MachineCat: Design and Implementation

Rui (Ray) Zhou

The University of Prince Edward Island

CSIT-15
July 17, 2012
# Contents

1 Introduction 4  
  1.1 What is MachineCat? 4  
  1.2 The Objective of MachineCat 4  

2 User Guide 4  
  2.1 Basic Concepts 5  
    2.1.1 Finite States Machine 5  
    2.1.2 Regular Expression 5  
    2.1.3 Finite State Language 5  
  2.2 Usages By Example 5  
    2.2.1 Executes Implemented Filters 5  
    2.2.2 The "--infoPage" Flag 6  
    2.2.3 Other Flags 6  
    2.2.4 Using MachineCat as Basement 6  

3 General Designs and Key Features 6  
  3.1 C++ and Standard Library 6  
  3.2 Efficiency and Stability 7  
  3.3 Organization of Source Files 7  

4 Architecture of BackBone System 8  
  4.1 Debug Mode 8  
  4.2 BackBone (Data Structure) 9  
    4.2.1 McArray 9  
    4.2.2 McSet 10  
    4.2.3 McMap 11  
    4.2.4 McInt 12  
  4.3 Theory Objects 12  
    4.3.1 McState 12  
    4.3.2 McTransition 13  
    4.3.3 McNFA 13  
    4.3.4 McDFA 15  
    4.3.5 Symbol Table Design 15  
    4.3.6 McString and McLanguage 16  
  4.4 API Support 17  

5 Implemented Algorithms 17  
  5.1 McMachineAnalyze 17  
  5.2 McNFAtoDFA Set 17  
  5.3 McDFAMinimize TBF 17  
  5.4 McNFAEnum 18  
  5.5 More algorithms... 18  

6 Filters and Main Function 18  
  6.1 Meta-Programming with C++ 18  
  6.2 Auto-Collection: Algorithms 18  
  6.3 Auto-Collection: Filters 19  
  6.4 Generate Symbolic Links 20  
  6.5 Why Separate Algorithm and Filter 20  
  6.6 Main Function 21  
  6.7 Implemented Filters 23  


1 Introduction

1.1 What is MachineCat?

The MachineCat is not a cat made with metal gears and rods. It is a C++ software package for computation with finite machines, finite languages and other objects in the theory of computing. Users can use previously implemented filters (think of them as different executables with different usages) to do various tests. Users can implement their own algorithms and filters as they want for MachineCat and put them into correct folders, MachineCat will automatically integrate their code and generate the executables. Users can also implement any specific program for any specific purpose very easily, by including MachineCat and use it as the base.

Initially, MachineCat was built as Rui (Ray) Zhou’s Honors Project under the supervision of Dr. Cezar Câmpeanu[1], but it is destined to be much more than a undergraduate thesis project. Although still in its development phase, MachineCat has shown its great abilities and potentials.

1.2 The Objective of MachineCat

The objective of MachineCat is to be an excellent software package and compete with other software packages designed for the theory of computing. Compared to Grail+ [2], a widely used symbolic computation environment/library for the theory of computing, MachineCat is better in the following aspects:

MachineCat has a better backbone with code up to today’s standard. Grail+ was made before the standards of C++ programming language were established[3]. Although Grail+ is a wonderful package, its code is very old styled and sometimes a little corrupted. Grail+ also integrates code that supports systems which are rarely used or not used anymore. MachineCat uses standard C++ and takes advantage of the standard library, so MachineCat will always be updated with the development of the C++ language and its standard library as long as the interfaces stay consistent.

The stability of MachineCat has been given special attention. A lot of input data or operations that crash Grail+ will be acceptable in MachineCat, and as a result more agile way of data presentation is possible.

The development of MachineCat has been a process of pursuing great features while keeping performance in mind. While Grail+ is famous for its ability to process large scale tests at rapid speed, MachineCat is comparable or even better than Grail+.

The majority of MachineCat is composed of different modules. The principle of object-oriented programming is carefully followed. Moreover, the modularization feature is especially well designed when it comes to the implementation of filters and algorithms. In MachineCat, we can attach and remove algorithms and filters just by moving folders.

With a finite machine integrated with the input processing system, MachineCat is able to take inputs rapidly and correct errors such as incorrect indentation, miss-spelled keywords and so on. MachineCat will also be compatible with other packages regarding the output and input formats.

Because of its rich functionality with great usability, MachineCat will be able to support many features which were not in other packages. Many newly designed features in MachineCat are very user-friendly, such as the filter info page.

2 User Guide

This section is for those who need to start using MachineCat right away and make tests with previously implemented filters. The basics of MachineCat and some usage examples are presented. Users will be able to start using MachineCat very quickly. Potential developers should read the entire manual if they want to do further development for MachineCat.
2.1 Basic Concepts

The following basic concepts in theory of computing is required to use MachineCat.

2.1.1 Finite States Machine

Finite Automata (Finite-state machine) is a behavior model composed of a finite number of states, transitions between those states, and actions, similarly to a flow graph in which one can inspect the way logic runs when certain conditions are met.[4]

Here is an normal way to present a finite machine in text file:

```
(START) |− 0
  0 a 1
  0 b 2
  0 c 3
  1 a 4
  2 a 4
  3 a 4
  4 a 5
  5 −| (FINAL)
```

2.1.2 Regular Expression

In computing, a regular expression provides a concise and flexible means for “matching” (specifying and recognizing) strings of text, such as particular characters, words, or patterns of characters. [5] Examples:

- \( a(b^*) = \{a, ab, abb, abbb, \ldots\} \)
- \( a|b = \{a, b\} \)
- \( (a|b)^* = \{\epsilon, a, b, aa, bb, ab, aabb, \ldots\} \)

2.1.3 Finite State Language

In theoretical computer science and formal language theory, a regular language is a formal language that can be expressed using a regular expression.

A specific subset within the class of regular languages is the finite languages -those containing only a finite number of words. These are regular languages, as one can create a regular expression that is the union of every word in the language.[6]

Examples:

- \( \{\} \)
- \( \{\epsilon\} \)
- \( \{a, ab, abb, aabb\} \)

2.2 Usages By Example

In the following subsections, examples of how to use MachineCat at user level are presented. For the introduction of MachineCat, please read section 1.

2.2.1 Executes Implemented Filters

Here is an example of how to use the implemented filters in MachineCat to do computation and tests:

1. Navigate to MachineCat/Bin/ directory.
2. Invoke any filters, enter the test data, or provide the name of the file including the test data.
3. Watch the output.

4. Here is the test execution of the filter fmenum:

```bash
1 MachineCat/Bin$ ./fmenum
2 enter nfa
3 (START) | 0
4 0 A 1
5 1 B 2
6 2 | (FINAL)
7
8 enter max length
9 2
10 enter max num
11 2
12 done!
13 A B
```

2.2.2 The “–infoPage” Flag

Instead of putting all the information about an algorithm or a filter into the comments in their source code, In MachineCat, we always have one HTML file for every filter. The HTML page is the place for the programmer to document the information about the filter, such as the usage examples, the algorithms invoked and other useful links to outer sources.

Although algorithms and filters are coded as separated parts, for one algorithm, we will always have at least one filters which invokes it, otherwise it is an algorithm used by none and does not make any sense to be there. The information for an algorithm should be documented in the HTML page of the filters which uses the algorithm frequently or uses the algorithm as its main computation.

To open the local HTML page for informations, please use the flag “–infoPage” after the filter name to pop up the information page. For example, figure[5] shows thee webpage for ./fmenum--infoPage :

The HTML page will be opened with the default browser. The page will contain usage examples and any other information written by the developer for the filter. Users must make sure you have a default browser supporting basic HTML.

In other situations when MachineCat is running on a remote connection without a remote X window[7], the flag “–info” can be used to see the source of the HTML manual pages instead of trying to pop up the default browser, which may cause a lot of annoying problems when the system can not find a valid display.

2.2.3 Other Flags

Along with the “–info” flag which should be valid with all the filters, there are also other filter-specific flags you can use for different purposes. For example, the filter “fmunion” takes two finite machine and union them together. This filter will check the existence of the “–unique” flag. If the flag is found, the filter will make sure the result finite machine does not contain duplicated lines in the transitions. Otherwise, by default, there will be no check for duplicates done for performance concerns. The user should be able to use “filter_name --infoPage” or “filter_name –info” to see the available flags for specific filters.

2.2.4 Using MachineCat as Basement

Instead of using filters in MachineCat to do tests, using MachineCat as basement helps you when you do actual programming, so the details have been written in section[7.2]

3 General Designs and Key Features

3.1 C++ and Standard Library

MachineCat is a project developed with careful consideration to coding styles and philosophy. One obvious feature of MachineCat is the frequent usage of and dependency on the C++ standard library(C++ STL). The C++ STL
already implements a lot of useful functions, which perform fast and stable. As the C++ STL is so widely used and approved, there is few bugs in them, and there is absolute no need to code up anything from scratch again if a (similar) class already exists in the C++ STL.

Other than the C++ STL, MachineCat is also using native C++ and compiled with g++, so it will be able to run in any environment where standard C++ and g++/gcc is supported. Some scripts are used to enable various of automatic features, currently the scripts are made for unix-like system which support bash and sh (we use bash by default). Alternative scripts for other platforms can be implemented easily.

However, MachineCat was developed in Ubuntu with Eclipse, and the recommended platform for it is Linux. Eclipse has been very helpful in MachineCat’s development. It can also be used with Grail+ to provide lots of convenience.

3.2 Efficiency and Stability

Another important feature of this package is its speed. Nowadays computers usually have a memory big enough for researching packages like MachineCat to take advantage of, so MachineCat will prefer to choose efficiency when there is a potential trade-off between speed and space. However this does not mean MachineCat has to take more memory for speed. MachineCat also tries to provide great stability and compatibility along with good speed. For example, in the operation to attach transitions to their start states, Grail+ uses an array and indexes the array with states numbers. This provides very fast access time but could be very problematic when the states are not canonically named. In MachineCat when we attach transitions, we apply a McMap and use the start states as keys, which makes it slower ($O \log(n)$ vs $O(1)$) to access the elements, but provides very stable and readable code styles. In fact, using the current MachineCat attach() approach, the affected MachineCat filters sometimes run slightly slower, but used less memory. This may sounds opposite to the preference of speed, but it is actually a balanced way to deal with all the concerns, guaranteeing an overall great performance in most cases. With McMap, MachineCat never crashes for non-canonical numbered states.

Atentions were also given to the code style when we try to achieve the target of efficiency. MachineCat uses references more than pointers. Objects are mostly stored in the stack instead of the heap, which reduces the process of accessing objects by pointers and make it less likely for programmers to accidentally leak memory. Inline is used widely in this project, basically functions with only one statement will be in-lined. It results in an increase in the executable size, but allows higher speed to be achieved. However, Bigger blocks will not be in-lined, because a big increase in executable size will also decrease efficiency. The const keyword is also used a lot in MachineCat, to protect objects from inappropriate operations. This makes the whole project more robust. Const is also important because of its ability to make sure MachineCat does not create unnecessary temp objects and process even faster when we call functions.

3.3 Organization of Source Files

MachineCat is organized in the following way:

1. One header file MachineCat.h is put into the root folder. User can use MachineCat as basement by including it.
2. There is a folder called Backbone/, which holds the code for all of the abstract container data structures like array or set. One head file “McBackBone.h” includes everything in the folder in the sequence of dependency.
3. Beside Backbone/ folder, there is a folder named “TheoryObjects/”, this folder contains the implementation of various objects in theory of computing, such as machine and language. The head file “TheoryObjects.h” includes all the files in the folder.
4. The codes in folder DebugControl/ implement the debug system, all the debug triggers are also defined here.
5. Different from many packages that integrated algorithms into theory objects classes as member functions, MachineCat places all the algorithms in the Algorithms/ folder. They are not members of any classes,
but global accessible methods that perform algorithms using the Theory Objects classes and Backbone data structures. This approach is significant better that we keep the algorithms independent from Theory objects, so we can easily add, update and delete different algorithms in different separated pieces of source code as we want, without messing up the implementation of any Theory objects class. On the other side, updating the theory objects will not affect the implementation of algorithms as long as we keep the interfaces consistent in the theory objects classes. MachineCat is highly modularized with this design, that every component can be added, replaced or deleted without necessarily affecting any irrelevant components.

6. The folder Filters/ contains the definition and implementation of the filters. These filters are invoked from the main function, which is defined in the MainFunction/ folder. Filters are also highly modularized.

7. The Bin/ folder is empty by default. But when we compile the project the filters will be generated in this folder. Please note that anything in this folder will be cleaned up when a re-compilation happens or “make clean” is invoked.

8. In MachineCat root folder, there are also Test folder and Documentation folder, they are used to put all the test code and develop notes.

4 Architecture of BackBone System

The Backbone system is carefully designed to perform rapidly and robustly. It is the basement of the entire MachineCat.

4.1 Debug Mode

Why we need a debug system? The obvious reason is that if the program somehow starts to perform incorrectly, we need to have debug output in different steps, to see where exactly has went wrong. Another great reason to have a debug system and output information on the go is to trace the computation process of an algorithm, to obtain not only the final results but also the internal work and data. This helps us to understand algorithms in more depth.

MachineCat supports precise debugging output by using the following approach:

For any scope in any piece of code, such as a function or a class definition, it can be issued a unique integer defined in DebugControl/ McDebug.h as its debug trigger. An global integer array “DEBUG_TRIGGERS” is used to store all the triggers for those scopes of code that we want to debug. Any debug-able scope will check if its associated debug trigger is in the array. If yes, then the debug information in this scope is outputted.

For example, in DebugControl/McDebug.h, we have:

```c
//2. debug trigger names
#define DEBUG_ALL 0
#define DEBUG_McNFAtoDFA_Set 1
```

If we want to enable the debug information for McNFAtoDFA_Set():

```c
DEBUG_TRIGGER(DEBUG_McNFAtoDFA_Set);
//ENABLE DEBUG FOR THIS ALGO
//before use it
McNFAtoDFA_Set(nfa, d);
```

And the function’s debug mode is triggered in its scope:

```c
void McNFAtoDFA_Set(McNFA& nfa, McDFA& dfa )
{
    //DEBUG ?
    bool debug =false;
    if(DEBUG_TRIGGERED(DEBUG_McNFAtoDFA_Set) || DEBUG_TRIGGERED(DEBUG_ALL))
    {
        //DEBUG_CODE
    }
```
```cpp
defined=true;
std::cout<<"Debug output for McNFAdcDFA_Set (NFA, DFA) is enabled."<<std::endl;
```

4.2 BackBone (Data Structure)

4.2.1 McArray

McArray is the most basic data container class in the MachineCat project. It is actually a wrapper class manipulating one standard library vector class as its member.

Standard mathematical functions are defined for McArray, such as comparison, assignment, plus-equal(add element) so on.

McArray is labeled with sorted or not. Whether sorted or not is a very interesting and important attribute of an McArray object in MachineCat, because a sorted array can be processed with many fast algorithms which depend on order. We would like to always keep a McArray sorted if possible.

Three sorting algorithms are integrated in McArray class. The heap sort function takes advantage of the existing heap sort method in the STL::algorithms. The other two are both quicksort, the difference is the usage of stdlib::qsort and algorithm::sort. The efficiency of the three sorting function could vary and need further test. But so far from what we have observed, the sorting algorithm with STD::sort() seems to be the best, which over-performs STD::qsort() a little but significantly over heapSort().

The comparative operators returns the comparison result of the vectors inside McArrays.

[Enumeration of some Useful API]

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int size()</td>
<td>return the size of this array. Complexity: 1</td>
</tr>
<tr>
<td>void clear()</td>
<td>Clear the content of the array, sorted is set to true. Complexity: O(1)</td>
</tr>
<tr>
<td>bool isSorted()</td>
<td>return the value of sorted. Complexity: O(1)</td>
</tr>
<tr>
<td>void heapSort()</td>
<td>Heap sort this McArray, the STL heap make_heap and sort_heap for std::vector is called. Complexity: At most, ( N \log N ) comparisons, where ( N ) is (last-first).</td>
</tr>
<tr>
<td>void quickSortWithQsort()</td>
<td>This is a quick sort using qsort declared in “stdlib.h”.The majority operations are pointer operations/manipulations. Complexity: at most ( O(n^2) )</td>
</tr>
<tr>
<td>void quickSortWithSort()</td>
<td>This method calls the algorithm::sort() method on the vector. Complexity: Approximately ( N \log N ) comparisons on average (where ( N ) is last-first). In the worst case, up to ( N^2 ), depending on specific sorting algorithm used by library implementation.</td>
</tr>
<tr>
<td>void swap(McArray&lt;T&gt; &amp;a)</td>
<td>Swap this array with the parameter Complexity: linear.</td>
</tr>
</tbody>
</table>
```plaintext
int member(const T&) const
Search for the parameter in this array. If this McArray is sorted, binary search is used, otherwise linear search is used.
Complexity: \( \log_2 n \) to \( n \)

static void getIntersection(const McArray<T>& a1, const McArray<T>& a2, McArray<T>& inter)
Store the intersection of the first two McArray Parameter into the last one.
Complexity: Suppose there are \( a \) elements in \( a_1 \) and \( b \) elements in \( a_2 \), if \( a_2 \) is not sorted, complexity is \( a \cdot b \), if \( a_2 \) is sorted, complexity is \( a \cdot \log_2 b \). Possible Improvement could be done for the case when 2 arrays are both sorted.

McArray<T>& mergeSortedArrays(McArray<T>&)
Merge the parameter McArray into this array, programmer should make sure both McArray are sorted. This McArray is returned.
Complexity: linear, associated with the number of elements of the shorter array.

void unique_array()
First sort the array, then remove duplicated elements
Complexity: sort \( O(n \log n) \), remove is linear, so total \( O(n \log n) \)

4.2.2 McSet
McSet inherits from McArray using public inheritance. A McSet is nothing but a McArray that (should) enforces the uniqueness of the elements in it. This enforcement is carried when we add elements using "+=" to a McSet or when we cast a McArray to a McSet using McSet<T>::McSet<T>(const McArray&).
McSet uses the comparative operators inherited from McArray.

[Enumeration of Useful API]
```
McSet<T>& intersect(McSet<T>&, McSet<T>&)
Get the intersection of the two parameter sets, and store the result in this McSet. The two parameter sets are heapSorted first and then the std::set_intersection is called.
complexity(growth order): $n \log_2 n$

4.2.3 McMap
Currently McMap is just a wrapper for std::map, used as a “dictionary” data container.
With McMap, we can achieve very agile code style with both great readability and wonderful efficiency. Here is an example:
Attach transitions to start state:

```
for (int a = 0; a < transitions.size(); a++)
    transitionsByState[transitions[a].getStart()].disjoint_union(
        transitions[a]);
```

And get the transitions set by start state:
```
return transitionsByState[s];
```

The memory allocation and management are all maintained by McMap, and we do not have any potential issues like segmentation fault when we use arrays.

The iterator of McMap is declared as:
```
typedef typename std::map<Key, T>::iterator McMapItator;
```
The following is some useful API, for this stage basically everything is wrapping the functions from std::map.

```
McMapItatorbegin()
get the iterator to the head of the Dictionary(Map).
complexity(growth order): 1
```
```
McMapItatorbegin()
get the iterator to the head of the Dictionary(Map).
complexity(growth order): 1
```
```
McMapItator end()
Get the end iterator to of the Dictionary.
complexity(growth order): 1
```
```
McMap<Key, T>& erase(const Key&)
erase the element with thesis key as the entry.
complexity(growth order): 1
```
```
T& operator[](const Key&k)
return the element associated with this Key.
complexity(growth order): 1
```
4.2.4 McInt

McInt is essentially a continuous chunk of memory, it is implemented as a char array and can be as large as we want until we reach the limitation of the memory (seriously, why we need such huge thing?).

Why we need McInt? The idea is to use the minimal information to represent specific data type. To explain the idea in more depth, we need to check the implementation in details:

There are different constructors for McInt, for different usages.

The first constructor takes in a McSet with McStates in it, and construct a McInt in such way:

1. check to find the biggest state number in the set, we note it as \( n \).
2. allocate(using new) a char array, which has \( \text{int}(n/8)+1 \) byte units.
3. consider the char array as memory chunk, we start to read the state numbers.
4. For a state with number \( a \), we mark the No. \( a + 1 \) digit from right to the left in the memory chunk “1”.
5. After looping through the McSet, the McInt contains the information of the McSet of McStates.

The reason we want use a McInt to represent a set of McStates is the consideration of efficiency benefits brought by this idea of minimal information representation. In many algorithms, we need to do recursively comparison between set of McStates. Let’s take a look at why McInt is better in comparison:

For example, we have a set 0, 1, 5, and we want to compare it with 2,3,4. In most situations and systems, the sets are not sorted. We can either sort them and do at least one integer comparison, or we need do multiple integer comparisons. However, if we do the liner complexity transformation and represent them as a McInt, the comparison will take place between: 0x00100011 and 0x00011100, which are two chars, It is significantly better! The advantages hold for most cases, unless we are dealing with sets like 9999999,10000000. In the case of small set containing big numbers, McInt is not a good choice as it takes so much memory.

The second constructor transfers a String to a McInt by direct memory copy. Note that the raw bit representation of std::string in C++ is not obviously associated with the content of the string, we need to use the string::c_str() method and copy the memory chunk from the string’s C string representation.

The benefits of representing a string as an McInt is primarily in the comparison process. The default comparison between strings in C++ is done by comparing two string char by char, but McInt compare length first, if the memory chunk length is same, McInt falls back to use the default comparison of memory.

[API]

\[
\begin{array}{|l|}
\hline
\text{McInt(const McSet<McState>&)} \\
\text{Transfer a McSet of McStates to McInt} \\
\text{complexity(growth order): linear with the number of elements in the Set.} \\
\hline
\end{array}
\]

\[
\begin{array}{|l|}
\hline
\text{McInt(const std::string&)} \\
\text{Initialize a McInt from a String.} \\
\text{complexity(growth order):1} \\
\hline
\end{array}
\]

4.3 Theory Objects

Theory objects refers to the data structures and classes that implements the concepts in computation theory, such as languages and machines. Theory objects are implemented based on the data structures in the backbone. In MachineCat, theory objects are implemented with careful consideration of both program efficiency and logics in theory of computing.

4.3.1 McState

This class represents the object “state” in a machine. It used to be composed with an integer state number and two boolean variables, indicating whether it is a start/final state. Later the author realized that a single state could not
really has start nor final, because start and final is only meaningful when we are talking about states in machines. So currently McState is as simple as a class wrapping an integer and providing useful interfaces.

Functions like set and get state number are implemented mostly with in-lined functions. The comparison between McState is eventually the comparison of the state number.

```cpp
int getNumber() const  
get the number of the state.
complexity(growth order): 1

void setNumber(int n)
Set the number of the state.
complexity(growth order): 1
```

### 4.3.2 McTransition

This class implements the concept of a transition in finite machines. Its protected members are two McStates and one integer as the label of transitions. There is no templates implementation for the transition labels because the symbol table system is established to translate any finite machine to a new one with integers as transition labels. This guarantees fast and stable machine processing in MachineCat. Resulted output will be translated back to the labels used in the original machine with the same table.

The comparison between two transitions are carried as such: first we compare the start McStates, if they are not equal then we return the comparison result. If they are equal, we compare the labels. If the labels are not equal, we return the comparison results. If the labels are still equal, we return the comparison results of the end McStates in the McTransitions.

```cpp
McTransition(int, int, int)
This constructor is a convenient way to initialize a McTransition with 3 numbers, which will be used as the start, label and final.
complexity(growth order): O(1)

void null()
Set the transition to NULL start, NULL end, NULL label.
complexity(growth order): O(1)

void reverse()
Reverse the transition.
complexity(growth order): O(1)

bool sameStartAndLabel(const McTransition&)
Compare if the parameter transition has the same start and label as this one.
complexity(growth order): O(1)
```

### 4.3.3 McNFA

McNFA is the class modeling a non-deterministic finite machine. McNFA is one of the most important classes in the entire project. Many algorithms essentially work with McNFA. This class contains these variable members:

```cpp
protected:
McSet<McState> startStates;
McSet<McState> finalStates;
McSet<McTransition> transitions;
bool isAttached;
McMap<McState, McSet<McTransition>> transitionsByState; // the dictionary
// in which transitions are attached to keys
```
//to rename states like "1 3 5" to "0 1 2"
bool statesRenamed;
McArray<int> statesMapping;

Other than the start and final states as well as the transitions, McNFA also contains a Boolean member “isAttached” and a dictionary(McMap) member “transitionsByState”. When calling McNFA::attach(), all the transitions in this NFA will be copied into the dictionary, into various sets under different keys equal to the start states. After attaching, it is very easy to access the transitions started from any state by:

```cpp
McSet<McTransition>& getTransitionsStartFrom( const McState&) {...}
```

Please note that MachineCat will not delete the original transitions when attach(), and unattach() simply clear the dictionary.

Two other interesting members are “stateRenamed” and “statesMapping”. A McNFA can rename its states if there is gaps between state numbers. Calling McNFA:rename() will do such operations:

```
(START) -| 0
0 a 3
3 b 5
5 |- (FINAL)
```

will be renamed to

```
(START) -| 0
0 a 1
1 b 2
2 |- (FINAL)
```

and we have “statesMapping” as

```
0 0
1 3
2 5
```

It is useful when some algorithms need canonical numbered machines to function. This operation, however, is not required in most cases, because MachineCat’s architecture by default allows non-canonical numbered machines be processed correctly.

One of the highlights in the design of McNFA is the input system. As shown in appendix(section 11), a deterministic finite machine is hard coded into the input system for McNFA. This finite machine takes the input stream char by char and assemble the input McNFA in memory transition by transition. The benefits of such design is obvious: the speed is very fast, and the compatibility for errors is great because we design the machine to allow ambiguity. For example:

```
strt- 0
0 apple 1
1 |eNd
```

is taken into MachineCat and transferred into a McNFA correctly as if the input is:

```
(START) |- 0
0 apple 1
1 |- (FINAL)
```
So we are less likely to run into annoying situations when we fail a test just because one in-obvious space was missed in input.

**[API documents]**

```cpp
void reachable_states(McSet<McState>&);
This function takes in a set and filled in with all the reachable states in this Machine.
complexity(growth order):\(O(n \log(n))\)

void reachable();
This function will remove all the un-reachable states in the finite machine as well as the transitions associated with them.
complexity(growth order):\(O(n \log(n))\)

McNFA& reverse();
This function reverses the finite machine.
complexity(growth order):\(O(n)\)

McNFA& select(const McState&, int, McNFA&) const;
creates a sub McNFA (third parameter) by selecting transitions from this McNFA with a given state (first parameter) as the source or sink state (depending on the value of the second parameter which can be SOURCE, SINK or EITHER). The function returns a reference to the new McNFA.
complexity(growth order):\(O(\log n)\)(sorted) to \(O(n)\)

McSet<McState>& sinks(McSet<McState>&) const; McNFA& reverse();
This functions gets all the end states of transitions
complexity(growth order):\(O(n)\)
```

### 4.3.4 McDFA

This inherits from McNFA. Basic operations are similar, some functions like addTransition() is overrided to make sure we do not make a dfa become nfa when adding transitions. Note that the copy constructor which takes a NFA as parameter simply copy the data from the nfa, so it is the programmers responsibility to make sure the parameter dfa is determined.

**[Useful API]**

```cpp
bool addTransition(const McTransition&);
dd a transition, each transition to be add will be checked to make sure this is still a DFA.
complexity(growth order):linear with the size of this DFA.

bool getDestination(const McState&, const int, McState&);
Find the sink states, given the start and transition Label. If found the sink state is stored in the third parameter and true is returned. Other wise false is returned.
complexity(growth order):linear with the size of this DFA.
```

### 4.3.5 Symbol Table Design

One of MachineCat’s smart designs is that it will accept anything as the middle labels of transitions, and will be able to use those labels to make strings and languages. While internally, MachineCat only use integer as the labels of transitions. This guarantees efficiency and stability.
To achieve this, we construct symbol table at the input stage. What happens is that when MachineCat takes in transitions, it will transfer the label as a string to a McInt; if the McInt is in our symbol table, we use the index of the McInt in the symbol table as the internal label for this transition; otherwise we add the McInt first and then use its index as the internal label for this transition.

The original label, is stored in a dictionary associated with the symbol table that use McInt as entry. So we can retrieve the internal numbers to its original form at output stage.

[API]

```cpp
text

int getSymbolTableIndex(const string& str)

Takes in a string, if this string is not in symbol table, make a McInt from it, store the McInt in the Array of the symbol table and stores the string in the dictionary with the McInt as the entry key. The index of the McInt in the array is returned.

complexity(growth order): linear with the size of the symbol table.

const string& indexToString(int index)

Get the string back, with the index used as the label of transitions internally.

complexity(growth order): Logarithmic in size.
```

4.3.6 McInt and McLanguage

Because of the design of the symbol tables system, the string in MachineCat is essentially an array of integers. The initial implementation is as simple as:

```cpp
#ifndef MCSTRING
class McString: public McArray<int> {
  protected:
    public:
};
#define MCSTRING
#endif
```

and the language in MachineCat is just an set of McString:

```cpp
#ifndef MCLANGUAGE
class McLanguage: public McSet<McString> {
  protected:
    public:
      //Constructor family
};
#define MCLANGUAGE
#endif
```

The input and output of McString and McLanguage is closely associated with the symbol table. When user input one string, the string is analyzed, the new basic “labels” in the string is transfered to McInt and added to the dictionary of the symbol table, and the string is represented as an array of integers, which are the values of the indexes of the McInt in the Array of symbol table. When user output an string or language, the integers are compared against the symbol table and translated back to the original values. Because the “labels” in MachineCat can be anything, so we add spaces in McStrings: “ABABAB” in Grail+ with chars will be represented as: “A B A B A B” in MachineCat.
4.4 API Support

While implementing MachineCat with our own thoughts and design, we also tried to implement or integrate many of the interfaces available in Grail+’s backbone system. With current architecture of MachineCat, we are already able to transfer most of the algorithms from Grail+ to MachineCat, with very little modifications.

It is in the plan for MachineCat to support algorithms from packages other than Grail+, which means more work will be done to implement more interfaces from more packages.

The ultimate goal for MachineCat is to have a strong API support system, that not only provides MachineCat styled API for algorithms designed specific with the consideration of MachineCat’s features, but also supports many interfaces implemented in other packages, so that we will be able to take advantages of the implemented algorithms in other packages with very simple transportation.

5 Implemented Algorithms

Here we introduce some of already implemented algorithms. Please note that it is extremely easy to add, modify and delete algorithms in MachineCat, so please refer to the project to see the latest algorithms implemented.

5.1 McMachineAnalyze

This is a test algorithm added when initializing the system, it does little things, but provides a place to put test code as you want.

This algorithm Takes in a pointer of McNFA, thus by polymorph-ism could be applied to add object of classes inherited from McNFA. For now, this algorithm only output the size of the finite machine.

```cpp
void McMachineAnalyze(const McNFA& fm)
This function output information of this nfa.
complexity(growth order): depends
```
[source]: I made up it.

5.2 McNFAtoDFA_Set

Debug Trigger defined as:
#define DEBUG_ McNFAtoDFA_Set 1

This performs a transformation from a NFA to DFA with the power set construction algorithm. It takes in a NFA to transfer from and a dfa reference to store the result. If the “-d” or “-da” is provided, the set information in the process will be presented.

```cpp
void McNFAtoDFA_Set(McNFA& nfa, McDFA& dfa)
Transfer the first McNFA to a McDFA, store the McDFA to the second parameter.
Complexity(growth order): Up to 2^States number in nfa)
```
[source]: The power set construction algorithms [8]

5.3 McDFAMinimize_TBF

Debug Trigger defined as:
#define DEBUG_ McDFAMinimize_TBF 4

```cpp
void McDFAMinimize_TBF(McDFA& dfa);
Takes in a dfa and minimize it with table filling algorithm.
complexity(growth order):average O(n log n), worst O(n^2)
```
This takes in a DFA and minimizes it with the table filling algorithm, which makes an equivalence table and looping until no more distinguishable states can be found. The passed in DFA will be changed.

Table filling Algorithm.

5.4 McNFAEnum

This algorithm enumerates strings accepted by an NFA.

```
int McNFAEnum(McNFA& nfa, McArray<McString>& words, int max_num, int max_length)
```

Enumerate strings in nfa and store them in the McArray words, by lexicographical order. max_num is the maximum number of words we want, max_length is the maximum length of words we want. The program stops when either of the max value is reached, or the end of the language.

"Complexity (growth order): linear with the number of strings in McArray\"words\"."

5.5 More algorithms...

With current approach, algorithms in MachineCat are completely modularized (see 6), thus can be added and removed quickly and easily. Please check the package for most up-to-date algorithms implemented.

6 Filters and Main Function

We have brought MachineCat's modularization feature to a new level that provides extreme convenience for adding filters and algorithms. When the users need to add in new algorithms or filters, then simply need to make the code pieces for the new algorithms and filters. MachineCat is designed to automatically pick up new algorithms and filters code put into specific folders.

This is great because we can treat algorithms and filters as modules, plug into or unplug them from MachineCat by simply dropping in the code pieces or delete them, no other code needs to be changed. For example, see section 7.

The following discuss how is the task accomplished in MachineCat.

6.1 Meta-Programming with C++

Meta programming basically means some program A is generated by B which manipulate A's code as its data. It is very common in scripting languages like JavaScript. C++ by default does not provide native support for any meta-programming techniques, as it is not usually needed in the circumstances where C++ is used.

However, in MachineCat, in order to automatically pick up any new algorithms dropped in the algorithms folder, the header file McAlgorithms.h in the folder MachineCat/Algorithms/ needs to be updated. To automatically generate new filters, the header file MachineCat/ Filters/ Filters.h needs to be updated; the function pointers to the new filter functions need to be loaded in Main(), and the symbolic links (the "filters") need to be made and put in the right folder.

C++ can not do self-modification at run-time, so in MachineCat, the ultimate solution is:

 Configure the program and re-write some code pieces with scripts before we compile. Currently Bash is used for Linux system.

6.2 Auto-Collection: Algorithms

When the user navigate to the root folder MachineCat/ and enter "make", the script “AlgorithmCollector” in MachineCat/Algorithms/ is executed. Here is the contents in that script:
What it does is:

1. Remove the old header file.
2. Generate the code to include all the “.h” files in the new McAlgorithms.h.
3. Generate the code to include all the “.cpp” files in the new McAlgorithms.h.
4. Done, including new algorithms is very easy.

6.3 Auto-Collection: Filters

Immediately after the process of algorithms collection, the filters collection and generation script is executed:

1. Remove the old “Filters.h” file.
2. Include all the header files of the filter functions into the “Filters.h” file.

The following tasks are carried out:

1. Remove the old “Filters.h” file.
2. Include all the header files of the filter functions into the “Filters.h” file.
3. Include all the “.cpp” files of the filter functions into the “Filters.h” file.

4. The script then generates a function called load_filters( ), in the following Way:

(a) Write the beginning of the function:

```cpp
map<string, void (*)()> mc_filters;
void load_filters()
```

(b) For each filter function, find its header file “BLABLA.h”, and generate a line like this:

```cpp
mc_filters["BLABLA"]=BLABLA;
```

This assume the name of the filter function is same as its header file, and add the function pointer to the filter function into the map “mc_filters”.

(c) Close the function by writing “}”.

After algorithms collection and filters collection, the entire project is compiled with g++.

### 6.4 Generate Symbolic Links

After compilation, the script MachineCat/Filters/FilterLinker is executed:

```bash
#!/bin/bash
#add the filters
for i in 'ls */*.h';
do ii="$i/../\""; 
   ln -s ../MainFunction/a.out ../Bin/${ii/%.h/}
done
```

This script search for the header files of all the filter functions, for each header file “BLABLA.h”, it creates a symbolic link called “BLABLA” in MachineCat/Bin, which links to the a.out file in MachineCat/MainFunction/.

Again, the name of the header files determines the name of the filters.

### 6.5 Why Separate Algorithm and Filter

Here is an example to illustrate the reason:

In filter fnmin_tbl, we want to invoke the table filling algorithm to minimize the dfa inputted by user, but we want to make sure the input machine is a dfa, so here is the code:

```cpp
void fnmin_tblf()
{
   McDFA dfa;
   ...
   IO process 
   ....
   //if this a real dfa?
   if (!McIsDeterm(dfa))
   { 
      cout << "Not a dfa!" << endl;
      exit(1);
   }
   //invoke the table filling algorithm
```
McDFAMinimize_TBF(dfa);

If we make algorithms and filter in one piece of code, then that piece of code has to also handles IO, then we can not use “McIsDeterm” in the table filling filter any more, instead we have to re-code it in, which brings duplicates and difficulty to maintain the project. So separating filters and algorithms is just to make things clear: to split the real algorithms and the householder-code like IO; also we have more flexibility with this approach. We can code pure algorithms without thinking about other stuff. Then we can make filters to take advantage of the algorithms in any way we want.

6.6 Main Function

The main function is very simple:

```cpp
#include <iostream>
#include <sstream>
#include "../MachineCat.h"

/*
 * General variables
 */
//all the arguments
McArray<string> args;
//if the -info flag exits
bool show_info = false;
bool specific_debug = false;

#include "../Filters/Filters.h"

int main(int argc_in, char** argv_in)
{
    //load all the filters
    load_filters();
    // the link name
    string filter_name(argv_in[0]);
    //remove leading "." if there is one
    if (filter_name.find(".") == 0)
    {
        filter_name = filter_name.substr(2);
    }
    //record the rest arguments
    for (int a = 1; a < argc_in; a++)
    {
        args += string(argv_in[a]);
    }
    //the debug flag -d and -da
    if (args.member("-d") >= 0)
    {
        cout << 
```
specific_debug = true;
args -= "-d";
}

//open the info page via browser
if (args.member("--infoPage") >= 0)
{
    show_info = true;
    args -= "--infoPage";

    //open the local link and exit
    char* prog[3];
    prog[0] = (char*)"/usr/bin/xdg-open";
    stringstream ss;
    ss << "../Filters/";
    ss << filter_name;
    ss << "/";
    ss << filter_name;
    ss << ".html";
    prog[1] = (char*) ss.str().c_str();
    prog[2] = (char*) '\0';
    execvp(prog[0], prog);
    exit(0);
}

//open the info page via more
if (args.member("--info") >= 0)
{
    show_info = true;
    args -= "--info";

    //open the local link and exit
    char* prog[3];
    prog[0] = (char*)"more";
    stringstream ss;
    ss << "../Filters/";
    ss << filter_name;
    ss << "/";
    ss << filter_name;
    ss << ".html";
    prog[1] = (char*) ss.str().c_str();
    prog[2] = (char*) '\0';
    cout << endl << "You can also try the flag --infoPage to see this html file with your default web browser!" << endl;
    execvp(prog[0], prog);
    exit(0);
}

//version flag
if (args.member("--version") >= 0)
{
    cout << "Version 1.0" << endl;
    exit(0);
}

//help flag
The main function does the following things in order:

1. Call the function `load_filters()`, which is assembled by the “FilterCollector” and collect all the function pointers pointing to the filter functions.
2. Find the filter name used to call the function.
3. Handle the flags that are available for all the filters.
4. Invoke the associated filter function.

Compared to Grail+, MachineCat has a very short and organized main function, which makes the logic very clear in the code and also accelerate the time to load the algorithms when a filter is used. With the careful designed filters which generates code and header files, we do not need to change the main function at all when we add algorithms or filters. With a map of function pointers to filters, we completely abandoned the old ugly way to use “if else” blocks.

6.7 Implemented Filters

The filters, as mentioned before, is householder functions to handle IO an calls algorithms. Filters are completely modularized in MachineCat and can be added/removed easily. For the information about algorithms, see section 5. Some bench marks are available for part of the implemented filters in section 8.2. For most up to date implemented filters, please refer to the package and use “--infoPage” (see section 2.2.2).

7 Programming Guide

7.1 Guide: How to Add a New Filter

Programmers are able to code their own pieces of algorithms and filters without changing any existing parts of MachineCat. Simply drop the new code in the right folders and re-make the project, MachineCat will automatically pick up new algorithms and add entry for new filters.

This means there can be multiple people making several different new algorithms/filters for MachineCat, and they can simply put their work together into MachineCat and MachineCat will pick them up, clean and safe.

Removal algorithms and filters will be as simple as removal of associated code. No changes need to be done to original MachineCat, either.

7.1.1 Add New Algorithm

Add a new algorithm into MachineCat is very simple. Now We use an example to illustrate the detailed process to accomplish such task. Say we want to add a new algorithm, called “McMachineAnalyze” which simply takes in an non-deterministic finite machine and out put the size. Following is the steps:

1. Goto the folder MachineCat/Algorithms/, and make an folder called “McMachineAnalyze”, inside which we create a file called “McMachineAnalyze.cpp”.

```cpp
if (args.member("--help") >= 0)
{
    cout << "Please use -info to see all the help information" << endl;
    exit(0);
}

//execute the filter
(*mc_filters[filter_name])();
return 0;
```
2. Open the file “McMachineAnalyze.cpp”, and code the function. The function can implement any algorithm we want, the code inside this function can call any functions that is in MachineCat library or any other function that is already implemented in the Algorithms/ folder.

For this easy example, we code the function as such:

```cpp
//This algorithm is to analyze a finite machine
using namespace std;

void McMachineAnalyze(const McNFA& fm)
{
    cout << "Analyzing DFA:" << endl;
    McSet<McTransition> ts = fm.getTransitions();
    cout << "This machine has" << ts.size() << " transitions" << endl;
}
```

Note that all the functions in the algorithms folder will be a global function. MachineCat is designed in such way that none particular algorithms need to be put into any theory object class as its member function unless the algorithm logically belongs to the theory object. This design is more logically intuitive because algorithms uses theory objects, not the other way around. Make algorithms global also helps to make smaller theory objects class and keep the project modularized.

3. Besides the cpp file, add a file called “McMachineAnalyze.h”, in which we add the declaration of the function that we implemented:

```cpp
//Algorithms set : The analysis of finite machines
//the methods declaration
void McMachineAnalyze(const McNFA&); //Analyze a FA
```

documents.

4. In terminal, Navigate to the folder MachineCat/ and re-make. MachineCat will automatically pick up the new algorithms, re-generate include files and add the function to the library.

5. All done! So easy!

* Please note that it is a good practice to keep the names of the function, the header and cpp file as well as the folder holding them the same, as some “meta-programming” process during the compilation may need to gather information from the file names.

### 7.1.2 Add New Filter

A filter is an easy way to invoke the algorithms we implemented in MachineCat, to enable fast everyday test!

Now we assume we want to add a new filter, which will invoke the algorithm “McMachineAnalyze” and output the result.

Steps:

1. Open the folder MachineCat/Filters/, inside which we add a new folder called “fmanalyze”. In the newly created folder, add a file “fmanalyze.cpp”

2. Implement the filter in the cpp file, usually the filter simply handles I/O and call the associated function:

```cpp
void fmanalyze()
{
    McNFA nfa;
    std::cout << "enter nfa" << std::endl;
```
```cpp
std::cin >> nfa;
McMachineAnalyze(nfa);
```

Please note that the filter function must be in the format: `void function_name(void)`, to keep the consistency of function pointers. This is required for MachineCat to pick up the filter.

3. Beside the cpp file, create a file named “fmanalyze.h”, in which we code the declaration of the filter function:

```cpp
void fmanalyze();
```

4. Re-make the project, MachineCat will automatically pick up this filter code and generate a new filter in the MachineCat/bin directory.

5. All done, you now have a new filter! Please remember to also put an simple HTML file beside it to hold the information about the filter.

* Please note that it is a must to keep the names of the function, the header and cpp file as well as the folder holding them the same, as the auto filter collection process does depend on them when it comes to filter generation.

### 7.2 Program with MachineCat as Basement

Using MachineCat as basement is very simple. The programmer can include the entire package simply by including the `MachineCat.h` header file in the root folder of the package. Programmers need to check the interfaces implemented in each class and algorithms coded for different usages.

Recommended IDE with MachineCat is Eclipse for C++. In fact the entire package was developed with Eclipse. The auto-completion feature in Eclipse is very useful when developing with a large package.

As this package is in standard C++ with clear and simple Makefile, many other IDE should also work properly.

It is clear that in the near future, MachineCat will be compiled into a real library and make it even easier to include.

### 7.3 Changes to BackBone and Other Core Components

The BackBone, TheoryObjects and other supporting components should not be changed unless approved, because the changes in these parts will not automatically merged by MachineCat.

### 8 Comparison and Benchmarks

In this section, I compare MachineCat with Grail+, the targeted role model and supposed competitor of MachineCat, in different aspects regarding general features and performance.

#### 8.1 General Features Comparison

8.1.1 The Good

Although MachineCat has not been fully implemented, it already integrated a number of great features which can not be found in any others including Grail+. We provide some selected typical instances:
| Feature                  | MachineCat                                                                 | Grail+                                                                 | MachineCat can support any inputs of continuous blocks as the transition labels, and process them rapidly. Example:

```
MachineCat/Bin$ ./fmenum
enter nfa
(START) |- 0
  0 APPLE 1
  1 PEAR 2
  2 PEACH_TASTE_THE_BEST 3
  3 I_EAT_"ALL"_ANY_WAY 0
  3 -| (FINAL)
enter max length
3
enter max num
1
done!
APPLE PEAR PEACH_TASTE_THE_BEST
```

MachineCat takes in any continuous block as the transition label. Internally everything label is taken in as string and transferred into a McInt. Both of them will be stored in the symbol table and the indexes will be used as the internal transition labels. This means the machines in MachineCat is always using ints as alphabets. This makes MachineCat stable and efficient.

Grail+ supports different inputs, too. But Grail+ achieve this using generic programming skills and templates. We need re-compile everything if we want to switch from machine with “ints” as alphabets to machine with “strings” as alphabets. Currently, only the “char” mode is constantly used and tested in UPEI. Generic programming and templates make its harder to program in Grail+ as we need always think about the different possible instantiation of the templates.

<table>
<thead>
<tr>
<th>Debug System</th>
<th>MachineCat has a precise debugging system that can be customized agilely to output any specific information in program execution. Details can be found at section 4.1</th>
<th>Grail+ does not provide debug system by default, we have to add debug output manually. Although efforts was made at UPEI trying to add debug system into Grail+, the system can not provide the precise debug output of a certain scope as MachineCat does with rich debug triggers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Structure</td>
<td>MachineCat is built up in a very strict hierarchy that actually make sense, No(or very little) circular dependency exists.</td>
<td>Grail+ does not have a clear hierarchy of classes and algorithms. Circular references are common in Grail+.</td>
</tr>
<tr>
<td>Cont’d</td>
<td>MachineCat</td>
<td>Grail+</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>Modularization</strong></td>
<td>MachineCat is highly object-oriented and modularized. Same as most object oriented software, we can easily change any single class in the package. As long as the function interface is kept the same, updating any single class will not affect other pieces of code. Moreover, the best part in modularization of MachineCat is achieved by involving the “meta programming” idea. Several well designed bash scripts are made to detect code files in certain folders and generate the code to add entries of them into the project. With the help of the scripts, the algorithms and filters in MachineCat becomes completely independent modulars. We can drop the code pieces of algorithms and filters into the project and re-compile to install them, and remove them simply by deleting their folders and re-compile. With the efforts to enhance modularization, the entire project is extremely clean and clear. The piece of code for main function in MachineCat is as short as a page and always stay the same regardless of the newly implemented algorithms and filters. While in Grail+ the main function is huge, inelegant and has to be changed to make entries for any new filters.</td>
<td>Grail+ is also object oriented. But Grail+ has a huge main method with tens of if statements. Adding a filter or an algorithm into Grail+ requires modification in many pieces of files, including the actual code of the filter and the algorithms as well as the “main method”; associated header files; several “Makefiles” and so on.</td>
</tr>
</tbody>
</table>

| **Code Style & System Supporting** | MachineCat is based on standard C++, most of the data structure wraps the associated data structure in standard C++ library. This is good because most of the part can automatically upgrade with the upgrading of C++. Non third party library was used at all, not even boost lib. MachineCat does not intend to specify any platform, it should be able to run in any platform supporting standard C++. The only system-specific requirement is that we need to provide the equivalences of the bash scripts for algorithms and filters collection and linkage, if we want MachineCat to fully keep its highly modularized feature. | Original was developed before C++ standards were released. It now conflicts with the standard library, and there are lots of redundant code in it, which was originally made to support various of very old system types, some of which are not used widely these days. |

### 8.1.2 The “Need More Work”

MachineCat is far from its ultimate target to be one of the best theory of computing package. A lot of features in Grail+ is implemented in MachineCat, but still many of them are not yet in.
Some components of the core backbone could further optimize for both performance and stability. More details please refer to Section 9.

8.2 Tests and Benchmarks

MachineCat is an individual project and has not been fully completed due to the size of the possible features set and the strict requirements of both architecture and performance. I only compare the features that is already in MachineCat.

All the tests were carried out in a virtual machine which occupies 3 threads from a Intel Core i7-2670QM@2.20GHz CPU which provides 4 cores 8 threads. The memory allocated to the virtual machine was 2GB. The guest system is Ubuntu 10.04,32 bits. The host is Win7, 64 bits.

8.2.1 Input Speed Tests of Finite Machines

The input system is crucial when we have to take in very big finite machines and test on them. The duplication check is done after the system takes in a entire machine. Here is the process to remove duplicates in Grail+:

1. Takes in all the lines of transitions.
2. Sort all the lines.
3. (Remove duplicates by checking continuous lines.)

The complexity order is \( O(n \log n) \), due to the sorting process.

In MachineCat, the finite machine is taken in using a hard-coded finite machine which handles input. No sort is done after taking in a machine.

We did the tests for finite machines of \( 2^0, 2^1, 2^6 ... 2^{20} \) lines with each line starting by a different state. and obtained the results in Figure 2. In this test statistics, Grail+ and MachineCat are very comparable in speed. In some of the previous random tests, MachineCat’s input system is better or at least as good as the input system of Grail+; the hard-coded finite machine handles input efficiently with more compatibility to potential errors.

Note that xaxis (the number of states) increases exponentially.

8.2.2 Speed Tests of Nfatodfa

There are two nfatodfa algorithms in MachineCat, one uses exact same implementation as it is in Grail+, the other apply McInt.

Tests were made to take in finite machines of \( 2^0, 2^4, 2^6 ... 2^{20} \) lines, with each line starting by a different state, and do a nftadfa. The result is in Figure 3. Although the internal work is different and MachineCat cares more on stability and code clarity, with same algorithm, MachineCat is as fast as Grail+.

8.2.3 Speed Tests of Deterministic Check

The task is to check if the inputed finite machine is a nfa or a dfa.

Tests were made to takes in finite machines of \( 2^0, 2^4, 2^6 ... 2^{20} \) lines, with each line starting by a different state, and check if they are deterministic. The results are in Figure 4. Grail+ and MachineCat uses different algorithms to test determinism. Basically Grail+ looks at each state and check if there are two or more destinations with one transition label from this state, the process goes on until it finds the proof of non-deterministic or reaches the end of the states set. MachineCat takes all the start states number and the transition labels and construct an array of pairs, sort it and check if there are two equivalent adjacent pairs, if such pairs are found the machine is non-deterministic.

The performance is very close with both algorithms, I assume it is because the two algorithms essentially do the same kind check.
8.2.4 Speed Tests of Completion Check

Figure 5 is the test results to check if inputted DFA is completed. Grail+ checks the completeness of a finite machine with the same approach as it checks whether a machine is deterministic. For each state, Grail+ selects all the transitions with current state as start, and see if those transitions have involved all the alphabets. If so, Grail+ starts to check the next state until it goes to the last state, otherwise we return a false, which indicates the machine is not completed.

MachineCat has implemented two algorithms to check completeness of a machine. The first one, associated with \texttt{filter fmiscomp} is essentially doing the same operations as Grail+. From Figure 5 we can see that the performance is also very close. The second algorithm is a little different. The algorithm called by \texttt{filter fmiscomp2} simply count how many unique pairs of start state and transition label exist in the machine, if the number is equal to \( \text{(states\_number \times \text{alphabets\_number})} \), the algorithm returns true, which indicates the machine is completed. The second algorithm is faster, especially when the machine is indeed completed.

8.2.5 Speed Test of Enumeration

Figure 6 is the test results to enumerate 100 words with finite machines of \( 2^0, 2^4, 2^6 \ldots 2^{20} \) lines, with each line starting by a different state. The algorithm in MachineCat was imported from Grail+, after some modification, it works greatly with the backbone of MachineCat. This algorithm develop words fragment lexical graphically, until we have enough words or no more words to enumerate. The performance of two packages are comparable.

8.2.6 Speed Test of Union

Figure 7 is the test results to enumerate 100 words with finite machines of \( 2^0, 2^4, 2^6 \ldots 2^{20} \) lines, with each line starting by a different state. This is, in both package, a direct call to add all the content in one machine’s set of transitions to another one. MachineCat takes a extra parameter, which indicates if we need to check duplicates afterwards.

9 Future Work

MachineCat is Rui (Ray) Zhou’s honors project for my 2011-2012 academic year, but it is more than that. Dr. Câmpeanu and Ray will continue to work on it, and we are looking forward to see it become a successful package.

9.1 More Core Features

Here is a list of more core features to implement. These features will greatly enrich MachineCat.

<table>
<thead>
<tr>
<th>Features To Implement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure: McList</td>
<td>The list is data structure is not yet implemented in MachineCat, as for most of the algorithm, the speed of accessing certain elements is more important. However the list structure can be very useful for certain situations when insertion and deleting is the primary operations. As an major data structure in computer science areas, it will be good to have it in MachineCat.</td>
</tr>
<tr>
<td>Theory Object: McReg-Exp</td>
<td>Grammar and expressions are other important portion of Theory of computing. It is for sure that MachineCat will support the computation for them, but a careful design and implementation need be done. This will be the first component to add after my honor’s presentation.</td>
</tr>
<tr>
<td>Theory Object: Mc-Transducer</td>
<td>Transducer is a different finite machine that outputs information during the states transitions. Grail+ does not support transducer, but MachineCat will implement it soon.</td>
</tr>
</tbody>
</table>
### 9.2 Promising Features of Interests

Here is a list of promising features to add, those features may not be necessary, yet could be very useful.

<table>
<thead>
<tr>
<th>Promising Features</th>
<th>Description.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile MachineCat as library</td>
<td>MachineCat has a very clear hierarchy in its structure and the class dependency relations are all logically set up. It will be very easy to compile the entire project as a library.</td>
</tr>
<tr>
<td>Graphical tool for finite Machine(2D)</td>
<td>During the previous two summers when I was working on Grail+, I made a Java tool, called “drawfm.java” which takes in finite machines and present it with colored 2d machine diagrams. Users can also drag the states around in the image. I would like to integrate this tool into MachineCat, and make it co-operate with other components.</td>
</tr>
<tr>
<td>Graphical tool for finite Machine(3D)</td>
<td>Similar to the 2D tool, I also made another piece of code which takes in finite machine in text form and transfer it to 3D image with OpenGL. Integrating this tool will be challenging and interesting.</td>
</tr>
<tr>
<td>Enriched backbone to support code(algorithms)</td>
<td>Besides its own well designed APIs. MachineCat can provide more APIs which can be found in other packages in MachineCat’s backbone. So it will be very easy to do transportation of implemented algorithms from other package to MachineCat.</td>
</tr>
<tr>
<td>IO compatibility with different packages.</td>
<td>When I have time I hope to make a filter in MachineCat to transfer machines between different formats supported by different packages.</td>
</tr>
<tr>
<td>Other Great Features.</td>
<td>Anything that can be useful and interesting can be integrated.</td>
</tr>
</tbody>
</table>
10 Acknowledgement

Thanks To:

Dr. Cezar Câmpeanu,
For being my reliable mentor and guiding me over the past two and half years with great passion and great sense of responsibility.

The Computer Science Department,
For changing my life and leading me to a new world of Computer Science.

My Friends and Family,
For always being there for me, during the good and the hard times.
References


Appendix

11.1 Input System of McNFA

```cpp
//MCNFA
inline istream& operator>>(istream& is, McNFA& nfa)
{
    /*
    string temp;
    while (true)
    {
        temp.clear();
        getline(is, temp); //get a line
        if (temp.find("START") != string::npos)
        {
            /*
            string temps(temp.substr(temp.find("|-") + 2));
            temps >> s;
            nfa.addStart(s);
        }
        else if (temp.find("FINAL") != string::npos)
        {
            /*
            string temps(temp.substr(0, temp.find("-|")));
            temps >> s;
            nfa.addFinal(s);
        }
        else if (temp.find(EOF) != string::npos)
        {
            break;
        }
        else if (temp.length() == 0) /// skip empty line
        {
            break;
        }
        else
        {
            McTransition t;
            temp >> t;
            nfa.getTransitions().disjoint_union(t);
            //we can use nfa.addtransition(), but that check for duplication, kind slow
        }
    }
    nfa.getTransitions().unique_array();
    //nfa.unqiuTransitionsByAttach();
    */

    //read in the finite machine, using a finite machine
    string state1("*");
    string label("*");
    string state2("*");
    int current_state = 0; //initial state
    char c;
    while (is.good())
    {
        c = is.get();
    }
}
```
//cout << current_state;
//cout << c << endl;

//for comments

switch (current_state)
{
  case -3:
    switch (c)
    {
      case '#':
        current_state = 0;
        break;
      default:
        //ignore some token
        break;
    }
    break;
  case 0:
    switch (c)
    {
      case '{':
      case 'S':
      case 's':
        current_state++;
        break;
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
        current_state = 3;
        state1 += c;
        break;
      case '#':
        current_state = -3;
        break;
      default:
        //ignore some token
        break;
    }
    break;
  case 1:
    switch (c)
    {
      case 'S':
      case 's':
      case 'T':
      case 't':
      case 'A':
      case 'a':
      case 'R':
      case 'r':
      case ')':
      case '|':
      case '-':
      case '(':  
      case '#':
        break;
    }
    break;
}
case '1':
    //LOL
    break;

case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
    current_state++;
    state1 += c;
    break;

default:
    current_state = -1; //error
    break;

break;

case 2:
switch (c)
{
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
    state1 += c;
    break;

default:
    nfa.addStart(atoi(state1.c_str()));
    state1.clear();
    current_state = 0;
    break;

break;

case 3:
switch (c)
{
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
    state1 += c;
    break;

case '|':
case '−':
    nfa.addFinal(atoi(state1.c_str()));
    state1.clear();
    current_state++;
    break;

case '(':  
    current_state = 5;
    break;

default:    
    current_state = -1;
    break;
}
break;
case 4:  
    switch (c)  
    {
    case 'I':
    case 'i':
    case 'N':
    case 'n':
    case 'A':
    case 'a':
    case 'L':
    case 'l':
    case '(':  
    case ')':
    case '|':
    case '-':
    case ':':
    case '−':
    break;
    default:  
    current_state = 0;
    break;
}
break;
case 5:  
    switch (c)  
    {
    case '|':
    case '−':
    nfa.addFinal(atoi(state1.c_str()));
    state1.clear();
    current_state = 4;
    break;

case '(':  
    break;

default:  
    label += c;
    current_state++;
    break;
}
break;
case 6:
switch (c)
{

case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
    state2 += c;
    break;

default:
    label += c;
    break;
}
break;

case 7:
switch (c)
{

case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
    state2 += c;
    break;

case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
    break;

default:
    MnTransition temp(atoi(state1.c_str()),
                     getSymbolTableIndex(label),
                     atoi(state2.c_str()));
    nfa.getTransitions().disjoint_union(temp);
    state1.clear();
    label.clear();
    state2.clear();
    current_state = 0;
    break;
}
break;

if (current_state == -1)
{
    cout << "Input error at token" << c << ")" << endl;
    exit(1);
}

// nfa.getTransitions().unique_array();
return is;
Figure 1: fcmenum –infoPage

Figure 2: Input System Test
Figure 3: NFAtoDFA test

Figure 4: IsDeterm test
Figure 5: IsComp test

Figure 6: fmenum test
Figure 7: fmunion test