CSIT
Research
Report Series

Guide to Extending Grail

Brodie Champion and Marcus Trenton

University of Prince Edward Island

CSIT-4
August 27, 2007

Computer Science and Information Technology @
the University of Prince Edward Island

Contents

Guide to Extending Grail

Brodie Champion and Marcus Trenton [*][†]
Department of Computer Science and Information Technology,
University of Prince Edward Island,
Charlottetown, PE C1A 4P3, Canada

August 27, 2007

Abstract

We describe all the steps which were necessary expand upon Grail [7], namely by adding filters. We provide the example of adding the filter *fmshuffle* and *fmshuffleq* to Grail. We also describe how to work with a few of the classes in Grail. Grail already possess excellent documentation [2, 3, 4, 5, 6], but has a few peculiarities.

1 Introduction

Here is a brief description of the steps needed to add a new filter to Grail:

- Create the symbolic links to the Grail filter by editing */char/makefile*, */char/names.h*, and */char/dosnames.h*
- Edit */char/grail.cpp* to handle the command line input
- Edit *include.cpp* for the appropriate core class to add new include statements
- Edit the appriate class' header files to include your new function
- Create a *src* file to provide the body of the new function

2 Grail Architecture

Grail is highly modularized program that relies heavily on templates. The symbols used to represent inputs and states are abstracted throughout the implementation of Grail. Only in `grail.cpp`, which contains the main function, is the template class defined as being a character. There are no other implementations yet, though byte, int, or long would make good candidates. The core classes as the nouns of automata: `inst` (instructions), `state`, `fm` (finite machines), `fl` (finite language), and so on. Each of the major functions (not set, get, or supporting functions but instead funtions like complete, cross product, make deterministic) are implemented in a seperate `src` files to keep

[*]Supervisor: Cezar Câmpeanu, Deptartment of Computer Science and Information Technology, University of Prince Edward Island, Charlottetown, PE C1A 4P3, Canada

[†]All correspondence to Cezar Câmpeanu: ccampeanu@upei.ca

the core classes to a manageable size. Then there are a number of overhead classes to make maintainance easier: *names.h* holds the strings that represent each of the commands implemented in Grail, several of the core classes have an *include.h* to hold all of the include statements. The classes highly overload the operators, particularly the $\leq$, $\geq$, =, ==, += and [ ] operators.

3 Grail Filters

The numerous Grail filters are simply symbolic links to one or more executable files. To add a new filter we must create a new symbolic link. We will add a symbolic link to the executable *char.out*. This can be done by editing the makefile */char/makefile*. The *install* target creates symbolic links for the filters:

```
install:
...
$(CP) $(BIN)char.out $(I_BIN)fcmmin$(EXT)
$(CP) $(BIN)char.out $(I_BIN)fcmtofm$(EXT)
$(CP) $(BIN)char.out $(I_BIN)fmcomp$(EXT)
$(CP) $(BIN)char.out $(I_BIN)fmcment$(EXT)
$(CP) $(BIN)char.out $(I_BIN)fmcat$(EXT)
$(CP) $(BIN)char.out $(I_BIN)fmcross$(EXT)
...
```

`fmcomp` (make the finite machine complete), `fmcross` (produce the cross product of 2 finite machines), and the others are symbolic links for existing Grail commands. We add the command

`$(CP) $(BIN)char.out $(I_BIN)fmshuffle$(EXT)`

to that list to create the symbolic link for our newly created function, `fmshuffle`.

4 Grail main() Function

A Grail executable, such as *char.out*, has access to the name of the filter calling it through its first command line argument. The main function in *char.out* resides in the file */char/grail.cpp*. If we examine */char/grail.cpp*, we see that there are a number of if statements which take action based on the name of the filter being called. */char/names.h* and */char/dosnames.h* holds the strings that represent filters (like `fmcomp`), with the path portion of the string removed. This is done for portability; in this way, *fmshuffle* can be *fmshuffle.exe* on a windows system. The variable my_name holds the filter the user has requested. Here is a simplified code snippet from */char/grail.cpp*.

```
if (strcmp(my_name, fmcomp) == 0)
{
   ...
}

if (strcmp(my_name, fmcment) == 0)
{
   ...
}
```

```
if (strcmp(my_name, fmcat) == 0)
{
   ...
}


... and so on!
```

To get our `fmshuffle` working, we must add a line to recognize the name of the filter. We add a block of code:

```
if (strcmp(fm_shuffle, fmshuffle) == 0)
{
    //Input state machines that represent symbols and
    //states with characters
    fm<char> a;
    fm<char> b;

    //Output state machine
    fm<char> c;

    //Read in input
    get_two(a, b, argc, argv, my_name);

    //Shuffle!
    c.shuffle(a, b);
    cout << c;
    return 0;
}
```

Now, we would like our shuffle filter to operate on two automata, and then output a third automata. To do this, we use three instances of the *fm<char>* class:

```
 //Input state machines
fm<char> a;
fm<char> b;

//Output state machine
fm<char> c;
```

We desire to read in the automata a and b from files. There are functions provided in */char/grail.cpp* to read in automata, finite languages, and regular expressions. The function:

`get_one(X, int argc, char** argv, char* my_name)`

is overloaded such that X may be a reference to a finite machine *fm<char>*, a finite language *fl<char>*, a regular expression *re<char>*, and so on. X is a output variable and will hold whatever construct that was read in. Similarly, there is an overloaded function:

`get_two(X,Y, int argc, char** argv, char* my_name)`

where X and Y are *both* finite machines, finite languages, regular expressions, etc.

In the case of shuffle, our operation is rather complex, so we chose to implement it as a method in the `fm` class.[1]

5 Adding Methods to the Grail Classes

In order to keep consistency it is important to know what class should contain which methods. Should a method that converts a finite language to a finite machine be located in the finite language or finite machine class? The general rule is that the method goes in the more complex class. The methods for converting between a finite language and a finite machine are found in the finite machine class. Similarly, finite cover machines are more comlpex than finite machines.

Most likely you will want to edit the larger classes and leave the smaller ones alone. The larger classes are `fm`, `re`, and `fl` among others. In our example we were wanted to shuffle finite machines and hence we added onto the `fm` (finite machine) class. Adding a method to the `fm` class required changing the following files:

- */classes/fm/fm.h*

  This file contains the definition of the fm class. We add the function prototype:

  `void  shuffle(const fm<Label>& a, const fm<Label> b);`

- */classes/fm/include.h*

  This file includes all of the code needed to compile the `fm` class. It includes */char/fm.h*, as well as a number of `.src` files. Where each `.src` file typically contains the implementation only one method in the `fm` class. This is done to break up what would otherwise be an incredibly large and unmanageable class. We add the line `#include ‘‘shuffle.src’’`

- */classes/fm/shuffle.src*

  In this file we add the implementation of our shuffle method.

6 Existing Grail Classes

When writing a filter, or adding a new method to a Grail class, we have access to a wide variety of pre-existing classes and methods. There are, however, a few peculiarities in these methods. We will look at parts of our implementation of shuffle, to see how these Grail classes can be used. In general, the comments in the Grail source code are quite extensive and well written. Only the most important or quirky methods will be mentioned in this section.

Grail makes extensive use of templates. How states and instructions are represented (be it integers, chars, or bytes) is not determined until compile time. Until that time they are refered to as generic Items or Labels.

Several classes have a += operator. This is used to add something to the object: instructions to finite machines, items to sets, and items to finite languages. As well, the array and finite machine classes have an indexing operator, [ ]. Almost all classes have many of the comparison operators overloaded. To avoid memory issues, the copy constructor or copying action is usually associated with the = operator.

[1]This would allow us to add the shuffle operation for different executables, not just char.out - that is, perform shuffle on other types, not just fm<char>.

Sadly, Grail lacks consistency with regard to using capitalization. We can only suggest that you follow the capitalization standards already present in the file you are editing.

6.1 The Set Class

Grail stores a lot of its data in set objects. Due to the use of templates the of the objects must be of same type. A finite machine stores its transitions, start states, and final states in sets. Input alphabets for states can also be stored in a set. We will describe some of the more useful functions in the set class here.

- `Set<Item>()`

  Creates an empty set.

- `Set<Item>(const Set<Item> &)`

  Creates a new set which is a copy of a given set.

- `void operator+=(Item)`

  Adds a new element to the set while ensuring that each element is unique. Effectively this means we can add an element, `e`, to a set `s`, by saying

  ```
  s += e;
  ```

- `void operator+=(const set<Item>&)`

  Adds another set to this set while ensuring that each element of the resulting set is unique.

- `void operator-=(Item)`

  Removes an element from the set.

- `void disjoint_union(Item)`

  Adds a new element to the set but does not check for uniqueness.

- `int size()`

  Returns the number of elements in the set.

- `Item operator[ ](int n)`

  Returns the ith element of the set. We may iterate over all items in a set using `size()` and `operator[ ](int n)`.

  ```
  for(int i = 0; i < my_set.size(); i++)
  {
  //Do something with the ith element
  Item my_item = my_set[i];
  }
  ```

- `void intersect(const set<Item> &, const set<Item>&)`

  Makes this set the intersection of two given sets.

- `int member(Item)`

  Returns the index of the desired Item, or -1 if the Item is not in the set.

6.2 State Class

Grail represents states internally as integers. Beware, the internal representation of a state in the state class is not the same as the number in the interface methods. So respect the state class' encapsulation.

The state class has the following functions:

- `state(int)`

  Creates a state object representing the given int.

- `start()`

  Makes this state the pseudo start state, i.e. (START)

- `final()`

  Makes this state the pseudo final state, i.e. (FINAL) in grail instructions

6.3 Inst Class

Grail thinks of all inputs (instructions) as triples. An arc has 3 elements: the orginiating (source) state, the input symbol, and the destination (sink) state. The problem is that not all instructions are naturally triples. For example, designating a state as being final only involves 1 element: the state. However, for ease of internal workings, such Grail instructions are treated as triples. The input text (not code) to designate state 5 as a final state is "$5 \ -| \ (FINAL)$". When Grail reads this instruction from a file it parses it in its usual way: 5 is a source state, $-|$ is an input symbol, and (FINAL) is a destination state. This makes no intuitive sense and Grail gets around the problem using the idea of "pseudo states". A pseudo state is not actually part of the automata and is simply used as a marker. The instruction is now processed by using a useless, arbitrary value for the input which forms an arc from state 5 to the final state. The end result is that 5 is treated exactly the way we would expect, as a final state. There is also an analgous process with start states, except that the pseudo state of start is the source and the normal state of 5 is the sink. So, why should we have explained all of these internal mechanics if the end result is exactly what you expect? If the function you are writting needs to manually assign start or final states, you must create them in the manner consistant with the internal workings of Grail.

The states 0 and 1 are pseudo start and final states, used for representing special grail instructions. Since 0 is used to represent (START) and 1 is used to represent (FINAL), a state label n is actually stored as an int $n + 2$. This change is purely internal and does not affect the interface methods in any way.

- `inst()`

  The default constructor does nothing.

- `inst<Label>`

  `assign(const state s1, const Label r, const state s2)`

  Defines the current instruction to use the given source state (s1), input label (r), and sink state (s2).

- `Label get_label()`

  Returns the input label for this instruction. In other words, what is the symbol for the arc?

- `state get_sink()`

  Returns the sink state.

- `state get_source()`

  Returns the source state for this instruction.

Consider the following code snippet from *shuffleq.src*

```
state start_state; //Create a pseudo state.

start_state.start(); //Designate pseudo state as a start state.

inst<Label> new_inst; //Create a new instruction.

//Define the new instruction.

new_inst.assign(start_state, alphabet[0], 0);

//Add the instruction to the fm.

quotient.add_instruction(new_inst);
```

In this code snippet a new state is created using the default constructor and is called `start_state`. It will serve the purpose of being a pseudo state representing a starting state. A new instruction is then created (using a label so that we are not tying ourselves down to a character or integer representation) and then assigned a triple that represents what the action is. The source state is designated as being the pseudo start state. The input symbol is the first symbol of the finite machine's alphabet (but it could be anything, since it is arbitrary). The variable's alphabet was previous declared in the method in code outside of this snippet. The sink state is designated as being state 0. The newly formed instruction is then added to the finite machine called `quotient`.

6.4 Fm Class

This is a large class that contains many methods. Those adding to Grail will likely add onto the larger classes, such as this:

- `fm()`

  Creates an empty finite machine.

- `fm<Label>operator+=(const fm<Label>)`

  Adds all instructions, start states, and finals states in the parameter to the current fm.

- `inst<Label>operator[](int i) const`

  Returns the ith instruction in this fm.

- `void add_instruction(const inst<Label> i)`

  Adds the instruction to the current finite machine's instructions.

- `fm<Label>clear()`

  Removes all instructions/arcs/states from the finite machine.

- `set<state>finals(set<state> r) const`

  Returns a copy of the set of final states in this finite machine.

6.5 Pair Class

This is just a container class that hold a pair of something. All of its methods are get, set, or comparison based.

References

[1] C. Câmpeanu, K. Salomaa, S. Yu, S.: Tight lower bound for the state complexity of shuffle of regular languages, *Journal of Automata, Languages and Combinatorics*, 7, 2002, 303310.

[2] D. Raymond, Programmer's guide to Grail, version 2.5, `http://www.csd.uwo.ca/Research/grail/.papers/prog.ps`, 1996.

[3] D. Raymond and Roger Robson, Release Notes for Grail, version 2.5, `http://www.csd.uwo.ca/Research/grail/.papers/notes.ps`, 1996.

[4] D. Raymond and D.Wood, Grail: Engineering Automata in C++, version 2.5, `http://www.csd.uwo.ca/Research/grail/.papers/engine.ps`, 1996.

[5] D. Raymond and D.Wood,, User's guide to Grail, version 2.5, `http://www.csd.uwo.ca/Research/grail/.papers/user.ps`, 1996.

[6] D. Raymond and D.Wood, Grail: A C++ Library for Automata and Expressions, *Journal of Symbolic Computation*, **17** 4, 341-350, 1994.

[7] Yu, S.: Grail+: A symbolic computation environment for finite-state machines, regular expressions and finite languages, `http://www.csd.uwo.ca/Research/grail/` , 2002.