Natural Language Engineering **2** (4) 295–302. Printed in the United Kingdom

© 1996 Cambridge University Press

Implementing and using finite automata toolkits

BRUCE W. WATSON

Ribbit Software Systems Inc.
IST Technologies Research Group, Box 24040, 297 Bernard Ave., Kelowna, B.C., V1Y 9P9, Canada
e-mail: watson@RibbitSoft.com

(*Received 1 December 1996; Revised 9 February 1997*)

Abstract

Finite automata and various extensions of them, such as transducers, are used in areas as diverse as compilers, spelling checking, natural language grammar checking, communication protocol design, digital circuit simulation, digital flight control, speech recognition and synthesis, genetic sequencing, and Java program verification. Unfortunately, as the number of applications has grown, so has the variety of implementations and implementation techniques. Typically, programmers will be confused enough to resort to their text books for the most elementary algorithms. Recently, advances have been made in taxonomizing algorithms for constructing and minimizing automata and in evaluating various implementation strategies Watson 1995. Armed with this, a number of general-purpose toolkits have been developed at universities and companies. One of these, FIRE Lite, was developed at the Eindhoven University of Technology, while its commercial successor, FIRE Engine II, has been developed at Ribbit Software Systems Inc. Both of these toolkits provide implementations of all of the known algorithms for constructing automata from regular expressions, and all of the known algorithms for minimizing deterministic finite automata. While the two toolkits have a great deal in common, we will concentrate on the structure and use of the noncommercial FIRE Lite. The prototype version of FIRE Lite was designed with compilers in mind. More recently, computation linguists and communications protocol designers have become interested in using the toolkit. This has lead to the development of a much more general interface to FIRE Lite, including the support of both Mealy and Moore regular transducers. While such a toolkit may appear extremely complex, there are only a few choices to be made. We also consider a 'recipe' for making good use of the toolkits. Lastly, we consider the future of FIRE Lite. While FIRE Engine II has obvious commercial value, we are committed to maintaining a version which is freely available for academic use.

1 Introduction

In this paper, we present the structure and use of FIRE Lite (referred to as 'the toolkit' throughout this paper) — a toolkit for dictionaries, regular acceptors (simple finite automata) and regular transducers. The toolkit grew out of my Ph.D. thesis research Watson 1995, which had a particular emphasis on compiler-oriented algorithms. For commercial use and applications, the toolkit has been superseded by FIRE Engine II from Ribbit Software Systems, Inc., while FIRE Lite is still available for academic use, see `http://www.RibbitSoft.com/firelite/firelite.html`. The toolkit has a number of unique features:

https://doi.org/10.1017/S135132499700154X Published online by Cambridge University Press

- The toolkit was directly implemented from a taxonomy of algorithms for constructing finite automata. The taxonomy, which is fully presented in Watson 1995, provides full correctness arguments for the algorithms in a format which is easy to understand and which facilitates comparison of the algorithms. The use of the taxonomy ensured that the toolkit was extremely easy to implement and debug.
- All of the known finite automata construction algorithms are implemented in the toolkit. This enables client programmers to choose the algorithm and the type of automata they wish to use.
- The automata implementations are structured in such a way that multiple threads in the client program can make use of a single finite automaton.

There are a number of other general purpose finite automata toolkits — perhaps the most important being Grail, developed by Darrell Raymond and Derick Wood at the University of Waterloo and the University of Western Ontario Raymond and Wood 1996. Grail is an extremely flexible toolkit, well implemented and of great educational value. It is, however, a bit less efficient than FIRE Lite. Choosing between the two toolkits can be particularly difficult and usually comes down to a choice between flexibility and performance.

Much more extensive documentation on both FIRE Lite and FIRE Engine II is provided in Watson 1995 and on the web site `www.RibbitSoft.com`. We begin with a description of regular acceptors in the toolkit; this is followed by a description of transducers and a guide to making use of the toolkit. We conclude with an outline of future directions for the toolkit. In this paper, we assume some familiarity with the C++ programming language.

2 The structure of FIRE Lite

FIRE Lite is implemented as a set of C++ classes, making it a pure class library (it does not contain any functions/procedures and is fully object-oriented). A single class, `FAAbs`, forms the root of the inheritance hierarchy for all of the types of finite automata; the class defines a common interface for all of the different types of automata. Three concrete types of automata inherit from `FAAbs`: `FADFA` (deterministic automata), `FAEFFA` (automata without transitions on ε, the empty string), and `FAFA` (general, possibly nondeterministic, automata). Each of these classes has a constructor which is a template member function (it is type-parameterized). The single template argument to each of these automata is known as the *automaton seed* type. For each type of automaton, there are a number of choices for the seed, each of which leads to the use of a different automaton construction algorithm (again, each having different performance characteristics).

Class `RE` implements regular expressions while class `ERE` implements extended regular expressions (with complementation and intersection).

In the following subsections, we consider some of the specific implementation issues.

https://doi.org/10.1017/S135132499700154X Published online by Cambridge University Press

2.1 How FAs are built

Each of the concrete finite automaton template classes uses the automaton seed object to construct the transitions and the set of final states. The initial seed represents the start state of the automaton. It encodes additional information about the regular expression from which it was constructed. The automaton constructor will then map each such seed object to an integer representation of the state, which is, of course, much more time and space efficient than the full seed representation. The automaton template class can then request from the seed the set of out-transitions (from the state which the seed represents), yielding other seeds which will, in turn, be mapped to their own states. It can also ask the seed whether it represents a final state.

Following the construction of the transitions and the final states, all of the temporarily constructed seeds are destroyed. The result is an automaton which occupies very little memory, allowing automata which consist of millions of states.

2.2 Implementation techniques: templates versus inheritance

There are two basic philosophies for structuring class libraries in C++. We consider how an automaton toolkit, containing a number of different types of finite automata, would be implemented with the two techniques. In the first solution, an automaton class would be placed at the root of the inheritance hierarchy. It would declare virtual member functions which define the functionality of finite automata. It would not, however, give implementations of the automaton classes. The concrete automata classes would inherit from the root class, they would each define their own local data, depending upon the particular form of automaton, and they would override the root-defined member functions, providing an implementation, with the same externally-visible functionality, which uses the data structures specific to the automaton.

This solution allows a program to create automata and pass around pointers to the automata, while only declaring the pointers as having type 'pointer to the automaton base class'. The programmer can later change the type of the actual automaton in the code which constructs it without recompiling the entire application *(binary-level code sharing)*. The disadvantage to this solution is the time overhead of a virtual function call, usually tens, or even hundreds, of machine cycles, frequently one for each input symbol processed.

In the second solution, common data structures and functionality in the different types of automata would be factored out into a template class. The differences between the specific automata would then be placed in classes to be used as template arguments. This approach, part of a field known as *generic algorithms,* is heavily used in the C++ Standard Template Library Stepanov and Lee 1994.

This solution does not allow programs to pass around automaton pointers which are independent of the particular automaton type. This, in turn, leads to the requirement to recompile a program when a new automaton type is to be used (*source-level code sharing*). This solution has a speed advantage, where a C++ optimizing compiler is able to make inline calls to the member functions, yielding function calls of only a few machine cycles.

https://doi.org/10.1017/S135132499700154X Published online by Cambridge University Press

2.3 Multi-threading

When actually using an automaton to process a string, all of the *current state* information for an automaton is kept outside the automaton itself in an object of class `Configuration`. Most toolkits, including the predecessor of FIRE Lite known as FIRE Engine Watson 1994a; Watson 1994b, store the current state within the automaton. They then provide automaton member functions to advance along transitions to the next state, or to reset the automaton to the start state. Under that model, a single automaton cannot be shared between multiple threads in a program. A program with 1000 threads, each needing the same automaton, would have each thread create a copy of the automaton, leading to 1000 copies of the same data structure, potentially a massive waste of memory and run-time resources.

The interface to FIRE Lite makes use of *call-backs*. The client code invokes a 'process input' member function of the automaton, passing it a pointer to the input string to be processed, and a pointer to a function known as the *call-back* function. When the automaton member function makes a transduction or enters an accepting (final) state while processing the input string, it calls the call-back function, passing it information such as the amount of string processed and the most recent transduction. Since any number of threads may simultaneously invoke the automaton's member function, each passing a call-back function, no additional copies need to be made of the automaton. Unfortunately, the call-back interface proved to be clumsy for programmers to use and requires the programmer to provide a call-back function; the current interface to FIRE Engine II was developed as a result.

In FIRE Engine II, each thread need only maintain its own `Configuration`. It can reset the automaton to the start state by resetting the `Configuration`. The finite automaton member functions that make a transition take a reference to the `Configuration`, designating the current state. All of the threads are then able to share the same finite automaton. This has been particularly useful in compilers and pattern matching applications.

2.4 Data structuring decisions

The entire toolkit is implemented for maximum performance. States are encoded as integers —which allows us to implement sets of states as bit vectors. For example, testing whether the current state is a final state is a simple bit operation on most machines— usually resulting in fewer than three machine instructions. The set of transitions out of a particular state is implemented as an array of label/destination state pairs[1]. Making a transition out of a state is as simple as traversing the array, looking for the matching transition label. Again, this is a particularly high-speed operation on most current processors. Finally, the mapping from a state to its set of transitions is done through another array, indexed by the state number.

All of these details are comfortably hidden within the C++ classes of the toolkit.

[1] In FIRE Engine II, a number of alternative implementations are provided to cater to different types of automata (such as ones with dense versus sparse transition graphs) found in various application domains.

https://doi.org/10.1017/S135132499700154X Published online by Cambridge University Press

The level of abstraction does, however, require a high quality C++ compiler, with inlining of member function invocations, to obtain good performance.

3 Beyond simple automata

In this section, we discuss the structure of more complex automata and transducers within FIRE Lite.

3.1 Complex transition labels

Traditionally, the C++ primitive data type `char` has been used as the transition label. In some of the newer application areas, it may be more applicable to use a much smaller alphabet (such as *a, c, g,* and *t* for genetic applications) or a much larger one (such as using an entire class).

In FIRE Lite, arbitrary alphabets are supported through type parameterization (making them templates) of all automata classes by the type of the transition labels. Traditional automata are then implemented by using `char` (for ASCII characters) or `wchar_t` (for Unicode) as the template argument. More complex labels can then be implemented by providing their type as the template argument.

The toolkit's use of transition labels goes a step further than simply allowing transitions of a different type. When making a transition, traditional automata compare the transition label with the input symbol, with equality meaning that the transition is taken. A generalization of this is to consider the label to be a predicate, taking a single argument of the type of the input symbols. For the given current state, only those out-transitions with predicate labels which hold (are *true*) are taken. The predicates are actually implemented as C++ objects, which support the `operator()` member function. Supporting that member function means that the label itself can be used syntactically the same way a predicate would be used. Since additional information can be stored in such 'predicate objects' between applications of the predicate, the predicate can be an arbitrary computation. Of course, such arbitrary transductions take us beyond the realm of regular transductions; the implementation is new enough that we have not yet seen a practical use of nonregular transduction.

3.2 Mealy (transition-output) transducers

Mealy transducers are a special type of automaton which provide an output every time a transition is taken while processing the input. They can be cascaded (the output of one is used as the input of another) to create complex transformations on the input string, while using much less memory than would be required for the same transformation with a single transducer.

Mealy transducers are implemented by embedding the output information in the transition labels. The labels are still predicates (deciding whether a transition is taken), but they also support a member function which provides the corresponding output. The member function is frequently as simple as returning a constant value.

https://doi.org/10.1017/S135132499700154X Published online by Cambridge University Press

However, they could conceivably be an arbitrary computation, much the same as the predicates.

One remaining difficulty is how to cope with the differences between deterministic and nondeterministic Mealy transducers. In a deterministic transducer, the automaton will only be in one state at a time and at most one out-transition from the current state will be valid for the input symbol, resulting in a single output symbol. In a nondeterministic transducer, the automaton may be in a *set* of current states and a number of out-transitions may be applicable for the input symbol. This, in turn, leads to a *set* of output symbols for the single input symbol. The problem is solved by defining an associative and commutative (binary) operation on the output alphabet. In a nondeterministic transducer, the operator is applied to the set of output symbols, yielding a single output symbol for every input symbol. The deterministic transducers are constructed such that their output is the same as the output of the nondeterministic transducer with the operator applied. (As we explain later, a similar transduction difference exists for deterministic and nondeterministic Moore transducers. Unfortunately, no solution has yet been found for the difference between those types of Moore transducers.)

4 Planning your use of FIRE Lite

Despite the seemingly complex implementation of toolkits such as FIRE Lite and Grail, using them within other software is remarkably simple. The following sequence of development steps can be used to begin using the toolkit. First, choose an input alphabet. Typically, this is the `char` or `wchar_t` primitive data types in C++, however, it could also be an enumeration (`enum`) data type representing some other alphabet. Through the use of predicates as transition labels, a much more complex input alphabet could also be used, such as larger structures, objects, or arrays.

Next, choose a type of finite automaton. The choices range from a deterministic automaton, having the fastest transition time, but the slowest construction time, to a general nondeterministic automaton, having a relatively slow transition time, but extremely fast construction times. The precise performance characteristics of the various types of automata are detailed in Watson 1995.

Finally, if the automaton is to be used as a transducer, outputs will be required for each transition. The output portion of a transition's label is usually a simple addition to the normal transition label. For example, an automaton which is transducing `char` input to `float` output will only need a simple transition label type such as `class TDLabel { char in; float out; };` In addition to this, when a deterministic transducer is constructed from a nondeterministic one, some of the transition labels (with identical inputs, or `in` fields) will be combined. This requires that the user define a label combination operator. In the above example, we could take the *average* of the two `float` components of the labels being combined.

https://doi.org/10.1017/S135132499700154X Published online by Cambridge University Press

5 Future enhancements

The next version of FIRE Lite will contain a number of enhancements to the regular transduction provided in the current version[2]. In particular, the following deficiencies will be addressed with new features:

- The current (experimental) implementation of Moore machines displays a few anomalies. Mealy machines are implemented in such a way that the transduction given by a nondeterministic Mealy transducer will be identical to that given by a deterministic Mealy transducer. With the current implementation of Moore machines, a deterministic Moore transducer may not give a transduction which is identical to a nondeterministic Moore transducer constructed from the same specification.
- In the current FIRE Lite version, only regular expressions can be used to construct automata from scratch. Two additional methods for automata construction will be supported in the future: regular (also known as *linear*) grammars and graphical tools. With graphical tools, a finite automaton could be put together by a user one transition (or state) at a time. Preliminary exploration shows that such graphical construction would be more of interest to computational linguists, educators and communications protocol designers than to compiler writers.
- At present, automata must be nonrecursive. Transition labels must be simple objects, either predicates or mappings which produce a Mealy output. A future version of the toolkit will allow automata in which transition labels are automata themselves. This could allow recursively structured automata, in which an automaton is also one of its own transition labels — perhaps yielding succinct forms of nonregular transduction.

6 Conclusions

In this paper, we have outlined the structure and implementation of a toolkit called FIRE Lite and hinted at the core structure of FIRE Engine II toolkit. The toolkit provides extensive support for regular acceptors and regular transducers. All of the known algorithms for constructing automata are implemented in FIRE Lite, using their abstract versions presented in the taxonomy in Watson 1995. The toolkit has already been heavily used in compilers and in various pattern matching applications — both of which have made extensive use of multithreading. Extensive benchmarking, also reported in Watson 1995, combined with client programmer experiences with other toolkits shows that FIRE Lite's performance is significantly better than that of other toolkits. Since transduction was originally added to the toolkit at the suggestion of computational linguists, it will be interesting to see what kinds of new applications are devised for the toolkit.

[2] The current version of FIRE Engine II already contains full or experimental implementations of these enhancements.

https://doi.org/10.1017/S135132499700154X Published online by Cambridge University Press

Acknowledgements

I would like to thank András Kornai for his input in structuring this paper, Franz Guenthner for providing ideas for enhancements to FIRE Lite, Richard Watson for assisting with some core design issues, Kees Hemerik, Franz Kruseman Aretz, and Gerard Zwaan for their advice during the design of the original FIRE Engine and FIRE Lite toolkits, and Nanette Saes for proofreading the paper.

References

Carroll, M.D. and Ellis, M.A. 1995. *Designing and coding reusable C++*. Addison-Wesley, Reading, MA.

Raymond, D. and Wood, D. 1996. The Grail papers. Computer Science Technical report, University of Waterloo, Canada. Available by ftp from `CS-archive.uwaterloo.ca`

Stepanov, A. and Lee, M. 1994. STL: the C++ standard template library. Computer Science Report, Hewlett-Packard Laboratories, CA.

Watson, B.W. 1994. An introduction to the FIRE Engine: A C++ toolkit for Finite automata and Regular Expressions. Computing Science Report 94/21, Eindhoven University of Technology, The Netherlands. Available via `www.RibbitSoft.com`

Watson, B.W. 1994. The design and implementation of the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions. Computing Science Report 94/22, Eindhoven University of Technology, The Netherlands. Available via `www.RibbitSoft.com`

Watson, B.W. 1995. *Taxonomies and Toolkits of Regular Language Algorithms*. Ph.D dissertation, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands.

https://doi.org/10.1017/S135132499700154X Published online by Cambridge University Press