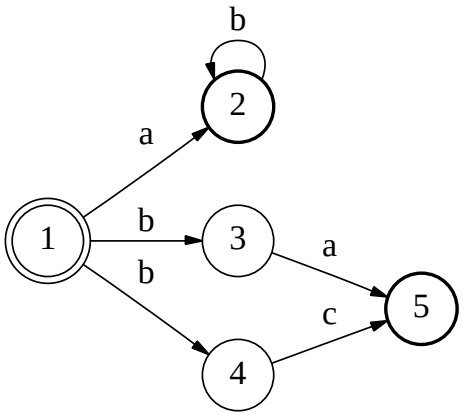

Figure 1: Example of NFA

The language recognized by an automaton A is defined by :

$$L(A) = \{w \in \Sigma^* \mid \delta_1^*(I, w) \cap F \neq \emptyset\}$$

that is, the labels of the paths leading from an initial state to a final state.

An automaton is said to be *deterministic* iff I is a singleton and there is at most one transition per state which is labeled by a given alphabet letter, that is, $|\delta_1(q)| \leq 1$, $\forall q \in Q$. In this case, δ_1 is defined as :

$$\delta_1(q, \sigma) = \begin{cases} p & \text{if } (q, \sigma, p) \in \Delta \\ 0 & \text{otherwise} \end{cases}$$

The sink state acts as failure value for the transition function.

Example (figure 1) A is a non-deterministic automaton with $\Sigma = \{a, b, c\}$, $Q = \{1, 2, 3, 4, 5\}$, $I = \{1\}$, $F = \{2, 5\}$ and $\Delta = \{(1, a, 2), (2, b, 2), (1, b, 3), (3, a, 5), (1, b, 4), (4, c, 5)\}$.
$L(A) = \{ab^*, ba, bc\}$.

2.2 Containers & Cursors

3 Getting Started

This section contains code example ranging from basic automaton operations to cursors manipulation. They cover all aspect of the library functionnalities and aim at providing an insight of what can be done and how[1].

3.1 Declaring a Container

The automaton containers are class templates parameterized by two types: the alphabet traits and the tag type. By default, the alphabet trait is `plain` (8 bits `char`) and the tag type is `empty_tag` (no tags needed). Also a predefined alphabet traits called `range` is provided. It must be instanciated with a builtin integral type `T` followed by two constants `x` and `y` of type `T` defining a subset `[x, y]` of the domain `T`. For example, `plain` is defined as follow :

```
typedef range<char, (char) -128, (char) 127> plain;
```

[1]The code examples make no use of the ASTL name space so `ASTL_USES_NAMESPACE` should not be defined to compile them as is.

3.1.1 DFA

ASTL provides eight DFA containers sharing the same interface. They have homogeneous interfaces and behaviors but heterogeneous implementations and complexities allowing to choose at utilization time which one fits best the situation. The following piece of code declares a variety of containers :

```
#include <astl.h>
#include <dfa.h>
#include <string>

int main()
{
  DFA_map<> A1;
  DFA_matrix<range<char, 'A', 'Z'> > A2;
  DFA_tr<range<int, 0, 1023> > A3;
  DFA_mtf<french, std::string> A4;
  DFA_hash<std::char_traits<char> > A5;
  DFA_bin<ASCII, int> A6;
}
```

The declaration of `A1` uses the template default parameters, it is equivalent to :

```
DFA_map<plain, empty_tag> A1;
```

`A5` uses the standard character traits `std::char_traits` as alphabet traits. It can be used to instanciate all containers except `DFA_matrix` and must only be used with builtin types alphabets.

3.1.2 Compact DFA

A compact automaton is a container adapter parameterized by the adapted automaton type. It is constructed by copy :

```
#include <astl.h>
#include <dfa_mtf.h>
#include <dfa_compact.h>

int main()
{
  DFA_mtf<> A;
  DFA_compact<DFA_mtf<> > C(A);
}
```

The constructor of `C` builds a copy of `A` in a compact representation.

3.1.3 NFA

```
#include <astl.h>
#include <nfa_mmap.h>

int main()
{
  NFA_mmap<plain, int> A;
}
```

3.2 Constructing an Automaton

```
#include <astl.h>
#include <dfa.h>
#include <vector>
#include <iterator>

int main()
{
  using namespace std;
  DFA_matrix<> A;
  DFA_matrix<>::state_type q = A.new_state();
  DFA_matrix<>::state_type p = A.new_state();
  A.set_trans(q, 'a', p);
  A.initial(q);
  A.final(p) = true;
  DFA_matrix<>::state_type Q1[10];
  A.new_state(10, Q1);
  vector<DFA_matrix<>::state_type> Q2(10);
  A.new_state(10, Q2.begin());
  vector<DFA_matrix<>::state_type> Q3;
  A.new_state(10, back_inserter(Q3));
}
```

3.3 Accessing States Transitions

```
#include <astl.h>
#include <dfa.h>
#include <iostream>

int main()
{
  using namespace std;
  typedef DFA_matrix<> DFA;
  DFA A;
  // Construction...
  DFA::state_type q = A.initial();
  DFA::state_type p = A.delta1(q, 'a');
  if (p == A.null_state) cout << "undefined" << endl;
  DFA::edges_type e = A.delta2(q);
  if (e.find('b') == e.end()) cout << "undefined" << endl;
  DFA::edges_type::const_iterator i;
  for(i = e.begin(); i != e.end(); ++i)
    cout << " source " << q
         << " letter " << i->first
         << " aim "    << i->second;
}
```

3.4 Minimizing Acyclic DFAs

```
#include <astl.h>
#include <dfa.h>
#include <minimize.h>
```

```
int main()
{
  DFA_matrix<plain, minimization_tag> A;
  // Construction...
  acyclic_minimization(A);
}
```

3.5 Matching

```
#include <astl.h>
#include <dfa.h>
#include <cursor.h>
#include <language.h>
#include <string>
#include <iostream>

int main()
{
  using namespace std;
  DFA_matrix<> A;
  string w = "word";
  // Construction...
  cursor<DFA_matrix<> > c(A);

  string::const_iterator i = w.begin();
  for(c = A.initial(); i != w.end() && c.forward(*i); ++i);
  if (i == w.end() && c.src_final()) cout << "recognized";

  if (is_in(w.begin(), w.end(), plainc(A))) cout << "recognized too";

  if (is_in(istream_iterator<char>(cin), istream_iterator<char>(),
            plainc(A))) cout << "found on stdin";
}
```

3.6 Using a Forward Cursor

```
#include <astl.h>
#include <dfa.h>
#include <language.h>
#include <cursor.h>
#include <iostream>

int main()
{
  DFA_matrix<> A;
  // Construction...
  forward_cursor<DFA_matrix<> > c(A, A.initial());
  if (c.first())
    do
      std::cout << " source " << c.src()
                << " letter " << c.letter()
                << " aim "    << c.aim();
    while (c.next());
```

```
  const char *w = "word";
  if (is_in(w, w + 4, forwardc(A))) std::cout << "recognized";

}
```

3.7 Using a Depth-First Cursor

3.8 Language Extraction

```
#include <astl.h>
#include <dfa.h>
#include <language.h>
#include <cursor.h>
#include <iostream>

int main()
{
  DFA_matrix<> A;
  // Construction...
  language(std::cout, dfirstc(A));
}
```

3.9 Copying

```
#include <astl.h>
#include <dfa.h>
#include <language.h>
#include <cursor.h>
#include <ccopy.h>
#include <iostream>

int main()
{
  DFA_matrix<> A, B;
  // Construction...
  DFA_matrix<>::state_type i = ccopy(A, dfirstc(B));
  A.initial(i);

  DFA_matrix<> C, D;
  // Construction...
  C.initial(clone(C, dfirstc(D)));
}
```

3.10 Streams Input/Output

```
#include <astl.h>
#include <dfa.h>
#include <ccopy.h>
#include <stream.h>

int main()
{
  DFA_matrix<> A;
```

```
  // Construction...
  dump(cout, dfirstc(A)); // don't save tags
  full_dump(cout, dfirstc(A)); // save tags

  DFA_matrix<> B;
  restore(B, cin);
}
```

3.11 On-the-fly Processing

```
#include <astl.h>
#include <dfa.h>
#include <language.h>
#include <set_operation.h>
#include <cursor.h>
#include <string>
#include <iostream>

int main()
{
  DFA_matrix<> A, B;
  // Constructions
  std::string w = "word";
  if (is_in(w.begin(), w.end(),
            intersectionc(forwardc(A), forwardc(B))))
     std::cout << "recognized";
}
```

3.12 By-copy Processing

```
#include <astl.h>
#include <dfa.h>
#include <language.h>
#include <set_operation.h>
#include <cursor.h>
#include <neighbor.h>

int main()
{
  DFA_matrix<> A, B;
  // Constructions
  ccopy(A, dfirstc(neighborc(forwardc(B), "word", 2)));

  DFA_matrix<> C;
  clone(C, dfirstc(notc(diffc(forwardc(A), forwardc(B)))));
}
```

3.13 Lazy-construction Processing

```
#include <astl.h>
#include <dfa.h>
#include <lazy.h>
#include <regexp.h>
#include <language.h>
```

```
#include <iostream>

int main()
{
  regexp_cursor e("a|b*");
  const char *w = "aaabb";
  if (is_in(w, w + 5, lazyc(e))) std::cout "recognized";

  lazy_cursor<regexp_cursor> c(e);
  if (is_in(w, w + 5, c)) std::cout "recognized too";
}
```

3.14 Virtual-traversal Processing

```
#include <astl.h>
#include <dfa.h>
#include <stream.h>
#include <iostream>

// save the automaton:

int main()
{
  DFA_matrix<> A;
  // Construction...
  dump(cout, dfirstc(A));
}

// cut ....................................
```

```
#include <astl.h>
#include <dfa.h>
#include <ccopy.h>
#include <language.h>
#include <iostream>

// extract language:

int main()
{
  clone_cursor<plain> c(std::cin);
  language(std::cout, c);
}
```

3.15 Determinizing

```
#include <astl.h>
#include <dfa.h>
#include <nfa.h>
#include <determinize.h>
#include <ccopy.h>

int main()
{
```

```
  NFA_mmap<> N;
  // Construction...
  DFA_matrix<> A;
  A.initial(clone(A, dfirstc(forwarddc(N))));
}
```

3.16 Displaying and Printing

```
#include <astl.h>
#include <dfa.h>
#include <dot.h>
#include <ccopy.h>
#include <iostream>

int main()
{
  DFA_map<> A, B;
  // Construction...
  dot(std::cout, dfirstc(A)); // don't write tags
  full_dot(std::cout, dfirstc(A)); // write tags
}
```

4 Coding Standards

4.1 Namespace

ASTL components may optionally be enclosed in a namespace `astl` by defining the symbol `ASTL_USES_NAMESPACE`.

4.2 Exceptions & RTTI

So far, ASTL makes no use of the C++ exceptions mechanism nor RTTI (RunTime Type Information). They can be safely turned off on the compiler command line (`g++ -fno-exceptions -fno-rtti`).

4.3 Types and Functions Naming

ASTL follows the C++ standard way for types naming: types and functions names contain only lower-case letters and compound words contain underscores separating components. Most of the time, words are used literaly without any abbreviation. Examples: `forward_cursor`, `acyclic_minimization`.
For formal template parameters, upper-case letters are used and no underscore appears, thus minimizing the probability for names collision between real types and formal parameters. For instance, the definition:

```
template <typename ForwardCursor>
class A
{ };
```

should avoid any confusion between the symbol `ForwardCursor` and the existing type `forward_cursor`.

4.4 Helper Functions

Whenever it is possible and useful, a helper function is provided to make component building and initializing easier.

4.5 Testing & Debugging

`src/check_dfa.cc`
`src/check_nfa.cc`
`src/check_cursor.cc`
`check.h` (coverage test)
`debug.h` debug cursor and trace cursor

5 Concepts

5.1 Alphabet

5.2 Edges

5.3 Container

5.4 Cursor

6 Models

6.1 Alphabets

6.2 Containers

6.3 Cursors

cursor<DFA>

Category : cursors **Component Type :** Type

Description

A cursor is a pointer to an automaton state that is able to move along defined transitions. Its purpose is to implement simple traversals testing if a word is in the language recognized by an automaton.

Example

```
DFA_matrix<> A;
const char *w = "word";
add_word(A, w, w + 4);
cursor<DFA_matrix<> > c(A);
for(c = A.initial(); *w && c.forward(*w); ++w);
assert (*w == 0 && c.src_final());
```

Definition

`cursor.h`

Template parameters

Parameter	Description	Default
`DFA`	the automaton type	

Model of

plain cursor.

Type requirements

- `DFA` is a model of DFA

Public base classes

`plain_cursor_concept`

Member	Where defined	Description
`state_type`	plain cursor	The type of the states handles of the underlying DFA, that is, `DFA::state_type`
`char_type`	plain cursor	The type of the transitions letters, `DFA::char_type`
`tag_type`	plain cursor	The type of the tags associated to states, `DFA::tag_type`
`char_traits`	plain cursor	Character traits associated to `char_type`
`cursor()`	plain cursor	Default constructor
`cursor(const cursor&)`	plain cursor	Copy constructor

Member	Where defined	Description
`state_type src() const`	plain cursor	Return the state handle the cursor is pointing to
`cursor& operator=(state_type q)`	plain cursor	Set the cursor to point to state `q`
`cursor& operator=(const cursor&)`	plain cursor	Assignment operator
`bool operator==(const cursor&) const`	plain cursor	Return `true` iff both cursors point to the same state
`bool sink() const`	plain cursor	Return `true` iff the cursor points to the sink state `DFA::null_state`
`bool forward(const char_type &a)`	plain cursor	Move along transition labeled with `a` if defined, otherwise move to sink state and return `false`
`bool exists(const char_type &a) const`	plain cursor	Return `true` if a transition labeled with `a` is defined
`bool src_final() const`	plain cursor	Return `true` if pointed state is final
`tag_type src_tag() const`	plain cursor	Return the object associated to pointed state

Members

New members

These members are not defined in the plain cursor requirements but are specific to `cursor`.

Member	Description
`cursor(const DFA &A)`	Construct a cursor pointing to a DFA `A`
`cursor(const DFA &A, state_type q)`	Construct a cursor pointing to the state `q` of the DFA `A`

Helper functions

```
template <typename DFA>
cursor<DFA> plainc(const DFA &A, DFA::state_type q = A.initial());
```

Notes

A default-constructed cursor, or a cursor that has not been set to point to a valid state has a singular value which means the only operation allowed is the assignment. The sink state is not considered as a valid state.

See also

`forward_cursor`, DFA.

forward_cursor<DFA>

Category : cursors **Component Type :** Type

Description

A `forward_cursor` is a pointer to an automaton transition, that is, a triple (source state, letter, aim state). It provides all the functionnalities of the plain cursor and some means to iterate through the sequence of the outgoing transitions of the source state.

Example

```
string w[4] = { "forward", "cursor", "code", "example" };
DFA_bin a;
add_word(a, w, w + 4);
forward_cursor c(a, a.initial());
assert(c.first());
do cout << "src " << c.src() << " letter " << c.letter
        << " aim " << c.aim();
while (c.next());
```

Definition

`cursor.h`

Template parameters

Parameter	Description	Default
`DFA`	The automaton type	

Model of

plain cursor, forward cursor.

Type requirements

- `DFA` is a model of DFA.

Public base classes

`forward_cursor_concept`.

Member	Where defined	Description
`state_type`	plain cursor	The type of the states handles of the underlying DFA, that is, `DFA::state_type`
`char_type`	plain cursor	The type of the transitions letters, `DFA::char_type`
`tag_type`	plain cursor	The type of the tags associated to states, `DFA::tag_type`
`char_traits`	plain cursor	Character traits associated to `char_type`
`cursor()`	plain cursor	Default constructor

Member	Where defined	Description
`cursor(const cursor&)`	plain cursor	Copy constructor
`state_type src() const`	plain cursor	Return the state handle the cursor is pointing to
`cursor& operator=(state_type q)`	plain cursor	Set the cursor to point to state `q`
`cursor& operator=(const cursor&)`	plain cursor	Assignment operator
`bool operator==(const cursor&) const`	plain cursor	Return `true` iff both cursors point to the same state
`bool sink() const`	plain cursor	Return `true` iff the cursor points to the sink state `DFA::null_state`
`bool forward(const char_type &a)`	plain cursor	Move along transition labeled with `a` if defined, otherwise move to sink state and return `false`
`bool exists(const char_type &a) const`	plain cursor	Return `true` if a transition labeled with `a` is defined
`bool src_final() const`	plain cursor	Return `true` if pointed state is final
`tag_type src_tag() const`	plain cursor	Return the object associated to pointed state
`char_type letter() const`	forward cursor	Return the letter on the pointed transition. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`bool first()`	forward cursor	Make the cursor point to the first element of the transitions sequence of the source state. Return `true` if there is such an element (if any transition is defined), otherwise the pointed transition is undefined.
`bool next()`	forward cursor	Move the cursor to the next element of the transitions sequence of source state. Return `true` if there is such an element (the cursor is not at the end of the sequence), otherwise the pointed transition is undefined. `first` must have been successfully called prior to using this method.
`bool find(const Alphabet &a)`	forward cursor	Make the cursor point to the transition labelled with `a`. Return `true` if such a transition exists, otherwise the pointed transition is undefined.
`void forward()`	forward cursor	Move forward on the currently pointed transition. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.

Member	Where defined	Description
`bool aim_final() const`	forward cursor	Return `true` if the aim state is final. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`tag_type aim_tag() const`	forward cursor	Return the object associated with the aim state. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`state_type aim() const`	forward cursor	Return the handle of the aim state the cursor is point to. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.

Members

New members

These members are not defined in the forward cursor requirements but are specific to `forward_cursor`.

Member	Description
`forward_cursor(const DFA &A)`	Construct a forward cursor pointing to a DFA `A`
`forward_cursor(const DFA &A, state_type q)`	Construct a forward cursor pointing to the state `q` of the DFA `A`

Helper functions

```
template <typename DFA>
forward_cursor<DFA> forwardc(const DFA &A, DFA::state_type q = A.initial());
```

Notes

A default-constructed forward cursor, or a forward cursor that has not been set to point to a valid state and a valid transition has a singular value which means the only operation allowed is the assignment. The sink state is not considered as a valid state.

See also

DFA, `cursor`, `stack_cursor`, `queue_cursor`.

stack_cursor<ForwardCursor, Container>

Category : cursors **Component Type :** Type

Description

A `stack_cursor` is a forward cursor storing its path in a stack of cursors. Each forward move along a transition pushes a new forward cursor onto the stack top and an extra method `backward` allows to pop. The depth-first traversal cursor `dfirst_cursor` relies on the `stack_cursor`.

Definition

`cursor.h`

Template parameters

Parameter	Description	Default
`ForwardCursor`	The type of the cursors which are stored in the stack	
`Container`	The type of the sequential container implementing the stack	`vector<ForwardCursor>`

Model of

plain cursor, forward cursor, stack cursor.

Type requirements

- `ForwardCursor` is a model of forward cursor.
- `Container` is a model of back insertion sequence.
- `Container::value_type` must be `ForwardCursor`.

Public base classes

`cursor_concept`, `forward_cursor_concept`, `stack_cursor_concept`.

Member	Where defined	Description
`state_type`	plain cursor	The type of the states handles of the underlying DFA, that is, `DFA::state_type`
`char_type`	plain cursor	The type of the transitions letters, `DFA::char_type`
`tag_type`	plain cursor	The type of the tags associated to states, `DFA::tag_type`
`char_traits`	plain cursor	Character traits associated to `char_type`
`cursor()`	plain cursor	Default constructor
`cursor(const cursor&)`	plain cursor	Copy constructor

Member	Where defined	Description
`state_type src() const`	plain cursor	Return the state handle the cursor is pointing to
`cursor& operator=(state_type q)`	plain cursor	Set the cursor to point to state `q`
`cursor& operator=(const cursor&)`	plain cursor	Assignment operator
`bool operator==(const cursor&) const`	plain cursor	Return `true` iff both cursors point to the same state
`bool sink() const`	plain cursor	Return `true` iff the cursor points to the sink state `DFA::null_state`
`bool forward(const char_type &a)`	plain cursor	Move along transition labeled with `a` if defined, otherwise move to sink state and return `false`
`bool exists(const char_type &a) const`	plain cursor	Return `true` if a transition labeled with `a` is defined
`bool src_final() const`	plain cursor	Return `true` if pointed state is final
`tag_type src_tag() const`	plain cursor	Return the object associated to pointed state
`char_type letter() const`	forward cursor	Return the letter on the pointed transition. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`bool first()`	forward cursor	Make the cursor point to the first element of the transitions sequence of the source state. Return `true` if there is such an element (if any transition is defined), otherwise the pointed transition is undefined.
`bool next()`	forward cursor	Move the cursor to the next element of the transitions sequence of source state. Return `true` if there is such an element (the cursor is not at the end of the sequence), otherwise the pointed transition is undefined. `first` must have been successfully called prior to using this method.
`bool find(const Alphabet &a)`	forward cursor	Make the cursor point to the transition labelled with `a`. Return `true` if such a transition exists, otherwise the pointed transition is undefined.
`void forward()`	forward cursor	Move forward on the currently pointed transition. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.

Member	Where defined	Description
`bool aim_final() const`	forward cursor	Return `true` if the aim state is final. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`tag_type aim_tag() const`	forward cursor	Return the object associated with the aim state. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`state_type aim() const`	forward cursor	Return the handle of the aim state the cursor is point to. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`bool backward()`	stack cursor	pop the stack top. Return `false` if the resulting stack is empty.

Members

New Members

Membre	Description
`stack_cursor(const ForwardCursor &c)`	Construct a stack cursor with a stack containing `c`.
`stack_cursor()`	Construct a stack cursor with an empty stack.

Helper Functions

```
template <class ForwardCursor>
stack_cursor<ForwardCursor> stackc(const ForwardCursor &x);
```

Notes

- Calls to the `stack_cursor` methods are only valid iff the stack is non-empty.
- The default constructor is used by the depth-first cursor to implement ends of range: the empty stack serves as stop condition for the traversal.

See Also

forward cursor, stack cursor, depth-first cursor, `dfirst_cursor`.

queue_cursor<ForwardCusor, Container>

Category : cursors **Component Type :** Type

Description

A `queue_cursor` is a forward cursor storing its path in a queue of cursors. Each move through the sequence of the outgoing transitions of the source state (`next`) enqueues a forward cursor. An extra method `dequeue` allows to dequeue and to implement the breadth-first traversal.

Definition

`cursor.h`

Template parameters

Parameter	Description	Default
`ForwardCursor`	The type of the cursors stored in the queue	
`Container`	The type of the sequential container implementing the queue	`deque<ForwardCursor>`

Model of

plain cursor, forward cursor, queue cursor.

Type requirements

- `ForwardCursor` is a model of forward cursor.
- `Container` is a model of front insertion sequence.
- `Container::value_type` must be `ForwardCursor`.

Public base classes

`cursor_concept`, `forward_cursor_concept`, `queue_cursor_concept`.

Member	Where defined	Description
`state_type`	plain cursor	The type of the states handles of the underlying DFA, that is, `DFA::state_type`
`char_type`	plain cursor	The type of the transitions letters, `DFA::char_type`
`tag_type`	plain cursor	The type of the tags associated to states, `DFA::tag_type`
`char_traits`	plain cursor	Character traits associated to `char_type`
`cursor()`	plain cursor	Default constructor
`cursor(const cursor&)`	plain cursor	Copy constructor
`state_type src() const`	plain cursor	Return the state handle the cursor is pointing to

Member	Where defined	Description
`cursor& operator=(state_type q)`	plain cursor	Set the cursor to point to state `q`
`cursor& operator=(const cursor&)`	plain cursor	Assignment operator
`bool operator==(const cursor&) const`	plain cursor	Return `true` iff both cursors point to the same state
`bool sink() const`	plain cursor	Return `true` iff the cursor points to the sink state `DFA::null_state`
`bool forward(const char_type &a)`	plain cursor	Move along transition labeled with `a` if defined, otherwise move to sink state and return `false`
`bool exists(const char_type &a) const`	plain cursor	Return `true` if a transition labeled with `a` is defined
`bool src_final() const`	plain cursor	Return `true` if pointed state is final
`tag_type src_tag() const`	plain cursor	Return the object associated to pointed state
`char_type letter() const`	forward cursor	Return the letter on the pointed transition. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`bool first()`	forward cursor	Make the cursor point to the first element of the transitions sequence of the source state. Return `true` if there is such an element (if any transition is defined), otherwise the pointed transition is undefined.
`bool next()`	forward cursor	Move the cursor to the next element of the transitions sequence of source state. Return `true` if there is such an element (the cursor is not at the end of the sequence), otherwise the pointed transition is undefined. `first` must have been successfully called prior to using this method.
`bool find(const Alphabet &a)`	forward cursor	Make the cursor point to the transition labelled with `a`. Return `true` if such a transition exists, otherwise the pointed transition is undefined.
`void forward()`	forward cursor	Move forward on the currently pointed transition. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`bool aim_final() const`	forward cursor	Return `true` if the aim state is final. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.

Member	Where defined	Description
`tag_type aim_tag() const`	forward cursor	Return the object associated with the aim state. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`state_type aim() const`	forward cursor	Return the handle of the aim state the cursor is point to. The cursor must have been set to point to a defined transition beforehand by succesfully calling `first`, `next()` or `find`.
`bool dequeue()`	queue cursor	dequeue the current cursor and return `false` is the resulting queue is empty

Members

New Members

Membre	Description
`queue_cursor(const ForwardCursor &c)`	Construct a cursor with a queue storing `c`.
`queue_cursor()`	Construct a cursor with an empty queue.

Helper Functions

```
template <class ForwardCusor>
queue_cursor<ForwardCusor> queuec(const ForwardCusor &x);
```

Notes

- Calls to the `queue_cursor` methods are only valid iff the queue is non-empty.
- The default constructor is used by the breadth-first cursor to implement ends of range: the empty queue serves as stop condition for the traversal.

See Also

forward cursor, queue cursor, breadth-first cursor, `bfirst_cursor`.

dfirst_cursor<StackCursor>

Category : cursors **Component Type :** Type

Description

A `dfirst_cursor` implements the depth-first traversal on acyclic deterministic automata. It is in some sense an iterator on a sequence of transitions ordered according to the depth-first traversal algorithm. The method `forward` allows to increment the cursor, making it point to the next transition in the sequence. This methods returns `true` if the transition reached has been pushed onto the stack (forward move) and `false` otherwise (pop and backward move).
The `dfirst_cursor` is fundamental because it is used much in the same way as the iterators on sequence to define ranges for algorithms.

Example

```
string w[4] = { "forward", "cursor", "code", "example" };
DFA_bin a;
add_word(a, w, w + 4);
dfirst_cursor<stack_cursor<forward_cursor<DFA_bin<> > > >
  first = dfirstc(a), last;
vector<char> word;
while (first != last) {
  do {
    word.push_back(first.letter());
    if (first.aim_final()) {
      copy(word.begin(), word.end(), ostream_iterator<char>(cout));
      cout << endl;
    }
  } while (first.forward());

  while (!first.forward()) word.pop_back();
}
```

Definition

`cursor.h`

Template parameters

Parameter	Description	Default
`StackCursor`	The type of the underlying stack cursor	

Model of

depth-first cursor.

Type requirements

- `StackCursor` is a model of stack cursor.

Public base classes

`dfirst_cursor_concept`.

Member	Where defined	Description
`state_type`	depth-first cursor	The type of the states handles of the underlying DFA.
`char_type`	depth-first cursor	The type of the transitions letters.
`tag_type`	depth-first cursor	The type of the tags associated to states.
`dfirst_cursor()`	depth-first cursor	Construct a cursor with an empty stack useful as end-of-range iterator.
`dfirst_cursor( const dfirst_cursor &c)`	depth-first cursor	Copy constructor.
`state_type src() const`	depth-first cursor	Return the state handle the cursor is pointing to.
`bool src_final() const`	depth-first cursor	Return `true` if the pointed state is final.
`char_type letter() const`	depth-first cursor	Return the letter on the pointed transition.
`bool aim() const`	depth-first cursor	Return the handle of the aim state the cursor is pointing to.
`bool aim_final() const`	depth-first cursor	Return `true` if the aim state is final.
`bool operator==( const dfirst_cursor &c) const`	depth-first cursor	Return `true` iff both stacks are equal.
`bool forward()`	depth-first cursor	Increment the cursor making it point to the next transition in the sequence. Return `true` if the transition reached has been pushed onto the stack.
`Tag src_tag() const`	depth-first cursor	Return the tag associated to the source state.
`Tag aim_tag() const`	depth-first cursor	Return the tag associated to the aim state.

Members

New Members

Membre	Description
`dfirst_cursor(const StackCursor &c)`	Construct a cursor with `c` as stack.

Helper Functions

The function `dfirstc` returns a `dfirst_cursor` constructed from the object passed as argument. This object is allowed to be a model of DFA, forward cursor or stack cursor.

Notes

- Calls to method other than `operator==` are valid iff the cursor stack is non-empty.
- This cursor works only on acyclic structures, use `dfirst_mark_cursor` on cyclic DFAs.

See Also

depth-first cursor, stack cursor, `dfirst_mark_cursor`.

dfirst_mark_cursor<StackCursor, Marker>

Category : cursors **Component Type :** Type

Description

A `dfirst_mark_cursor` implements the depth-first traversal on cyclic deterministic automata. It works exactly as the `dfirst_cursor` does except that a state-mark function garantees that each transition is only reached twice

Definition

`cursor.h`

Template parameters

Parameter	Description	Default
`StackCursor`	The type of the underlying stack cursor	
`Marker`	The type of the state-mark object function	`set_marker`

Model of

depth-first cursor.

Type requirements

- `StackCursor` is a model of stack cursor.
- `Marker` is a model of state marker.

Public base classes

`dfirst_cursor_concept`.

Member	Where defined	Description
`state_type`	depth-first cursor	The type of the states handles of the underlying DFA.
`char_type`	depth-first cursor	The type of the transitions letters.
`tag_type`	depth-first cursor	The type of the tags associated to states.
`dfirst_cursor()`	depth-first cursor	Construct a cursor with an empty stack useful as end-of-range iterator.
`dfirst_cursor( const dfirst_cursor &c)`	depth-first cursor	Copy constructor.
`state_type src() const`	depth-first cursor	Return the state handle the cursor is pointing to.
`bool src_final() const`	depth-first cursor	Return `true` if the pointed state is final.
`char_type letter() const`	depth-first cursor	Return the letter on the pointed transition.

Member	Where defined	Description
`bool aim() const`	depth-first cursor	Return the handle of the aim state the cursor is pointing to.
`bool aim_final() const`	depth-first cursor	Return `true` if the aim state is final.
`bool operator==( const dfirst_cursor &c) const`	depth-first cursor	Return `true` iff both stacks are equal.
`bool forward()`	depth-first cursor	Increment the cursor making it point to the next transition in the sequence. Return `true` if the transition reached has been pushed onto the stack.
`Tag src_tag() const`	depth-first cursor	Return the tag associated to the source state.
`Tag aim_tag() const`	depth-first cursor	Return the tag associated to the aim state.

Members

New Members

Membre	Description
`dfirst_cursor(const StackCursor &c)`	Construct a cursor with `c` as stack.

Helper Functions

The function `dfirst_markc` returns a `dfirst_mark_cursor` constructed from the object passed as argument. This object is allowed to be a model of DFA, forward cursor or stack cursor.

Notes

- Calls to method other than `operator==` are valid iff the cursor stack is non-empty.
- This cursor is designed for cyclic structures. Using it on acyclic DFAs results in memory and time penalties but should not disrupt the processing beyond that. On a acyclic structures, a `dfirst_cursor` is more appropriated.

See Also

depth-first cursor, stack cursor, `dfirst_cursor`.

bfirst_cursor<QueueCursor>

Category : cursors **Component Type :** Type

Description

A `bfirst_cursor` implements the breadth-first traversal on acyclic deterministic automata. It is an iterator on a sequence of transitions ordered according to the breadth-first traversal algorithm. The method `next` allows to increment the cursor, making it point to the next transition in the sequence. This methods returns `true` if the transition reached has been enqueued and `false` otherwise (dequeue).
The `bfirst_cursor` is used in the same way as the iterators on sequence to define ranges for algorithms.

Example

```
string w[4] = { "forward", "cursor", "code", "example" };
DFA_bin a;
tree_build(a, w, w + 4);
bfirst_cursor<queue_cursor<forward_cursor<DFA_bin<> > > >
  first = bfirstc(a), last;
while (first != last) {
  do
    cout << "src " << first.src() << " letter " << first.letter()
         << " aim " << first.aim() << endl;
  while (first.next());
}
```

Definition

`cursor.h`

Template parameters

Parameter	Description	Default
`QueueCursor`	the type of the underlying queue cursor	

Model of

breadth-first cursor.

Type requirements

- `QueueCursor` is a model of queue cursor.

Public base classes

`bfirst_cursor_concept`.

Member	Where defined	Description
`state_type`	breadth-first cursor	The type of the states handles of the underlying DFA.
`char_type`	breadth-first cursor	The type of the transitions letters.

Member	Where defined	Description
`tag_type`	breadth-first cursor	The type of the tags associated to states.
`bfirst_cursor()`	breadth-first cursor	Construct a cursor with an empty queue useful as end-of-range iterator.
`bfirst_cursor( const bfirst_cursor &c)`	breadth-first cursor	Copy constructor.
`state_type src() const`	breadth-first cursor	Return the state handle the cursor is pointing to.
`bool src_final() const`	breadth-first cursor	Return `true` if the pointed state is final.
`char_type letter() const`	breadth-first cursor	Return the letter on the pointed transition.
`bool aim() const`	breadth-first cursor	Return the handle of the aim state the cursor is pointing to.
`bool aim_final() const`	breadth-first cursor	Return `true` if the aim state is final.
`bool operator==( const bfirst_cursor &c) const`	breadth-first cursor	Return `true` iff both queues are equal.
`Tag src_tag() const`	breadth-first cursor	Return the tag associated to the source state.
`Tag aim_tag() const`	breadth-first cursor	Return the tag associated to the aim state.
`bool next_transition()`	breadth-first cursor	Increment the cursor making it point to the next transition in the sequence. Return `true` if there are transitions left.

Members

New Members

Member	Description
`bfirst_cursor(const QueueCursor &c)`	Construct a cursor with `c` as queue.

Helper Functions

The function `bfirst_c` returns a `bfirst_cursor` constructed from the object passed as argument. This object is allowed to be a model of DFA, forward cursor or queue cursor.

Notes

- Calls to method other than `operator==` are valid iff the cursor queue is non-empty.
- This cursor works only on acyclic structures, use `bfirst_mark_cursor` on cyclic DFAs.

See Also

breadth-first cursor, queue cursor, `bfirst_mark_cursor`.

6.4 Cursor Adapters

6.5 Algorithms

first_match

Category : Algorithm **Component Type :** Function

Prototype

```
template <typename ForwardI, typename Cursor>
ForwardI
first_match(Cursor &c, ForwardI first, ForwardI last);

template <typename ForwardI, typename Cursor>
ForwardI
first_match(const Cursor &c, ForwardI first, ForwardI last);
```

Description

Returns a past-the-end iterator on the shortest prefix of the word `[first, last)` recognized by the automaton pointed by `c`. If none is found, `first` is returned which means that either no final state has been reached during the traversal or a transition was undefined.
The first version of the algorithm does not copy the cursor for efficiency reasons as this function is to be called intensively during pattern recognition processings. The second version allows the algorithm to be called with constant cursors like those returned by the helper functions as `forwardc()`.

Definition

`match.h`

Requirements on types

- `ForwardI` is a model of forward iterator.
- `Cursor` is a model of plain cursor.
- `ForwardI::value_type` is convertible to `Cursor::char_type`.

Preconditions

- `[first, last)` is a valid range.
- `c.sink()` is false.

Complexity

At most `last - first` calls to `Cursor::forward`.

Example

See Also

`match`, `longest_match`, `match_count`

longest_match

Category : Algorithm **Component Type :** Function

Prototype

```
template <typename ForwardI, typename Cursor>
ForwardI
longest_match(Cursor &c, ForwardI first, ForwardI last);

template <typename ForwardI, typename Cursor>
ForwardI
longest_match(const Cursor &c, ForwardI first, ForwardI last);
```

Description

Returns a past-the-end iterator on the longest prefix of the word `[first, last)` recognized by the automaton pointed by `c`. If none is found, `first` is returned which means that either no final state has been reached during the traversal or a transition was undefined.
The first version of the algorithm does not copy the cursor for efficiency reasons as this function is to be called intensively during pattern recognition processings. The second version allows the algorithm to be called with constant cursors like those returned by the helper functions as `forwardc()`.

Definition

`match.h`

Requirements on types

- `ForwardI` is a model of forward iterator.
- `Cursor` is a model of plain cursor.
- `ForwardI::value_type` is convertible to `Cursor::char_type`.

Preconditions

- `[first, last)` is a valid range.
- `c.sink()` is false.

Complexity

At most `last - first` calls to `Cursor::forward`.

Example

See Also

`match`, `first_match`, `match_count`

match_count

Category : Algorithm **Component Type :** Function

Prototype

```
template <typename InputI, typename Cursor>
int match_count(Cursor &c, InputI first, InputI last);

template <typename InputI, typename Cursor>
int match_count(const Cursor &c, InputI first, InputI last);
```

Description

Returns a count of the prefixes of the sequence `[first, last)` recognized by the automaton pointed to by c, in other words, a count of states where `c.src_final()` is true which were reached during the automaton traversal along the path `[first, last)`.
The first version of the algorithm does not copy the cursor for efficiency reasons as this function is to be called intensively during pattern recognition processings. The second version allows the algorithm to be called with constant cursors like those returned by the helper functions as `forwardc()`.

Definition

`match.h`

Requirements on types

- `InputI` is a model of input iterator.
- `Cursor` is a model of plain cursor.
- `InputI::value_type` is convertible to `Cursor::char_type`.

Preconditions

- `[first, last)` is a valid range.
- `c.sink()` is false.

Complexity

At most `last - first` calls to `Cursor::forward`.

Example

See Also

`match`, `first_match`, `longest_match`

dump

Category : Algorithm **Component Type :** Function

Prototype

```
template <typename DFirstC>
ostream&
dump(ostream &out, DFirstC first, DFirstC last = DFirstC());
```

Description

Writes the automaton defined by the range `[first, last)` to the output stream `out` in an ASCII representation (letters shall have an operator `<<` defined). Such a representation can be read through the algorithm `restore` or the input stream cursor `istream_cursor`. This version of the algorithm does not add states tags to the representation, see `full_dump` for a complete one.

Definition

`stream.h`

Requirements on types

- `DFirstC` is a model of depth-first cursor.
- `ostream& operator<<(ostream&, DFirstC::char_type&)`.

Preconditions

- `[first, last)` is a valid range.

Complexity

$O(n \log n)$ where n is `last - first`.

Example

```
#include <astl.h>
#include <stream.h>
#include <minimize.h>
#include <iostream>

int main()
{
  DFA_bin<plain, minimization_tag> A;
  // construct with words from stdin:
  add_words(A, istream_iterator<string>(std::cin),
               istream_iterator<string>());
  // minimize:
  acyclic_minimization(A);
  // dump to stdout getting rid of minimization_tag:
  dump(std::cout, dfirst_markc(A));
}
```

Notes

- Since this algorithm does not take in account states tags, it is a good way to get rid of tags which were added temporarily to the automaton in order to apply an algorithm.

See Also

full_dump, restore, istream_cursor

full_dump

Category : Algorithm **Component Type :** Function

Prototype

```
template <typename DFirstC>
ostream&
full_dump(ostream &out, DFirstC first, DFirstC last = DFirstC());
```

Description

Writes the automaton defined by the range `[first, last)` to the output stream `out` in an ASCII representation (letters shall have an operator `<<` defined). Such a representation can be read through the algorithm `restore` or the input stream cursor `istream_cursor`. This version of the algorithm adds states tags to the representation imposing that an operator `<<` be defined for tags, see `dump` for a tagless representation.

Definition

`stream.h`

Requirements on types

- `DFirstC` is a model of depth-first cursor.
- `ostream& operator<<(ostream&, DFirstC::char_type&)`.
- `ostream& operator<<(ostream&, DFirstC::tag_type&)`.

Preconditions

- `[first, last)` is a valid range.

Complexity

$O(n \log n)$ where n is `last - first`.

Example

```
#include <astl.h>
#include <stream.h>
#include <hash.h>
#include <iostream>

int main()
{
  DFA_map<plain, hash_tag> A;
  // construct with words from stdin:
  add_words(A, istream_iterator<string>(std::cin),
               istream_iterator<string>());
  // turn A into a hashing automaton:
  make_hash(A);
  // dump to stdout with hashing tags:
  full_dump(std::cout, dfirst_markc(A));
}
```

See Also

dump, restore, istream_cursor

restore

Category : Algorithm **Component Type :** Function

Prototype

```
template <typename FA>
void restore(FA &a, istream &in);
```

Description

Reads the automaton ASCII representation from the input stream `in` and construct `a` accordingly. `FA` tags and alphabet types shall match those used to generate the representation (through `dump` or `full_dump`). The first state defined by the representation is set as the automaton initial state.

Definition

`stream.h`

Requirements on types

- `FA` is a model of FA.

Preconditions

- `FA::tag_type` and `FA::char_type` must match those used to write the ASCII representation.

Complexity

$O(n \log n)$ where n is the transition count of the input representation

Example

```
#include <astl.h>
#include <stream.h>
#include <hash.h>
#include <iostream>

int main()
{
  // reconstruct the hashing automaton of dull_dump example:
  DFA_map<plain, hash_tag> A;
  restore(A, std::cin);
}
```

See Also

dump, full_dump, istream_cursor

7 Advanced Examples