miniGrail

Cezar Câmpeanu Rui Zhou

MiniGrail is a Grail+ clone. However, the code is totally diferent because its main structure is designed differently.

For the moment, it has limited functionality and only a few of the functions present in Grail+ are implemented in miniGrail.

MiniGrail is based on the MachineCat software an UPEI Honour's project of Rui Zhou under the supervision of Cezar Câmpeanu.

As of today July 14, 2012 the following filters are implemented in miniGrail:

1. `fcmenum`: enumeration of words accepted by a DFCA
2. `fltofm`: converts a finite language to a finite machine
3. `fmcomp`: make a finite machine complete
4. `fmenum`: enumeration of words accepted by a DFA
5. `fmunion`: the finite machine recognizing the union of two languages represented by two finite machines
6. `iscomp`: tests if an automaton is complete
7. `iscomp2`: tests if an automaton is complete(different algorithm)
8. `isdeterm`: checks if an automaton is complete
9. `fmmin_tbf`: The table filling algorithm for minimizing DFAs
10. `mGManual`: This Document (pdf format).
11. `nfatodfa`: transforming a nondetermninistic machne to a deterministic one (subset construction)
12. `nfatodfa_mG`: transforming a nondetermninistic machne to a deterministic one (subset construction), but different implementation

13. `fmanalyze`: does nothing

14. `inputTest`: tests Input read, used for developmemnt

1 Command line options

The following commend line options are implemented in miniGrail:

− − `infoPage` Launches the default browser with the location of a HTML documnet containing detailed information about thet filter/algorithm.

− − `info` Displays the same HTML file as the option − − `infoPage`, but it is filtered to text mode.

−`unique` Eliminates duplicate lines in the output for nondeterministic finite machines

− − `version` Outputs the version of miniGrail

2 Project Structure

`miniGrail`is built up in a very strict hierarchy that actually make sense, No(or very little) circular dependency exists. Circular references are common in `Grail+` , where an hierarchy of classes and algorithms is harder to achieve.

`miniGrail` Dependency Map.

3 Addind and Removing Code in miniGrail

3.0.1 Add New Algorithm

Add a new algorithm into `miniGrail` 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 "mGMachineAnalyze" which simply takes in an non-deterministic finite machine and out put the size. Following is the steps:

1. Goto the folder `miniGrail/Algorithms/`, and make an folder called "`mGMachineAnalyze`", inside which we create a file called "`mGMachineAnalyze.cpp`".

2. Open the file "`mGMachineAnalyze.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 `miniGrail` library or any other function that is already implemented in the Algorithms/ folder.

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

```
 //This algorithm is to analyze a  finite machine
using namespace std;
void mGMachineAnalyze(const mGNFA& fm)
{
 cout << "Analyzing DFA:" << endl;
 mGSet<mGTransition> 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. `miniGrail` 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 "`mGMachineAnalyze.h`", in which we add the declaration of the function that we implemented:

```
//Algorithms set : The analyis of finite machines
//the methods declaration
void mGMachineAnalyze(const mGNFA&); //Analyze a FA
```

documents.

4. In terminal, Navigate to the folder `miniGrail/` and re-make. `miniGrail` 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.

3.0.2 Add New Filter

A filter is an easy way to invoke the algorithms we implemented in `miniGrail`, to enable fast everyday test! Now we assume we want to add a new filter, which will invoke the algorithm "`mGMachineAnalyze`" and output the result. Steps:

1. Open the folder `miniGrail/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:

```
void fmanalyze()
{
 mGNFA nfa;
 std::cout << "enter nfa" << std::endl;
 std::cin >> nfa;
 mGMachineAnalyze(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 `miniGrail` 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:

```
void fmanalyze();
```

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

4 Using miniGrail

In the this section we include some runni9ng examples for the use with `miniGrail`:

```
miniGrail$ ls
Algorithms/
BackBone/
Bin/
DebugControl/
Documentation/
Filters/
MainFunction/
Makefile
miniGrail.h
Tests/
TheoryObjects/
a
b
miniGrail$cd Bin
Bin]$ ./isdeterm b
Oops, miniGrail can't open b
[Bin]$ ./isdeterm ../b
no
[Bin]$ ./isdeterm ../a
yes
```

```
[Bin]$ cat ../a
$ cat ../a
(start) |- 0
0 a 1
1 a 2
2 a 4
3 a 5
5 a 2
4 b 1
1 b 3
2 -|(FINAL)
[Bin]$ ./fmenum -n 7 ../a
a a
a b a a
a a a b a
a a a b b a a
a b a a a b a
a a a b a a b a
a b a a a b b a a
Bin]$ ./fmenum  -n 7 ../a| sed -e "s/ //g"
aa
abaa
aaaba
aaabbaa
abaaaba
aaabaaba
abaaabbaa
$ fmenum ../a | head -7
aa
abaa
aaaba
aaabbaa
abaaaba
aaabaaba
abaaabbaa
```

```
$ cat ../b
(start) |- 0
0 a 1
0 b 4
1 a 5
1 a 2
2 a 4
3 a 5
5 a 2
4 b 1
1 b 3
2 -|(FINAL)
0 -|(FINAL)

$ ./fmenum -n 7 ../b

a a
a a a
b b a
a b a a
b b a a
a a a b a
b b b a a
[Bin]$ ./fmenum -n 7 ../b| sed -e "s/ //g"

aa
aaa
bba
abaa
bbaa
aaaba
bbbaa
$ fmenum ../b | head -7

aa
aaa
bba
abaa
bbaa
aaaba
```

```
 Bin]$ cat ../c
l=12
(START) |- 0
0 a 1
1 b 2
2 a 3
3 b 4
4 a 5
5 b 2
5 a 0
2 b 1
0 -| (FINAL)
 Bin]$ fcmenum ../c

ababaa
abbbabaa
abbbbbabaa
abababababaa
abbbbbbbabaa
abbbababababaa
ababaaababaa
abababbbabaa
Bin]$ ./fcmenum ../c| sed -e "s/ //g"

ababaa
abbbabaa
abbbbbabaa
abababababaa
abbbbbbbabaa
abbbababababaa
ababaaababaa
abababbbabaa
 Bin]$ ./fcmenum -n 3 ../c| sed -e "s/ //g"

ababaa
abbbabaa
```

5 Speed Tests. Comparison with Grail+

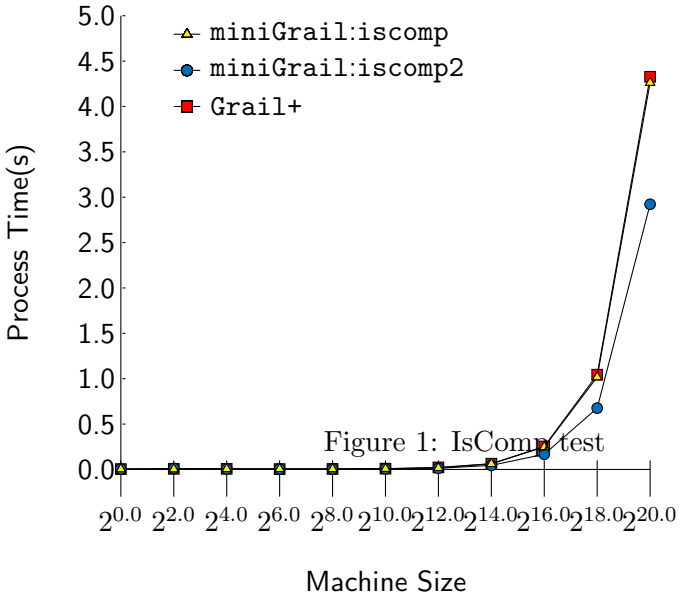

Figure 1: IsComp test

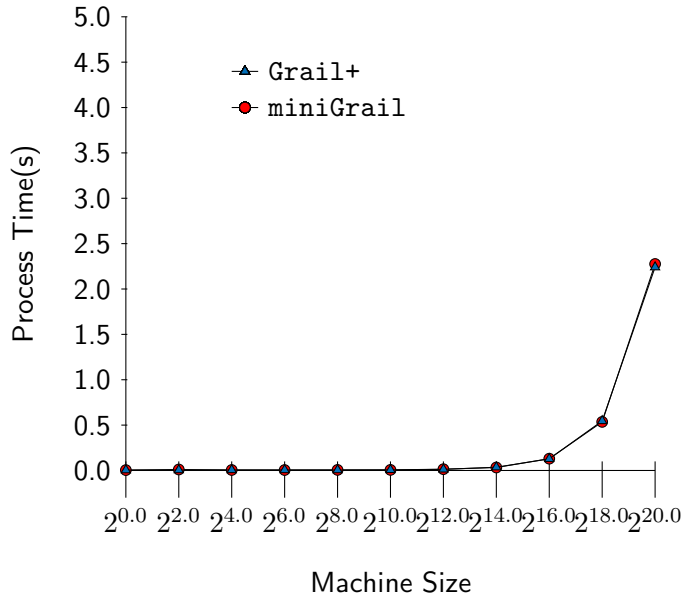

Figure 2: IsDeterm test

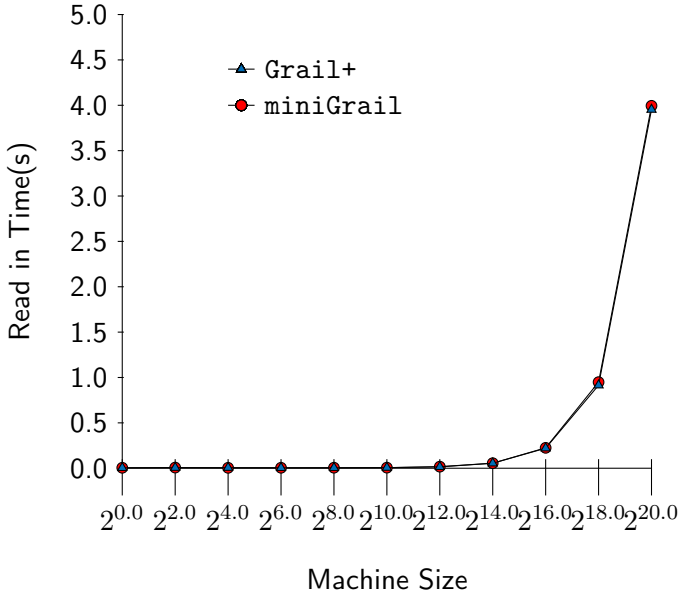

Figure 3: fmenum test

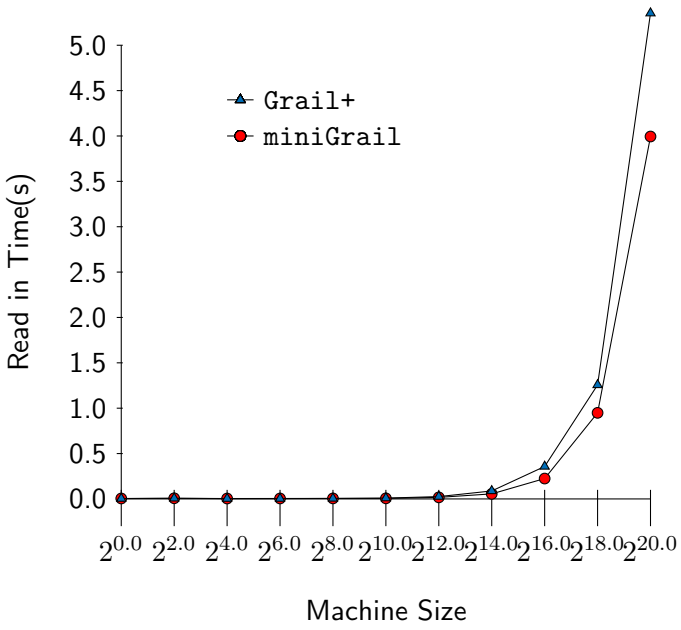

Figure 4: fmunion test