J. Symbolic Computation (1995) **11**, 1–000

Grail: A C++ Library for Automata and Expressions

Darrell Raymond[†]
Derick Wood[‡]

(Received January 1994)

Grail is a package for symbolic manipulation of finite-state automata and regular expressions. It provides most standard operations on automata and expressions, including minimization, subset construction, conversion between automata and regular expressions, and language enumeration and testing. *Grail*'s objects are parameterizable; users can provide their own classes to define the input alphabet of automata and expressions. *Grail*'s operations are accessible either as individual programs or directly through a C++ class library.

1. Introduction.

Grail is a symbolic computation environment for finite-state automata, regular expressions, and other formal language theory objects (Floyd and Beigel 1994; Wood 1987). Using *Grail*, one can input automata or expressions, convert them from one form to the other, minimize, make deterministic, complement, and perform many other operations. Currently, *Grail* consists of a collection of process-based filters for computing with finite-state automata and regular expressions, and a C++ class library for customized automata programming.

We are investigating the implementation of automata and expressions for three reasons. The first reason is to provide an environment that supports experimental research on automata. We are particularly interested in learning about the tradeoffs of alternative algorithms for solving problems, and in learning more about average case complexity through simulation. The second reason is to develop efficient software that can support the application of formal language objects to various programming problems. Traditionally, automata are used for parsing and text transduction, but they have also proved to be useful for other tasks, such as specifying network protocols, hardware testing, and describing various types of device controller. The third reason is to develop an environment that supports the teaching of formal language concepts. Most formal language courses are highly mathematical; the emphasis is on proof, not on engineering. While this bias is appropriate for students who will proceed to do graduate work in formal language theory, the vast majority of students would benefit from a better understanding of the problems and opportunities involved in implementing the objects they study.

[†] Department of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada
[‡] Department of Computer Science, University of Western Ontario, London, ON N6A 5B7, Canada

0747–7171/90/000000 + 00 $03.00/0

The first project to bear the name 'Grail' began some years ago, as an extension of Johnson's INR (Johnson 1986). INR is a program for computing with rational relations. INR's efficiency in manipulating very large automata gave it a unique role both in research and in practical applications (Kazman 1986). Johnson, Wood, and Seger developed the first version of *Grail* as a layer of C code on top of INR, using INR to provide automata support. The resulting system was able to handle context-free grammars and automata with regular expressions as transition labels. The system had a complex and monolithic architecture, however, and work on it was discontinued.

The second attempt at *Grail* was developed subsequently by the present authors. This version, written in C, had the express goals of modularity, clarity, and extendibility. Unfortunately, neither C nor our good intentions were enough to ensure these characteristics. We subsequently turned to C++, and its stricter type checking (and our practical experience) led to a more robust and modular library that is the basis of the present implementation of *Grail*.

2. Using *Grail*.

The primary objects in version 2.0 of *Grail* are finite-state machines and regular expressions. Regular expressions look much the same as they do elsewhere; *Grail* supports catenation, union, and Kleene star for regular expressions, along with parentheses to specify precedence. The following are examples of regular expressions acceptable to *Grail*:

```
a+b
((a+bcde*)+c)*
{}
""+a
```

The notation {} indicates the empty set, and the notation "" indicates the empty string.

A deterministic finite-state machine (or automaton) is normally specified by a 5-tuple:

$$< Q, \Sigma, \delta, s, F >$$

where Q is the set of states, Σ is the input alphabet, δ is a partial transition function $\delta : Q \times \Sigma \rightarrow Q$, s is the start state, and F is a set of final states. To simplify *Grail*'s input, we represent finite-state machines as sets of instructions. A machine for the language ab, for example, is specified by the instructions:

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

Each instruction is a triple that consists of source state, instruction label, and target state. The start and final states of the machine are indicated by means of special *pseudo-instructions*, whose labels are special symbols that can be thought of as endmarkers for the input string. The states (START) and (FINAL) are pseudo-states; they indicate that the other state in the instruction is a start or final state. The set of (non-pseudo) instructions is an enumeration of the transition relation. The input alphabet of the machine is

Table 1. *Grail* filters

fmcment	complement a machine
fmcomp	complete a machine
fmcat	catenate two machines
fmcross	cross product of two machines
fmenum	enumerate strings in the language of a machine
fmexec	execute a machine on a given string
fmmin	minimize a machine by Hopcroft's method
fmminrev	minimize a machine by reversal
fmplus	plus of a machine
fmreach	reduce a machine to reachable submachine
fmrenum	canonical renumbering of a machine
fmreverse	reverse a machine
fmstar	star of a machine
fmtore	convert a machine into a regular expression
fmunion	union of two machines
fmdeterm	convert an NFA into a DFA by subset construction
iscomp	test a machine for completeness
isdeterm	test a machine for determinism
isomorph	test two machines for isomorphism
isuniv	test a machine for universality
isempty	test a regular expression for equivalence to empty set
isnull	test a regular expression for equivalence to empty string
recat	catenate two regular expressions
remin	minimal bracketing of a regular expression
restar	star of a regular expression
retofm	convert a regular expression into a machine
reunion	union of two regular expressions
xfmcat	catenate two extended machines
xfmplus	plus of an extended machine
xfmreach	reduce an extended machine to reachable submachine
xfmreverse	reverse an extended machine
xfmstar	star of an extended machine
xfmtore	convert an extended machine into a regular expression
xfmunion	union of two extended machines

given implicitly; it is the set of symbols that appear on (non-pseudo) instructions. *Grail*'s machines differ from conventional automata in that we permit multiple start states as well as multiple final states.

To the user, *Grail* is the set of filters shown in Table 1. The filters can be used from a command shell, such as *sh* or *csh*. Each filter takes a machine or regular expression as input, and produces a machine or regular expression as output. Regular expressions and machines can be entered directly from the keyboard or (more usually) redirected from files. To convert a regular expression into a finite-state machine, for example, one might issue the following command:

```
% echo "(a+b)*(abc)" | retofm
```

whose output would be

```
0 a 1
2 b 3
0 a 0
0 a 2
2 b 0
2 b 2
4 a 1
4 a 0
4 a 2
4 b 3
4 b 0
4 b 2
(START) |- 4
1 a 6
3 a 6
4 a 6
6 b 8
8 c 10
10 -| (FINAL)
```

The filter `retofm` converts the input regular expression into a nondeterministic finite-state machine, which it prints on its standard output. This output can be the input for another filter; for example, one that converts the machine back into a regular expression (folded here to fit onto the page):

```
% echo "(a+b)*(abc)" | retofm | fmtore
abc+aabc+aa*aabc+ba*aabc+babc+a(b+ba*a)*ba*aabc+a(b+ba*a)*babc
+b(b+ba*a)*ba*aabc+b(b+ba*a)*babc+aa*a(b+ba*a)*ba*aabc+aa*a(b+
ba*a)*babc+ba*a(b+ba*a)*ba*aabc+ba*a(b+ba*a)*babc
```

We may choose to make the machine deterministic before converting it into a regular expression:

```
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmtore
aa*b(aa*b+bb*aa*b)*c+bb*aa*b(aa*b+bb*aa*b)*c
```

Or we may choose to minimize the deterministic machine before converting it into a regular expression:

```
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmmin | fmtore
b*aa*b(aa*b+bb*aa*b)*c
```

This set of pipelines illustrates a range of possibilities for developing regular expressions for a given language. As a teaching tool, the preceding examples show that algorithms that preserve language equivalence do not preserve identity (it also emphasizes the point that manually checking for language equivalence is extremely tedious except for very simple languages). As a software tool, the preceding sequence is useful for generating variant expressions of a single language, that then can be used to exercise other algorithms.

Shell scripts are not limited to pipelines. *Grail* filters can be combined with control flow constructs and other filters to develop machine programs. In the following session, we apply cross product to a machine (crossed with itself), use *wc* to compute the size of the resulting machine, and use *time* to compute the time consumed in the computation. This procedure is then applied to the result three more times.

```
$ cat nfm
(START) |- 0
0 a 1
0 a 2
1 -| (FINAL)
2 -| (FINAL)
$ for i in 1 2 3 4
> do
>  time fmcross nfm nfm >nfm.tmp
>  wc nfm.tmp
>  mv nfm.tmp nfm
> done
        0.1 real         0.0 user         0.0 sys
       9      27      89 nfm.tmp
        0.0 real         0.0 user         0.0 sys
      33      99     349 nfm.tmp
        0.1 real         0.0 user         0.0 sys
     513    1539    6413 nfm.tmp
       19.8 real        16.4 user         2.1 sys
  131073  393219 2162701 nfm.tmp
$
```

In four steps, the size of the result jumps from 9 to 131,000 instructions, and the time taken for computation in the last step is about 20 seconds. Approximately 20 Mbytes of memory are needed to compute the last machine.

The filter-based approach to a symbolic computation environment has several advantages. First, it is relatively easy to add or modify elements of the package simply by adding new filters, which can be written in any programming language. Second, we can take advantage of the large number of shell languages that exist, as well as users' familiarity with them, rather than having to develop and justify our own customized language. Third, it is relatively easy to distribute or to multiprocess a computation using multiple filters, because the separate processes can be placed on different processors without the need for customized programming.

For those who want to avoid the cost of I/O implicit in the use of the filter approach, *Grail* can also be accessed directly as a C++ library. The filter command

```
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmmin | fmtore
```

can also be written directly in C++:

```
#include        "grail.h"

main()
{
     re<char>   r;
     fm<char>   a;

     cin >> r;
     retofm(r, a);
     r = fmtore(a.subset().min_by_partition());

     cout << r << "\n";
}
```

The program first reads a regular expression from standard input. The `retofm` function then converts the expression into a machine. The machine is then made deterministic (via `subset`), minimized (via `min_by_partition`), converted back to an expression (via `fmtore`), and then output.

Using the C++ library directly enables the parameterization of *Grail*'s machines and expressions with a specific input alphabet. Any C++ type or class can serve as an input alphabet; in the preceding program, the objects use an alphabet of `char`s, but it is also possible to define more complex alphabets. *Grail* includes filters for *extended* machines, which have regular expressions as their input alphabet.

3. Design and implementation.

Grail is designed in several layers. The outermost layer, employed by the casual user, is the set of filters shown in Table 1. This layer can be used with little or no programming experience. The second layer is made up of the parameterizable classes for finite-state machines and regular expressions; this layer can be employed without much knowledge of *Grail*'s internals. The filters, for example, are simply I/O wrappers around calls to the parameterized classes. The final layer is the functionality of the individual classes themselves. *Grail* includes definitions for 15 classes, organized in the relatively flat hierarchy shown in Table 2.

The main classes are `fm` (finite-state machines) and `re` (regular expressions). These classes provide the capabilities that make *Grail* useful for symbolic computation with machines and expressions. Each class defines input and output functions, standard class routines (constructor, destructor, assignment, copy constructor) and comparison functions for identity testing. The classes also define the basic abstract functions for their objects, such as union, catenation, Kleene closure, and the conversion functions from regular expression into machine and vice versa.

There are two types of support classes. The first type implements basic container classes: `set`, `list`, and `string` are containers of general utility. The second type of support class implements substructures that are specific to finite-state machines and expressions: `state` implements the states of a finite-state machine, `inst` implements the instructions of a finite-state machine, and `subexp` and its derived classes implement regular expressions. `subexp` is an abstract base class; it serves only to define the functions common to all of its derived classes. Each regular expression contains a single subexpres-

Table 2. *Grail* classes

`fm`	
`inst`	
`list`	
`re`	
`set`	
`state`	
`string`	
`subexp`	
	`null_exp`
	`empty_set`
	`empty_string`
	`symbol_exp`
	`cat_exp`
	`plus_exp`
	`star_exp`

sion which may be an empty set (`empty_set`), empty string (`empty_string`), any single symbol (`symbol_exp`), a catenation expression (`cat_exp`), a union expression, (`plus_exp`), or a star expression (`star_exp`). The `null_exp` is used only for initialization.

Version 2.0 of *Grail* is approximately 7000 lines of C++. Complete details about the implementation can be found in the *Grail* technical report (Raymond and Wood 1994).

4. Some empirical lessons.

Developing *Grail* has taught us much about implementing algorithms for finite-state machines. C++ is an important contributor to the robustness of our code, mainly because of its strict type checking and its encapsulation of function with data structure. We have also learned some lessons that apply to the construction of mathematical libraries in general. One of these is that a library of routines is only half the battle; the other half is in developing a library of test data, and the provision of a mechanism for automatic testing and performance evaluation. In the early stages of development, *Grail*'s filters were tested with simple machines, and the results were checked by hand. As the pace of development increased, however, this was no longer acceptable; one cannot very well test tens of programs on each of several test cases by hand, and one cannot test very large machines or expressions by hand at all, since the probability of a manual error in checking rapidly becomes higher than the probability of an error in the code. Thus, it becomes necessary to automate testing. Automation is also essential in performance evaluation, which relies on large inputs in order to thoroughly exercise the code. One approach to generating large test cases is to apply filters that generate non-isomorphic machines that are language equivalent. Repeatedly converting between machine and regular expression, for example, will result in a large machine that accepts a known language. Hence, the result of processing such a machine can be tested by minimizing and comparing the result to the known minimal machine. Another related tactic is to repeatedly take the cross product of a nondeterministic machine with itself; there will be an exponential blowup in the size of the result, which is still language equivalent with the original.

A second important lesson is that a sound theoretical understanding of an algorithm is not the same as a sound implementation. To paraphrase a popular saying, a little knowledge of worst-case performance is a dangerous thing. Algorithms that have bad worst-case performance may be quite acceptable for most practical uses. Subset construction, in particular, is exponential in the worst case, but empirical study shows that the number of machines that exhibit this behavior is small (Leslie 1992). Moreover, it appears to be predictable from the input whether an exponential result is likely to occur. Since most users do not want to store or further use exponential output, predicting this result may be sufficient.

On the other hand, a sloppy implementation of a well-known algorithm with reasonable average case performance may be unacceptable for *every* large input. Linear-time algorithms can easily become quadratic-time if careful attention is not paid to problems such as the proper management of sets.

5. Related work.

There are several systems with similar goals to *Grail*, besides the aforementioned INR. The earliest effort we know of is Leiss's REGPACK, which was written in SPITBOL (Leiss 1977). More recently, Champarnaud and Hansel have produced the AUTOMATE system, written in C (Champarnaud and Hansel, 1991). AUTOMATE supports finite-state machines and finite semigroups, and can compute both syntactic and instruction monoids. Jansen and Potthoff's AMORE system also handles syntactic monoids, and produces graphical displays (Jansen, Potthoff, Thomas and Wermuth 1990). Hannay has built a Hypercard-based system for simulating automata, including Turing machines (Hannay 1992). Krischer's FANCY, the Finite AutomatoN Checker of nancY, is intended to support formal hardware verification. It provides equivalence and inclusion checking for finite-state automata and is accessible through a graphical user interface. FADELA, the Finite Automaton DEbugging LAnguage, is an effort directed by Gjalt de Jong (van der Zanden 1990). FADELA can derive deterministic automata, regular expressions, and Müller machines. It can also handle languages with infinite words. FLAP, the Formal Languages and Automata Package comes from Rensselaer Polytechnic Institute (LoSacco and Rodger 1993; Caugherty and Rodger, to appear). FLAP supports drawing and executing finite-state automata, pushdown machines and Turing machines. FLAP can handle nondeterministic machines, provides a stepping capability, and supports paper output.

Grail shares one or more goals with each of these programs—to serve as a research and teaching environment, to facilitate the study of machine implementations, or to be used in evaluating protocols and designs. *Grail* differs from these systems in several ways, however.

The first difference is in system design. Most other systems are provided as closed environments accessible through a special-purpose language, whereas *Grail* provides a set of independent processes that can be used with any command shell language. The advantage of the *Grail* approach is that it is easier for users to add to or to change its functionality; they need only to write a replacement filter that can read and write *Grail*'s machines. Naturally, it is possible to make use of *Grail*'s existing code to write such filters, but it is also possible to write filters in any desired language. More integrated systems do not provide this kind of flexibility.

A second difference is that *Grail* does not share existing systems' asymmetrical view of machines and regular expressions. INR, REGPACK, AUTOMATE, and AMORE make the implicit assumption that a regular expression (or rational relation) is the desired type of input, and that a (minimal, deterministic) finite-state machine is the desired output. *Grail* makes no such assumption. Unlike these other systems, *Grail* facilitates symbolic computation of the objects themselves, rather than computation of the languages they denote. Thus *Grail* treats both machines and regular expressions as first-class citizens. Partly because we have not chosen an asymmetrical approach, *Grail* provides a larger set of operations for dealing with its objects than do the other systems. REGPACK, for example, has 6 routines, whereas *Grail* has 34.

A third difference is that *Grail* is written in C++, and provides machines and regular expressions as a class library. The use of C++ confers an additional degree of modularity that permits us to safely change the internals of the system, while leaving the upper level interfaces unchanged. The ability to use different input alphabets is a direct result of the modularity provided by the C++ template mechanism.

6. Future work.

There is much future work we wish to do on *Grail*.

Our main and continuing activity is to increase the number of operations that we can apply to machines and regular expressions, and to improve the performance of existing operations. In the near future we will be looking at various kinds of mappings within classes (that is, from one machine or expression into another), and will also work on the specification of transduction expressions and transducers.

We will also increase the number of types of objects that are available to *Grail* programmers. In addition to more high-level classes, such as one for monoids and one for languages, we also expect to implement low-level support classes for digraphs and relations. The latter are particularly important in simplifying existing machine operations, since many machine operations can be reduced to operations on low-level classes. Testing reachability in a machine, for example, should not be done with a special-purpose algorithm, but using a reachability-testing algorithm for general digraphs. Similarly, many of the operations we apply to sets of instructions ought to be handled as special cases of operations on relational databases.

In the near future we will exploit *Grail*'s parameterizability to create many interesting types of machine, by instantiating our classes with appropriate input alphabets. (Floyd and Beigel 1994; Salomaa, Wood, and Yu, 1994). A finite-state machine whose instruction labels are symbol pairs, for example, can describe a transducer. Another interesting use of parameterized finite-state machines is to represent pushdown automata. Such machines can be represented with instruction labels of the form $aA^{-1}\alpha$, where a is an input symbol, A^{-1} is the top string that is to be popped from the stack, and α is the symbol to be pushed on the stack. This approach to pushdown automata was introduced by Floyd (as reported by Kurki-Sunio 1975) and is used by Salomaa *et al.* (Salomaa, Wood, and Yu 1994).

7. Acknowledgements.

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada. Darrell Raymond was also supported by an IBM Canada Research Fellowship. We would like to thank Howard Johnson for his assistance and encouragement.

Darrell Raymond can be reached at `drraymon@daisy.uwaterloo.ca`. Derick Wood can be reached at `dwood@csd.uwo.ca`.

References

J.M. Champarnaud and G. Hansel. AUTOMATE: A computing package for automata and finite semigroups. *Journal of Symbolic Computation*, 12:197–220, 1991.

R.W. Floyd and R. Beigel. *The Language of Machines: An Introduction to Computability and Formal Languages*. Computer Science Press, New York, N.Y., 1994.

D.G. Hannay. Hypercard automata simulation: Finite-state, pushdown, and turing automata. *SIGSCE Bulletin*, 24(2):55–58, June 1992.

V. Jansen, A. Potthoff, W. Thomas, and U. Wermuth. A short guide to the AMORE system. Aachener Informatik-Berichte (90) 02, Lehrstuhl für Informatik II, Universitat Aachen, Aachen, Germany, January 1990.

J. H. Johnson. INR: A program for computing finite automata. unpublished manuscript, Department of Computer Science, University of Waterloo, Waterloo, Canada, January 1986.

R. Kazman. Structuring the text of the Oxford English Dictionary through finite state transduction. Technical report CS-86-20, Department of Computer Science, University of Waterloo, Waterloo, Canada, June 1986.

R. Kurki-Suonio. Describing automata in terms of languages associated with their peripheral devices. Technical report STAN-CS-75-493, Computer Science Department, Stanford University, Stanford, California, 1975.

E. Leiss. REGPACK: An interactive package for regular languages and finite automata. Technical report CS-77-32, Department of Computer Science, University of Waterloo, Waterloo, Canada, October 1977.

T.K.S. Leslie. Efficient approaches to subset construction. Technical report CS-92-29, Department of Computer Science, University of Waterloo, Waterloo, Canada, April 1992.

M. LoSacco and S. Rodger. FLAP: A tool for drawing and simulating automata. *ED-MEDIA 93, World Conference on Educational Multimedia and Hypermedia*, pages 310–317, June 1993.

D. Raymond and D. Wood. The *Grail* papers, version 2.0. Technical report CS-94-01, Department of Computer Science, University of Waterloo, Waterloo, Ontario, January 1994.

K. Salomaa, D. Wood, and S. Yu. Rediscovering pushdown automata. unpublished manuscript, Department of Computer Science, University of Western Ontario, London, Ontario, January 1994.

J.G.N.M. van der Zanden. FADELA: Finite Automata DEbugging Language. Master's thesis, Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, August 1990.

D. Wood. *Theory of Computation*. John Wiley & Sons, New York, N.Y., 1987.