Implementing and using finite automata toolkits[1]

Bruce W. Watson
Ribbit Software Systems
Box 24040, 297 Bernard
Kelowna, BC V1Y 9P9
Canada
watson@RibbitSoft.com

Abstract. Finite automata (and various extensions of them) are used in areas as diverse as: compilers, digital flight control, speech recognition, 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 automata) and in evaluating various implementation strategies. Armed with this, a number of general-purpose toolkits have been developed at universities. One of these, known as the FIRE Lite, was developed at the Eindhoven University of Technology, in the Netherlands. We will consider the structure and use of this toolkit, which provides implementations of all of the known algorithms for constructing automata from regular expressions, and all of the known algorithms for minimizing deterministic finite automata.

The first edition of the toolkit was designed with compilers in mind. More recently, computation linguists have become interested in using the toolkit. This has lead to the development of a much more general interface to the toolkit, including the support of both Mealy (transition labeled) and Moore (state labeled) 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 the toolkit. While this toolkit has some obvious commercial value (and it is being commercialized), we are committed to maintaining a version which is freely available for non-commercial use.

1 INTRODUCTION

In this paper, we briefly present the structure and use of the FIRE Lite — a toolkit for regular acceptors (simple finite automata) and regular transducers. The toolkit grew out of thesis research [2], which had a particular emphasis on compiler-oriented algorithms. The toolkit has a number of unique features:

[1] A great deal of this research was performed while I was at the Eindhoven University of Technology, The Netherlands.

© 1996 B.W. Watson
Proceedings of the ECAI 96 Workshop
Extended Finite State Models of Language
Edited by A. Kornai.

- The toolkit was directly implemented from a taxonomy of algorithms for constructing finite automata. The taxonomy, which is fully presented in [2], 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 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 — such as Grail (developed by Darrell Raymond and Derick Wood at the University of Waterloo and the University of Western Ontario) [1]. 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 the toolkit is provided in [2]. 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

The 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.* 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. In the future, support will be provided for regular grammars.

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

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. The automaton template class can then request (from the seed) the set of out-transitions (from the state the seed represents); 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 the minimum amount of memory.

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:

1. 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, and would each define their own local data (depending upon the particular form of automaton), and 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 (this is known as *binary-level code sharing*). The disadvantage to this solution is the cost of a virtual function call (usually tens, or even hundreds, of machine cycles).
2. In the second solution, common data structures and functionality in the different types of automata would be factored out into a template class. The difference between the automata would then be placed in classes to be used as template arguments. (This approach, part of a field known as *generic*[2]*algorithms,* is heavily used in the C++ Standard Template Library.)
   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 (this is known as *source, but not binary, 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.

[2] Not *genetic.*

2.3 Multi-threading

All of the *current state* information for an automaton is kept outside the class, in an object of class `Configuration`. Most other toolkits (and, indeed, the first version of the FIRE Lite, known as the FIRE Engine) 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.

When using the FIRE Lite, 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 which 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 both 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. 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. The level of abstraction does, however, require a high quality C++ compiler to obtain good performance.

3 BEYOND SIMPLE AUTOMATA

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

In the FIRE Lite, the alphabets are supported type parameterizing (making them templates) all of the automata classes by the type of the transition labels. Traditional automata are then implemented by using `char` 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 happen to support the `operator()` member function. Supporting that member function means that the label itself can be used syntactically as a function or 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 (ones with output on transitions) 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, 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 ever 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 in 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 (state output) transducers. Unfortunately, no solution has yet been found for the difference.)

4 PLANNING YOUR USE OF FIRE LITE

Despite the seemingly complex implementation of toolkits such as FIRE Lite and Grail, using them in other software use is remarkably simple. The following sequence of development steps can be used to begin using the toolkit:

1. Choose an input alphabet. Typically, this is the `char` primitive data type 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.
2. 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 [2].
3. 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.

5 FUTURE ENHANCEMENTS

The next version of FIRE Lite will contain a number of enhancements to the regular transduction provided in this version. 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: from regular (also known as *linear*) grammars,

and using 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 than compiler users.

- 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 the FIRE Lite. The toolkit provides extensive support for regular acceptors and regular transducers. All of the known algorithms for constructing automata are implemented in the toolkit, using their abstract versions presented in the taxonomy in [2]. 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 [2]) shows that the toolkit'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.

Part of the toolkit is available for non-commercial use, while some newer parts of the toolkit are only available for licensing. For either type of use, please contact the author via e-mail.

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, Kees Hemerik, Franz Kruseman Aretz, and Gerard Zwaan for their assistance in developing the original FIRE Engine and FIRE Lite toolkits, and Nanette Saes for proofreading the paper.

REFERENCES

[1] D. Raymond and D. Wood, 'The grail papers', *University of Waterloo technical report series*, (1996).

[2] B.W. Watson, *Taxonomies and Toolkits of Regular Language Algorithms*, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.