Automata and performances

Guillaume Lazzara

Technical Report n^o0606, June 2006
revision 878

Summary
Implementing a mathematic theory on computers is not always as easy as it seems to be. Usually, the most intuitive implementation is not the fastest.

So, with this point of view, we will introduce you performance problems with automata, together with a study of the best methods that could be used in Vaucanson in the future.

Résumé
Passer d'une théorie mathématique à une implémentation sur machine n'est pas toujours évident. Bien souvent d'ailleurs, l'implémentation la plus intuitive n'est pas la plus performante.

Les automates n'échappent pas à cette règle. Être capable de pousser les machines au-delà des limites, optimiser leur implémentation et les algorithmes associés est une réelle nécessité.

C'est donc dans cet esprit que nous présenterons les problèmes de performances liés aux automates booléens, ainsi qu'une étude des méthodes les plus pertinentes qui pourraient être utilisées à l'avenir dans Vaucanson.

Keywords
Vaucanson, automata, performance, implementation.

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

Copying this document

Copyright © 2006 LRDE.

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

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

Contents

Introduction

Nowadays, few automaton libraries are available. Generic automata libraries are even harder to find than specialized one. FSM is one of the best known automata libraries but it is proprietary. As there is no free library like FSM in the open-source community, Vaucanson has to be a free generic open source alternative.

Willing to be a free open source alternative is nice, but to be considered as an alternative, it is required to have the same performances or, at least, almost the same. Currently, it is not really the case, and particulary for big automata like, for instance, a million states automaton. Under certain circumstances, FSM is up to a thousand time faster than Vaucanson. One reason is the fact that FSM is written in C and Vaucanson in C++. C++ allows us to be more generic and to keep mathematical objects in the implemention, but it is costly: resolving virtual functions symbols is very costly compared to C for which this cost does not exist. The Performances of Vaucanson are all the more bad that FSM is almost as generic as Vaucanson... Meaning FSM and Vaucanson both manipulate boolean automata and transducers. Perhaps, FSM have a specific implementation and specific algorithms for each kind of automaton whereas Vaucanson does not, but it cannot be the only reason.

We will see through this report that the performance issues of Vaucanson can be mostly fixed by changing automata structures and associative data structures used in algorithms. Usually, the standard set (std::set) and map (std::map) from the STL are used to implement such algorithms or structures. However, those structures can quickly reach their limit under certain circumstancies due to their internal implementation. The performances of those standard containers depend on the number of elements stored in it. Thus, using millions of elements in those containers can lead to a significant slow down. That's why we decided to use a different structure, which is not provided in the STL: the hashtables. We will set out a comparative between several hashtables subject to help us in improving Vaucanson.

From this first comparative, we will also compare different implementations which use different container structures and different automata implementation. The benchmark will use the two most costly algorithms: determinization and minimization. Thanks to this benchmark, we will be able to determine which implementation is the best and which one would, hopefully, help us to be almost as fast as FSM.

Chapter 1

How to improve performances ?

1.1 The performance issues

The performances of an automaton implementation are, like many other kind of libraries, based on the way the algorithms are coded. But, if we search a little bit more, we realize that even the best algorithms cannot help us if we do not have a structure adapted to the way automata will be used.

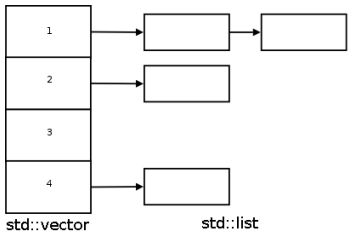

Figure 1.1: Adjacency list structure

For instance, in almost all algorithms working on automata, we have to iterate over all the states and the transitions. So, to store all the states, it would be worth using a vector instead of a list, especially because we rarely add or remove states. On the contrary, transitions can be deleted more often and we have to privilege this functionality. So, a list is probably more interesting for it. This implementation we have just described (fig. 1.1) is the classical one. It is used in Vaucanson and many programs using automata or graphs. The cost in memory and the access time to get an element are globally sufficent to fit the usual use cases: it is a good compromise between memory cost and access time.
But, in our case, in Vaucanson, when one starts manipulating automata with millions of states, this structure starts to reach its limits. Usually, the more you have states, the more you have transitions. With millions of states, accessing information about a state can be retrieved in a constant time, but when one wants to get a specific transition, iterating over a list with probably millions of elements, it becomes very costly! This is due to the list's $O(n)$[1] complexity. Here

[1] Where n is the number of states. We will keep this notation all along the report.

one can see there is a possible improvement. Perhaps, one could decrease the complexity using another structure to store transitions.

As we have said at the beginning, the performances can be increased by changing the structure used in the algorithms. The choice of the container structures is very important because, depending on their functionnalities, algorithms can be implemented differently: one can relinquish temporary structures or one can avoid several tests. The best examples are the determinization and the minimization. Those two algorithms use a lot of associative structures. One has to create a lot of associations between states and new temporary virtual states. The STL provides such kind of containers: std::map and std::set. Those containers are perfect for basic use of Vaucanson with small automata with thousands of states, for instance. However, those structures are implemented using red/black trees which complexity is better than a usual list: $O(log(n))$. Here again, like the list used to store the transitions, the more you would have transitions, the more your search for a specific transition would be costly.

After this short overview, we have seen one would be really interested in a solution offering a constant complexity: $O(1)$. This solution exists and can be used in the two contexts explained above: the hashtables. As one can imagine, there is not only one implementation of such structure. That is why, before starting to solve our problem, we compared different hashtable implementations.

1.2 Hashtables comparative

Before introducing the results of the comparative, we would like to remind how a "classical" hashtable works. A hashtable is a data structure that associates keys with values. The primary operation it supports efficiently is a lookup: given a key, find the corresponding value. It works by transforming the key using a hash function into a hash, a number that the hash table uses to locate the desired value.

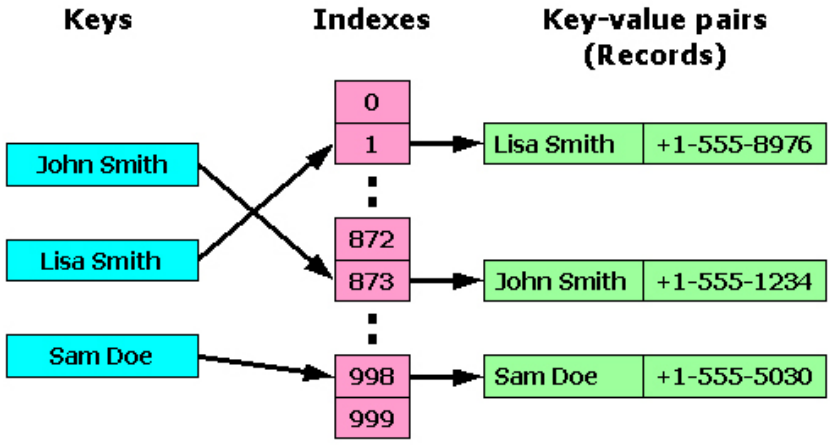

Figure 1.2: How a hashtable works.

Figure 1.2 illustrates the hashtable definition given above. First, you give a name; thanks to the hash function, the hashtable gets an id and is able to access directly to a pair structure containing the key, i.e. the name, and the data associated, i.e. the phone number associated.

As one saw above, before trying to solve our performance problem with the first hashtable found on the Internet, with the help of Samuel Charron, we tried to compare different implementations. We searched many hashtables over the Internet for our comparative and, finally, we retained four different implementations :

- Google dense hash map
- Boost Multi Index (BMI)
- GNU hashmapGNU hashmap
- Qt QHashmapQt QHashmap

We chose the Google dense hash map because they were known for their good performances (memory use and execution time). We chose also Boost Multi Index because of their flexibility and the reactivity of the community. The GNU Hashmap was elected as it has been developed in order to complete the STL. Finally, we wanted to try also Qt QHashmap because Qt is well known for its easiness and its good performances. It is also well known as it is used in KDE. We found other hashtables like the Mozilla hashtable, or GTK hashtable, but the first one was too specialized for Mozilla products and the second one was written in C. As we wanted to use hashtables in a C++ project, we had eliminated directly those which were written in another language.

1.2.1 How did we proceed?

Each test depends on some parameters we changed at compile time. So, for one algorithm tested, some parameters fixed and an implementation, we had one executable. The execution time only includes the algorithm execution time.

To test each algorithm, we have to insert elements in the container. We have chosen integer numbers for convenience.
The integers inserted are generated once, before compiling all test binaries. They are stored in four constant integer arrays, allowing us to share those integers with other test executables.
The reason we generated four constant integer arrays was that, as we use hashtables, we specified our hash function. For integers, we thought that an identity function would be adapted and would make tests easier. So, having tables with numbers different from those inserted in the container is interesting when one want to do random deletion: numbers may or may not be in the container.

For the test, we used a struct with two fields for insertion: key and value. Both fields were an integer.

One test is composed of four parameters:

- SIZE: The number of integers to generate and insert in the container.
- ALGO: The algorithm one wants to test.

- IALGO: The way one wants to insert integer numbers (collision, no collision...).
- RESERVE: The size one wants to pre-allocate in the container.

The benchmark was done in two steps:

- Addition of SIZE elements in the tested hashtable.
- Launch of the algorithm.

The performance measurement is done before and after the algorithm execution. All elements are distinct. Three hash functions are used to control collisions.

- No collision: the function returns the element itself.
- Always collisions: the function always returns 0.
- Randomized collisions: an array of size SIZE elements returns a randomized value.

1.2.2 Algorithms

Eleven algorithms were chosen for the tests:

- Insertion only.
- Forward remove.
- Backard remove.
- Randomized remove.
- Iterate: iterates over all elements of the hash table and dereferences the iterator.
- Forward search.
- Backward search.
- Randomized search.
- Forward extraction.
- Backward extraction.
- Randomized extraction.

All the algorithms were launched through this loop, except the forward algorithm for which the 0 and the SIZE values were switched.

```
for i <- 0 to SIZE do
  Algorithm(hash[i])
```

For the random algorithms, the rand() function from the STL library was used.

1.2.3 Comparative results

To be able to realize the benefit of using a hashtable instead of a std::map, we included the STL std::map in the comparative. It has the same usage as a hashtable and is the actual container used in Vaucanson.

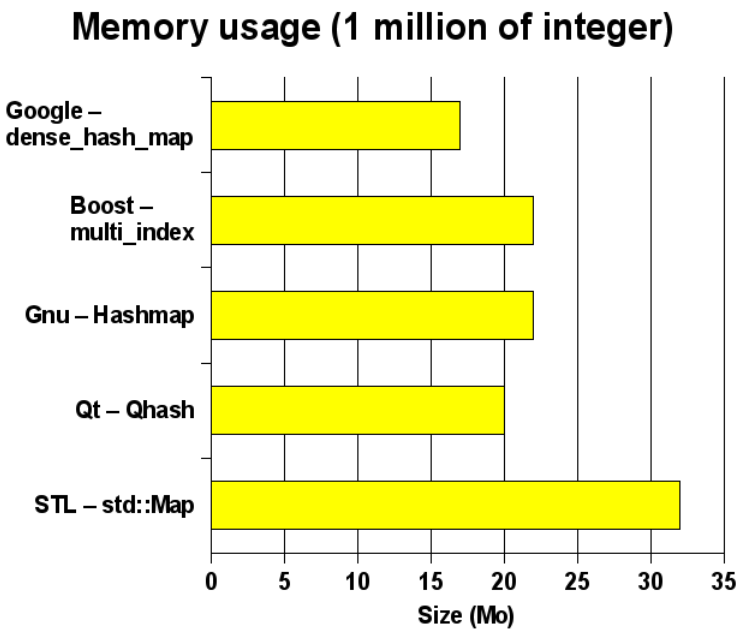

Figure 1.3: memory usage for 1 million of integer inserted

On this first chart (fig. 1.3) we compare the memory used by the structure to store a million of elements. As one can see, hashtables are globally ligther in memory than a standard std::map. The use of memory reaches an average of 20 Mo for the hashtables where std::map uses 32Mo! The benefit reaches the value of 30% which is not negligible. Here one can see clearly that Google dense hash maps are the best when we consider the memory consumption.

Before commenting the next table (table 1.1), we would like to introduce how the score has been calculated. As the main goal is to compare the hashtables to the std::map, we used the std::map as a reference. So, first, we calculated the time made by a the std::map to pass all the tests. The total time will be written down T_{ref} and the time made to fulfill a test t_i. For information, the reference time T_{ref} of the std::map was 6.39s.

$$T_{ref} = \sum t_i$$

Then to calculate the score T' of each hashtable, the formula below has been used:

$$T' = \frac{\sum t'_i}{T_{ref}}$$

HashTable	Score
Google dense_hash_map	3.65
Boost multi index	4.89
GNU hashmap	5.49
Qt QHash	8.22
STL std::map	100

Table 1.1: Score illustrating the speed of each hashtables to execute all the tests.

Using this formula implies that lower is the score, better is the implementation. As one can see, Google dense hash map is the best implementation. It is about 25% faster than Boost multi index and 95% faster than the the std::map! Google is unmistakably the best. Obviously, this implementation has been chosen for the implementation of the algorithms and the automata. At the beginning, we wanted to choose only one implementation of hashtable, but we realized quickly that, even if Boost Multi Index does not have the best performance, it offers many interesting functionalities we will describe further. So, finally, two hashtable implementations has been retained in the automata implementation: Boost Multi Index (BMI) and Google dense hash map.

1.3 Hashtables overview

Before introducing the different automata implementations, we would like to give an overview of the two hashtables we have chose.

1.3.1 Google dense hash map

As one saw thanks to the comparative, Google dense hashmap is really fast for everything. Furthermore, the memory consumption compared to the fastness is amazing. Google hashtables really seems to be our ultimate solution. But behind those attractive advantages, Google hashtable have also very bad defects for our needs :

- Like the std::map, those hashtables can only have a single key to access a specific data. It could be a little bit weird to consider this as a defect, but we will see later that this kind of hashtables could be very interesting for us.
- A default value is needed to delete an empty key. This is a bit restrictive because if, for example, one uses it with letters extracted from an alphabet, one will have to choose a symbol which will not be used.
- Keys and Data must be plain old data. It means using c++ objects directly by using copy constructor is not possible. The objects allocation and the memory management has to be done manually. It is not a serious defect but it remains restrictive.
- Inserting an element invalidates all iterators. This is a critical point: one wants to have permanent validity iterators. In many algorithms one have to iterate over a container and to modify it during the iteration. Non permanent validity iterators force us to make copies and/or make search for elements. When one iterates a million times it is too costly!
- Finally, another critical point: this hashtable cannot behave like a multimap. Too many times, in the algorithms, one wants to store elements with the same key. Unfortunatly, we cannot do this with this structure. It forces to create more temporary structures and finally slows down the algorithms.

As one can see, the favorite hashtable is not adapted to this problem. The good performances of this structure implies too many defects to be usable. The constraints were so important that I decided to fully renounce to use this hashtable. So this is the last time in this report where we will talk about Google hashtables.

1.3.2 Boost Multi Index (BMI)

Fortunately, Boost Multi Index go past our initial needs. It is the most flexible hashtable tested in this comparative:

- It can behave like a multimap. As we said before, one really wants this functionnality. Optimizations are possible with this.
- It certifies permanent validity iterators.

The functionalities we have just introduced are quiet basic and common. But BMI offers other innovative functionalities exported from the world of databases:

- Several different keys can be specified to access the same data, in the same hashtable.
- It is also possible to create composed keys, meaning, you have to specify two or several elements to access the associated data.
- Finally, the most useful and wonderful functionnality: you can maintain simultaneously different sorting of the data!

As one can see, BMI offers the possibility to access data by different ways. The fact it can maintain simultaneous different sorting is really interesting because we don't have to create temporary structures in algorithms to sort the data. Accessing the data by different keys implies the same benefit as well. BMI will be used in almost all my automata implementations in the future for thoses reasons. However, careful, BMI seems to be a little bit "magic" but its functionnalities have a cost and they can slow down performances if one abuses of multiple sorting and multiple keys.

We created a small example which shows the differences of performance between a hashtable with a simple sorting and another one with three different sortings. Here is the code:

```
struct s_transition
{
  s_transition (unsigned int src_, unsigned int dst_, char lbl_)
      : src(src_),
        dst(dst_),
        lbl(lbl_)
        {}

  unsigned int  src;
  unsigned int  dst;
  char          lbl;
};


int main()
{
  hash_t hashtable;

  for (int i = 0; i < 10000000; ++i)
    hashtable.insert(s_transition(i, i, i));

  return 0;
}
```

In this very simple exemple, we add a structure, containing 2 integers and one char, 10 millions time in the hashtable. For this test, we used two different hashtables we will call light_hashtable (fig. 1.4) and heavy_hashtable (fig. 1.5). The light_hashtable use the "composed key" feature of Boost. There is only one way to get some data from this hashtable: provide a "src" and "lbl" value. Each structure s_transition are identified by this pair. Then, in the heavy_hashtable there are three different ways to get an element. Either you provide a pair (src,lbl), or (dsl,lbl), or just (src). This structure is of course heavier to maintain. Thanks to Boost, everything is done automatically but it still takes some time to be computed as shown by the results below. We also wanted to compare the Boost multi key feature with a normal way of simulating this behavior: the use of several hashtables. The several_hashtable results correspond to the time made by a heavy_hashtable splitted into three different hashtables.

HashTable	**Time**
light_hashtable	12.491s
heavy_hashtable	24.168s
several_hashtable	35.175s

As expected, maintaining 3 different keys with boost is less expensive than maintaining three structures at the same time: about 33% faster! This example shows also that maintaining several keys with Boost really slows down the performances: between a hashtable with only one key and another one with three keys, the performances are divided by 2! Thus, Boost features should be used carefully.

Figure 1.4: Structure of the light_hashtable

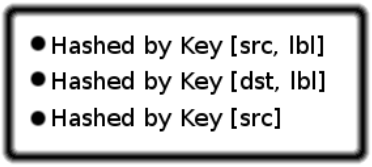

Figure 1.5: Structure of the heavy_hashtable

Last point, we have said above that not having a multimap would lead us to create more local variables. Let's just see a simple exemple. In the minimization implementation for the adjacency list one uses this complex structure:

```
147  vector<map<char, pairSet > >          pred (A.order ());
```

Thank to boost, we can merge all those maps and this vector into one unique structure detailed in the boost-container.hh file page 106 between lines 244 and 263.
As we've said before, maintaining a Boost hashtable with several keys is faster than maintaining several maps. Thus, here, we save computing time. Of course, with Google hash map, we would have had the same lack of performance, like the adjacency list structure, as we would have had to create at least two hashtables.

Chapter 2

Introduction to the algorithms

2.1 Minimization

Two algorithms can be used to minimize an automaton: Hopcroft's or Moore's. The first one, Hopcroft's, has a $nlog(n)$ complexity whereas Moore's has a n^2 complexity. That is the main reason why we will focus more on the Hopcroft algorithm in this report. Here is a formal description of the algorithm:

function minimization_Hopcroft

Variables:

Π:	set of set of states.
Q:	set containing all the states.
F:	set containing the final states only.
T:	list of set of states.

begin

Initialize $\Pi = \{Q \setminus F, F\}$
 Put F in the "To treat" list T
While T is not empty
 Remove a set S of T
 For each letter in Σ
  Compute $\left| \begin{array}{l} X : the list of states going in S \\ Y : the list of states that don't \end{array} \right.$
  For each sets π in Π
   if $\pi \cap X \neq \emptyset$ and $\pi \cap Y \neq \emptyset$
    Split π in $\pi \cap X$ and $\pi \cap Y$
    Insert the smaller one in T

end

When we will implement this algorithm, one will have one set built already, the Q set, which is the automaton, actually.
Here, one will also have to find the final states quickly. So, maybe it could be interesting to have a set containing them in the automata to pick them all directly. One will also need to know

which states are going to a specific state. So, here, one would really like the automaton to store both outgoing and incoming edges.

In the algorithm itself, there are a lot of sets. Compared to mathematics, in computing, one wants to avoid manipulating this structure as often as possible. For instance, huge sets lead to too many comparisons if an intersection is made with another one. Thus, in the implementation we will have to try to find another way to represent sets in the memory.

2.2 Determinization

function determinization

Variables

A2: automaton.
P: list of set of states.
A: alphabet of the source automaton.

Begin

initialize $A2 =$ new automaton
Add a new state $sinkState$ to $A2$
initialize $P = I$
while $P \neq \emptyset$
remove one set S of P
 for each letter a of A
  compute set E: successors of all the states in S whose transition is labeled by a
  if E has not been treated yet
   create new state NS in $A2$
   add E to P
   add an edge between NS and S
   if E contains an initial state
    mark NS as initial in $A2$
   if E contains a final state
    mark NS as final in $A2$
  else
   add an edge between NS and $sinkState$ in $A2$
end

In this algorithm, first, we have to remove all the espilon transitions in the automaton. This operation requires all the states and transitions to be iterated. So, it is costly if we are sure the current automaton doesn't have any espilon transition. Here it could be interesting if the automaton could provide this information with a boolean for example.
Then in the main part of the algorithm, one have to compute sets of successors for each letter in the alphabet. As they are sets of succesors of a set of states, it means, if one doesn't choose a correct implementation, one will have a triple loop which is completly not optimized as we would have to test if the current item matchs one or two parameters, the letter and the state for example. This is a critical part of the algorithm where we will have to be careful during the implementation.

Chapter 3

Different implementations

In this section, we will expose all the implementations we have made. we will try to show the differences from an algorithm point of view and an automaton structure point of view.

3.1 Adjacency list

3.1.1 Automata structure

First, we chose to implement a simple automata structure using adjacency list. This structure is quiet simple as we explained before: a std::vector is used to store each states. Each state contains a list corresponding to its transitions. This implementation exists in two variants:

- The first one, **list-stl**, is fully implemented using the STL, meaning both automata structure and algorithms use it. This implementation has its importance because it is a sort of reference implementation.
- Then, **list-boost** has also an automata structure using STL but containers in the algorithms have been changed by Boost Multi Index. The reason we decided to make such an implementation is that we wanted to know if the automata structure had an impact on the algorithms performances even if they use really fast structures.

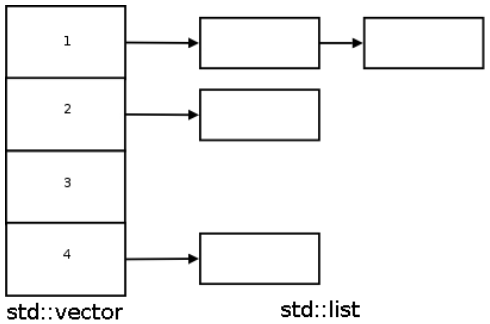

Figure 3.1: Adjacency list structure

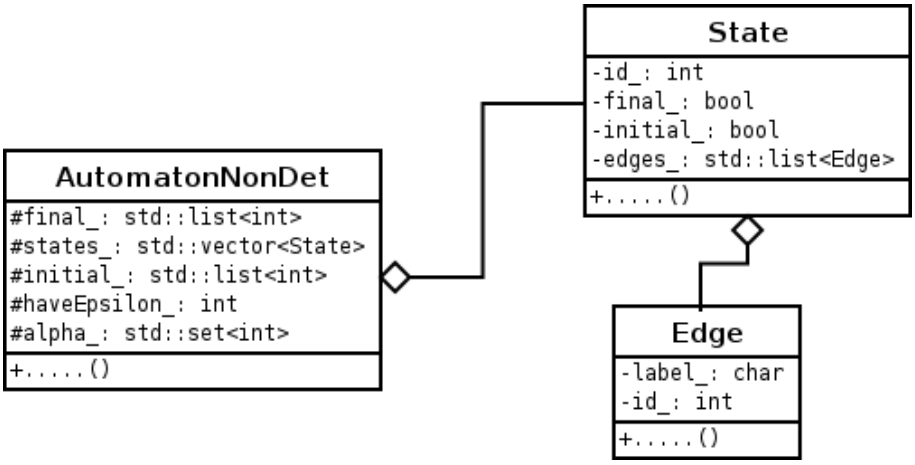

Figure 3.2: UML diagram of the Adjacency list structure

The whole implementation of those structures can be found on pages 37, 46, 58, 59, 61 and 62.

Here are the core of the automaton object and the two associated objects for edges and states (fig. 3.2). First, one can see this automaton structure require three objects and is a little bit heavy in memory. We can notice that we use a lot of list too. It is better but not always efficient. As searching in a list to know if a state is final or initial is costly, we added a bool in the state structure to check this quickly. As one can see, this structure being limited, we have to find tricks to keep it fast which is at the expense of the memory consumption.

One last thing which is common to all implementations coming next, we have added an integer called haveEpsilon_ telling if there are epsilon transitions or not. It can avoid the removing epsilon transition step for the trim algorithm for example.

3.1.2 Algorithms implementation

Determinization algorithm

The whole implementation of this algorithm can be found in section 6.8 page 63.

The main function to use this algorithm is:

```
69 AutomatonDet
70 determinize (const AutomatonNonDet& A)
```

This function, with this implementation, needs a lot of temporary structures. We will see in details the use of each of them.

```
72   AutomatonNonDet                    tmp (A);
```

First, this copy of the automaton passed in parameter allow us to trim it if needed.

```
73   AutomatonDet                       res;
```

This other temporary automaton will be used to store the determinized automaton.

```
74   set<set<int> >                     newNodes;
```

This set is very important. During the determinization process, we create "virtual states" which match states sets from the initial automaton. Each time we find a new state for our determinized automaton, we add it to this set and we will apply the algorithm to it.

```
75   std::map<char, set<int> >          trans;
```

This map is also very useful when we are processing a state of our new determinized automaton. The label and the outgoing state of edges, starting from this current state, are stored in it. This structure is costly because for each state of the determinized automaton we have to clear it thanks to the function :

```
37 initTrans (map<char, set<int> >& trans, const AutomatonNonDet& A)
```

```
76   std::map<std::set<int>, int>       nodes;
```

This structure allows us to get the state number of state created in the determinized automaton from a particular states set.

In all those temporary structures, one can see we already use two std::map and one std::set, and we don't count the numerous std::set used in those containers. Obviously, you can imagine that with millions of states, those structures will consume a lot of memory and, most of all, they will be very slow.

Minimization algorithm

The whole implementation of this algorithm can be found in section 6.9 page 66.

the main function for this algorithm is:

```
144 AutomatonDet
145 minimize (const AutomatonDet& A)
```

This function use big temporary structures we are going to detail below.

```
147   vector<map<char, pairSet > >          pred (A.order ());
```

This one is the biggest structure used in this algorithm, memory speaking. It holds the predecessors for each states for each particular labels. Unfortunatly, we have to keep it in memory from the beginning to the end of the algorithm due to structures limitations. Furthermore, the initialization of this structure takes a long time with a lot of states.

```
184         set<int>          X;
185         set<int>          Y;
186         set<int>          tmp;
187         std::insert_iterator<set<int> > X_ins(X, X.begin());
188         std::insert_iterator<set<int> > Y_ins(tmp, tmp.begin());
```

With the Hopcroft algorithm, we have to create several sets and to split them under certain conditions. First we have to create a set, X, with the predecessors of the state of the minimized automaton currently processed, and then, we have to create another set, Y, with the rest of the states. With the STL and the STL automaton structure, we have to use the temporary structure containing information on the predecessors and the X and Y set, as described above. We have to make intersection or union of several sets. Of course, with few elements, it is quick to compute but it becomes quickly a nightmare when you have thousands of elements.

One sees the use of sets is not a good idea. Unfortunately, the automaton structure impose us a way of writing the code which is not ideal for such algorithms. If we want to write this code differently with this structure, bidirectional maps would be a good solution. That is why Boost could help us and one will see that it will help us more than we expected.

3.2 Adjacency Boost Multi Index (ABMI)

3.2.1 Automaton structure

The full implementation of this structure can be found on pages 71, 81, 88, 99, 100 and 102.

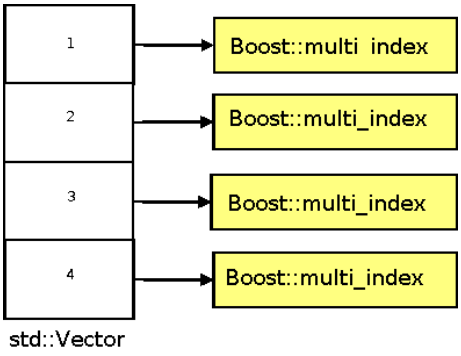

Figure 3.3: Adjacency BMI structure

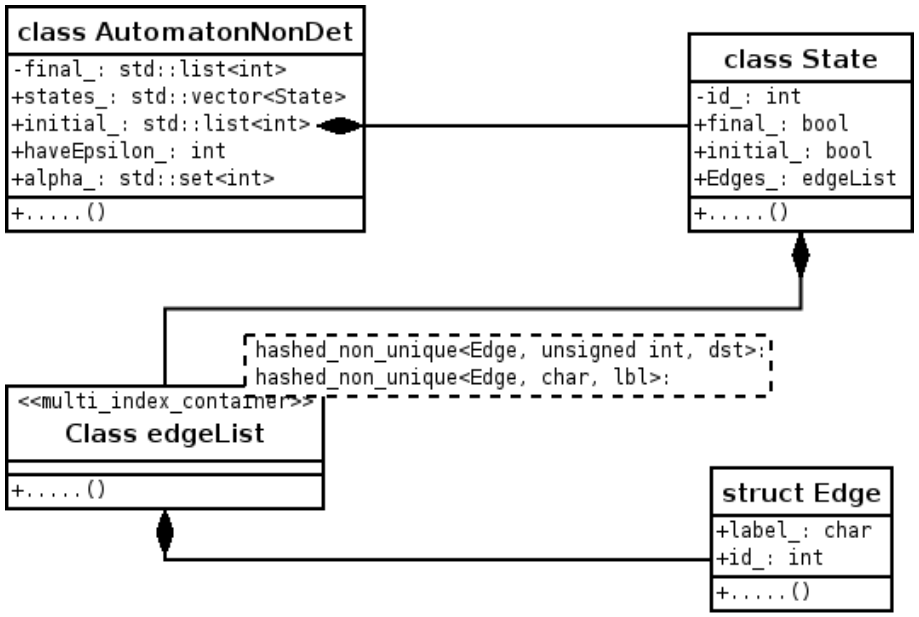

Figure 3.4: Adjacency BMI UML diagram

As one can see on figure 3.4, this implementation is very similar to the previous one. One still have a std::vector to store states information. The states are identical as well. The main difference is the way the transitions are stored: in the State class, instead of a simple std::list we have

now a BMI (class EdgeList, see fig. 3.4). This structure provides constant access to a specific transition! This is not negligible. Moreover, thanks to this structure, transitions are accessible by two different ways: either by specifying the destination state or the label.

In this implementation, we use BMI in both the algorithms and the automaton structure.

As we have said, we save computing time. However, this new structure takes more memory: about 30% more than the adjacency list (see table 3.1). It could be a drawback but, as one is going to see further, all the information stored now don't need to be maintained anymore and can be used directly, whereas with the adjacency list, we need to create new temporary structures and clear them during the execution of the algorithm which implies memory allocations and deallocations which is costly.

HashTable	**Size**
Adjacency BMI	336Mo
Adjacency List	235Mo

Table 3.1: Comparison of the size used to store 10 000 000 millions of Edge elements.

3.2.2 Algorithms implementation

Determinization algorithm

The whole implementation of this algorithm can be found in section 6.17 page 103.

Here is the declaration of a temporary variable type:

```
struct s_enstonb
{
  s_enstonb(const int_set& f, unsigned int i)
    : first(f),
      second(i)
  {}

  int_set       first;
  unsigned int  second;
};

typedef multi_index_container
<
  s_enstonb,
  indexed_by
  <
    hashed_unique
    <
      BOOST_MULTI_INDEX_MEMBER(s_enstonb, int_set, first),
      boost_hash,
```

```
      boost_equal
    >
> ensToNb_map;
```

In fact, this type ensToNb_map is a hash map which use a struct s_enstonb to store elements and its field 'first' as key. The algorithm uses also other types created to replace the STL's ones:

- **int_setlist**: a simple list using the Boost Multi Index structure without hash system.
- **int_set**: a simple int set using the Boost Multi Index structure without hash system.

The whole implementation of those structures can be found in section 6.11 on page 74.

Now one has seen the types, let's see what are the differences and similitudes compared to the STL implementation. First, it is interesting to notice we don't have the temporary variable, called "trans", anymore. Thus, we do not need to created sets of successors anymore. With all this, we already save computing time. Otherwise, we keep the same structure and variables. Now everything is written with Boost, one can wonder if it makes the code harder to read. Fortunately not. For proof, for instance, with the STL structure you had in the determinize function:

```
106     initTrans (trans, A);
107
108     for (set_it = node->begin (); set_it != node->end (); ++set_it)
109       for (it = tmp.edge_begin (*set_it);
110            it != tmp.edge_end (*set_it); ++it) {
111
112         trans[it->label_get ()].insert (it->id_get ());
113       }
```

This was a little bit longer to write and of course really slow because we had to create the sets. Now, with ABMI, we can delete this code and add in the updateNodes function a new test:

```
28          AutomatonNonDet::transitionRange trans;
29          trans = A.getTransFromSrcLbl(*it2, *it);
30
31          for (index_iterator<edgeList, lbl>::type elt = trans.first;
32               elt != trans.second; ++elt)
33            {
34              if (A.isFinal(elt->dst))
35                haveFinal = true;
36
37              ens.insert(elt->dst);
38            }
```

We just have to get transitions labeled by a certain letter and starting from a certain state. Then if it does not exist we do nothing or, otherwise, we add it to our new "virtual state". Easy and fast: searching in the nodes object does not cost anything thanks to hashtables.

Minimization algorithm

The whole implementation of this algorithm can be found in section 6.18 page 106.

Before introducing the differences and similitudes between ABMI and STL on this algorithm, we would like to introduce two new types:

- **graph_t**: this is a multi index using hashtables. It contains complete information about transitions: source, destination and label. For each triplet you can access data using source or destination as key.
- **EquivSet**: this is a multi index containing a pair of int. It uses also hashtables. Each element of one pair is a key to access the pair.

First, at the beginning of the algorithm we have to create a graph_t structure and fill it with all information about transitions. It take some time but, then, accessing an element is fast. So for constructing the structure we are almost as slow as the STL implementation but during the use in the algorithm we save a lot of time.

Then, remember, for each state of the minimized automaton we had to create two sets: X and Y. Here it is really different and really faster! Just check by yourself on page 106:

```
147     std::vector<bool>           X(A.order(), false);
```

ABMI makes possible a new way of iterating the automaton: instead of "for each states and for each letter" we can write "for each letter and for each states" without spoiling the algorithm results. It means the number of iterations doesn't depend on the number of states anymore, but on the number of letters in the alphabet. Furthermore, we can access critical data directly, at anytime, thanks to the hashtables. So, thanks to this, we succeed in replacing sets by only one vector of bool. Of course, in the splitPi function we have done the same: every set have been replaced by a simple call to a search function in the hashtable.

Finally, when the base automaton has been totally processed, we call a function called createMinimized which create the final automaton. Thanks to this structure we do not need any equivalence table for the states whereas STL structure does.

3.3 Boost Multi Index (BMI)

3.3.1 Automaton structure

The full implementation of this structure can be found on pages 110 and 121.

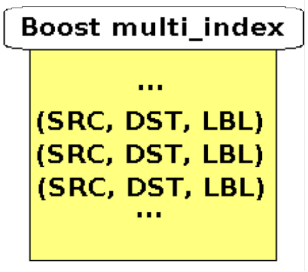

Figure 3.5: BMI structure

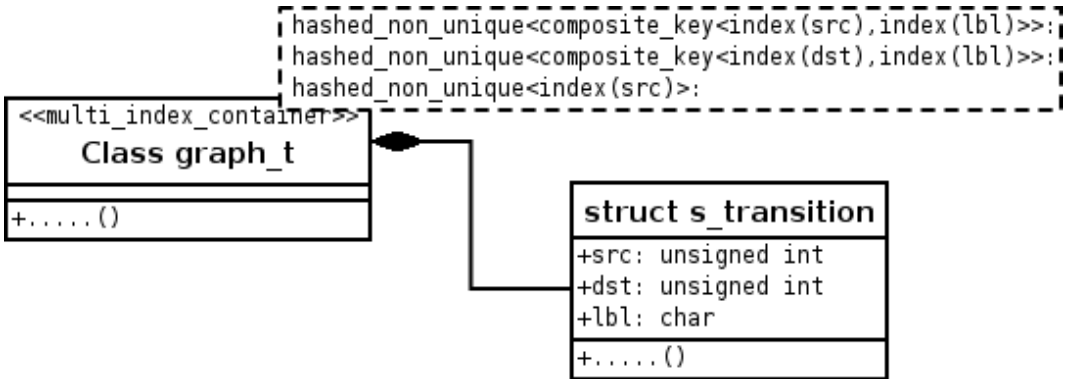

Figure 3.6: BMI structure UML diagram

This structure is a bit particular. It is far from any mathematics theory about automata: there is not any notion of states set. Here we will only find transitions information. The automata is a multi index composed of triplets: source, destination and label.
As one can see on the UML diagram (fig. 3.6), one has 3 possible keys to get data:

- A composed key specifying source and label.
- Another composed key specifying destination and label.
- A simple key specifying a label.

3.3.2 Algorithms implementation

Determinization algorithm

For this algorithm, ABMI and BMI are very similar. There is nothing to add.

Minimization algorithm

Here again BMI behaves like ABMI. Nevertheless, there is an improvement. For ABMI, we filled an object graph_t which tooks a certain time:

```
117  for (AutomatonNonDet::node_const_iterator it_node = A.begin();
118       it_node != A.end(); ++it_node) {
119
120    AutomatonNonDet::edge_const_iterator it_edge;
121    for (it_edge = it_node->begin();
122         it_edge != it_node->end(); ++it_edge)
123
124      pred.insert(s_transition(it_node->id_get(),
125                               it_edge->dst, it_edge->lbl));
126  }
```

Now, we do not need this anymore because we have all information in our automaton structure.

Chapter 4

Results on the initialization of automata

Even though we focused on the execution time of the main automata algorithms, we wanted to compare the time needed to create the different structures.

Basically, all the implementations using Boost are heavier in memory and take more time to be constructed, as we store more information in the automaton structure, . Here follow two tables (table 4.1 and 4.2) which compare the three implementations described before.

When one adds 100 000 new states to the automaton (see table 4.1), one can see clearly that and adjacency BMI is about 12 times slower and about 20 times heavier in memory compared to the adjacency list: it has to create a multi index for each state added, which means high memory allocation, thus, more time to instantiate a new state. However, the structure of the BMI allows it to be really fast when adding a new state, about 7 times faster than the adjacency list, as it only has to increment a counter: only the transitions are stored in this structure.

Add new transitions is also very costly compared to a adjacency list as the multi index have to create the different keys and sorting (see table 4.2). The adjacency list, for instance, is 6 times slower than the adjacency list and takes 3 times more memory. Fortunately, the BMI structure is better: it is still 5 times slower thant the adjacency list but requires only 2 times more memory. It could seem to be a lot of memory, however with the BMI structure we store 3 times more information than the adjacency list.

HashTable	**Size**	**Time**
Adjacency BMI	615Mo	12.45s
Adjacency List	28Mo	0.72s
BMI	1.5Mo	0.1s
Vaucanson	14Mo	0.21s

Table 4.1: Results for each hashtable when adding 100 000 states.

HashTable	**Size**	**Time**
Adjacency BMI	897Mo	29.07s
Adjacency List	255Mo	5.81s
BMI	462Mo	26.22s
Vaucanson	1900Mo	93.14s

Table 4.2: Results for each hashtable when adding 100 000 states and 100 transitions per state.

Chapter 5

Results on the algorithms

5.1 How did we proceed?

For the tests we used the "ladybird" automata. This is a kind of automata which is very interesting for testing algorithms capacity: if the base automata has n states, then the deterministic automaton would have 2^n states. On each test, we incremented the number of states by one.

As we said at the beginning of this report, we have tested the automaton structure with the determinization and minimization algorithm. As it exists two algorithms for the minimization, we decided to test both. But it is important to know that Hopcroft's algorithm ($n \log(n)$) is better than the Moore's one (n^2).

We used a dedicated machine for the tests. Here is the configuration : two Pentium 4 Xeon HyperThreading, 4Go DDR Ram.

5.2 Determinization algorithm

Here are the first results of our tests (chart 5.1). Without any surprise, Vaucanson is out really early on the graph: too slow and too much memory consumption. Then, one has an interesting result, even if it is not the best yet. The list-stl is faster than list-boost implementation. Thanks to this result, one can see the automata structure has to be adapted to the algorithms. Even with good containers and good algorithms, we get mitigated results. Then, if one continues with the chart 5.1, one finds FSM which is not so bad. One would even say 'good' because if one compares its memory consumption to the rest, it is globally the best.

Finally, there are the two last implementations which use Boost. First, BMI is very good. It can reach 2^22 states without problem. Unfortunately, for the ABMI implementation, it cannot reach 2^22 due to the memory size limitation on 32 bits architecture (chart 5.2). So, for the determinization algorithm, the BMI implementation seems to be the best.

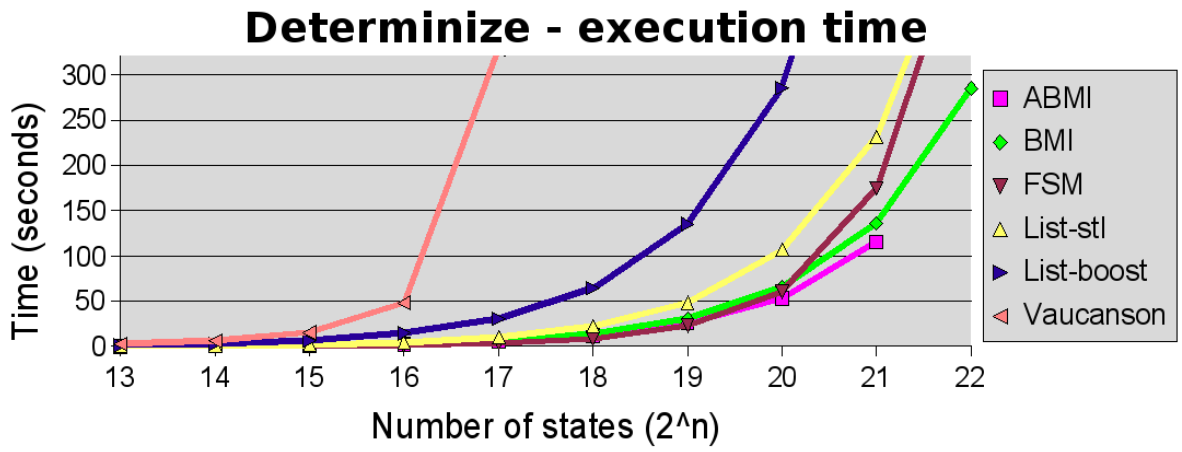

Figure 5.1: Execution time of the determinization algorithm according to the number of states.

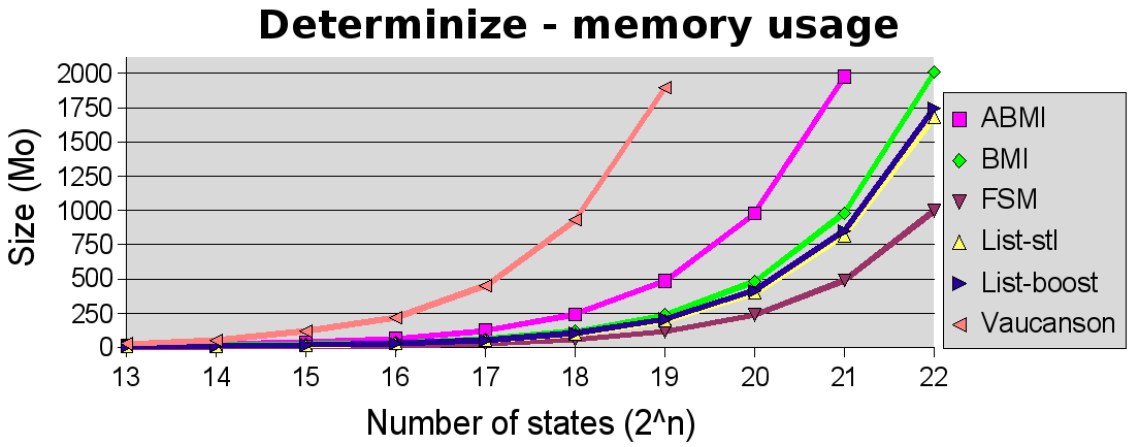

Figure 5.2: Memory usage of the determinization algorithm according to the number of states.

5.3 Minimization algorithm (Hopcroft)

First, on chart 5.3, one can see that the list-stl and list-boost implementations are very bad, worse than Vaucanson... So, we really think we have badly implemented those one. After Vaucanson, one sees ABMI implementation which is really better than Vaucanson. Finally, near the number of states axis, one sees BMI and FSM. For sure, BMI is not far from FSM but we do not really know their difference of performance. So, one can see the second chart, chart 5.4, with a log scale. On this chart, BMI is clearly not as fast as FSM but there is an important advantage compared to Vaucanson and the others implementations: the algorithm has a $n \log(n)$ complexity whatever the number of states. So, this is not a full victory against FSM but hope is here!

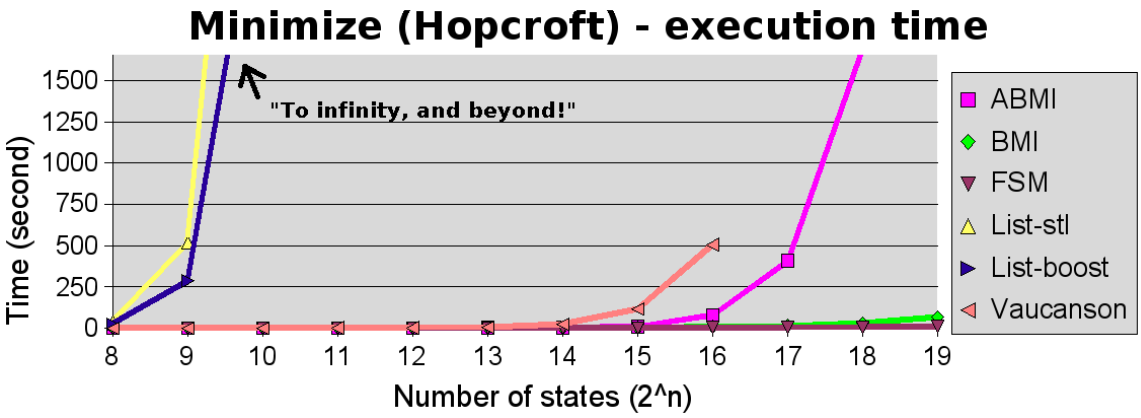

Figure 5.3: Execution time of the Hopcroft's minimization algorithm according to the number of states.

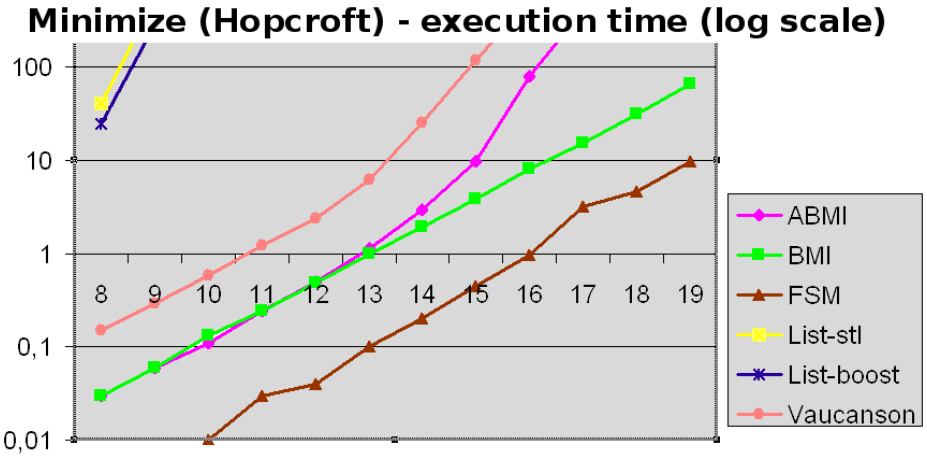

Figure 5.4: Execution time of the Hopcroft's minimization algorithm according to the number of states in log scale.

5.4 Minimization algorithm (Moore)

We decided not to suppress this chart (chart 5.5) but, if one reads the result, one will see they are not interesting compared to those got with Hopcroft. Even if those results should not be very good, they should not be as bad as they are. We have probably badly implemented this algorithm too.
Last point: we put the FSM results on this chart but there is no way to choose the minimization algorithm with FSM. It was a reference value to improve the comparison facility.

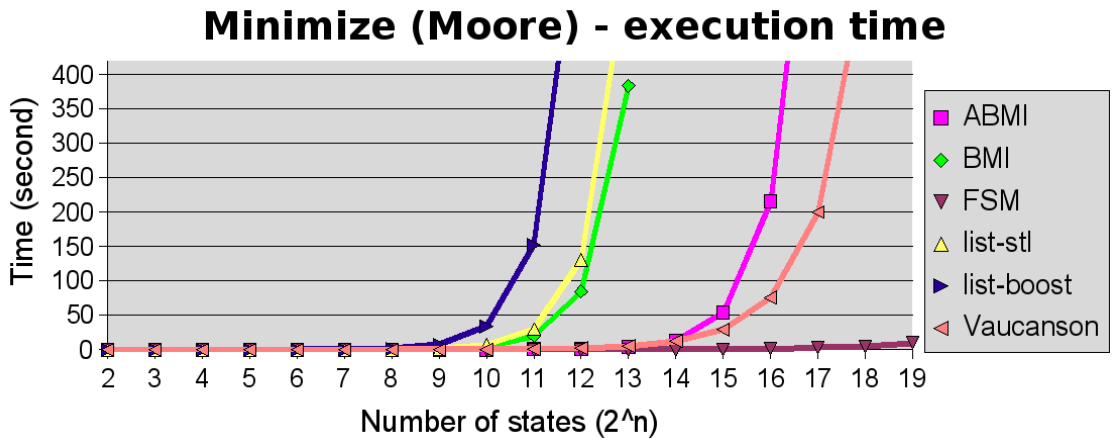

Figure 5.5: Execution time of the Moore's minimization algorithm.

Conclusion

Through this report, one has seen that hashtables are a very good alternative for a huge amount of associative data. The Boost Multi Index was a very good surprise too. On the contrary to Google Dense Hash Map, they allow several functionalities similar to a database; that is exactly what we wanted: bidirectional maps and constant access time to element, Boost gave it to us.

Thanks to the benchmarks, one discovered different things about the several implementations. First, with the list-stl and the list-boost implementation, one has seen the importance of the automata structure to have good results with algorithms. Good containers and good algorithms are not sufficient. Finally, pure Boost implementation seems to be the best. ABMI is a good one but it still requires too much memory in the algorithm, in particularly in the minimize algorithm. Finally, the most balanced choice is the BMI implementation: fast, "light" and "compact".

During the test, we hoped to be able to be better than FSM on the two algorithms. Finally it was only with the determinization algorithm. For the minimization, we stayed really near from FSM but we are still worse.

Thanks to this report, we have seen that improving Vaucanson is possible by changing the automaton structure with Boost Multi Index. But for now, a question persists: will this new structure need to rewrite totally Vaucanson? We have not totally the answer but what it is sure is that programming algorithm for list-stl or BMI implementation is very different. So we can imagine that with Vaucanson we would have to rewrite a big part of the existing code.

If Vaucanson would require most of this code to be rewritten, this would allow new possibilites for improving performances. Indeed, other ways of improving Vaucanson are possible but would require to rewrite the code as well:

- **Threading**: Some algorithms could be threaded. For instance, the minimization algorithm can be splitted into different parts to create a producter/consumer pattern. One could create up to four threads in this algorithm.
- **Shared memory between objects**: like the shared memory mechanism used in the Qt library, each class used in Vaucanson could share its attributes with other copies of this object until one of them is modified. So, whatever the size of the class, a non modified copy of an object would have the size of an integer.
- **Dedicated hashtables**: in this report one tried different hashtable implementations which were mostly generic. But, now one have found our needs for the hashtables, why not implementing a hashtable perfecly adapted to our needs?

- **A new memory allocator**: both the STL and Boost offer the possibility to set a new memory allocator function. This new function could reduce the number of system call used to increase the size of the heap and increase significantly the performance. For instance, instead of allocating thousands of small areas one could allocate hundreds of larger areas.

Chapter 6

Appendix

In this section, the full souce code of all the implementations is gathered. Few headers might be missing, like lib.hh, because they are not relevant in this report. The missing files usually contains forward declarations only.

6.1 automaton-non-det.hh - Adjacency list

```
1  #ifndef AUTOMATONNONDET_HH_
2  # define AUTOMATONNONDET_HH_
3
4  # include <vector>
5  # include <set>
6  # include "edge.hh"
7  # include "node.hh"
8
9  class Node;
10 class Edge;
11
12 class AutomatonNonDet
13 {
14   public:
15
16     //Iterator Typedef
17     typedef std::vector<Node>::iterator          node_iterator;
18     typedef std::vector<Node>::const_iterator    node_const_iterator;
19     typedef std::list<Edge>::iterator            edge_iterator;
20     typedef std::list<Edge>::const_iterator      edge_const_iterator;
21     typedef std::list<int>::iterator             initial_iterator;
22     typedef std::list<int>::iterator             final_iterator;
23     typedef std::list<int>::const_iterator       initial_const_iterator;
24     typedef std::list<int>::const_iterator       final_const_iterator;
25     typedef std::set<int>::iterator              alpha_iterator;
26     typedef std::set<int>::const_iterator        alpha_const_iterator;
27
28     //Constructors - destructor
29     virtual                      ~AutomatonNonDet () {}
30                                  AutomatonNonDet (int nb = 0);
31
32     //Interface
33     virtual void                 createNbNodes (int);
34     virtual node_iterator        nodeAdd (bool initial = false, bool final = false);
35     virtual node_iterator        nodeDel (int i);
36     virtual node_iterator        nodeDel (node_iterator);
37     virtual node_iterator        nodeDelWoClean (node_iterator);
38     virtual node_iterator        nodeDelWoClean (int i);
39     virtual void                 edgeAdd (Node& u, Node& v, char c);
40     virtual void                 edgeAdd (int u, int v, char c);
41     virtual void                 edgeAddBack (int u, int v, char c);
42     virtual edge_iterator        edgeDel (Node&, edge_iterator&);
43     virtual edge_iterator        edgeDel (int, edge_iterator&);
44     virtual bool                 initialAdd (int i);
45     virtual initial_iterator     initialDel (initial_iterator);
46     virtual void                 initialClear ();
47     virtual void                 finalAdd (int i);
```

```
48      virtual final_iterator       finalDel (final_iterator);
49      virtual void                 finalClear ();
50      virtual void                 print () const;
51      virtual void                 printSorted () const;
52      virtual void                 setFinalState (const AutomatonNonDet &, int = 0);
53      virtual void                 setInitialState (const AutomatonNonDet &A,
54                                                    int offset);
55
56      virtual node_iterator        nodeToIt (Node& it);
57      virtual const Node&          node (int) const;
58
59      virtual int                  haveEpsilon () const;
60      virtual unsigned             order () const;
61      virtual unsigned             initial_nb () const;
62
63      //Iterator getter
64      virtual edge_const_iterator          edge_begin (int i) const;
65      virtual edge_iterator                edge_begin (int i);
66      virtual edge_const_iterator          edge_end (int i) const;
67      virtual edge_iterator                edge_end (int i);
68      virtual node_iterator                begin ();
69      virtual node_iterator                end ();
70      virtual node_const_iterator          begin () const;
71      virtual node_const_iterator          end () const;
72      virtual initial_const_iterator       initial_begin () const;
73      virtual initial_iterator             initial_begin ();
74      virtual final_iterator               final_begin ();
75      virtual final_const_iterator         final_begin () const;
76      virtual initial_const_iterator       initial_end () const;
77      virtual initial_iterator             initial_end ();
78      virtual final_iterator               final_end ();
79      virtual final_const_iterator         final_end () const;
80      virtual alpha_iterator               alpha_begin();
81      virtual alpha_iterator               alpha_end();
82      virtual alpha_const_iterator         alpha_begin() const;
83      virtual alpha_const_iterator         alpha_end() const;
84
85      AutomatonNonDet&                     operator= (const AutomatonNonDet&);
86
87    protected:
88      void                  edgeUpdate (edge_iterator it);
89      void                  statesUpdate (std::list<int>& l, node_iterator it);
90      std::list<int>        final_;
91      std::vector<Node>     nodes_;
92      std::list<int>        initial_;
93      int                   haveEpsilon_;
94      std::set<int>         alpha_;
95  };
96
```

```
 97
 98 AutomatonNonDet           operator+ (const AutomatonNonDet&, const AutomatonNonDet&);
 99 AutomatonNonDet           operator* (const AutomatonNonDet&, const AutomatonNonDet&);
100 AutomatonNonDet           operator* (const AutomatonNonDet&);
101
102 # include "automaton-non-det.hxx"
103
104 #endif /* !AUTOMATONNONDET_HH_ */
```

6.2 automaton-non-det.hxx - Adjacency list

```
1  /**
2   *
3   * Return a state iterator on the beginning
4   * of the state list
5   *
6   */
7  /** @{ */
8  inline
9  AutomatonNonDet::node_iterator
10 AutomatonNonDet::begin ()
11 {
12   return nodes_.begin ();
13 }
14
15
16
17 inline
18 AutomatonNonDet::node_const_iterator
19 AutomatonNonDet::begin () const
20 {
21   return nodes_.begin ();
22 }
23 /** @} */
24
25
26 /**
27  *
28  * Return a state iterator on the end
29  * of the state list
30  *
31  */
32 /** @{ */
33 inline
34 AutomatonNonDet::node_iterator
35 AutomatonNonDet::end ()
36 {
37   return nodes_.end ();
38 }
39
40 inline
41 AutomatonNonDet::node_const_iterator
42 AutomatonNonDet::end () const
43 {
44   return nodes_.end ();
45 }
46 /** @} */
47
```

```
48
49 /**
50  *
51  * Return an iterator on the beginning
52  * of the initial state list
53  *
54  */
55 /** @{ */
56 inline
57 AutomatonNonDet::initial_const_iterator
58 AutomatonNonDet::initial_begin () const
59 {
60   return initial_.begin ();
61 }
62
63
64 inline
65 AutomatonNonDet::initial_iterator
66 AutomatonNonDet::initial_begin ()
67 {
68   return initial_.begin ();
69 }
70 /** @} */
71
72
73 /**
74  *
75  * Return an iterator on the beginning
76  * of the final state list
77  *
78  */
79 /** @{ */
80 inline
81 AutomatonNonDet::final_iterator
82 AutomatonNonDet::final_begin ()
83 {
84   return final_.begin ();
85 }
86
87
88 inline
89 AutomatonNonDet::final_const_iterator
90 AutomatonNonDet::final_begin () const
91 {
92   return final_.begin ();
93 }
94 /** @} */
95
96
```

```
 97 /**
 98  *
 99  * Return an iterator on the end
100  * of the initial state list
101  *
102  */
103 /** @{ */
104 inline
105 AutomatonNonDet::initial_const_iterator
106 AutomatonNonDet::initial_end () const
107 {
108   return initial_.end ();
109 }
110
111
112 inline
113 AutomatonNonDet::initial_iterator
114 AutomatonNonDet::initial_end ()
115 {
116   return initial_.end ();
117 }
118 /** @} */
119
120
121 /**
122  *
123  * Return an iterator on the end
124  * of the final state list
125  *
126  */
127 /** @{ */
128 inline
129 AutomatonNonDet::final_iterator
130 AutomatonNonDet::final_end ()
131 {
132   return final_.end ();
133 }
134
135
136 inline
137 AutomatonNonDet::final_const_iterator
138 AutomatonNonDet::final_end () const
139 {
140   return final_.end ();
141 }
142 /** @} */
143
144
145 /**
```

```
146   *
147   * Return the number of state
148   *
149   */
150  inline
151  unsigned
152  AutomatonNonDet::order () const
153  {
154    return nodes_.size ();
155  }
156
157
158  /**
159   *
160   * Return an iterator on the beginning
161   * of the ith state transition list
162   *
163   */
164  /** @{ */
165  inline
166  AutomatonNonDet::edge_const_iterator
167  AutomatonNonDet::edge_begin (int i) const
168  {
169    return nodes_[i].begin ();
170  }
171
172
173
174  inline
175  AutomatonNonDet::edge_iterator
176  AutomatonNonDet::edge_begin (int i)
177  {
178    return nodes_[i].begin ();
179  }
180  /** @} */
181
182
183  /**
184   *
185   * Return an iterator on the end
186   * of the ith state transition list
187   *
188   */
189  /** @{ */
190  inline
191  AutomatonNonDet::edge_const_iterator
192  AutomatonNonDet::edge_end (int i) const
193  {
194    return nodes_[i].end ();
```

```
195 }
196
197
198
199 inline
200 AutomatonNonDet::edge_iterator
201 AutomatonNonDet::edge_end (int i)
202 {
203   return nodes_[i].end ();
204 }
205 /** @} */
206
207
208 /**
209  *
210  * Return the number of initial state
211  *
212  */
213 inline
214 unsigned
215 AutomatonNonDet::initial_nb () const
216 {
217   return initial_.size ();
218 }
219
220
221 inline
222 void
223 AutomatonNonDet::initialClear ()
224 {
225   initial_iterator      it;
226
227   for (it = initial_.begin (); it != initial_.end ();)
228     it = initialDel (it);
229   initial_.clear ();
230 }
231
232
233 inline
234 void
235 AutomatonNonDet::finalClear ()
236 {
237   final_iterator        it;
238
239   for (it = final_.begin (); it != final_.end ();)
240     it = finalDel (it);
241   final_.clear ();
242 }
243
```

```
244
245 inline
246 int
247 AutomatonNonDet::haveEpsilon () const
248 {
249   return haveEpsilon_;
250 }
251
252
253 inline
254 AutomatonNonDet::alpha_iterator
255 AutomatonNonDet::alpha_begin()
256 {
257   return alpha_.begin();
258 }
259
260
261 inline
262 AutomatonNonDet::alpha_iterator
263 AutomatonNonDet::alpha_end()
264 {
265   return alpha_.end();
266 }
267
268
269 inline
270 AutomatonNonDet::alpha_const_iterator
271 AutomatonNonDet::alpha_begin() const
272 {
273   return alpha_.begin();
274 }
275
276
277 inline
278 AutomatonNonDet::alpha_const_iterator
279 AutomatonNonDet::alpha_end() const
280 {
281   return alpha_.end();
282 }
```

6.3 automaton-non-det.cc - Adjacency list

```
1  #include <vector>
2  #include <iostream>
3  #include <map>
4  #include "automaton-non-det.hh"
5
6
7  /**
8   * @brief Other constructor
9   *
10  * Initialize an AutomatonNonDet and create
11  * nb states.
12  *
13  */
14 AutomatonNonDet::AutomatonNonDet (int nb)
15   : haveEpsilon_(0)
16 {
17   createNbNodes (nb);
18 }
19
20
21 /**
22  * @brief Create nb states in the automaton
23  *
24  * Create nb states in the automaton.
25  */
26 void
27 AutomatonNonDet::createNbNodes (int nb)
28 {
29   for (int i = 0; i < nb; ++i)
30     nodeAdd ();
31 }
32
33
34 /**
35  * @brief Add a state to the automaton.
36  *
37  * Add a state to the automaton.
38  * You can specify if it is final and/or
39  * an initial one.
40  *
41  */
42 AutomatonNonDet::node_iterator
43 AutomatonNonDet::nodeAdd (bool initial, bool final)
44 {
45   if (initial)
46     initial_.push_back (nodes_.size ());
47   if (final)
```

```
48      final_.push_back (nodes_.size ());
49    return nodes_.insert (nodes_.end (), Node (nodes_.size (), initial, final));
50  }
51
52
53  /**
54   *
55   * Remove a state from the automaton.
56   *
57   */
58  /** @{ */
59  AutomatonNonDet::node_iterator
60  AutomatonNonDet::nodeDel (int i)
61  {
62    nodeDel (nodes_.begin () + i);
63  }
64
65  AutomatonNonDet::node_iterator
66  AutomatonNonDet::nodeDel (node_iterator it)
67  {
68    node_iterator node;
69    edge_iterator edge;
70
71    statesUpdate (initial_, it);
72    statesUpdate (final_, it);
73    for (node = nodes_.begin (); node != nodes_.end (); ++node)
74    {
75      if (node->id_get () > it->id_get ())
76        node->id_update ();
77      for (edge = node->begin (); edge != node->end (); )
78        if (edge->id_get () == it->id_get ())
79          edge = node->edgeDel (edge);
80        else
81        {
82          if (edge->id_get () > it->id_get ())
83            edgeUpdate (edge);
84          ++edge;
85        }
86    }
87    return nodes_.erase (it);
88  }
89  /** @} */
90
91
92  /**
93   *
94   * Remove a state from the automaton without deleting transitions
95   * pointing to it from other states.
96   *
```

```
97   */
98  /** @{ */
99  AutomatonNonDet::node_iterator
100 AutomatonNonDet::nodeDelWoClean (node_iterator it)
101 {
102   return nodes_.erase (it);
103 }
104
105 AutomatonNonDet::node_iterator
106 AutomatonNonDet::nodeDelWoClean (int i)
107 {
108   return nodes_.erase (nodes_.begin () + i);
109 }
110 /** @} */
111
112 /**
113  *
114  * Add a transition between 2 states
115  *
116  */
117 /** @{ */
118 void
119 AutomatonNonDet::edgeAdd (Node& u,
120                           Node& v,
121                           char  c)
122 {
123   if (c != 1)
124     alpha_.insert(c);
125   else
126     ++haveEpsilon_;
127   u.edgeAdd (v, c);
128 }
129
130 void
131 AutomatonNonDet::edgeAdd (int   u,
132                           int   v,
133                           char  c)
134 {
135   if (c != 1)
136     alpha_.insert(c);
137   else
138     ++haveEpsilon_;
139   nodes_[u].edgeAdd (v, c);
140 }
141
142 void
143 AutomatonNonDet::edgeAddBack (int       u,
144                               int       v,
145                               char      c)
```

```
146 {
147   if (c != 1)
148     alpha_.insert(c);
149   else
150     ++haveEpsilon_;
151   nodes_[u].edgeAddBack (v, c);
152 }
153 /** @} */
154
155
156 /**
157  *
158  * Delete a transition between 2 states
159  *
160  */
161 /** @{ */
162 AutomatonNonDet::edge_iterator
163 AutomatonNonDet::edgeDel (Node&                 itnode,
164                           edge_iterator&        itedge)
165 {
166   if (itedge->label_get() == 1)
167     --haveEpsilon_;
168   return itnode.edgeDel (itedge);
169 }
170
171 AutomatonNonDet::edge_iterator
172 AutomatonNonDet::edgeDel (int                   itnode,
173                           edge_iterator&        itedge)
174 {
175   if (itedge->label_get() == 1)
176     --haveEpsilon_;
177   return nodes_[itnode].edgeDel (itedge);
178 }
179 /** @} */
180
181
182 /**
183  *
184  * Return a state iterator from its reference.
185  *
186  */
187 AutomatonNonDet::node_iterator
188 AutomatonNonDet::nodeToIt (Node& it)
189 {
190   node_iterator it_node = nodes_.begin ();
191
192   for (; it_node != nodes_.end () && &(*it_node) != &it; ++it_node)
193     ;
194   return it_node;
```

```
195 }
196
197
198 /**
199  *
200  * Display the automaton on stdout
201  *
202  */
203 void
204 AutomatonNonDet::print () const
205 {
206   node_const_iterator   node;
207   edge_const_iterator   edge;
208   final_const_iterator  it;
209
210   for (node = nodes_.begin (); node != nodes_.end (); ++node)
211   {
212     std::cout << node->id_get ();
213     if (node->isInitial ())
214       std::cout << " (I)";
215     else if (node->isFinal ())
216       std::cout << " (F)";
217     else
218       std::cout << "    ";
219     std::cout << " : " ;
220     for (edge = node->begin (); edge != node->end (); ++edge)
221       std::cout << " " << "(" << edge->id_get ()
222                 << "," << edge->label_get () << ")";
223     std::cout << std::endl;
224   }
225   std::cout << "Final states :";
226   for (it = final_begin (); it != final_end (); ++it)
227     std::cout << " " << *it;
228   std::cout << std::endl << "------" << std::endl;
229 }
230
231
232 /**
233  *
234  * Display the automaton on stdout
235  * Transitions are sorted.
236  *
237  */
238 void
239 AutomatonNonDet::printSorted () const
240 {
241   node_const_iterator   node;
242   edge_const_iterator   edge;
243   final_const_iterator  it;
```

```
244   std::multimap<int, char>       sort;
245
246   for (node = nodes_.begin (); node != nodes_.end (); ++node)
247   {
248     std::cout << node->id_get ();
249     if (node->isInitial ())
250       std::cout << " (I)";
251     else if (node->isFinal ())
252       std::cout << " (F)";
253     else
254       std::cout << "    ";
255     std::cout << " : " ;
256     sort.clear ();
257     for (edge = node->begin (); edge != node->end (); ++edge)
258       sort.insert (std::make_pair (edge->id_get (), edge->label_get ()));
259     for (std::multimap<int, char>::iterator it = sort.begin ();
260          it != sort.end (); ++it) {
261
262         std::cout << " " << "(" << it->first << "," << it->second << ")";
263     }
264     std::cout << std::endl;
265   }
266   std::cout << "Final states : {";
267   it = final_begin ();
268   std::cout << *it;
269   for (++it; it != final_end (); ++it)
270     std::cout << "; " << *it;
271   std::cout << "}" << std::endl << "-----" << std::endl;
272 }
273
274
275
276 /**
277  *
278  * Set a state as an initial state
279  *
280  */
281 bool
282 AutomatonNonDet::initialAdd (int i)
283 {
284   nodes_[i].initial_set (true);
285   initial_.push_back (i);
286   return true;
287 }
288
289
290 /**
291  *
292  * Delete an initial state
```

```
293   *
294   */
295  AutomatonNonDet::initial_iterator
296  AutomatonNonDet::initialDel (initial_iterator it)
297  {
298    nodes_[*it].initial_set (false);
299    return initial_.erase (it);
300  }
301
302
303  /**
304   *
305   * Delete a final state
306   *
307   */
308  AutomatonNonDet::final_iterator
309  AutomatonNonDet::finalDel (final_iterator it)
310  {
311    nodes_[*it].final_set (false);
312    return final_.erase (it);
313  }
314
315
316
317  /**
318   *
319   * Set a state as a final state
320   *
321   */
322  void
323  AutomatonNonDet::finalAdd (int i)
324  {
325    nodes_[i].final_set (true);
326    final_.push_back (i);
327  }
328
329  /**
330   *
331   * Return a reference upon a state from its number
332   *
333   */
334  const Node&
335  AutomatonNonDet::node (int i) const
336  {
337    return nodes_[i];
338  }
339
340
341  /**
```

```
342  *
343  * Update edge id when deleting a state
344  *
345  */
346 void
347 AutomatonNonDet::edgeUpdate (edge_iterator edge)
348 {
349   initial_iterator      init;
350   int                   i;
351
352   edge->id_update ();
353   if (nodes_[edge->id_get ()].isInitial ())
354     for (init = initial_.begin (); init != initial_.end ();)
355       if (*init == edge->id_get ())
356       {
357         i = *init - 1;
358         initial_.erase (init++);
359         initial_.push_back (i);
360       }
361       else
362         ++init;
363   if (nodes_[edge->id_get ()].isFinal ())
364     for (init = final_.begin (); init != final_.end ();)
365       if (*init == edge->id_get ())
366       {
367         i = *init - 1;
368         final_.erase (init++);
369         final_.push_back (i);
370       }
371       else
372         ++init;
373 }
374
375
376 /**
377  *
378  * Update initial or final state id when deleting a state
379  *
380  */
381 void
382 AutomatonNonDet::statesUpdate (std::list<int>& l, node_iterator it)
383 {
384   initial_iterator      init;
385   int                   i;
386
387   for (init = l.begin (); init != l.end ();)
388     if (*init == it->id_get ())
389       l.erase (init++);
390     else
```

```
391       {
392         if (*init > it->id_get ())
393         {
394           i = *init - 1;
395           l.erase (init);
396           l.push_back (i);
397         }
398         ++init;
399       }
400   }
401
402
403   /**
404    *
405    * Replace final states by the automaton's ones
406    *
407    */
408   void
409   AutomatonNonDet::setFinalState (const AutomatonNonDet &A, int offset)
410   {
411     final_const_iterator  it;
412
413     finalClear ();
414     for (it = A.final_begin (); it != A.final_end (); ++it)
415       finalAdd (*it + offset);
416   }
417
418
419
420   /**
421    *
422    * Replace initial states by the automaton's ones
423    *
424    */
425   void
426   AutomatonNonDet::setInitialState (const AutomatonNonDet &A, int offset)
427   {
428     initial_const_iterator        it;
429
430     initialClear ();
431     for (it = A.initial_begin (); it != A.initial_end (); ++it)
432       initialAdd (*it + offset);
433   }
434
435
436   /**
437    *
438    * Assign a new automaton to this.
439    *
```

```
440   */
441  AutomatonNonDet&
442  AutomatonNonDet::operator= (const AutomatonNonDet& A)
443  {
444    initial_ = A.initial_;
445    final_ = A.final_;
446    nodes_ = A.nodes_;
447  }
448
449
450  void
451  updateInitial (const AutomatonNonDet& A, AutomatonNonDet& tmp, int initial)
452  {
453    AutomatonNonDet::initial_const_iterator        it_init;
454
455    for (it_init = A.initial_begin (); it_init != A.initial_end (); ++it_init)
456      tmp.edgeAdd (initial, *it_init, 1);
457    tmp.initialClear ();
458    tmp.initialAdd (initial);
459  }
460
461  void
462  updateFinal (const AutomatonNonDet& A, AutomatonNonDet& tmp, int final)
463  {
464    AutomatonNonDet::initial_const_iterator        it_init;
465
466    for (it_init = A.final_begin (); it_init != A.final_end (); ++it_init)
467      tmp.edgeAdd (*it_init, final, 1);
468    tmp.finalClear ();
469    tmp.finalAdd (final);
470  }
471
472
473  /**
474   *
475   * Make an union between 2 automatons
476   *
477   */
478  AutomatonNonDet
479  operator+ (const AutomatonNonDet& A, const AutomatonNonDet& B)
480  {
481    AutomatonNonDet                                tmp (A);
482    AutomatonNonDet::node_const_iterator           it_node;
483    AutomatonNonDet::edge_const_iterator           it_edge;
484    AutomatonNonDet::initial_const_iterator        it_init;
485
486    int initial = tmp.nodeAdd ()->id_get ();
487    int final = tmp.nodeAdd ()->id_get ();
488
```

```
489   updateInitial (A, tmp, initial);
490   updateFinal (A, tmp, final);
491
492   int base = tmp.order ();
493
494   for (int i = 0; i < B.order (); ++i)
495     tmp.nodeAdd ();
496
497   for (it_node = B.begin (); it_node != B.end (); ++it_node)
498     for (it_edge = it_node->begin (); it_edge != it_node->end (); ++it_edge)
499       tmp.edgeAdd (base + it_node->id_get (),
500                    base + it_edge->id_get (),
501                    it_edge->label_get ());
502
503   for (it_init = B.initial_begin (); it_init != B.initial_end (); ++it_init)
504     tmp.edgeAdd (initial, *it_init + base, 1);
505
506   for (it_init = B.final_begin (); it_init != B.final_end (); ++it_init)
507     tmp.edgeAdd (*it_init +  base, final, 1);
508   return tmp;
509 }
510
511
512 /**
513  *
514  * Make an intersection between 2 automatons
515  *
516  */
517 AutomatonNonDet
518 operator* (const AutomatonNonDet& A, const AutomatonNonDet& B)
519 {
520   if (!B.order ())
521     return AutomatonNonDet (A);
522   if (!A.order ())
523     return AutomatonNonDet (B);
524
525   AutomatonNonDet::node_const_iterator          it_node;
526   AutomatonNonDet::edge_const_iterator          it_edge;
527   AutomatonNonDet::initial_const_iterator       it_init;
528   AutomatonNonDet                               tmp (A);
529
530   int base = A.order ();
531   for (int i = 0; i < B.order () + 1; ++i)
532     tmp.nodeAdd ();
533   for (it_node = B.begin (); it_node != B.end (); ++it_node)
534     for (it_edge = it_node->begin (); it_edge != it_node->end (); ++it_edge)
535       tmp.edgeAdd (base + it_node->id_get (),
536                    base + it_edge->id_get (),
537                    it_edge->label_get ());
```

```
538   tmp.edgeAdd (*(A.final_begin ()), tmp.order () - 1, 1);
539   tmp.edgeAdd (tmp.order () - 1, *(B.initial_begin ()) + base, 1);
540   tmp.setFinalState (B, base);
541   return tmp;
542 }
543
544
545 /**
546  *
547  * Add transitions to apply a star
548  *
549  */
550 AutomatonNonDet
551 operator* (const AutomatonNonDet& A)
552 {
553   AutomatonNonDet         tmp (A);
554   int initial = tmp.nodeAdd ()->id_get ();
555   int final = tmp.nodeAdd ()->id_get ();
556
557   tmp.edgeAdd (initial, *(tmp.initial_begin ()), 1);
558   tmp.edgeAdd (*(tmp.final_begin ()), final, 1);
559   tmp.edgeAdd (initial, final, 1);
560   tmp.edgeAdd (*(tmp.final_begin ()), *(tmp.initial_begin ()), 1);
561   tmp.finalClear ();
562   tmp.finalAdd (final);
563   tmp.initialClear ();
564   tmp.initialAdd (initial);
565   return tmp;
566 }
```

6.4 node.hh - Adjacency list

```
1  #ifndef NODE_HH_
2  # define NODE_HH_
3
4  # include <list>
5  # include "edge.hh"
6
7  class Node
8  {
9    public:
10
11     typedef std::list<Edge>::iterator edge_iterator;
12     typedef std::list<Edge>::const_iterator edge_const_iterator;
13
14                          Node (int id, bool initial = false, bool final = false)
15                            : id_ (id), final_ (final), initial_ (initial) {}
16     edge_iterator        edgeAdd (Node& v, char c);
17     edge_iterator        edgeAdd (int v, char c);
18     void                 edgeAddBack (int v, char c);
19     edge_iterator        edgeDel (edge_iterator it);
20     void                 id_update () { --id_; }
21     int                  id_get () const { return id_; }
22     void                 final_set (bool final) { final_ = final; }
23     bool                 isFinal () const { return final_; }
24     void                 initial_set (bool initial) { initial_ = initial; }
25     bool                 isInitial () const { return initial_; }
26     int                  haveEdge (char c) const;
27     edge_iterator        begin () { return Edges_.begin (); }
28     edge_iterator        end () { return Edges_.end (); }
29     edge_const_iterator  begin () const { return Edges_.begin (); }
30     edge_const_iterator  end () const { return Edges_.end (); }
31
32   private:
33     int                  id_;
34     bool                 final_;
35     bool                 initial_;
36     std::list<Edge>      Edges_;
37 };
38
39
40 #endif /* !NODE_HH_ */
```

6.5 node.cc - Adjacency list

```
1  #include <iostream>
2  #include "node.hh"
3  #include "automaton-non-det.hh"
4
5  /**
6   *
7   * Add a transition between this state and the state v
8   *
9   */
10 /** @{ */
11 Node::edge_iterator
12 Node::edgeAdd (Node& v, char c)
13 {
14   return Edges_.insert (Edges_.end (), Edge (c, v.id_get ()));
15 }
16
17
18 Node::edge_iterator
19 Node::edgeAdd (int v, char c)
20 {
21   return Edges_.insert(Edges_.end (), Edge (c, v));
22 }
23
24 void
25 Node::edgeAddBack (int v, char c)
26 {
27   return Edges_.push_back (Edge (c, v));
28 }
29 /** @} */
30
31
32 int
33 Node::haveEdge (char c) const
34 {
35   edge_const_iterator   it;
36
37   for (it = Edges_.begin (); it != Edges_.end (); ++it)
38     if (it->label_get () == c)
39       return it->id_get ();
40   return -1;
41 }
42
43 /**
44  *
45  * Delete a transition
46  *
47  */
```

```
48 Node::edge_iterator
49 Node::edgeDel (edge_iterator it)
50 {
51   return Edges_.erase (it);
52 }
```

6.6 edge.hh - Adjacency list

```
1  #ifndef EDGE_HH_
2  # define EDGE_HH_
3
4
5  class Edge
6  {
7    public:
8
9                          Edge (char c, int id) : label_ (c), id_ (id) {}
10     int                 id_get () const { return id_; }
11     void                id_update () { --id_; }
12     void                id_set (int id) { id_ = id; }
13     char                label_get () const { return label_; }
14     bool                operator== (const Edge& e);
15
16   private:
17     char                label_;
18     int                 id_;
19 };
20
21
22 #endif /* !EDGE_HH_ */
```

6.7 edge.cc - Adjacency list

```
1  #include "edge.hh"
2
3  /**
4   *
5   * Overloading of the == operator for Edge
6   *
7   */
8  bool
9  Edge::operator== (const Edge& e)
10 {
11   return (label_ == e.label_get ())
12     && (id_ == e.id_get ());
13 }
```

6.8 determinize.cc - Adjacency list

```
1  #include <iostream>
2  #include <map>
3  #include <set>
4  #include <vector>
5  #include "lib.hh"
6
7  using std::map;
8  using std::vector;
9  using std::set;
10
11 void
12 updateNodes (AutomatonNonDet&                    A,
13               set<set<int> >&                    newNodes,
14               map<char, set<int> >&              trans,
15               std::map<std::set<int>, int>&      nodes,
16               int                                current,
17               int                                sink_state)
18 {
19   map<char, set<int> >::iterator          it;
20   int                                     i = 0;
21
22   for (it = trans.begin (); it != trans.end (); ++it)
23     if (it->second.size ())
24     {
25       if (nodes.find (it->second) == nodes.end ())
26       {
27         nodes[it->second] = A.nodeAdd ()->id_get ();
28         newNodes.insert (it->second);
29       }
30       A.edgeAdd (current, nodes[it->second], it->first);
31     }
32     else
33       A.edgeAdd (current, sink_state, it->first);
34 }
35
36 void
37 initTrans (map<char, set<int> >& trans, const AutomatonNonDet& A)
38 {
39   for (AutomatonNonDet::alpha_const_iterator al = A.alpha_begin();
40        al != A.alpha_end(); ++al)
41     trans[*al].clear ();
42 }
43
44
45 void
46 checkState (const AutomatonNonDet&               A,
47             AutomatonNonDet&                     res,
```

```
48              std::map<std::set<int>, int>&       nodes)
49 {
50   vector<bool>                                   final (A.order (), false);
51   AutomatonNonDet::final_const_iterator          it;
52   std::map<std::set<int>, int>::iterator         it_map;
53   set<int>::iterator                             it_set;
54
55   for (it = A.final_begin (); it != A.final_end (); ++it)
56     final[*it] = true;
57
58   for (it_map = nodes.begin (); it_map != nodes.end (); ++it_map)
59     for (it_set = it_map->first.begin ();
60          it_set != it_map->first.end (); ++it_set)
61       if (final[*it_set])
62       {
63         res.finalAdd (it_map->second);
64         break;
65       }
66 }
67
68
69 AutomatonDet
70 determinize (const AutomatonNonDet& A)
71 {
72   AutomatonNonDet                          tmp (A);
73   AutomatonDet                             res;
74   set<set<int> >                           newNodes;
75   std::map<char, set<int> >                trans;
76   std::map<std::set<int>, int>             nodes;
77   set<int>::iterator                       set_it;
78   AutomatonNonDet::edge_const_iterator     it;
79   std::set<int>                            initSet;
80   int                                      sink_state;
81
82
83   sink_state = res.nodeAdd()->id_get();
84
85   for (AutomatonNonDet::alpha_const_iterator al = A.alpha_begin();
86        al != A.alpha_end(); ++al)
87     {
88       res.edgeAdd(sink_state, sink_state, *al);
89       trans[*al] = set<int>();
90     }
91
92   if (A.haveEpsilon())
93     removeEpsilon (tmp);
94
95   for (AutomatonNonDet::initial_const_iterator init = A.initial_begin ();
96        init != A.initial_end (); ++init)
```

```
97      initSet.insert(*init);
98
99    nodes[initSet] = res.nodeAdd (true)->id_get ();
100   newNodes.insert (initSet);
101   set<set<int> >::iterator node;
102
103   while (newNodes.size ())
104   {
105     node = newNodes.begin ();
106     initTrans (trans, A);
107
108     for (set_it = node->begin (); set_it != node->end (); ++set_it)
109       for (it = tmp.edge_begin (*set_it);
110            it != tmp.edge_end (*set_it); ++it) {
111
112           trans[it->label_get ()].insert (it->id_get ());
113       }
114     updateNodes (res, newNodes, trans, nodes, nodes[*node], sink_state);
115     newNodes.erase (node);
116   }
117
118   checkState (tmp, res, nodes);
119   return res;
120 }
121
122
123
124 AutomatonDet
125 determinize (const AutomatonDet& A)
126 {
127   return A;
128 }
```

6.9 minimize.cc - Adjacency list

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <set>
5  #include <map>
6  #include "lib.hh"
7
8  using std::map;
9  using std::set;
10 using std::multimap;
11 using std::vector;
12 using std::pair;
13 using std::cout;
14 using std::endl;
15
16 void
17 fill_pred (const AutomatonDet&                    A,
18             vector<map<char, pairSet > >&         pred,
19             set<int>&                             full)
20 {
21   AutomatonNonDet::node_const_iterator            it_node;
22   AutomatonNonDet::edge_const_iterator            it_edge;
23   vector<multimap<char, int> >                    tab (A.order ());
24   pair<mapIt, mapIt>                              res;
25
26   for (it_node = A.begin (); it_node != A.end (); ++it_node)
27     for (it_edge = it_node->begin (); it_edge !=  it_node->end (); ++it_edge)
28       tab[it_edge->id_get ()].insert (std::make_pair(it_edge->label_get (),
29                                       it_node->id_get ()));
30
31   for (int j = 0; j < tab.size (); ++j)
32     for (AutomatonNonDet::alpha_const_iterator al = A.alpha_begin();
33          al != A.alpha_end(); ++al)
34       {
35         res = tab[j].equal_range (*al);
36         if (res.first != tab[j].end ())
37           {
38             set<int>    predSet;
39             set<int>    otherSet;
40
41             std::insert_iterator<set<int> > otherSet_ins(otherSet,
42                                                          otherSet.begin());
43             for (multimap<char, int>::iterator it = res.first;
44                  it != res.second; ++it)
45               predSet.insert (it->second);
46             set_difference(full.begin (), full.end (),
47                            predSet.begin (), predSet.end (),otherSet_ins);
```

```
48
49              pred[j][*al] = pairSet (predSet, otherSet);
50            }
51        }
52  }
53
54
55  void
56  splitPi (set<set<int> >&        pi,
57           set<set<int> >&        T,
58           set<int>&              X,
59           set<int>&              Y,
60           bool&                  atomic)
61  {
62    set<set<int> >::iterator      it;
63
64    for (it = pi.begin (); it != pi.end ();)
65    {
66      if (it->size () == 1)
67      {
68        ++it;
69        continue;
70      }
71
72      set<int>    tmp1;
73      set<int>    tmp2;
74      std::insert_iterator<set<int> > tmp1_ins(tmp1, tmp1.begin());
75      std::insert_iterator<set<int> > tmp2_ins(tmp2, tmp2.begin());
76
77      set_intersection (it->begin (), it->end (),
78                        X.begin (), X.end (), tmp1_ins);
79      set_intersection (it->begin (), it->end (),
80                        Y.begin (), Y.end (), tmp2_ins);
81
82      if (tmp1.size () > 1 || tmp2.size () > 1)
83        atomic = false;
84
85      if (tmp1.size () && tmp2.size ())
86      {
87        pi.erase(it++);
88        pi.insert (tmp1);
89        pi.insert (tmp2);
90
91        if (tmp1.size () > tmp2.size ())
92          T.insert (tmp2);
93        else
94          T.insert (tmp1);
95      }
96      else
```

```
97        ++it;
98    }
99  }
100
101
102 void
103 createMinimized (const set<set<int> >&          pi,
104                  const AutomatonNonDet&         A,
105                  AutomatonDet&                  tmp)
106 {
107   vector<int>   equiv (A.order ());
108   int           i = 0;
109
110   tmp.createNbNodes (pi.size ());
111   for (set<set<int> >::const_iterator it_sset = pi.begin ();
112        it_sset != pi.end (); ++it_sset, ++i)
113     for (set<int>::const_iterator it_set = it_sset->begin ();
114          it_set != it_sset->end (); ++it_set)
115       equiv[*it_set] = i;
116
117   i = 0;
118   for (set<set<int> >::const_iterator it_sset = pi.begin ();
119        it_sset != pi.end (); ++it_sset, ++i)
120   {
121     set<char>                   alpha;
122
123     for (set<int>::const_iterator it_set = it_sset->begin ();
124          it_set != it_sset->end (); ++it_set)
125     {
126       if (A.node (*it_set).isFinal ())
127         tmp.finalAdd (i);
128       if (A.node (*it_set).isInitial ())
129         tmp.initialAdd (i);
130
131       for (AutomatonNonDet::edge_const_iterator it_edge = A.edge_begin (*it_set);
132            it_edge != A.edge_end (*it_set); ++it_edge)
133         if (alpha.find(it_edge->label_get ()) == alpha.end())
134         {
135           tmp.edgeAdd (i, equiv[it_edge->id_get ()],
136                        it_edge->label_get ());
137           alpha.insert(it_edge->label_get ());
138         }
139     }
140   }
141 }
142
143
144 AutomatonDet
145 minimize (const AutomatonDet& A)
```

```
146 {
147   vector<map<char, pairSet > >          pred (A.order ());
148   AutomatonDet                          tmp;
149   set<set<int> >                        T;
150   set<int>                              S;
151   set<set<int> >                        pi;
152   set<int>                              full;
153   bool                                  atomic = false;
154
155   for (int i = 0; i < A.order (); ++i)
156     full.insert (i);
157
158   fill_pred (A, pred, full);
159
160   for (AutomatonNonDet::node_const_iterator it = A.begin ();
161        it != A.end (); ++it)
162     if (!it->isFinal ())
163       S.insert (it->id_get ());
164
165   pi.insert (S);
166   S.clear ();
167
168   for (AutomatonNonDet::final_const_iterator it = A.final_begin ();
169        it != A.final_end (); ++it)
170     S.insert (*it);
171
172   T.insert (S);
173   pi.insert (S);
174
175   while (!T.empty () && !atomic)
176   {
177     S = *(T.begin ());
178     T.erase (T.begin ());
179     atomic = true;
180
181     for (AutomatonNonDet::alpha_const_iterator al = A.alpha_begin();
182          al != A.alpha_end(); ++al)
183       {
184         set<int>        X;
185         set<int>        Y;
186         set<int>        tmp;
187         std::insert_iterator<set<int> > X_ins(X, X.begin());
188         std::insert_iterator<set<int> > Y_ins(tmp, tmp.begin());
189
190         for (set<int>::iterator it = S.begin (); it != S.end (); ++it)
191           {
192             pairSet& it2 = pred[*it][*al];
193
194             set_union (X.begin (), X.end (),
```

```
195                         it2.first.begin (),
196                         it2.first.end (),
197                         X_ins);
198
199             if (!Y.size ())
200               Y = it2.second;
201             else
202               {
203                 set_intersection (Y.begin (), Y.end (),
204                                   it2.second.begin (),
205                                   it2.second.end (),
206                                   Y_ins);
207                 Y = tmp;
208               }
209           }
210
211         splitPi (pi, T, X, Y, atomic);
212       }
213   }
214
215   pred.clear ();
216   createMinimized (pi, A, tmp);
217
218   return tmp;
219 }
220
221 AutomatonDet
222 minimize (const AutomatonNonDet& A)
223 {
224   return minimize (determinize (A));
225 }
```

6.10 automaton-non-det.hh - ABMI

```
1  #ifndef AUTOMATONNONDET_HH_
2  # define AUTOMATONNONDET_HH_
3
4  # include "boost-containers.hh"
5  # include "node.hh"
6  # include <vector>
7  # include <set>
8
9  class Node;
10
11 class AutomatonNonDet
12 {
13   public:
14
15     //Iterator Typedef
16     typedef std::vector<Node>::iterator          node_iterator;
17     typedef std::vector<Node>::const_iterator    node_const_iterator;
18     typedef edgeList::iterator                   edge_iterator;
19     typedef edgeList::const_iterator             edge_const_iterator;
20     typedef states_set::iterator                 initial_iterator;
21     typedef states_set::iterator                 final_iterator;
22     typedef states_set::const_iterator           initial_const_iterator;
23     typedef states_set::const_iterator           final_const_iterator;
24     typedef alphabet_t::iterator                 alpha_iterator;
25     typedef alphabet_t::const_iterator           alpha_const_iterator;
26     typedef std::pair<index_iterator<edgeList, lbl>::type,
27                       index_iterator<edgeList, lbl>::type> transitionRange;
28
29     //Constructors - destructor
30                                  AutomatonNonDet () {}
31     virtual                      ~AutomatonNonDet () {}
32                                  AutomatonNonDet (int nb);
33
34     //Interface
35     virtual void                 createNbNodes (int);
36     virtual node_iterator        nodeAdd (bool initial = false,
37                                           bool final = false);
38     virtual node_iterator        nodeDel (int i);
39     virtual node_iterator        nodeDel (node_iterator);
40     virtual node_iterator        nodeDelWoClean (node_iterator);
41     virtual node_iterator        nodeDelWoClean (int i);
42     virtual void                 edgeAdd (Node& u, Node& v, char c);
43     virtual void                 edgeAdd (int u, int v, char c);
44     virtual edge_iterator        edgeDel (Node&, edge_iterator&);
45     virtual edge_iterator        edgeDel (int, edge_iterator&);
46     virtual bool                 initialAdd (int i);
47     virtual initial_iterator     initialDel (initial_iterator);
```

```
48      virtual void                initialClear ();
49      virtual void                finalAdd (int i);
50      virtual final_iterator      finalDel (final_iterator);
51      virtual void                finalClear ();
52      virtual void                print () const;
53      virtual void                printSorted () const;
54      virtual void                setFinalState (const AutomatonNonDet &, int = 0);
55      virtual void                setInitialState (const AutomatonNonDet &A,
56                                                   int offset);
57      virtual transitionRange     getTransFromSrcLbl(int src, char c) const;
58
59      virtual node_iterator       nodeToIt (Node& it);
60      virtual const Node&         node (int) const;
61      virtual bool                isFinal(int i) const;
62      virtual bool                isInitial(int i) const;
63      virtual int                 alpha_size() const;
64
65      virtual unsigned            order () const;
66      virtual int                 haveEpsilon() const;
67      virtual unsigned            initial_nb () const;
68
69      //Iterator getter
70      virtual edge_const_iterator         edge_begin (int i) const;
71      virtual edge_iterator               edge_begin (int i);
72      virtual edge_const_iterator         edge_end (int i) const;
73      virtual edge_iterator               edge_end (int i);
74      virtual node_iterator               begin ();
75      virtual node_iterator               end ();
76      virtual node_const_iterator         begin () const;
77      virtual node_const_iterator         end () const;
78      virtual initial_const_iterator      initial_begin () const;
79      virtual initial_iterator            initial_begin ();
80      virtual final_iterator              final_begin ();
81      virtual final_const_iterator        final_begin () const;
82      virtual initial_const_iterator      initial_end () const;
83      virtual initial_iterator            initial_end ();
84      virtual final_iterator              final_end ();
85      virtual final_const_iterator        final_end () const;
86      virtual alpha_iterator              alpha_begin();
87      virtual alpha_const_iterator        alpha_begin() const;
88      virtual alpha_iterator              alpha_end();
89      virtual alpha_const_iterator        alpha_end() const;
90
91      AutomatonNonDet&                    operator= (const AutomatonNonDet&);
92
93    protected:
94      void                    statesUpdate (states_set& l, node_iterator it);
95      states_set              final_;
96      std::vector<Node>       nodes_;
```

```
97      states_set          initial_;
98      alphabet_t          alpha_;
99      int                 haveEpsilon_;
100 };
101
102
103 AutomatonNonDet         operator+ (const AutomatonNonDet&,
104                                    const AutomatonNonDet&);
105 AutomatonNonDet         operator* (const AutomatonNonDet&,
106                                    const AutomatonNonDet&);
107 AutomatonNonDet         operator* (const AutomatonNonDet&);
108
109 # include "automaton-non-det.hxx"
110
111 #endif /* !AUTOMATONNONDET_HH_ */
```

6.11 boost-containers.hh - ABMI

```
1  #ifndef BOOST_CONTAINERS_HH_
2  # define BOOST_CONTAINERS_HH_
3
4  #include <boost/multi_index_container.hpp>
5  #include <boost/multi_index/member.hpp>
6  #include <boost/multi_index/ordered_index.hpp>
7  #include <boost/multi_index/hashed_index.hpp>
8  #include <boost/functional/hash/hash.hpp>
9  #include <boost/multi_index/sequenced_index.hpp>
10 #include <boost/multi_index/composite_key.hpp>
11 #include <iostream>
12
13 using namespace ::boost;
14 using boost::multi_index_container;
15 using namespace boost::multi_index;
16
17
18 //[BOOST] int_set declaration
19 typedef multi_index_container
20 <
21   int,
22   indexed_by
23   <
24         ordered_unique
25         <
26                 identity<int>
27         >
28   >
29 > int_set;
30
31 struct boost_hash;
32 struct boost_equal;
33
34 //[BOOST] int_setset declaration
35 typedef multi_index_container
36 <
37   int_set,
38   indexed_by
39   <
40         hashed_unique
41         <
42                 identity<int_set>,
43                 boost_hash,
44                 boost_equal
45         >
46   >
47 > int_setset;
```

```
48
49  struct s_map
50  {
51      int_set     key;
52      int         value;
53
54      s_map (const int_set& key_, int value_) : key(key_), value(value_) {}
55      s_map () {}
56  };
57
58  struct s_enstonb
59  {
60    s_enstonb(const int_set& f, unsigned int i)
61      : first(f),
62        second(i)
63    {}
64
65    int_set       first;
66    unsigned int  second;
67  };
68
69
70  struct boost_hash
71  {
72    inline
73    std::size_t operator()(const s_enstonb& v) const
74    {
75      return hash_range(v.first.begin(), v.first.end());
76    }
77    inline
78    std::size_t operator()(const s_map& v) const
79    {
80      return hash_range(v.key.begin(), v.key.end());
81    }
82    inline
83    std::size_t operator()(const int_set& v) const
84    {
85      return hash_range(v.begin(), v.end());
86    }
87  };
88
89  struct boost_equal
90  {
91    inline
92    std::size_t operator()(const s_enstonb& x1, const s_enstonb& x2) const
93    {
94      return (x1.first == x2.first) && (x1.second == x2.second);
95    }
96    inline
```

```
97    std::size_t operator()(const s_map& x1, const s_map& x2) const
98    {
99      return (x1.key == x2.key) && (x1.value == x2.value);
100   }
101   inline
102   std::size_t operator()(const int_set& x1, const int_set& x2) const
103   {
104     int_set::iterator it1, it2;
105
106     for (it1 = x1.begin (), it2 = x2.begin ();
107          it1 != x1.end () && it2 != x2.end () && *it2 == *it1; ++it1, ++it2)
108       ;
109     return (it1 == x1.end ()) && (it2 == x2.end ());
110   }
111 };
112
113 //[BOOST] map_set declaration
114 typedef multi_index_container
115 <
116   s_map,
117   indexed_by
118   <
119         hashed_non_unique
120         <
121                 BOOST_MULTI_INDEX_MEMBER(s_map, int_set, key),
122                 boost_hash,
123                 boost_equal
124         >
125   >
126 > map_set;
127
128
129 typedef struct s_pair
130 {
131   int_set       first;
132   int_set       second;
133
134   s_pair (const int_set& f, const int_set& s) : first(f), second(s) {}
135 } pairSet;
136
137 struct s_pairmap
138 {
139   char          first;
140   s_pair        second;
141
142   s_pairmap (const char c, const s_pair& p) : first(c), second(p) {}
143 };
144
145 //[BOOST] pairSet declaration
```

```
146 typedef multi_index_container
147 <
148   s_pairmap,
149   indexed_by
150   <
151         hashed_unique
152         <
153                 BOOST_MULTI_INDEX_MEMBER(s_pairmap, char, first)
154         >
155   >
156 > pairMap;
157
158 typedef std::pair<char, int>            edgePair;
159
160 //[BOOST] pairSet declaration
161 typedef multi_index_container
162 <
163   edgePair,
164   indexed_by
165   <
166         hashed_non_unique
167         <
168                 BOOST_MULTI_INDEX_MEMBER(edgePair, char, first)
169         >
170   >
171 > edgeMap;
172
173 typedef multi_index_container
174 <
175   int_set,
176   indexed_by<ordered_unique<identity<int_set> > >
177 > list_boost;
178
179 typedef edgeMap::iterator               mapIt;
180
181
182 typedef multi_index_container
183 <
184   s_enstonb,
185   indexed_by
186   <
187         hashed_unique
188         <
189                 BOOST_MULTI_INDEX_MEMBER(s_enstonb, int_set, first),
190                 boost_hash,
191                 boost_equal
192         >
193   >
194 > ensToNb_map;
```

```
195
196
197 typedef multi_index_container
198 <
199   int_set,
200   indexed_by
201   <
202         hashed_unique
203         <
204                 identity<int_set>,
205                 boost_hash,
206                 boost_equal
207         >
208   >
209 > int_setset;
210
211
212 typedef multi_index_container
213 <
214   int_set,
215   indexed_by
216   <
217         ordered_non_unique
218         <
219                 identity<int_set>,
220                 boost_equal
221         >,
222         sequenced<>
223   >
224 > int_setlist;
225
226
227 struct s_transition
228 {
229   s_transition (unsigned int src_, unsigned int dst_, char lbl_)
230     : src(src_),
231       dst(dst_),
232       lbl(lbl_)
233   {}
234
235   unsigned int  src;
236   unsigned int  dst;
237   char          lbl;
238 };
239
240 struct src {};
241 struct dst {};
242 struct lbl {};
243
```

```
244 typedef multi_index_container
245 <
246   s_transition,
247   indexed_by
248   <
249         hashed_non_unique
250         <
251                 composite_key
252                 <
253                         s_transition,
254                         BOOST_MULTI_INDEX_MEMBER(s_transition,
255                                                  unsigned int,
256                                                  dst),
257                         BOOST_MULTI_INDEX_MEMBER(s_transition,
258                                                  char,
259                                                  lbl)
260                 >
261         >
262   >
263 > graph_t;
264
265 typedef std::pair<graph_t::iterator, graph_t::iterator> predPair;
266
267 typedef multi_index_container
268 <
269   char,
270   indexed_by
271   <
272         ordered_unique<identity<char> >
273   >
274 > alphabet_t;
275
276
277 typedef multi_index_container
278 <
279   int,
280   indexed_by<hashed_unique<identity<int> > >
281 > states_set;
282
283 struct s_ens
284 {
285   s_ens(int n_, int id_)
286     : n(n_),
287       id(id_)
288   {}
289
290   int   n;
291   int   id;
292 };
```

```
293
294 struct n{};
295 struct id{};
296
297 typedef multi_index_container
298 <
299   s_ens,
300   indexed_by<hashed_non_unique<tag<n>, BOOST_MULTI_INDEX_MEMBER(s_ens, int, n)>,
301              hashed_non_unique<tag<id>, BOOST_MULTI_INDEX_MEMBER(s_ens, int, id)>
302 > EquivSet;
303
304 #endif //! BOOST_CONTAINERS_HH_
```

6.12 automaton-non-det.hxx - ABMI

```
1  /**
2   *
3   * Return a state iterator on the beginning
4   * of the state list
5   *
6   */
7  /** @{ */
8  inline
9  AutomatonNonDet::node_iterator
10 AutomatonNonDet::begin ()
11 {
12   return nodes_.begin ();
13 }
14
15
16
17 inline
18 AutomatonNonDet::node_const_iterator
19 AutomatonNonDet::begin () const
20 {
21   return nodes_.begin ();
22 }
23 /** @} */
24
25
26 /**
27  *
28  * Return a state iterator on the end
29  * of the state list
30  *
31  */
32 /** @{ */
33 inline
34 AutomatonNonDet::node_iterator
35 AutomatonNonDet::end ()
36 {
37   return nodes_.end ();
38 }
39
40 inline
41 AutomatonNonDet::node_const_iterator
42 AutomatonNonDet::end () const
43 {
44   return nodes_.end ();
45 }
46 /** @} */
47
```

```
48
49 /**
50  *
51  * Return an iterator on the beginning
52  * of the initial state list
53  *
54  */
55 /** @{ */
56 inline
57 AutomatonNonDet::initial_const_iterator
58 AutomatonNonDet::initial_begin () const
59 {
60   return initial_.begin ();
61 }
62
63
64 inline
65 AutomatonNonDet::initial_iterator
66 AutomatonNonDet::initial_begin ()
67 {
68   return initial_.begin ();
69 }
70 /** @} */
71
72
73 /**
74  *
75  * Return an iterator on the beginning
76  * of the final state list
77  *
78  */
79 /** @{ */
80 inline
81 AutomatonNonDet::final_iterator
82 AutomatonNonDet::final_begin ()
83 {
84   return final_.begin ();
85 }
86
87
88 inline
89 AutomatonNonDet::final_const_iterator
90 AutomatonNonDet::final_begin () const
91 {
92   return final_.begin ();
93 }
94 /** @} */
95
96
```

```
97  /**
98   *
99   * Return an iterator on the end
100  * of the initial state list
101  *
102  */
103 /** @{ */
104 inline
105 AutomatonNonDet::initial_const_iterator
106 AutomatonNonDet::initial_end () const
107 {
108   return initial_.end ();
109 }
110
111
112 inline
113 AutomatonNonDet::initial_iterator
114 AutomatonNonDet::initial_end ()
115 {
116   return initial_.end ();
117 }
118 /** @} */
119
120
121 /**
122  *
123  * Return an iterator on the end
124  * of the final state list
125  *
126  */
127 /** @{ */
128 inline
129 AutomatonNonDet::final_iterator
130 AutomatonNonDet::final_end ()
131 {
132   return final_.end ();
133 }
134
135
136 inline
137 AutomatonNonDet::final_const_iterator
138 AutomatonNonDet::final_end () const
139 {
140   return final_.end ();
141 }
142 /** @} */
143
144
145 /**
```

```
146   *
147   * Return the number of state
148   *
149   */
150  inline
151  unsigned
152  AutomatonNonDet::order () const
153  {
154    return nodes_.size ();
155  }
156
157
158  /**
159   *
160   * Return an iterator on the beginning
161   * of the ith state transition list
162   *
163   */
164  /** @{ */
165  inline
166  AutomatonNonDet::edge_const_iterator
167  AutomatonNonDet::edge_begin (int i) const
168  {
169    return nodes_[i].begin ();
170  }
171
172
173
174  inline
175  AutomatonNonDet::edge_iterator
176  AutomatonNonDet::edge_begin (int i)
177  {
178    return nodes_[i].begin ();
179  }
180  /** @} */
181
182
183  /**
184   *
185   * Return an iterator on the end
186   * of the ith state transition list
187   *
188   */
189  /** @{ */
190  inline
191  AutomatonNonDet::edge_const_iterator
192  AutomatonNonDet::edge_end (int i) const
193  {
194    return nodes_[i].end ();
```

```
195 }
196
197
198
199 inline
200 AutomatonNonDet::edge_iterator
201 AutomatonNonDet::edge_end (int i)
202 {
203   return nodes_[i].end ();
204 }
205 /** @} */
206
207
208 /**
209  *
210  * Return the number of initial state
211  *
212  */
213 inline
214 unsigned
215 AutomatonNonDet::initial_nb () const
216 {
217   return initial_.size ();
218 }
219
220
221 inline
222 void
223 AutomatonNonDet::initialClear ()
224 {
225   initial_iterator      it;
226
227   for (it = initial_.begin (); it != initial_.end ();)
228     it = initialDel (it);
229   initial_.clear ();
230 }
231
232
233 inline
234 void
235 AutomatonNonDet::finalClear ()
236 {
237   final_iterator        it;
238
239   for (it = final_.begin (); it != final_.end ();)
240     it = finalDel (it);
241   final_.clear ();
242 }
243
```

```
244
245
246 inline
247 AutomatonNonDet::alpha_iterator
248 AutomatonNonDet::alpha_begin()
249 {
250   return alpha_.begin();
251 }
252
253
254 inline
255 AutomatonNonDet::alpha_const_iterator
256 AutomatonNonDet::alpha_begin() const
257 {
258   return alpha_.begin();
259 }
260
261
262 inline
263 AutomatonNonDet::alpha_iterator
264 AutomatonNonDet::alpha_end()
265 {
266   return alpha_.end();
267 }
268
269
270 inline
271 AutomatonNonDet::alpha_const_iterator
272 AutomatonNonDet::alpha_end() const
273 {
274   return alpha_.end();
275 }
276
277
278 inline
279 int
280 AutomatonNonDet::haveEpsilon() const
281 {
282   return haveEpsilon_;
283 }
284
285
286
287 inline
288 AutomatonNonDet::transitionRange
289 AutomatonNonDet::getTransFromSrcLbl(int src, char c) const
290 {
291   return nodes_[src].getTransFromLbl(c);
292 }
```

```
293
294
295 inline
296 bool
297 AutomatonNonDet::isFinal(int i) const
298 {
299   return nodes_[i].isFinal();
300 }
301
302
303 inline
304 bool
305 AutomatonNonDet::isInitial(int i) const
306 {
307   return nodes_[i].isInitial();
308 }
309
310
311 inline
312 int
313 AutomatonNonDet::alpha_size() const
314 {
315   return alpha_.size();
316 }
```

6.13 automaton-non-det.cc - ABMI

```
1  #include <vector>
2  #include <iostream>
3  #include <map>
4  #include "automaton-non-det.hh"
5
6
7  /**
8   * @brief Other constructor
9   *
10  * Initialize an AutomatonNonDet and create
11  * nb states.
12  *
13  */
14 AutomatonNonDet::AutomatonNonDet (int nb)
15 {
16   createNbNodes (nb);
17 }
18
19
20 /**
21  * @brief Create nb states in the automaton
22  *
23  * Create nb states in the automaton.
24  */
25 void
26 AutomatonNonDet::createNbNodes (int nb)
27 {
28   for (int i = 0; i < nb; ++i)
29     nodeAdd ();
30 }
31
32
33 /**
34  * @brief Add a state to the automaton.
35  *
36  * Add a state to the automaton.
37  * You can specify if it is final and/or
38  * an initial one.
39  *
40  */
41 AutomatonNonDet::node_iterator
42 AutomatonNonDet::nodeAdd (bool initial, bool final)
43 {
44   if (initial)
45     initial_.insert (nodes_.size ());
46   if (final)
47     final_.insert (nodes_.size ());
```

```
48    return nodes_.insert (nodes_.end (), Node (nodes_.size (), initial, final));
49  }
50
51
52  /**
53   *
54   * Remove a state from the automaton.
55   *
56   */
57  /** @{ */
58  AutomatonNonDet::node_iterator
59  AutomatonNonDet::nodeDel (int i)
60  {
61    nodeDel (nodes_.begin () + i);
62  }
63
64  AutomatonNonDet::node_iterator
65  AutomatonNonDet::nodeDel (node_iterator it)
66  {
67    node_iterator node;
68    edge_iterator edge;
69
70    statesUpdate (initial_, it);
71    statesUpdate (final_, it);
72    for (node = nodes_.begin (); node != nodes_.end (); ++node)
73    {
74      if (node->id_get () > it->id_get ())
75        node->id_update ();
76      for (edge = node->begin (); edge != node->end (); )
77        if (edge->dst == it->id_get ())
78          edge = node->edgeDel (edge);
79        else
80        {
81          if (edge->dst > it->id_get ())
82            {
83              node->edgeAdd(edge->dst - 1, edge->lbl);
84              edge = node->edgeDel(edge);
85            }
86          ++edge;
87        }
88    }
89    return nodes_.erase (it);
90  }
91  /** @} */
92
93
94  /**
95   *
96   * Remove a state from the automaton without deleting transitions
```

```
97   * pointing to it from other states.
98   *
99   */
100 /** @{ */
101 AutomatonNonDet::node_iterator
102 AutomatonNonDet::nodeDelWoClean (node_iterator it)
103 {
104   return nodes_.erase (it);
105 }
106
107 AutomatonNonDet::node_iterator
108 AutomatonNonDet::nodeDelWoClean (int i)
109 {
110   return nodes_.erase (nodes_.begin () + i);
111 }
112 /** @} */
113
114 /**
115  *
116  * Add a transition between 2 states
117  *
118  */
119 /** @{ */
120 void
121 AutomatonNonDet::edgeAdd (Node& u,
122                           Node& v,
123                           char  c)
124 {
125   if (c != 1)
126     alpha_.insert(c);
127   else
128     ++haveEpsilon_;
129   u.edgeAdd (v, c);
130 }
131
132 void
133 AutomatonNonDet::edgeAdd (int   u,
134                           int   v,
135                           char  c)
136 {
137   if (c != 1)
138     alpha_.insert(c);
139   else
140     ++haveEpsilon_;
141   nodes_[u].edgeAdd (v, c);
142 }
143 /** @} */
144
145
```

```
146 /**
147  *
148  * Delete a transition between 2 states
149  *
150  */
151 /** @{ */
152 AutomatonNonDet::edge_iterator
153 AutomatonNonDet::edgeDel (Node&                 itnode,
154                           edge_iterator&        itedge)
155 {
156   return itnode.edgeDel (itedge);
157 }
158
159 AutomatonNonDet::edge_iterator
160 AutomatonNonDet::edgeDel (int                   itnode,
161                           edge_iterator&        itedge)
162 {
163   return nodes_[itnode].edgeDel (itedge);
164 }
165 /** @} */
166
167
168 /**
169  *
170  * Return a state iterator from its reference.
171  *
172  */
173 AutomatonNonDet::node_iterator
174 AutomatonNonDet::nodeToIt (Node& it)
175 {
176   node_iterator it_node = nodes_.begin ();
177
178   for (; it_node != nodes_.end () && &(*it_node) != &it; ++it_node)
179     ;
180   return it_node;
181 }
182
183
184 /**
185  *
186  * Display the automaton on stdout
187  *
188  */
189 void
190 AutomatonNonDet::print () const
191 {
192   node_const_iterator   node;
193   edge_const_iterator   edge;
194   final_const_iterator  it;
```

```
195
196   for (node = nodes_.begin (); node != nodes_.end (); ++node)
197   {
198     std::cout << node->id_get ();
199     if (node->isInitial ())
200       std::cout << " (I)";
201     else if (node->isFinal ())
202       std::cout << " (F)";
203     else
204       std::cout << "    ";
205     std::cout << " : " ;
206     for (edge = node->begin (); edge != node->end (); ++edge)
207       std::cout << " " << "(" << edge->dst << "," << edge->lbl << ")";
208     std::cout << std::endl;
209   }
210   std::cout << "Final states :";
211   for (it = final_begin (); it != final_end (); ++it)
212     std::cout << " " << *it;
213   std::cout << std::endl << "-----" << std::endl;
214 }
215
216
217 /**
218  *
219  * Display the automaton on stdout
220  * Transitions are sorted.
221  *
222  */
223 void
224 AutomatonNonDet::printSorted () const
225 {
226   node_const_iterator   node;
227   edge_const_iterator   edge;
228   final_const_iterator  it;
229   std::multimap<int, char>      sort;
230
231   for (node = nodes_.begin (); node != nodes_.end (); ++node)
232   {
233     std::cout << node->id_get ();
234     if (node->isInitial ())
235       std::cout << " (I)";
236     else if (node->isFinal ())
237       std::cout << " (F)";
238     else
239       std::cout << "    ";
240     std::cout << " : " ;
241     sort.clear ();
242     for (edge = node->begin (); edge != node->end (); ++edge)
243       sort.insert (std::make_pair (edge->dst, edge->lbl));
```

```
244     for (std::multimap<int, char>::iterator it = sort.begin ();
245          it != sort.end (); ++it) {
246
247         std::cout << "␣" << "(" << it->first << "," << it->second << ")";
248     }
249     std::cout << std::endl;
250   }
251   std::cout << "Final␣states␣:␣{";
252   it = final_begin ();
253   std::cout << *it;
254   for (++it; it != final_end (); ++it)
255     std::cout << ";␣" << *it;
256   std::cout << "}" << std::endl << "-----" << std::endl;
257 }
258
259
260
261 /**
262  *
263  * Set a state as an initial state
264  *
265  */
266 bool
267 AutomatonNonDet::initialAdd (int i)
268 {
269   nodes_[i].initial_set (true);
270   initial_.insert (i);
271   return true;
272 }
273
274
275 /**
276  *
277  * Delete an initial state
278  *
279  */
280 AutomatonNonDet::initial_iterator
281 AutomatonNonDet::initialDel (initial_iterator it)
282 {
283   nodes_[*it].initial_set (false);
284   return initial_.erase (it);
285 }
286
287
288 /**
289  *
290  * Delete a final state
291  *
292  */
```

```
293 AutomatonNonDet::final_iterator
294 AutomatonNonDet::finalDel (final_iterator it)
295 {
296   nodes_[*it].final_set (false);
297   return final_.erase (it);
298 }
299
300
301
302 /**
303  *
304  * Set a state as a final state
305  *
306  */
307 void
308 AutomatonNonDet::finalAdd (int i)
309 {
310   nodes_[i].final_set (true);
311   final_.insert (i);
312 }
313
314 /**
315  *
316  * Return a reference upon a state from its number
317  *
318  */
319 const Node&
320 AutomatonNonDet::node (int i) const
321 {
322   return nodes_[i];
323 }
324
325
326
327 /**
328  *
329  * Update initial or final state id when deleting a state
330  *
331  */
332 void
333 AutomatonNonDet::statesUpdate (states_set& l, node_iterator it)
334 {
335   initial_iterator      init;
336
337   for (init = l.begin (); init != l.end ();)
338     if (*init == it->id_get ())
339       l.erase (init++);
340     else
341     {
```

```
342         if (*init > it->id_get ())
343         {
344           l.insert (*init - 1);
345           l.erase (init);
346         }
347         ++init;
348       }
349   }
350
351
352   /**
353    *
354    * Replace final states by the automaton's ones
355    *
356    */
357   void
358   AutomatonNonDet::setFinalState (const AutomatonNonDet &A, int offset)
359   {
360     final_const_iterator  it;
361
362     finalClear ();
363     for (it = A.final_begin (); it != A.final_end (); ++it)
364       finalAdd (*it + offset);
365   }
366
367
368
369   /**
370    *
371    * Replace initial states by the automaton's ones
372    *
373    */
374   void
375   AutomatonNonDet::setInitialState (const AutomatonNonDet &A, int offset)
376   {
377     initial_const_iterator        it;
378
379     initialClear ();
380     for (it = A.initial_begin (); it != A.initial_end (); ++it)
381       initialAdd (*it + offset);
382   }
383
384
385   /**
386    *
387    * Assign a new automaton to this.
388    *
389    */
390   AutomatonNonDet&
```

```
391 AutomatonNonDet::operator= (const AutomatonNonDet& A)
392 {
393   initial_ = A.initial_;
394   final_ = A.final_;
395   nodes_ = A.nodes_;
396 }
397
398
399 void
400 updateInitial (const AutomatonNonDet& A, AutomatonNonDet& tmp, int initial)
401 {
402   AutomatonNonDet::initial_const_iterator        it_init;
403
404   for (it_init = A.initial_begin (); it_init != A.initial_end (); ++it_init)
405     tmp.edgeAdd (initial, *it_init, 1);
406   tmp.initialClear ();
407   tmp.initialAdd (initial);
408 }
409
410 void
411 updateFinal (const AutomatonNonDet& A, AutomatonNonDet& tmp, int final)
412 {
413   AutomatonNonDet::initial_const_iterator        it_init;
414
415   for (it_init = A.final_begin (); it_init != A.final_end (); ++it_init)
416     tmp.edgeAdd (*it_init, final, 1);
417   tmp.finalClear ();
418   tmp.finalAdd (final);
419 }
420
421
422 /**
423  *
424  * Make an union between 2 automatons
425  *
426  */
427 AutomatonNonDet
428 operator+ (const AutomatonNonDet& A, const AutomatonNonDet& B)
429 {
430   AutomatonNonDet                                tmp (A);
431   AutomatonNonDet::node_const_iterator           it_node;
432   AutomatonNonDet::edge_const_iterator           it_edge;
433   AutomatonNonDet::initial_const_iterator        it_init;
434
435   int initial = tmp.nodeAdd ()->id_get ();
436   int final = tmp.nodeAdd ()->id_get ();
437
438   updateInitial (A, tmp, initial);
439   updateFinal (A, tmp, final);
```

```
440
441   int base = tmp.order ();
442
443   for (int i = 0; i < B.order (); ++i)
444     tmp.nodeAdd ();
445
446   for (it_node = B.begin (); it_node != B.end (); ++it_node)
447     for (it_edge = it_node->begin (); it_edge != it_node->end (); ++it_edge)
448       tmp.edgeAdd (base + it_node->id_get (),
449                    base + it_edge->dst,
450                    it_edge->lbl);
451
452   for (it_init = B.initial_begin (); it_init != B.initial_end (); ++it_init)
453     tmp.edgeAdd (initial, *it_init + base, 1);
454
455   for (it_init = B.final_begin (); it_init != B.final_end (); ++it_init)
456     tmp.edgeAdd (*it_init +  base, final, 1);
457   return tmp;
458 }
459
460
461 /**
462  *
463  * Make an intersection between 2 automatons
464  *
465  */
466 AutomatonNonDet
467 operator* (const AutomatonNonDet& A, const AutomatonNonDet& B)
468 {
469   if (!B.order ())
470     return AutomatonNonDet (A);
471   if (!A.order ())
472     return AutomatonNonDet (B);
473
474   AutomatonNonDet::node_const_iterator          it_node;
475   AutomatonNonDet::edge_const_iterator          it_edge;
476   AutomatonNonDet::initial_const_iterator       it_init;
477   AutomatonNonDet                               tmp (A);
478
479   int base = A.order ();
480   for (int i = 0; i < B.order () + 1; ++i)
481     tmp.nodeAdd ();
482   for (it_node = B.begin (); it_node != B.end (); ++it_node)
483     for (it_edge = it_node->begin (); it_edge != it_node->end (); ++it_edge)
484       tmp.edgeAdd (base + it_node->id_get (),
485                    base + it_edge->dst,
486                    it_edge->lbl);
487   tmp.edgeAdd (*(A.final_begin ()), tmp.order () - 1, 1);
488   tmp.edgeAdd (tmp.order () - 1, *(B.initial_begin ()) + base, 1);
```

```
489   tmp.setFinalState (B, base);
490   return tmp;
491 }
492
493
494 /**
495  *
496  * Add transitions to apply a star
497  *
498  */
499 AutomatonNonDet
500 operator* (const AutomatonNonDet& A)
501 {
502   AutomatonNonDet         tmp (A);
503   int initial = tmp.nodeAdd ()->id_get ();
504   int final = tmp.nodeAdd ()->id_get ();
505
506   tmp.edgeAdd (initial, *(tmp.initial_begin ()), 1);
507   tmp.edgeAdd (*(tmp.final_begin ()), final, 1);
508   tmp.edgeAdd (initial, final, 1);
509   tmp.edgeAdd (*(tmp.final_begin ()), *(tmp.initial_begin ()), 1);
510   tmp.finalClear ();
511   tmp.finalAdd (final);
512   tmp.initialClear ();
513   tmp.initialAdd (initial);
514   return tmp;
515 }
```

6.14 node.hh - ABMI

```
1  #ifndef NODE_HH_
2  # define NODE_HH_
3
4  # include <list>
5  # include "edge.hh"
6
7  class Node
8  {
9    public:
10
11     typedef edgeList::iterator                                    edge_iterator;
12     typedef edgeList::const_iterator                              edge_const_iterator;
13     typedef std::pair<index_iterator<edgeList, lbl>::type,
14                       index_iterator<edgeList, lbl>::type>        transitionRange;
15
16                       Node (int id, bool initial = false, bool final = false)
17                         : id_ (id), final_ (final), initial_ (initial) {}
18     edge_iterator     edgeAdd (Node& v, char c);
19     edge_iterator     edgeAdd (int v, char c);
20     edge_iterator     edgeDel (edge_iterator it);
21     void              id_update () { --id_; }
22
23     inline
24     int               id_get () const { return id_; }
25
26     void              final_set (bool final) { final_ = final; }
27     bool              isFinal () const { return final_; }
28     void              initial_set (bool initial) { initial_ = initial; }
29     bool              isInitial () const { return initial_; }
30     int               haveEdge (char c) const;
31     transitionRange   getTransFromLbl(char c) const;
32     bool              haveTransition(int dst, char c) const;
33     edge_iterator     begin () { return Edges_.begin (); }
34     edge_iterator     end () { return Edges_.end (); }
35     edge_const_iterator begin () const { return Edges_.begin (); }
36     edge_const_iterator end () const { return Edges_.end (); }
37
38   private:
39     int               id_;
40     bool              final_;
41     bool              initial_;
42     edgeList          Edges_;
43 };
44
45
46 #endif /* !NODE_HH_ */
```

6.15 node.cc - ABMI

```
1  #include <iostream>
2  #include "automaton-non-det.hh"
3  #include "node.hh"
4
5  /**
6   *
7   * Add a transition between this state and the state v
8   *
9   */
10 /** @{ */
11 Node::edge_iterator
12 Node::edgeAdd (Node& v, char c)
13 {
14   return Edges_.insert (s_edge (v.id_get (), c)).first;
15 }
16
17
18 Node::edge_iterator
19 Node::edgeAdd (int v, char c)
20 {
21   return Edges_.insert(s_edge (v, c)).first;
22 }
23 /** @} */
24
25
26 int
27 Node::haveEdge (char c) const
28 {
29   std::pair<index_iterator<edgeList,lbl>::type,
30     index_iterator<edgeList,lbl>::type> it =  Edges_.get<lbl>().equal_range(c);
31   if (it.first != it.second)
32     return it.first->dst;
33   return -1;
34 }
35
36 /**
37  *
38  * Delete a transition
39  *
40  */
41 Node::edge_iterator
42 Node::edgeDel (edge_iterator it)
43 {
44   return Edges_.erase (it);
45 }
46
47
```

```
48 Node::transitionRange
49 Node::getTransFromLbl(char c) const
50 {
51   return Edges_.get<lbl>().equal_range(c);
52 }
53
54
55 bool
56 Node::haveTransition(int dst, char c) const
57 {
58   transitionRange        res = Edges_.get<lbl>().equal_range(c);
59
60   for (index_iterator<edgeList, lbl>::type it = res.first;
61        it != res.second; ++it)
62     if (it->dst == dst)
63       return true;
64   return false;
65 }
```

6.16 edge.hh - ABMI

```
1  #ifndef EDGE_HH_
2  # define EDGE_HH_
3
4  #include <boost/multi_index_container.hpp>
5  #include <boost/multi_index/member.hpp>
6  #include <boost/multi_index/ordered_index.hpp>
7  #include <boost/multi_index/hashed_index.hpp>
8  #include <boost/functional/hash/hash.hpp>
9  #include <boost/multi_index/sequenced_index.hpp>
10 #include "boost-containers.hh"
11
12 struct s_edge
13 {
14   s_edge (unsigned int dst_, char lbl_)
15     : dst(dst_),
16       lbl(lbl_)
17   {}
18
19   bool operator==(const s_edge& s) const
20   {
21     return dst == s.dst && lbl == s.lbl;
22   }
23
24   void  updateDst() { dst -= 1; }
25   unsigned int  dst;
26   char          lbl;
27 };
28
29 struct dst;
30 struct lbl;
31
32 typedef multi_index_container
33 <
34   s_edge,
35   indexed_by
36   <
37         hashed_non_unique<tag<dst>, BOOST_MULTI_INDEX_MEMBER(s_edge, unsigned int,
38         hashed_non_unique<tag<lbl>, BOOST_MULTI_INDEX_MEMBER(s_edge, char, lbl)>
39   >
40 > edgeList;
41
42
43
44 #endif /* !EDGE_HH_ */
```

6.17 determinize.cc - ABMI

```
1  #include <iostream>
2  #include <vector>
3  #include "lib.hh"
4  #include "boost-containers.hh"
5
6  using std::vector;
7
8  void
9  updateNodes (const AutomatonNonDet&      A,
10              AutomatonNonDet&             res,
11              int_setlist&                 newNodes,
12              ensToNb_map&                 nodes,
13              int                          sink_state)
14 {
15   int    current = nodes.find(*(newNodes.begin()),
16                               boost_hash(),
17                               boost_equal())->second;
18
19   for (AutomatonNonDet::alpha_iterator it = A.alpha_begin();
20        it != A.alpha_end(); ++it)
21     {
22       int_set    ens;
23       bool       haveFinal = false;
24
25       for (int_set::iterator it2 = newNodes.begin()->begin();
26            it2 != newNodes.begin()->end(); ++it2)
27         {
28           AutomatonNonDet::transitionRange trans;
29           trans = A.getTransFromSrcLbl(*it2, *it);
30
31           for (index_iterator<edgeList, lbl>::type elt = trans.first;
32                elt != trans.second; ++elt)
33             {
34               if (A.isFinal(elt->dst))
35                 haveFinal = true;
36
37               ens.insert(elt->dst);
38             }
39         }
40
41       if (!ens.empty())
42         {
43           if (nodes.find (ens, boost_hash(), boost_equal()) == nodes.end ())
44             {
45               nodes.insert(s_enstonb(ens, res.nodeAdd()->id_get()));
46               newNodes.insert (ens);
47             }
```

```
48            res.edgeAdd (current, nodes.find(ens, boost_hash(),
49                                             boost_equal())->second, *it);
50
51            if (haveFinal)
52              res.finalAdd(res.order() - 1);
53          }
54        else
55          res.edgeAdd(current, sink_state, *it);
56      }
57  }
58
59
60  AutomatonDet
61  determinize (const AutomatonNonDet& A)
62  {
63    AutomatonDet   res;
64    int_setlist    newNodes;
65    ensToNb_map    nodes;
66
67    int_set        initSet;
68    int            sink_state = res.nodeAdd()->id_get();
69
70    for (AutomatonNonDet::initial_const_iterator init = A.initial_begin ();
71         init != A.initial_end (); ++init)
72      initSet.insert(*init);
73
74    nodes.insert(s_enstonb(initSet, res.nodeAdd(true)->id_get()));
75    newNodes.insert (initSet);
76
77    if (A.haveEpsilon())
78      {
79        AutomatonNonDet    tmp(A);
80
81        removeEpsilon (tmp);
82        while (!newNodes.empty ())
83          {
84            updateNodes (tmp, res, newNodes, nodes, sink_state);
85            newNodes.erase (newNodes.begin ());
86          }
87      }
88    else
89      while (!newNodes.empty ())
90        {
91          updateNodes (A, res, newNodes, nodes, sink_state);
92          newNodes.erase (newNodes.begin ());
93        }
94
95    return res;
96  }
```

```
 97
 98
 99
100 AutomatonDet
101 determinize (const AutomatonDet& A)
102 {
103   return A;
104 }
```

6.18 minimize.cc - ABMI

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include "lib.hh"
5  #include "boost-containers.hh"
6  #include "minimize.hh"
7
8  using std::vector;
9  using std::pair;
10 using std::cout;
11 using std::endl;
12
13
14 void
15 splitPi (int_set&        pi,
16          int_set&        T,
17          vector<bool>&   X,
18          int_set&        atomicPi,
19          int&            nbEns,
20          EquivSet&       contains)
21 {
22   int_set::iterator      it;
23
24   for (it = pi.begin (); it != pi.end ();)
25   {
26     int                  n1 = 0;
27     int                  n2 = 0;
28
29     std::pair<EquivSet::iterator, EquivSet::iterator>
30       es = contains.equal_range(*it);
31
32     for (EquivSet::iterator it_equiv = es.first;
33          it_equiv != es.second;)
34       if (!X[it_equiv->id])
35         {
36           contains.modify_key(it_equiv++, update_equiv(nbEns));
37           ++n2;
38         }
39       else
40         {
41           ++n1;
42           ++it_equiv;
43         }
44
45     if (n1 && n2)
46       {
47         int tmp = *it;
```

```
48
49          pi.erase(it++);
50          if (n1 == 1)
51            atomicPi.insert(tmp);
52          else
53            pi.insert (tmp);
54          if (n2 == 1)
55            atomicPi.insert(nbEns);
56          else
57            pi.insert (nbEns);
58          if (n1 > n2)
59            T.insert (nbEns);
60          else
61            T.insert (tmp);
62          ++nbEns;
63        }
64      else
65        ++it;
66    }
67  }
68
69
70  void
71  createMinimized (const int_set&                pi,
72                   const AutomatonNonDet&        A,
73                   AutomatonDet&                 tmp,
74                   const EquivSet&               contains)
75  {
76    for (int_set::const_iterator it_set = pi.begin ();
77         it_set != pi.end (); ++it_set)
78      {
79        alphabet_t                          alpha;
80
81        std::pair<EquivSet::iterator, EquivSet::iterator>
82          es = contains.equal_range(*it_set);
83        for (EquivSet::iterator it_node = es.first;
84             it_node != es.second; ++it_node)
85          {
86            if (A.node (it_node->id).isFinal ())
87              tmp.finalAdd (*it_set);
88            if (A.node (it_node->id).isInitial ())
89              tmp.initialAdd (*it_set);
90
91            AutomatonNonDet::edge_const_iterator it_edge;
92            for (it_edge = A.edge_begin (it_node->id);
93                 it_edge != A.edge_end (it_node->id); ++it_edge)
94              if (alpha.find(it_edge->lbl) == alpha.end())
95                {
96                  tmp.edgeAdd (it_node->n, contains.get<id>().find(it_edge->dst)->n,
```

```
97                                  it_edge->lbl);
98                  alpha.insert(it_edge->lbl);
99                }
100         }
101     }
102 }
103
104
105 AutomatonDet
106 minimize (const AutomatonDet& A)
107 {
108   AutomatonDet  tmp;
109   int_set       T;
110   int           S;
111   int_set       pi;
112   int_set       atomicPi;
113   graph_t       pred;
114   EquivSet      contains;
115   int           nbEns = 2;
116
117   for (AutomatonNonDet::node_const_iterator it_node = A.begin();
118        it_node != A.end(); ++it_node) {
119
120     AutomatonNonDet::edge_const_iterator it_edge;
121     for (it_edge = it_node->begin();
122          it_edge != it_node->end(); ++it_edge)
123
124       pred.insert(s_transition(it_node->id_get(),
125                                it_edge->dst, it_edge->lbl));
126   }
127
128   for (AutomatonNonDet::node_const_iterator it = A.begin ();
129        it != A.end (); ++it)
130     if (it->isFinal ())
131       contains.insert (s_ens(0, it->id_get ()));
132     else
133       contains.insert(s_ens(1, it->id_get()));
134
135   pi.insert (0);
136   T.insert (0);
137   pi.insert (1);
138
139   while (!T.empty () && !pi.empty())
140   {
141     S = *(T.begin ());
142     T.erase (T.begin ());
143
144     for (AutomatonNonDet::alpha_iterator al = A.alpha_begin();
145          al != A.alpha_end() && !pi.empty(); ++al)
```

```
146       {
147         std::vector<bool>          X(A.order(), false);
148
149         std::pair<EquivSet::iterator, EquivSet::iterator>
150           es = contains.equal_range(S);
151         for (EquivSet::iterator it = es.first; it != es.second; ++it)
152           {
153             predPair pp = pred.equal_range(boost::make_tuple(it->id, *al));
154             for (graph_t::iterator it_trans = pp.first;
155                  it_trans != pp.second; ++it_trans)
156               X[it_trans->src] = true;
157           }
158
159         splitPi (pi, T, X, atomicPi, nbEns, contains);
160       }
161     }
162
163   tmp.createNbNodes (pi.size () + atomicPi.size());
164   createMinimized (pi, A, tmp, contains);
165   createMinimized (atomicPi, A, tmp, contains);
166
167   return tmp;
168 }
169
170 AutomatonDet
171 minimize (const AutomatonNonDet& A)
172 {
173   return minimize (determinize (A));
174 }
```

6.19 automaton-non-det.hh - BMI

```
1  #ifndef AUTOMATON_NON_DET_HH_
2  # define AUTOMATON_NON_DET_HH_
3  #include <iostream>
4  #include <set>
5  #include <boost/multi_index_container.hpp>
6  #include <boost/multi_index/member.hpp>
7  #include <boost/multi_index/ordered_index.hpp>
8  #include <boost/multi_index/hashed_index.hpp>
9  #include <boost/functional/hash/hash.hpp>
10 #include <boost/multi_index/sequenced_index.hpp>
11 #include <boost/tuple/tuple.hpp>
12 #include <boost/multi_index/composite_key.hpp>
13
14
15 using namespace ::boost;
16 using boost::multi_index_container;
17 using namespace boost::multi_index;
18
19
20 typedef multi_index_container
21 <
22   int,
23   indexed_by<ordered_unique<identity<int> > >
24 > int_set;
25
26
27 //typedef std::set<int> int_set;
28
29 struct s_enstonb
30 {
31   s_enstonb(const int_set& f, unsigned int i)
32     : first(f),
33       second(i)
34   {}
35
36   int_set        first;
37   unsigned int   second;
38 };
39
40
41 struct boost_hash
42 {
43   inline
44   std::size_t operator()(const s_enstonb& v) const
45   {
46     return hash_range(v.first.begin(), v.first.end());
47   }
```

```
48    inline
49    std::size_t operator()(const int_set& v) const
50    {
51      return hash_range(v.begin(), v.end());
52    }
53  };
54
55  struct boost_equal
56  {
57    inline
58    std::size_t operator()(const s_enstonb& x1, const s_enstonb& x2) const
59    {
60      return (x1.first == x2.first) && (x1.second == x2.second);
61    }
62    inline
63    std::size_t operator()(const int_set& x1, const int_set& x2) const
64    {
65      int_set::iterator it1, it2;
66
67      for (it1 = x1.begin (), it2 = x2.begin ();
68           it1 != x1.end () && it2 != x2.end () && *it2 == *it1; ++it1, ++it2)
69        ;
70      return (it1 == x1.end ()) && (it2 == x2.end ());
71    }
72  };
73
74
75  typedef multi_index_container
76  <
77    s_enstonb,
78    indexed_by
79    <
80          hashed_unique
81          <
82                  BOOST_MULTI_INDEX_MEMBER(s_enstonb, int_set, first),
83                  boost_hash,
84                  boost_equal
85          >
86    >
87  > ensToNb_map;
88
89
90  typedef multi_index_container
91  <
92    int_set,
93    indexed_by
94    <
95          hashed_unique
96          <
```

```
97                      identity<int_set>,
98                      boost_hash,
99                      boost_equal
100             >
101     >
102   > int_setset;
103
104
105   typedef multi_index_container
106   <
107     int_set,
108     indexed_by
109     <
110             ordered_non_unique
111             <
112                     identity<int_set>,
113                     boost_equal
114             >,
115             sequenced<>
116     >
117   > int_setlist;
118
119
120   typedef multi_index_container
121   <
122     unsigned int,
123     indexed_by
124     <
125             ordered_unique<identity<unsigned int> >
126     >
127   > states_t;
128
129
130   //typedef std::set<unsigned int> states_t;
131
132   typedef multi_index_container
133   <
134     char,
135     indexed_by
136     <
137             ordered_unique<identity<char> >
138     >
139   > alphabet_t;
140
141
142   //typedef std::set<char> alphabet_t;
143
144   struct s_transition
145   {
```

```
146   s_transition (unsigned int src_, unsigned int dst_, char lbl_)
147     : src(src_),
148       dst(dst_),
149       lbl(lbl_)
150   {}
151
152   unsigned int  src;
153   unsigned int  dst;
154   char          lbl;
155 };
156
157 struct succ {};
158 struct pred {};
159 struct src {};
160
161 typedef multi_index_container
162 <
163   s_transition,
164   indexed_by
165   <
166         hashed_non_unique
167         <
168                 tag<succ>,
169                 composite_key
170                 <
171                         s_transition,
172                         BOOST_MULTI_INDEX_MEMBER(s_transition, unsigned int, src),
173                         BOOST_MULTI_INDEX_MEMBER(s_transition, char
    , lbl)
174                 >
175         >,
176         hashed_non_unique
177         <
178                 tag<pred>,
179                 composite_key
180                 <
181                         s_transition,
182                         BOOST_MULTI_INDEX_MEMBER(s_transition, unsigned int, dst),
183                         BOOST_MULTI_INDEX_MEMBER(s_transition, char
    , lbl)
184                 >
185         >,
186         hashed_non_unique
187         <
188                 tag<src>,
189                 BOOST_MULTI_INDEX_MEMBER(s_transition, unsigned int, src)
190         >
191   >
192 > graph_t;
```

```
193
194 typedef std::pair<graph_t::iterator, graph_t::iterator> predPair;
195
196 class AutomatonNonDet
197 {
198 public:
199                                 AutomatonNonDet(unsigned int nb = 0);
200   virtual                       ~AutomatonNonDet() {}
201
202   typedef graph_t::iterator                                 iterator;
203   typedef graph_t::const_iterator                           const_iterator;
204   typedef index_iterator<graph_t, src>::type                src_iterator;
205   typedef index_const_iterator<graph_t, src>::type          src_const_iterator;
206   typedef states_t::iterator                                states_iterator;
207   typedef states_t::const_iterator                          states_const_iterator;
208   typedef alphabet_t::iterator                              alpha_iterator;
209   typedef alphabet_t::const_iterator                        alpha_const_iterator;
210   typedef std::pair<index_iterator<graph_t, pred>::type,
211                     index_iterator<graph_t, pred>::type>        predRange;
212   typedef std::pair<index_iterator<graph_t, src>::type,
213                     index_iterator<graph_t, src>::type>         srcRange;
214   typedef std::pair<graph_t::iterator,
215                     graph_t::iterator>                    transitionRange;
216
217   unsigned int                  nodeAdd(bool init = false, bool final = false);
218   void                          edgeAdd(unsigned int src,
219                                         unsigned int dst,
220                                         char lbl);
221   void                          finalAdd(unsigned int i);
222   void                          initialAdd(unsigned int i);
223   void                          createNbNodes(unsigned int n);
224
225   bool                          isFinal(unsigned int i) const;
226   bool                          isInitial(unsigned int i) const;
227
228   srcRange                      getSucc(unsigned int src) const;
229   transitionRange               getTransFromSrcLbl(unsigned int src,
230                                                    char lbl) const;
231   predRange                     getPredFromDst(unsigned int dst,
232                                                char lbl) const;
233
234   unsigned int                  size() const;
235   unsigned int                  alpha_size() const;
236   unsigned int                  haveEpsilon() const;
237
238   void                          print() const;
239
240   src_iterator                  begin();
241   src_const_iterator            begin() const;
```

```
242   src_iterator                end();
243   src_const_iterator          end() const;
244   states_iterator             init_begin();
245   states_const_iterator       init_begin() const;
246   states_iterator             init_end();
247   states_const_iterator       init_end() const;
248   states_iterator             final_begin();
249   states_const_iterator       final_begin() const;
250   states_iterator             final_end();
251   states_iterator             final_end() const;
252   alpha_iterator              alpha_begin();
253   alpha_const_iterator        alpha_begin() const;
254   alpha_iterator              alpha_end();
255   alpha_const_iterator        alpha_end() const;
256
257 private:
258   graph_t       graph_;
259   states_t      initial_;
260   states_t      final_;
261   alphabet_t    alpha_;
262   unsigned int  nb_;
263   unsigned int  haveEpsilon_;
264 };
265
266 # include "automaton-non-det.hxx"
267
268 #endif //! AUTOMATON_NON_DET_HH_
```

6.20 automaton-non-det.hxx - BMI

```
1
2
3  inline
4  void
5  AutomatonNonDet::edgeAdd(unsigned int src, unsigned int dst, char lbl)
6  {
7    graph_.insert(s_transition(src, dst, lbl));
8    if (lbl != 1)
9      alpha_.insert(lbl);
10   else
11     ++haveEpsilon_;
12 }
13
14
15 inline
16 void
17 AutomatonNonDet::initialAdd(unsigned int i)
18 {
19   initial_.insert(i);
20 }
21
22
23 inline
24 void
25 AutomatonNonDet::finalAdd(unsigned int i)
26 {
27   final_.insert(i);
28 }
29
30
31 inline
32 void
33 AutomatonNonDet::createNbNodes(unsigned int n)
34 {
35   nb_ += n;
36 }
37
38
39 inline
40 bool
41 AutomatonNonDet::isFinal(unsigned int i) const
42 {
43   return final_.find(i) != final_.end();
44 }
45
46
47 inline
```

```
48 bool
49 AutomatonNonDet::isInitial(unsigned int i) const
50 {
51   return initial_.find(i) != initial_.end();
52 }
53
54 inline
55 AutomatonNonDet::transitionRange
56 AutomatonNonDet::getTransFromSrcLbl(unsigned int src, char lbl) const
57 {
58   return graph_.equal_range(make_tuple(src, lbl));
59 }
60
61
62 inline
63 unsigned int
64 AutomatonNonDet::size() const
65 {
66   return nb_;
67 }
68
69
70 inline
71 unsigned int
72 AutomatonNonDet::alpha_size() const
73 {
74   return alpha_.size();
75 }
76
77
78 inline
79 unsigned int
80 AutomatonNonDet::haveEpsilon() const
81 {
82   return haveEpsilon_;
83 }
84
85
86 inline
87 AutomatonNonDet::src_iterator
88 AutomatonNonDet::begin()
89 {
90   return graph_.get<src>().begin();
91 }
92
93 inline
94 AutomatonNonDet::src_const_iterator
95 AutomatonNonDet::begin() const
96 {
```

```
97    return graph_.get<src>().begin();
98  }
99
100
101 inline
102 AutomatonNonDet::src_iterator
103 AutomatonNonDet::end()
104 {
105   return graph_.get<src>().end();
106 }
107
108
109 inline
110 AutomatonNonDet::src_const_iterator
111 AutomatonNonDet::end() const
112 {
113   return graph_.get<src>().end();
114 }
115
116 inline
117 AutomatonNonDet::states_iterator
118 AutomatonNonDet::init_begin()
119 {
120   return initial_.begin();
121 }
122
123
124 inline
125 AutomatonNonDet::states_const_iterator
126 AutomatonNonDet::init_begin() const
127 {
128   return initial_.begin();
129 }
130
131
132 inline
133 AutomatonNonDet::states_iterator
134 AutomatonNonDet::init_end()
135 {
136   return initial_.end();
137 }
138
139
140 inline
141 AutomatonNonDet::states_const_iterator
142 AutomatonNonDet::init_end() const
143 {
144   return initial_.end();
145 }
```

```
146
147
148 inline
149 AutomatonNonDet::states_iterator
150 AutomatonNonDet::final_begin()
151 {
152   return final_.begin();
153 }
154
155
156 inline
157 AutomatonNonDet::states_const_iterator
158 AutomatonNonDet::final_begin() const
159 {
160   return final_.begin();
161 }
162
163
164 inline
165 AutomatonNonDet::states_iterator
166 AutomatonNonDet::final_end()
167 {
168   return final_.end();
169 }
170
171
172 inline
173 AutomatonNonDet::states_iterator
174 AutomatonNonDet::final_end() const
175 {
176   return final_.end();
177 }
178
179
180 inline
181 AutomatonNonDet::alpha_iterator
182 AutomatonNonDet::alpha_begin()
183 {
184   return alpha_.begin();
185 }
186
187
188 inline
189 AutomatonNonDet::alpha_const_iterator
190 AutomatonNonDet::alpha_begin() const
191 {
192   return alpha_.begin();
193 }
194
```

```
195
196 inline
197 AutomatonNonDet::alpha_iterator
198 AutomatonNonDet::alpha_end()
199 {
200   return alpha_.end();
201 }
202
203
204 inline
205 AutomatonNonDet::alpha_const_iterator
206 AutomatonNonDet::alpha_end() const
207 {
208   return alpha_.end();
209 }
210
211 inline
212 AutomatonNonDet::predRange
213 AutomatonNonDet::getPredFromDst(unsigned int dst, char lbl) const
214 {
215   return graph_.get<pred>().equal_range(make_tuple(dst, lbl));
216 }
217
218 inline
219 AutomatonNonDet::srcRange
220 AutomatonNonDet::getSucc(unsigned int s) const
221 {
222   return graph_.get<src>().equal_range(s);
223 }
```

6.21 automaton-non-det.cc - BMI

```
1  #include <iostream>
2  #include "automaton-non-det.hh"
3
4  AutomatonNonDet::AutomatonNonDet(unsigned int nb)
5    : nb_(nb),
6      haveEpsilon_(0)
7  {
8
9  }
10
11 unsigned int
12 AutomatonNonDet::nodeAdd(bool init, bool final)
13 {
14   if (init)
15     initial_.insert(nb_);
16   if (final)
17     final_.insert(nb_);
18   return nb_++;
19 }
20
21
22 void
23 AutomatonNonDet::print() const
24 {
25   unsigned int  last = 1;
26
27   for(src_const_iterator it = graph_.get<src>().begin();
28       it != graph_.get<src>().end(); ++it)
29     {
30       if (last != it->src)
31         {
32           std::cout << std::endl << "[";
33           if (initial_.find(it->src) != initial_.end())
34             std::cout << "I";
35           else
36             std::cout << "␣";
37           if (final_.find(it->src) != final_.end())
38             std::cout << "F";
39           else
40             std::cout << "␣";
41           std::cout << "]␣";
42           std::cout << it->src << "␣:␣";
43           last = it->src;
44         }
45       std::cout << "(" << it->dst << "," << it->lbl << ")␣";
46     }
47   std::cout << std::endl;
```

```
48    std::cout << "alphabet␣:␣";
49    for (alpha_iterator it = alpha_.begin(); it != alpha_.end(); ++it)
50      std::cout << *it << ",";
51    std::cout << std::endl;
52  }
```

6.22 determinize.cc - BMI

```
1  #include <iostream>
2  #include <vector>
3  #include "lib.hh"
4
5  void
6  updateNodes (const AutomatonNonDet&      A,
7               AutomatonNonDet&            res,
8               int_setlist&                newNodes,
9               ensToNb_map&                nodes,
10              const std::vector<bool>&    isFinal,
11              int                         sink_state)
12 {
13   int    current = nodes.find(*(newNodes.begin()),
14                               boost_hash(),
15                               boost_equal())->second;
16
17   for (AutomatonNonDet::alpha_iterator it = A.alpha_begin();
18        it != A.alpha_end(); ++it)
19     {
20       int_set    ens;
21       bool       haveFinal = false;
22
23       for (int_set::iterator it2 = newNodes.begin()->begin();
24            it2 != newNodes.begin()->end(); ++it2)
25         {
26           AutomatonNonDet::transitionRange res = A.getTransFromSrcLbl(*it2,
27                                                                       *it);
28           if (res.first != res.second)
29             for (AutomatonNonDet::iterator elt = res.first;
30                  elt != res.second; ++elt)
31               {
32                 if (isFinal[elt->dst])
33                   haveFinal = true;
34                 ens.insert(elt->dst);
35               }
36         }
37
38       if (!ens.empty())
39         {
40           if (nodes.find (ens, boost_hash(),
41                           boost_equal()) == nodes.end ())
42             {
43               nodes.insert(s_enstonb(ens, res.nodeAdd()));
44               newNodes.insert (ens);
45             }
46
47           res.edgeAdd (current,
```

```
48                          nodes.find(ens, boost_hash(),
49                                     boost_equal())->second,
50                          *it);
51
52            if (haveFinal)
53              res.finalAdd(res.size() - 1);
54          }
55        else
56          res.edgeAdd (current, sink_state, *it);
57      }
58  }
59
60  AutomatonDet
61  determinize (const AutomatonNonDet&      A)
62  {
63    AutomatonDet            res;
64    int_setlist             newNodes;
65    ensToNb_map             nodes;
66    std::vector<bool>       isFinal(A.size(), false);
67    int_set                 initSet;
68    int                     sink_state;
69
70    sink_state = res.nodeAdd();
71    for (AutomatonNonDet::alpha_iterator al = A.alpha_begin();
72         al != A.alpha_end(); ++al)
73      res.edgeAdd(sink_state, sink_state, *al);
74
75    for (AutomatonNonDet::states_const_iterator it = A.final_begin();
76         it != A.final_end(); ++it)
77      isFinal[*it] = true;
78
79    for (AutomatonNonDet::states_const_iterator init = A.init_begin ();
80         init != A.init_end (); ++init)
81      initSet.insert(*init);
82
83    nodes.insert(s_enstonb(initSet, res.nodeAdd(true)));
84    newNodes.insert (initSet);
85
86    if (A.haveEpsilon())
87      {
88        AutomatonNonDet                 tmp(A);
89        removeEpsilon (tmp);
90
91        while (!newNodes.empty ())
92          {
93            updateNodes (tmp, res, newNodes, nodes, isFinal, sink_state);
94            newNodes.erase (newNodes.begin ());
95          }
96      }
```

```
97    else
98      while (!newNodes.empty ())
99        {
100         updateNodes (A, res, newNodes, nodes, isFinal, sink_state);
101         newNodes.erase (newNodes.begin ());
102       }
103
104   return res;
105 }
106
107
108
109 AutomatonDet
110 determinize (const AutomatonDet& A)
111 {
112   return A;
113 }
```

6.23 minimize.cc - BMI

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include "lib.hh"
5  #include "minimize.hh"
6
7  using std::vector;
8  using std::pair;
9  using std::cout;
10 using std::endl;
11
12
13 void
14 splitPi (int_set&        pi,
15          int_set&        T,
16          vector<bool>&   X,
17          int&            nbEns,
18          EquivSet&       contains)
19 {
20   for (int_set::iterator it = pi.begin (); it != pi.end ();)
21   {
22     int                   n1 = 0;
23     int                   n2 = 0;
24
25     std::pair<EquivSet::iterator, EquivSet::iterator>
26       es = contains.equal_range(*it);
27
28     for (EquivSet::iterator it_equiv = es.first;
29          it_equiv != es.second;)
30       if (!X[it_equiv->id])
31       {
32         contains.modify_key(it_equiv++, update_equiv(nbEns));
33         ++n2;
34       }
35       else
36       {
37         ++n1;
38         ++it_equiv;
39       }
40
41     if (n1 && n2)
42     {
43       if (n1 > n2)
44       {
45         pi.insert (*it);
46         T.insert (nbEns);
47       }
```

```
48        else
49        {
50          pi.insert (nbEns);
51          T.insert (*it);
52        }
53        pi.erase(it++);
54        ++nbEns;
55      }
56      else
57        ++it;
58    }
59  }
60
61
62  void
63  createMinimized (const int_set&                pi,
64                   const AutomatonNonDet&        A,
65                   AutomatonDet&                 tmp,
66                   const EquivSet&               contains)
67  {
68    tmp.createNbNodes (pi.size ());
69
70    for (int_set::const_iterator it_set = pi.begin ();
71         it_set != pi.end (); ++it_set)
72      {
73        alphabet_t        alpha;
74
75        std::pair<EquivSet::iterator, EquivSet::iterator>
76          es = contains.equal_range(*it_set);
77        for (EquivSet::iterator it_node = es.first;
78             it_node != es.second; ++it_node)
79          {
80            if (A.isFinal (it_node->id))
81              tmp.finalAdd (*it_set);
82            if (A.isInitial (it_node->id))
83              tmp.initialAdd (*it_set);
84
85            AutomatonNonDet::srcRange sr = A.getSucc(it_node->id);
86            for (AutomatonNonDet::src_const_iterator it_edge = sr.first;
87                 it_edge != sr.second; ++it_edge)
88              if (alpha.find(it_edge->lbl) == alpha.end())
89                {
90                  tmp.edgeAdd (it_node->n,
91                               contains.get<id>().find(it_edge->dst)->n,
92                               it_edge->lbl);
93                  alpha.insert(it_edge->lbl);
94                }
95          }
96      }
```

```
 97 }
 98
 99
100 AutomatonDet
101 minimize (const AutomatonDet& A)
102 {
103   AutomatonDet  tmp;
104   int_set       T;
105   int           S;
106   int_set       pi;
107   EquivSet      contains;
108   int           nbEns = 2;
109
110   for (int i = 0; i < A.size(); ++i)
111     contains.insert(s_ens(!A.isFinal (i), i));
112
113   pi.insert (0);
114   T.insert (0);
115   pi.insert (1);
116
117   while (!T.empty ())
118   {
119     S = *(T.begin ());
120     T.erase (T.begin ());
121     for (AutomatonNonDet::alpha_iterator al = A.alpha_begin();
122          al != A.alpha_end() && !pi.empty(); ++al)
123     {
124       std::vector<bool> X(A.size(), false);
125
126       std::pair<EquivSet::iterator, EquivSet::iterator>
127         es = contains.equal_range(S);
128       for (EquivSet::iterator it = es.first; it != es.second; ++it)
129         {
130           AutomatonNonDet::predRange pp = A.getPredFromDst(it->id, *al);
131           for (index_iterator<graph_t, pred>::type it_trans = pp.first;
132                it_trans != pp.second; ++it_trans)
133             X[it_trans->src] = true;
134         }
135       splitPi (pi, T, X, nbEns, contains);
136     }
137   }
138
139   createMinimized (pi, A, tmp, contains);
140   return tmp;
141 }
142
143 AutomatonDet
144 minimize (const AutomatonNonDet& A)
145 {
```

```
146     return minimize (determinize (A));
147 }
```

Chapter 7

Thanks to...

I would like to thank all the people who helped me to write this report or to do some testing:

- Jacques Sakarovitch
- Sylvain Lombardi
- Akim Demaille
- Alban Linard
- The Vaucanson staff (special thanks to Michael Cadilhac though)
- Samuel Charron
- And those I forgot to mention!