Grail: Engineering Automata in C++

Version 2.5

Darrell Raymond[1]
Derick Wood[2]

March 1996

1 Department of Computer Science, University of Western Ontario, London, Canada
2 Department of Computer Science, Hong Kong University of Science and Technology, Kowloon, Hong Kong

TABLE OF CONTENTS

INTRODUCTION

> I saw the Holy Grail, All pall'd in crimson samite.
>
> Tennyson, *Holy Grail*

> They seemed to seek some Hofbrauhaus of the spirit like a grail, hold a krug of Munich beer like a chalice.
>
> T. Pynchon, *V*

> This equipment can be used to counter heat-seeking missiles such as the Soviet SA-7 Grail shoulder-fired weapon, now extensively deployed in Third World countries.
>
> Daily Telegraph, Nov. 22, 1985

> We can't go doddering across Malaya behind an inspired crackpot following the Holy Grail, can we?
>
> H.M. Tomlinson, *Gallions Reach*

> The Edge was Fox's grail, that essential fraction of sheer human talent, nontransferable, locked in the skulls of the world's hottest research scientists.
>
> W. Gibson, *New Rose Hotel*

Grail is a symbolic computation environment for finite-state machines, regular expressions, and finite languages. Using *Grail*, one can input machines or expressions, convert them from one form to the other, minimize, determinize, complement, and perform many other operations. *Grail* is intended for use in teaching, for research into the properties of machines, and for efficient computation with machines.

This paper provides a basic introduction to *Grail* and describes some of its history and development. If you want to use *Grail*, you should also consult the *User's Guide to Grail* and the *man* pages for the individual filters. If you are installing *Grail*, or if you want to write C++ programs that use *Grail*, consult the *Programmer's Guide to Grail* and the *Release Notes*.

Grail is written in C++. It can be accessed either through a process library or through a C++ class library. The process library is used much like other filters; from a command shell, a user can execute processes on files or input streams, generating output that can be filtered by other processes. The C++ class library can be compiled into applications that need direct access to *Grail*, or that wish to minimize the costs of stream I/O.

The name 'grail' isn't necessarily an acronym, though it could be. In the past, we have sometimes suggested that *Grail* stands for something like '`G`rammars, `r`egular expressions, `a`utomata, `l`anguages' (we've never come up with something convincing for the `i`!). It's probably just as reasonable to think of our *Grail* experience as a search for the hofbrauhaus of formal language theory.

FEATURES OF *Grail*

Version 2.5 of *Grail* enables you to manipulate parameterizable finite-state machines, regular expressions, and finite languages. By 'parameterizable', we mean that the alphabet is not restricted to the usual twenty-six letters and ten digits. Instead, all algorithms are written in a type-independent manner, so that any valid C++ base type and any user-defined type or class can be the alphabet of a finite-state machine, regular expression, or finite language.

Regular expressions in *Grail* use the conventional notation of the theoretical community. *Grail* supports catenation, union, and Kleene star for regular expressions, along with parentheses to specify precedence (complement is not supported). The following are examples of regular expressions acceptable to *Grail*:

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

The expression {} denotes the empty set, and the expression "" denotes the empty string.

The traditional representation for automata is the 5-tuple:

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

where Q is the set of states, Σ is the input alphabet, δ is a partial relation $\delta : Q \times \Sigma \rightarrow \{Q\}$, $s \in Q$ is the start state, and $F \subseteq Q$ is a set of final states. In *Grail*, we represent machines as sets of instructions. A machine that accepts the language ab, for example, is specified by:

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

Each instruction is a triple consisting of a source state, an instruction symbol, and the corresponding 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 on the input tape. The states `(START)` and `(FINAL)` are *pseudo-states*; they simply 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 instruction relation. The alphabet of the machine is given implicitly; it is the set of symbols that appear in (non-pseudo) instructions. *Grail*'s machines differ from conventional machines in that we permit multiple start states as well as multiple final states. *Grail*'s machines are also parameterizable.

A finite language is simply a set of words (each of finite length) constructed from the characters of the alphabet.

To the user, *Grail* is a set of individual filter programs that operate on streams containing descriptions of finite-state machines, regular expressions, or finite languages. Most filters take a *Grail* object (that is, a finite-state machine, regular expression, or finite language) as input, and produce a *Grail* object as output. Objects 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

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

The filter *retofm* converts an 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, a filter that converts the machine back into a regular expression (folded here to fit onto the page):

```
% echo "(a+b)*(abc)" | retofm | fmtore
((aa*a+ba*a+a+b)(b+ba*a)*ba*aab+aab+aa*aab+ab+ba*aab+
((aa*a+ba*a+a+b)(b+ba*a)*b+b)ab)c
```

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 above filter command

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

can also be written directly in C++:

```
#include        "grail.h"

main()
{
        re<char>   r;
        char*      example = "(a+b)*(abc)\n";

        istrstream(example, strlen(example)) >> r;
        r.fmtore(r.retofm());

        cout << r << endl;
}
```

In the above program, the `istrstream` function is used to convert an internal string into input to be read as a regular expression; the `retofm` function converts the expression into a machine, and the `fmtore` function converts it back to an expression.

Grail's algorithms are independent of the type of the alphabet defined. We can have, for example, machines whose alphabet is the (infinite) set of ordered pairs of integers:

```
(START) |- 0
0 [1,2] 1
1 [2,2] 1
1 [3,4] 2
2 -| (FINAL)
```

Each of *Grail*'s filters can be compiled to work with this symbol set; thus, we can convert such a machine to a regular expression (of ordered pairs), enumerate its language (which is a set of strings of ordered pairs), and so on.

Grail's DESIGN

Most tools for working with machines and expressions are designed for a specific application, such as program parsing. *Grail*, on the other hand, is designed to be a general-purpose package for symbolic computation with machines and expressions. We intend for *Grail* to (eventually) fill all of the following needs:

- research

  Grail should facilitate the theoretical and practical investigation of machines and expressions, and the development of new algorithms for processing them. *Grail* has already been useful in investigating the properties of subset construction (Leslie 92).

- education

  Grail should facilitate teaching about machines. In part, it should do this by making it easier to experiment with machines, but we also hope that *Grail* will add a leavening of engineering to a subject that is mostly taught as theoretical mathematics. *Grail* has already been used for undergraduate teaching.

- application

  Grail should facilitate the use of machines in solving applied problems, such as protocol testing, embedded state machines, executing concurrent processes, parsing, string searching, and any other application that can be described by machines or expressions. Users of *Grail* have employed it for grammar transduction and natural language processing.

The key theme of *Grail*'s design is modularity. We seek modularity not just because it is the generally accepted route to a good software design, but because we expect that adding new facilities to *Grail* and developing new uses for *Grail* will be the most common activity of both its users and its designers. Modularity in *Grail* arises in four important areas:

- philosophy

  Other approaches to software for machines assume that minimal, deterministic machines are the desired end result of all processing. In *Grail* we do not make this assumption; we treat machines, languages, and expressions as equal first-class objects. Programmers will find in *Grail* a collection of useful tools and a number of ways to connect the tools to address new and interesting problems in formal language theory. Moreover, we intend to make many algorithms and implementations of algorithms accessible within *Grail*, both the (apparently) inefficient as well as the efficient, in order to facilitate experimentation and study, as well as to generate test cases.

- process-based software

  Instead of developing yet another language for writing machine programs, *Grail* is based on a set of individual processes that can be accessed by any command shell or any program that is capable of launching processes. Processes are modules whose encapsulation is enforced by the operating system; a process-based approach encourages programmers to develop simple, generally-applicable tools. A second advantage of this approach is that it is easy to distribute computation; by using the capabilities of `rsh` to set up internet pipes, we can run processes on different machines. A third advantage is that a process-based approach separates language issues from machines processing. It also leverages users' knowledge of shell programming; rather than requiring users to learn a new language, users can exploit *sh*, *csh*, *ksh*, *bash*, *perl*, and many other languages.

- textual interchange

  A multiple-process design requires some form of interprocess communication, since processes cannot access each others' data. We use a textual description of machines and regular expressions as the intermediary for *Grail*. Each process reads a textual description of the input machine, converts it into an internal form, processes it, and writes a textual description of an output machine. The advantage of this approach is that the input and output can be read, edited, and manipulated by standard utilities such as *vi*, *sort* and *wc*. The disadvantage is the extra cost of encoding and decoding between the language and internal forms, and the cost of process invocation and switching. In our experience, this cost is small compared to the cost of (for example) converting between objects or producing minimized machines.

- C++ class library

> C++ encourages encapsulation and the definition of interfaces, and hence encourages modularity in low-level code. In addition, we make extensive use of template classes, which in effect define operations on 'black boxes' that are ready to be instantiated with the user's choice of modules.

Grail's 41 filters are listed in Table 1.

A SHORT HISTORY OF *Grail*

Grail was preceded by two packages written at the University of Waterloo. The earlier effort was Leiss's `REGPACK` (Leiss 79), a package written in 1977 to support experimentation and research with finite-state machines. `REGPACK`, written in SPITBOL, supported the conversion of nondeterministic machines to deterministic machines, minimization of deterministic machines, and construction of syntactic monoids. While `REGPACK` did not directly influence the current effort, it is interesting to note that Leiss's goal of an environment for experimentation with machines is still one of our primary goals.

A program with more direct influence on *Grail* was Howard Johnson's `INR` (Johnson 86). `INR` was developed because of Johnson's interest in rational relations and their use in defining string similarity (Johnson 83). `INR` takes rational relations (including regular expressions) as input and converts them into finite-state machines, which can then be manipulated in various ways. `INR` can produce single- or multiple-tape machines; the latter are useful for describing transducers, since one tape can be considered an output tape for the other (input) tapes.

Johnson made special efforts to ensure that `INR` was a highly efficient and powerful tool for managing machines. His goal was the effective processing of machines with thousands of states and instructions. As a result, `INR` is written very compactly in C, and is especially efficient in handling potentially costly tasks such as memory allocation, subset construction, and minimization of machines. The basic algorithms for handling such tasks are well known, but there has been relatively little attention paid to efficient implementation of these algorithms. Johnson made the effort to develop efficient implementations, with the result that `INR` was the only software system

fmcment	complement a machine
fmcomp	complete a machine
fmcat	catenate two machines
fmcross	cross product of two machines
fmdeterm	make a machine deterministic
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 states and instructions
fmrenum	canonical renumbering of a machine
fmreverse	reverse a machine
fmstar	star of a machine
fmstats	print information about a machine
fmtofl	convert a machine to a finite language
fmtore	convert a machine to regular expression
fmunion	union of two machines
flappend	append a given string to every word
flexec	execute a finite language on a given string
flfilter	find intersection of finite language and finite-state machine
fllq	left quotient
flprepen	prepend a given string to every word
flprod	cross product of two finite languages
flreverse	reverse words in a finite language
flrq	right quotient
fltofm	convert a finite language to a finite-state machine
fltore	convert a finite language to a regular expression
flunion	union of two finite languages
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 for equivalence to empty set
isnull	test for equivalence to empty string
recat	catenate two regular expressions
remin	minimal bracketing of a regular expression
restar	Kleene star of a regular expression
retofm	convert a regular expression to a machine
retofl	convert a regular expression to a finite language
reunion	union of two regular expressions

Table 1.1: *Grail* filters

capable of handling the transduction of the *Oxford English Dictionary* (Kazman 86). Even today, many of INR's capabilities are more advanced than those of other software (though we like to think that *Grail* is catching up). The present effort has borrowed INR's philosophy of combining powerful capabilities with efficient design, as well as its notation for machines.

The first project to actually use the name 'Grail' was a joint effort between Howard Johnson, Carl-Johan Seger, and Derick Wood. This project extended INR to handle context-free grammars and machines with regular expressions as instruction labels. Software developed for this project consisted of a layer of code that used INR as an underlying computational engine. After some work, this effort was discontinued.

The *Grail* project was resuscitated by the present authors in 1990. We began with the observation that some issues were not satisfactorily handled either by INR or 'old Grail.' The first issue was obscurity. In pursuit of efficiency, INR had become a somewhat complex and monolithic piece of code. The layer of software added by 'old Grail' merely increased the complexity, because it was not easily maintainable or modifiable. The lack of documentation for INR and 'old Grail' made this software difficult to understand for anyone other than its programmers. Thus, the first order of business was to develop software that was more approachable and better documented, to improve maintainability and robustness, and to ensure that many programmers could work on the software.

The second issue was modularity. Much of the difficulty of building upon INR was a result of its tightly connected structure. Adding a new algorithm for subset construction, for example, required knowing much about the internals of INR, including its data structures, memory allocation, parser, and so on. We wanted a software environment in which programmers could work on algorithms without having to learn too much about the details of the existing code. This meant that we would have to build the software in a modular fashion, devising interfaces at several levels.

The third issue was generality. Like most systems that have appeared since, INR assumed that the user wanted to input regular expressions and receive deterministic, minimized machines as output. INR did not support the user who wanted to input machines and produce regular expressions as output. We wanted *Grail* to be a general purpose manipulation language, in which one could convert machines and expressions freely, with user control over minimization and determinism.

Grail version 0.5 was written in C, and consisted of the following filters:

`cross`	compute the cross product of two machines
`lreverse`	reverse the input using empty-string instructions
`min`	minimize the input by Hopcroft's partition algorithm
`min1`	minimize the input by reversal and subset construction
`percent`	compute the alternation (i.e. $(ab)^+$) of two machines
`plus`	compute `star`$-\epsilon$ of the machine
`quest`	compute the machine$+\epsilon$
`reverse`	reverse the input machine
`star`	compute the Kleene star of the input machine
`subset`	subset construction of the input machine
`union`	compute the union of two machines

These filters accessed a library of functions that did most of the actual work (the filters themselves were essentially simple I/O routines). The library contained procedures for handling I/O and for processing machines. The idea behind this decomposition was that the filters should be efficient enough for most problems involving machines; for very large or complex problems, a competent C programmer could access the library directly and thereby avoid any inefficiency introduced by process communication.

While the filters were reasonably successful, the library was not. Our C code was not particularly reliable, readable, or reusable. This latter problem was irritating both aesthetically and as a pure engineering problem. Operations on machines and regular expressions involve frequent manipulation of container structures such as sets and relations; it would be both elegant and efficient to use a single implementation of these structures for many different contents. Using C, however, one can provide this generality only by giving up strict type checking. In spite of these problems, version 0.5 did support a significant research project on subset construction (Leslie 92).

We decided to switch to C++ to re-implement *Grail*. We made this choice of language under the impression that we would develop an elegant class hierarchy that would greatly increase code reuse and the overall robustness of the system. While we have made some use of clases, C++ has been much more important for its better clarity and robustness, which are a result of its strict type checking and encapsulation. C++'s template facility is indispensable to *Grail*, and recent versions of the software have made more extensive use of inheritance and virtual functions.

Versions 0.8 through 1.2 of *Grail* saw the development of our C++ class library, which included the classes `set`, `list`, `string`, `regexp`, `trans` (transition), `state`, `fa`, `tset` (sets of transitions), and `xfa` (extended finite machine). This latter class defines machines that have regular expressions as transition labels. The `set` and `list` classes are template classes; they and `xfa` were our first attempt to rely on C++'s ability to support code reuse. In addition to rewriting our existing code in C++, we also added more functionality–the number of filters jumped from 11 to 34. Version 1.0 introduced an automatic testing facility that was used to check that changes to code still resulted in working filters. Version 1.1 introduced an automatic profiling facility that was used to test that purported improvements actually did lead to more efficient code. Version 1.2 was subjected to quality checks, both through the use of Purify and through correcting the bugs and inconsistencies that were discovered by compiling the code with two C++ compilers that are more strict than *cfront*.

Version 2 of Grail added support for parameterizable machines and expressions. Parameterizable finite-state machines can take any type as instruction label, and parameterizable regular expressions can take any type as a symbol class. Version 2 thus dispenses with the distinction between `xfa` and `fa` (each is an instance of the new parameterizable machine class `fm`), and has extended the reach of the `regexp` class (now called `re`) beyond strings of ASCII alphabetic characters. Version 2 also dispenses with the class `tset` and makes `string` a parameterized class.

The most recent version of *Grail* is Version 2.5. This version introduces a class for finite languages (`fl`) and for memory allocation (`pool`), and exhibits improved performance for large machines.

Recently, several systems for computing with machines have appeared in the literature or have been made available over the Internet.

Bruce Watson has written a C++ toolkit for finite-state machines and regular expressions called the FIRE Engine (Watson 94a, 94b). This package has the goals of efficiency and modularity, and implements more algorithms than does *Grail*. The FIRE Engine does not come with a non-programmer interface, such as *Grail*'s filters.

Champarnaud's AUTOMATE system, written in C, supports finite-state machines and finite semigroups (Champarnaud and Hansel 91). It can compute deterministic minimal machines, syntactic monoids, and transition monoids of regular languages.

The AMORE system, written in C, supports finite-state machines, regular expressions, and syntactic monoids (Jansen *et al.* 90). It can produce minimal DFAs, handle ϵ-NFAs, and perform various tests on syntactic monoids (for example, star-freeness, finiteness, and cofiniteness). AMORE can also display its machines graphically.

Both AMORE and AUTOMATE have goals similar to those of *Grail*—to serve as a research environment, to facilitate the study of machine implementations, and to provide a package for executing machines for other purposes (such as validating concurrent programs). Where *Grail* differs is in its emphasis on providing a full symbolic computing environment; in its provision of both filters and a class library; and in the fact that *Grail* does not attempt to provide its own graphical user interface or programming language. AMORE and AUTOMATE appear to be monolithic programs that attempt to provide a single interface to the user.

One use of machines is for hardware verification and protocol checking. FANCY, the Finite AutomatoN Checker of nancY, is Stefan Krischer's tool for formal hardware verification. It provides equivalence and inclusion checking for finite-state machines and is accessible through a graphical user interface.

FADELA, the Finite Automaton DEbugging LAnguage, is a project directed by Gjalt de Jong (van der Zanden 90). FADELA is designed to investigate ω-regular languages (that is, regular languages whose words are of infinite length). FADELA supports the production of deterministic Müller machines, and can convert these machines into regular expressions. FADELA also supports other operations on machines including minimization and complement.

An interesting experience is the development of machine tools in Nuprl, a proof language based on the lambda calculus (Kreitz 86). Definitions were constructed in Nuprl for finite sets, strings, tuples, and deterministic machines. Nuprl was then able to construct a proof of the pumping lemma. The main point of this work was not the development of an environment for manipulating machines, but an illustration of the utility of the Nuprl proof development system.

We know of several other systems whose motivation is primarily pedagogical. An early effort was GRAMPA, which was only partially implemented (Barnes 72). More recently, Hannay has built a Hypercard-based system for simulating machines (Hannay 92). This program appears to be useful for introductory teaching purposes, and for simulating small machines. FLAP, the Formal Languages and Automata Package, comes from Rensselaer Polytechnic Institute. FLAP supports the drawing and execution of finite-state machines, pushdown machines and Turing machines. FLAP can handle nondeterministic machines, provides the ability to step through the execution of a machine, and supports paper output (LoSacco and Rodger 93). The COLOS project at the University of Milan has a system called AUTOMATA, which permits students to work with finite-state automata, push-down automata, Mealy and Moore machines, regular expressions, and context-free grammars. It includes an X windows interface. Finally, *Turing's World* is a program for teaching the basics of finite-state machines and Turing machines (Barwise and Etchemendy 93). This program's strength is a nice graphical interface to the machines.

In addition to these systems, there is a vast amount of work on using grammars and machines in applications. Many operating system utilities understand a limited form of regular expression, for example, and almost every text editor provides general-purpose search-and-replace capabilities. The machines used in such tools are generally custom built, or perhaps adapted from custom code; operating systems have yet to offer a standard machine package for handling parameterizable machines and expressions in the same way that they offer parameterizable sorting and searching routines.

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 the code, mainly because of strict type checking. The C++ compiler has resisted many questionable constructs that were unquestioningly accepted by C. Consequently, programming bugs and errors less frequently show up in low-level operations. When bugs do appear, they are now almost always incorrect specifications of algorithms.

Grail has also taught us some lessons that apply to the construction of mathematical libraries in general. One lesson is that a library of routines is only half the battle; the other half is in developing a library of test data, and in 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 sufficient; 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 soon 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 it 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 behaviour is small (Leslie 92). 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. Another instance of this behavior is reported by Bruce Watson, who notes that Brzozowski's algorithm for minimization (applying reversal and subset construction twice) performs better than Hopcroft's algorithm in practice, even though worst-case analysis of the two algorithms suggests the opposite (Watson 95).

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.

HOW DO I OBTAIN *Grail*?

Grail is available without charge to researchers and students, or anyone who wishes to use the software for their own private education. Version 2.5 of *Grail* can be obtained from the Grail project's World Wide Web site, at

`http://www.csd.uwo.ca/research/grail`

This site also contains links to many other automata research projects.

Grail is not in the public domain. It cannot be sold, used for commercial purposes, or included as part of a commercial product without our permission.

ACKNOWLEDGEMENTS

This research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada. Darrell Raymond can be reached at `drraymon@csd.uwo.ca`. Derick Wood can be reached at `dwood@cs.ust.hk`.

REFERENCES

Barnes 72 K.R. Barnes, *Exploratory Steps Towards a Grammatical Manipulation Package (GRAMPA)* M.Sc. Thesis, McMaster University, Hamilton, Canada (1972).

Barwise and Etchemendy 93 J. Barwise, J. Etchemendy, *Turing's World 3.0: An Introduction to Computability Theory*, Center for the Study of Language and Information, Stanford, California (1993).

Champarnaud and Hansel 91 J.M. Champarnaud, G. Hansel, "AUTOMATE: A Computing Package for Automata and Finite Semigroups", *Journal of Symbolic Computation* **12** p. 197-220 (1991).

Hannay 92 D.G. Hannay, "Hypercard Automata Simulation: Finite-State, Pushdown, and Turing Machines", *SIGSCE Bulletin* **24**(2) p. 55-58 (June 1992).

Jansen *et al.* **90** V. Jansen, A. Potthoff, W. Thomas, U. Wermuth, "A Short Guide to the AMORE System", *Aachener Informatik-Berichte* **90**(02), Lehrstuhl für Informatik II, Universität Aachen, Aachen, Germany (January 1990).

Johnson 86 J.H. Johnson, "INR: A Program for Computing Finite Automata" unpublished manuscript, Department of Computer Science, University of Waterloo, Waterloo, Canada (January 1986).

Johnson 83 J.H. Johnson, "Formal Models for String Similarity", CS-83-32 Department of Computer Science, University of Waterloo, Waterloo, Canada (November 1983).

Kazman 86 R. Kazman, "Structuring the Text of the Oxford English Dictionary Through Finite State Transduction", CS-86-20, Department of Computer Science, University of Waterloo, Waterloo, Canada (June 1986).

Kreitz 86 C. Kreitz, "Constructive Automata Theory Implemented with the Nuprl Proof Development System", TR-86-779, Department of Computer Science, Cornell University, Ithaca, New York (September 1986).

Leiss 77 E. Leiss, "REGPACK: An Interactive Package for Regular Languages and Finite Automata", CS-77-32, Department of Computer Science, University of Waterloo, Waterloo, Canada (October 1977).

Leslie 92 T.K.S. Leslie, "Efficient Approaches to Subset Construction", CS-92-29, Department of Computer Science, University of Waterloo, Waterloo, Canada (April 1992).

LoSacco and Rodger 93 M. LoSacco, S. Rodger, "FLAP: A Tool for Drawing and Simulating Automata" *ED-MEDIA 93, World Conference on Educational Multimedia and Hypermedia* p. 310-317 (June 1993).

van der Zanden 90 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).

Watson 94a B.W. Watson, "An Introduction to the FIRE Engine: A C++ Toolkit for Finite Automata and Regular Expressions", Computing Science Note 94/21, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands (April 1994).

Watson 94b B.W. Watson, "The Design and Implementation of the FIRE Engine: A C++ Toolkit for Finite Automata and Regular Expressions", Computing Science Note 94/22, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands (April 1994).

Watson 95 B.W. Watson, "Taxonomies and Toolkits of Regular Language Algorithms", Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands (1995).