A System to Manipulate Regular Languages and Simplify Regular Expressions: A Demo Program

Baudouin Le Charlier
baudouin.lecharlier@uclouvain.be

UCLouvain – ICTEAM

1 Introduction

This paper is intended to show you how to use a demo program that simplifies regular expressions and also performs boolean operations, inclusion checks, and equivalence checks of regular expressions. One can also compute the minimal DFA corresponding to an expression, as a set of equations between expressions and their derivatives.

The demo program uses and is based on a set of java classes working on an internal representation of regular expressions and equations relating them. These classes constitue a tool box to build applications based on regular expressions and finite automata, not limited to the functionalities described here but the presentation here gives an "external" view of what can be done by the system. Information on how it works "internally" can be found in [8,9]. Copies of these documents are available at ArteFact together with the program and test data.[1]

2 The program

The program is available as a jar file. It was compiled on a MacBook Pro (Early 2015) with java version "1.8.0_131" . It should work on Mac OS, Linux, and Windows. It must be launched from a console window.

The program can be used interactively or by redirecting a text file to stdin. The second approach usually is simpler and we first focus on it. The text file must consist of command lines, regular expressions, comment and blank lines. The last line must consist of a single ";" character. (Subsequent lines are ignored.)

[1] Note however that these documents only give partial views of the system. They are not completely up-to-date either, but they at least provide some intuitions. Final versions are expected as soon as possible.

2.1 Simplifying expressions

As a first example, we run the program with the following console line:[2]

```
java -cp BLCTool.jar Demo <testdata/twoExprs.txt
```

It produces the following output:

```
SOI 1000000 2
[0]
// One of my favourite examples
------------------------------------------------------------
Original expression
c* + c*a(c*a + b)*c*
length = 21
------------------------------------------------------------
Normalized expression
c* + c*a(b + c*a)*c*
iExpr = 35 size = 18
------------------------------------------------------------
Simplified expression
(c + ab*)*
iExpr = 125 size = 7
============================================================
[1]
// An example from Han & Wood [6]
------------------------------------------------------------
Original expression
(aa + b)a*c(ba*c)*(ba*d + d) + (aa + b)a*d
length = 42
------------------------------------------------------------
Normalized expression
(b + aa)a*c(ba*c)*(d + ba*d) + (b + aa)a*d
iExpr = 42 size = 38
------------------------------------------------------------
Simplified expression
(b + aa)(a + cb)*(1 + c)d
iExpr = 192 size = 17
============================================================
[2]
```

[2] We use the term "console line" for a line typed in the console window, while "command line" is used for a command given to the running system.

```
Total time = 0.227 sec
Average size = 12.0 [simplified expressions]
iExprList.size() = 204
============================================================
```

The output first line `SOI 1000000 2` is the command line specifying the process to be performed. The command `SOI` indicates that the regular expressions in the file must be simplified using the three methods `S`, `O`, and `I`. The numbers 1,000,000 and 2 specify the size of the internal data structures usable by the algorithm: respectively the maximum number of identifiers available for expressions and the number of letters that can be used in expressions. This command line is a default. It could also be specified in the file `testdata/twoExprs.txt` as a command line preceding the expressions to be processed or in the console line launching the program, as

```
java -cp BLCTool.jar Demo <testdata/twoExprs.txt SOI 2 1000000
```

A different run of the program can be launched by typing

```
java -cp BLCTool.jar Demo <testdata/twoExprs.txt S 4 10000
```

It produces this (different) output:

```
S 4 10000
[0]
// One of my favourite examples
------------------------------------------------------------
Original expression
c* + c*a(c*a + b)*c*
length = 21
------------------------------------------------------------
Normalized expression
c* + c*a(b + c*a)*c*
iExpr = 35 size = 18
------------------------------------------------------------
Simplified expression
(c + a(b + c*a)*)*
iExpr = 106 size = 12
============================================================
[1]
// An example from Han & Wood [6]
------------------------------------------------------------
Original expression
(aa + b)a*c(ba*c)*(ba*d + d) + (aa + b)a*d
```

```
length = 42
------------------------------------------------------------
Normalized expression
(b + aa)a*c(ba*c)*(d + ba*d) + (b + aa)a*d
iExpr = 123 size = 38
------------------------------------------------------------
Simplified expression
(b + aa)a*(1 + c(ba*c)*(1 + ba*))d
iExpr = 143 size = 27
============================================================
[2]
Total time = 0.009 sec
Average size = 19.5 [simplified expressions]
iExprList.size() = 162
============================================================
```

Now, let us go back to the first output. We see that two expressions have been processed. Each of them produces three groups of lines, preceded by a comment (found in the input text file). The first group displays the original expression from the text file, with its length (as a string). The second group displays the normalized version of the original expression.[3] The third group shows the simplified expression computed by the system with the algorithms specified in the command line. The identifier and the size of the expression are also given. The size is the number of symbols making up the expression.[4] The last lines of the output specify the total running time of the simplification process (including printing), the average size of the simplified expressions, and the number of identifiers used in the end. It can be observed that both original expressions are significantly simplified. The first one clearly is minimized while the second one also looks so although it seems much more difficult to prove. However, it contains only one occurrence of the star symbol, which is a good indicator of simplicity.

Now let us compare the second output to the first. We see that both expressions are simplified to a lesser extend, because using the option `S` alone is less powerful that `SOI`, which uses additional algorithms. The time spent to compute the first output also is much greater than for the second one. But this is mostly because in the second case, the command `S 4 10000` specifies a wiser initialisation of the system internal data structures. Less identifiers are used but largely enough. Moreover, it is better to specify that the maximal number

[3] The normalization process is explained in the papers describing the internal working of the system.

[4] It is not the number of characters used to write the expression because parenthesis are not counted, while implicit concatenation operators are.

of different letters expected in expressions is 4, instead of 2. When the system encounters an expression containing more variables than the expected number, the entire internal data structure is reset, which is done twice in the first case, since the first expression uses three letters, and the second four.

2.2 Comparing expressions

We can compare expressions for both inclusion or equivalence of the regular languages they denote. Here is an example for inclusion:

```
java -cp BLCTool.jar Demo <testdata/someExprs.txt I 3 10000
```

This console command produces the following output:

```
I 3 10000
[0]
// First pair of expressions to compare
------------------------------------------------------------
Expr1
((a + ba)a*)* + (a + ba(1 + ba)ba)aa(1 + a)
iExpr = 42 size = 31
------------------------------------------------------------
Expr2
((a + b)a*)* + (a + b(1 + b)b)aa(1 + a)
iExpr = 51 size = 23
iExpr1 <= iExpr2
============================================================
[1]
// Second pair
------------------------------------------------------------
Expr1
a*(aab + bb*a + bb)*
iExpr = 123 size = 20
------------------------------------------------------------
Expr2
c* + c*a(b + c*a)*c*
iExpr = 156 size = 18
iExpr1 </= iExpr2
============================================================
[2]
Total time = 0.023 sec
iExprList.size() = 179
============================================================
```

Two pairs of expressions are compared. For the first pair, the system displays `iExpr1 <= iExpr2`, meaning that the first expression denotes a language included in the language denoted by the second. For the second pair, it displays `iExpr1 </= iExpr2` meaning that the first expression denotes a language containing at least one chain of letters not belonging to the language of the second. For checking equivalence, we can use the command `E` instead of `I`:

```
java -cp BLCTool.jar Demo <testdata/someOthers.txt E 10000
```

This console command produces the following output:

```
E 2 10000
[0]
// First pair
------------------------------------------------------------
Expr1
b*(ab*)*
iExpr = 31 size = 8
------------------------------------------------------------
Expr2
((a + b)a*)* + (a + b(1 + b)b)aa(1 + a)
iExpr = 44 size = 23
iExpr1 == iExpr2
============================================================
[1]
// Second pair
------------------------------------------------------------
Expr1
a*(aab + bb*a + bb)*
iExpr = 89 size = 20
------------------------------------------------------------
Expr2
((a + ba)a*)* + (a + ba(1 + ba)ba)aa(1 + a)
iExpr = 105 size = 31
iExpr1 =/= iExpr2
============================================================
[2]
Total time = 0.007 sec
iExprList.size() = 147
============================================================
```

This output means that the first two expressions denote the same language (actually, the same as (a + b)*) while the last two denote different ones.

Other commands for checking inclusion and equivalence are available in the system. They are more efficient in some cases. This is discussed later.

2.3 Computing and showing equations

The system is able to compute the minimal deterministic finite automaton (DFA) corresponding to an expression. (This is used for simplifying expressions, and so are the algorithms for checking inclusion and equivalence.) The demo program allows us to display this minimal DFA as a set of equations relating some expressions to their derivatives. Let us type the command:

```
java -cp BLCTool.jar Demo <testdata/conway.txt DFA 1000
```

We get the following output:

```
DFA 2 1000
[0]
// Conway's example [4]
------------------------------------------------------------
Original expression
a*(aab + bb*a + bb)*
length = 20
------------------------------------------------------------
Normalized expression
a*(aab + bb*a + bb)*
iExpr = 37 size = 20
------------------------------------------------------------
Simplified expression
a*(aab + b(b + b*a))*
iExpr = 131 size = 18
============================================================
Equations [MDFA] :
---------------
E131   =  1  + a.E155  + b.E156
E155   =  1  + a.E202  + b.E156
E156   =       a.E98   + b.E189
E202   =  1  + a.E202  + b.E199
E98    =  1  + a.E142  + b.E156
E189   =  1  + a.E210  + b.E156
E199   =  1  + a.E210  + b.E199
E142   =       a.E107
E210   =  1  + a.E141  + b.E156
E107   =                 b.E98
```

```
E0     =
E141   =       a.E107  + b.E98
============================================================
[1]
Total time = 0.012 sec
iExprList.size() = 360
============================================================
```

Every symbol `E`i represents a normalized expression, namely the expression having i as its identifier. For instance, `E131` is the (simplified) expression a*(aab + b(b + b*a))* . The notation `E`i is used instead of the expressions themselves for readability and because we do not need to know their actual value to reason on the set of equations considered as a DFA. The expression identifiers play the role of the states (or nodes) of the DFA. The equations indicate whether the state is accepting (final) (if they contain a 1) and specify the transitions to other states. Transitions to `E0` are omitted.[5]

2.4 Boolean (set) operators

The system is able to perform boolean operations on expressions. They compute a new expression denoting the intersection (`&`), the difference (`\`), or the symmetrical difference (`^`). It is also possible to compute the complementary (`!`). Practically, so-called *extended regular expressions*, containing boolean operators, can be given to the system and simplified using at least the command `O`. Let us show an example:

```
java -cp BLCTool.jar Demo <testdata/threeExtended.txt SOI 3 1000

SOI 3 1000
[0]
// A difference from Conway [4]
------------------------------------------------------------------
Original expression
((xy* + yx)* \ (y*x + xy)*)
length = 27
------------------------------------------------------------------
Normalized expression
((xy* + yx)* \ (y*x + xy)*)
iExpr = 37 size = 4294967314
------------------------------------------------------------------
```

[5] This example is borrowed from [4], pages 42–43. The reader can check that the set of equations above is equivalent to the DFA depicted in [4].

```
Simplified expression
(yx)*x((1 + y(yy*x)*)x)*y(y(1 + x))*y
iExpr = 200 size = 30
========================================================================
[1]
// An example from Antimirov [2]
------------------------------------------------------------------------
Original expression
(((a*b)* & (ab*)*) \ (ab)*)
length = 27
------------------------------------------------------------------------
Normalized expression
(((a*b)* & (ab*)*) \ (ab)*)
iExpr = 203 size = 8589934606
------------------------------------------------------------------------
Simplified expression
a(ba)*(a(a + b)*b + bb(1 + (a + b)*b))
iExpr = 275 size = 27
========================================================================
[2]
------------------------------------------------------------------------
Original expression
!0
length = 2
------------------------------------------------------------------------
Normalized expression
!0
iExpr = 276 size = 4294967296
------------------------------------------------------------------------
Simplified expression
(a + b + c)*
iExpr = 42 size = 6
========================================================================
[3]
Total time = 0.016 sec
Average size = 21.0 [simplified expressions]
iExprList.size() = 278
========================================================================
```

The minimal DFA of every "extended expression" is first computed and, then, solved as a set of equations. This results in a non extended expression, which

is simplified and returned as a simplified version of the extended one. The trick is that boolean operators are given a large weight (2^{32}) so that the size of the non extended expression has every chance to be less than the size of the extended one, even though it may contain much more symbols. Notice that `!0` is simplified to (a + b + c)* because, at this point, the system assumes that expressions may contain at most three letters.

2.5 Interactive execution

To run the programm interactively (without redirecting a text file), it is best to add the `'>'` option in the console line. The benefit is that the lines entered by the user are not echoed in the output. Additionally, a "prompt" `>` is issued by the program each time it expects an input from the user. Let us show an example:

```
java -cp BLCTool.jar Demo '>' 0 1000

SOI 0 1000
[0]
>a*(ca*)*
---------------------------------------------------------------
Normalized expression
a*(ca*)*
iExpr = 31 size = 8
---------------------------------------------------------------
Simplified expression
(a + c)*
iExpr = 33 size = 4
===============================================================
[1]
>((a + b + c + e)* \ (a + b + c + d)*)
---------------------------------------------------------------
Normalized expression
((a + b + c + e)* \ (a + b + c + d)*)
iExpr = 32 size = 4294967312
---------------------------------------------------------------
Simplified expression
(a + b + c)*e(a + b + c + e)*
iExpr = 71 size = 17
===============================================================
[2]
>CM 2
```

```
>I
Average size = 10.5 [simplified expressions]
iExprList.size() = 27
======================================================================
I
[0]
>(a*b)*aaa*
>(a + b)*a(a + b
syntax.ReaderException a ) was expected at position 15.
>(a + b)*a(a + b)
----------------------------------------------------------------------
Expr1
(a*b)*aaa*
iExpr = 33 size = 12
----------------------------------------------------------------------
Expr2
(a + b)*a(a + b)
iExpr = 37 size = 10
iExpr1 <= iExpr2
======================================================================
[1]
>;
iExprList.size() = 61
======================================================================
```

We see also that the user can re-type an expression in case of a syntactic error. This is not possible in the non interactive case (the program is stopped). Note also that the command line `CM 2` resets the system memory to two letters and 1,000 identifiers.

3 Summary of the commands

In this section, we complete the list of commands that can be used with the demo program. The reader is invited to try them.

3.1 Commands for simplification

A command for simplification consists of any combination of the letters `S`, `O`, `I`, except `I` alone. The letter `S` triggers the use of simplication rules, similar to those of [7,10]. The letter `O` triggers the creation of a minimal set of equations for the expressions. A new expression is created by solving this set of equations.

This expression can be simpler (shorter) than the one obtained using `S`. The letter `I` means that one asks to iterate the simplifications until no new expression is found. The reader should try the various combinations on some of the test files provided in the dropbox directory ArteFact. The more options are used, the better is the simplification but the execution time also is increased. When specifying a command, one can also specify the number of letters of expressions and the number of available identifiers. This can have an impact on the efficiency of the processing, especially when many and/or large expressions are considered. A compromise must be done: when many expressions are simplified, the results of previous simplifications can be reused, speeding up the process and/or giving unexpected simplifications. Of course, at some points, no more identifiers may be available. Then we can reset the memory, restarting simplifications within an empty context, possibly with another parametrization. It must also be noticed that some bounds are put on the size of the expressions and sets of equations on which algorithms are applied because most of them become exponential for large data.

3.2 Commands to compare expressions

There are two commands to check inclusion, and five to check equivalence of two expressions. The commands `I` and `E` trigger straightforward algorithms, which compute derivatives of the expressions, in parallel, until an incompatibility is found or all corresponding derivatives are found compatible. The command `IA` uses an algorithm based on an idea by Antimirov [1]. It is more efficient than `I` for some classes of expressions. For instance, it is linear in n for comparing pairs of the form (a*b)*a^{n}a* to pairs of the form (a+b)*a(a + b)$^{n-1}$, [6] while `I` is exponential. The reader may try both commands on the file `testdata/infEqAntimirov.txt`, which contains a list of such pairs for increasing values of n. The command `EA` uses the same idea than `IA` by checking a double inclusion in parallel. The command `EUF` uses a Union-Find data structure [5] to avoid comparing two derivatives that must be equivalent if previously encountered pairs are. Such an algorithm is used to compare expressions in [7]. The algorithm for `EP` uses a similar idea but a more elaborate relation based on congruence closure of expressions with respect to `+`. This is similar to the algorithm proposed in [3] for checking equivalence of NFAs. Finally, the algorithm for `EBLC` is a variant of `EP`, which appears to be more efficient in many cases.

[6] A classical example from [1]

4 Checking the demo program

The reader is invited to check the demo program on the test files available in the dropbox directory ArteFact. All test files belong to the subdirectory `testdata`. The files `basicExamples.txt` and `setOperations.txt` contains relatively short examples, which can be simplified with the various commands for simplification. The corresponding output can be compared to understand what the different options achieve. The directory `testdata` also contains files of randomly generated expressions, which can be used to assess and compare the efficiency of the various algorithms.

4.1 Simplifying and checking random test data

There are three sets of files, named `file-2-<length>-100.txt`, `infEq-2-<length>-100.txt`, and `true-2-<length>-100.txt`, where `<length>` = 10, 20, 40, ..., 2560.
Every file contains exactly 100 expressions using at most two letters, of size equal to `<length>`.

The files `file-2-<length>-100.txt` contain completely unrelated expressions. They can be used to check and compare the simplification options of the program, with respect to both precision of simplification, and execution time. They can also be used to check inclusion or equivalence of 50 pairs of expressions. The check should be negative, most of the time, and the execution fast, for all methods.

The files `infEq-2-<length>-100.txt` contain pairs of expressions such that the language of the first is included in the language of the second. They can be used to compare the efficiency of the algorithms `I` and `IA` when the check is positive.

Similarly, the files `true-2-<length>-100.txt` contain pairs of expressions such that the languages of the first and the second are equivalent. They are intended to be used to compare the efficiency of the algorithms `E`, `EA`, `EUF`, `EP`, `EBLC`.

4.2 Generating random test data

You may also generate new random test files with a different number of letters, different lengths, and a different number of lines than those proposed above. To do so you can use the following command:

```
java -cp BLCTool.jar syntax.GenRegExpr 3 10 5
1(a + a)1b*
a + (1 + b(b1*))
```

```
a1(a + (a* + 1))
(aa + a)*(a + 1)
1(a(bc*) + b)
;
```

The first number (here, 3) is the number of different letters you want to occur in the expressions, the second number (here, 10) is the length of every generated expression, the third number (here, 5) is the number of different expressions that are generated. The output is written on stdout. To get the result in a file, you can type a command such as:

```
java -cp BLCTool.jar syntax.GenRegExpr 3 10 5 >file.txt
```

5 Limitations and parameterization of the system

It may happen that no more free identifiers are available (the memory is "full"). Then the fancy message `It is time to call GC` is displayed and a `GCException` is raised. Specifically, the demo program catches the exception, resets the memory, and restarts the interrupted process "from scratch". In some cases, the restarted process raises an exception again. Then, the demo program displays the message `Failure on input [`*i*`].` This means that the restarted process is abandoned and the program moves on to the next task. Sometimes the problem can be solved by restarting the demo program with a higher maximal number of identifiers (and a minimal number of letters). However, other problems may arise due to java limitations (on your specific computer). It may help to increase the stack size and/or the "memory allocation pool" as in the following example:

```
java -Xss50m -Xms1000m -cp BLCTool.jar Demo <testdata/fold1.txt
 S 5000000
```

Last remark: For simplicity and readability, the demo program does not display expressions whose size exceeds a certain limit (it only displays their size). This can be changed by giving an higher value to the variable `LINELENGTH` at the beginning of the file `Demo.java` and recompiling it with

```
javac -cp BLCTool.jar Demo.java
```

But then the class path of the java command must be slightly modified as

```
java ... -cp .:BLCTool.jar Demo ...
```

6 Concrete syntax of expressions

The syntax of (extended) regular expressions accepted by the system is given by the BNF grammar below. Note that expressions using a binary set operator (`\ & ^`) must be put between parentheses (no operator priority). If an expression is not correctly written, the program displays an error message. Blanks can be inserted anywhere before, after, or inside an expression (on the same single line). Tabulation characters are not allowed.

```
<RE>      ::= <UNION>
<UNION>   ::= <CONC> | <UNION> + <CONC>
<CONC>    ::= <PSTAR> | <CONC> <PSTAR>
<PSTAR>   ::= <PCOMPL> | <PSTAR> *
<PCOMPL>  ::= <PLETR> | ! <PLETR>
<PLETR>   ::= (<UNION>) | (<UNION> \ <UNION>) |
              (<UNION> & <UNION>) |(<UNION> ^ <UNION>) | <ATOM>
<ATOM>    ::= a | b | ... | z | 0 | 1
```

References

1. Valentin M. Antimirov. Rewriting regular inequalities (extended abstract). In Horst Reichel, editor, *Fundamentals of Computation Theory, 10th International Symposium, FCT '95, Dresden, Germany, August 22-25, 1995, Proceedings*, volume 965 of *Lecture Notes in Computer Science*, pages 116–125. Springer, 1995.
2. Valentin M. Antimirov and Peter D. Mosses. Rewriting extended regular expressions. *Theor. Comput. Sci.*, 143(1):51–72, 1995.
3. Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 457–468. ACM, 2013.
4. J.H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall mathematics series. Dover Publications, Incorporated, 2012.
5. Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.
6. Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.
7. Stefan Kahrs and Colin Runciman. Simplifying regular expressions further. *J. Symb. Comput.*, 109:124–143, 2022.
8. Baudouin Le Charlier. Another look at derivatives of regular expressions, with an experimental evaluation. 2023. http://dx.doi.org/10.2139/ssrn.4592136.
9. Baudouin Le Charlier. A program that simplifies regular expressions (tool paper). *CoRR*, abs/2307.06436, 2023.
10. Alley Stoughton. *Formal Language Theory: Integrating Experimentation and Proof*. https://alleystoughton.us/forlan/book.pdf, 2003–2022. Open source book.