LanguageLab Extensions for Deterministic Finite Automata

Thomas E. O'Neil
Computer Science Department
University of North Dakota
oneil@cs.und.edu

Abstract

The LanguageLab is a collection of Java objects that implement formal languages, automata, and grammars in a hands-on development environment. Its original implementation, published three years ago, included a graphical interface for development and testing of deterministic finite automata (DFA). This paper describes recent enhancements to the LanguageLab, specifically the addition of a cache that holds multiple DFA and new methods that implement operations on automata such as union, intersection, complement, and state minimization. The DFA Builder is one component in the general LanguageLab context of acceptors and generators for sets of strings. It is implemented using Java Swing. The DFA class itself is designed to be independent of the interface components, so the DFA tools embedded within LanguageLab can easily be extracted for use in software development and research. As an instructional resource, the LanguageLab can be used to provide a laboratory experience for courses in formal languages and automata, compilers, and software engineering

Introduction

The LanguageLab is a laboratory tool designed to support instruction and research in formal languages and automata (O'Neil, 2003). It contains a hierarchy of Java classes for the implementation and testing of acceptors and generators for sets of strings over a finite alphabet. The set of strings is realized as a FormalLanguage object. Every FormalLanguage object has an Alphabet object and a StringSpecifier object that defines what strings are in the language. StringSpecifier has subclasses called Generator and Acceptor. Generators produce sequences of strings and Acceptors test individual strings for acceptance or rejection.

The deterministic finite automaton (DFA) is one of the most widely used systems in computing for defining sets of strings. The class of languages accepted by DFAs is called the class of Regular languages. This class contains all finite sets and some infinite sets of strings. While the class is limited to relatively simple infinite sets, it has a rich set of closure properties and decision algorithms. The class is closed under the fundamental set operations of union, intersection, and complement. DFAs also have a minimization procedure in which the number of states in an automaton can be reduced to a minimum dependent only upon the language it represents. The minimization algorithm makes it possible to determine whether two distinct DFAs actually accept the same language. We can simply minimize both DFAs and check to set whether the resulting transition graphs are isomorphic.

The rich set of closure properties provides the motivation for adding these powerful operations as tools for building DFAs in the LanguageLab. DFAs have direct applications in many practical computing contexts. The operations are also instructive illustrations of how formal systems can be automatically modified and combined. The

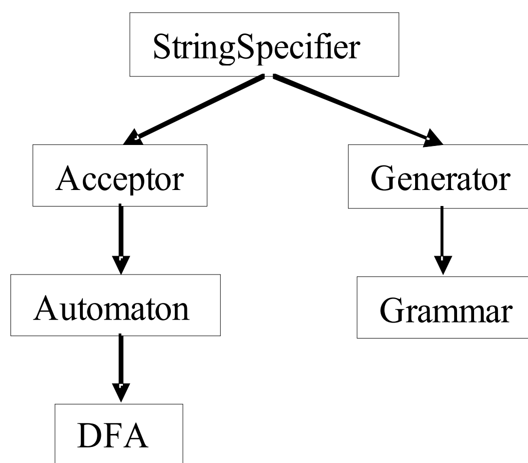

Figure 1: The StringSpecifier class hierarchy.

```
class FormalLanguage
{
    Alphabet alphabet;
    Acceptor acceptor;
    Generator generator;
    String name;
    String description;
// constructors
    public FormalLanguage();
    public FormalLanguage(Alphabet a);
// methods
    public boolean hasFixedAlphabet();
    public Alphabet getAlphabet();
    public void setAlphabet(Alphabet a);
    public String getName();
    public void setName(String n);
    public String getDescription();
    public void setDescription(String desc);
    public Generator getGenerator();
    public void setGenerator(Generator g);
    public Acceptor getAcceptor();
    public void setAcceptor(Acceptor a);
    private void setAlphabetsToAcceptor();
    private void setAlphabetsToGenerator();
}
```

Figure 2: The FormalLanguage Class

sections below describe the DFA class, its ancestors in the LanguageLab hierarchy, the operations for combining DFAs, and the interface that makes them accessible to users.

The Class Hierarchy

The FormalLanguage class represents a possibly infinite set of strings. Every FormalLanguage object has an Alphabet of symbols from which strings are made and a StringSpecifier object that determines what strings are in the language. The StringSpecifier can be either an Acceptor or a Generator. The FormalLanguage class has methods for getting and setting the language's name, its description, its alphabet, an acceptor, and a generator. A list of all the constructors and methods of the class can be found in Figure 2.

Every Acceptor object has a method that takes a string as a parameter and returns *true* or *false*, depending on whether the string is in the language of the Acceptor. An Automaton is a kind of acceptor that has states and state transition rules. An Automaton moves from state to state as it processes the characters in a string. Some of the states are designated to be accepting states. Acceptance of a string is determined by the state that the automaton reaches at the end of the computation. The constructors and methods of the Acceptor and Automaton classes are listed in Figure 3.

The deterministic finite automaton is implemented in the DFA class as an extension of the Automaton class. A complete listing of the methods of the class can be found in Figure 4. It has an alphabet, a state set, a rule set, and an *accepts*(String *s*) method, all inherited from the Automaton class. It also has methods *changeState*() and *run*(),which

```
abstract class Acceptor extends StringSpecifier
{
    public abstract boolean hasFixedAlphabet();
    public abstract boolean accepts(String string);
    protected boolean alphaTest(String string);
}
abstract class Automaton extends Acceptor
{
    StateSet stateset;
    State currentstate;
    String inputword;
    State startstate;
    int headposition;
    AutomatonRuleSet control;
    abstract public void changeState();
    abstract public void run();
    public void initialize(String inword);
    public State getCurrentState();
    public char getCurrentSymbol();
    public int getPosition();
    public String getInputWord();
    public int getStateCount();
    public StateSet getStateSet();
    public State getStartState();
    public void setStartState(State s);
    public boolean hasFixedAlphabet();
}
```

Figure 3: The Acceptor and Automaton Classes

are employed to determine the acceptance of an input string. It has methods *addState* (String *s*) and *deleteState*(String *s*), which facilitate editing of the DFA. The automaton is implemented with a StateSet object representing nodes of a graph and a DFAControl object containing a set of transition rules that represent arcs in the graph. The transition rules are stored in a matrix. This provides maximum run-time efficiency for testing the acceptance of a string, but it makes editing more complicated. If the addition of a new state exceeds the capacity of the matrix, the matrix is automatically doubled in size. The StateSet class contains a hash table that implements a mapping from state names to indexes in the control matrix. The deletion of a state causes a row to be removed from the control matrix.

Methods for Modifying and Combining DFAs

The algorithms for DFA modification require that the input DFA must have certain standard characteristics. For the purposes of the DFA class, an automaton is considered to be standard if it is both clean and complete. A clean automaton has no states that are unreachable from the start state, and a complete automaton has a transition rule for every possible *(state, input symbol)* pair. The methods that implement clean-up, completion, and standardization are *cleanCopy*(), *makeClean*(), *makeComplete*(), *renameStates*(), *standardCopy*(), *isStandard*(), and *standardize*(). Clean-up is accomplished by traversing the state transition arcs from the start state and marking the states encountered along the way. After the traversal, unmarked states are removed. If a DFA is not complete, a non-accepting error state is added to the automaton. Transitions to the error state are added

for any previously undefined *(state, symbol)* pairs. Finally, transitions are added for every input symbol from the error state back to itself.

The remaining methods of the DFA class provide implementations of the complement, intersection, union, and minimization operations. Given a DFA M, the complement operation produces a DFA that accepts the complementary set of strings. Following the standard textbook algorithm, this is accomplished by interchanging the accepting and non-accepting states of the original DFA (for example, see Kinber and Smith, 2001, p. 32). This approach works only if the set of transition rules is complete, so the *complement*() method calls the *makeComplete*() method before inverting the set of accepting states.

The union and intersection operations are very similar in their implementation. They both rely on construction of a product DFA. Given two DFAs M_1 and M_2, the state set of the product DFA is the product of the state sets of M_1 and M_2. The product DFA has a transition from state (p, q) to state (r, s) for input symbol a if M_1 has a transition from p to r on symbol a and M_2 has a transition from q to s on symbol a. Depending on how accepting states are defined, the product DFA accepts either the intersection or the union of the languages of M_1 and M_2. To get the intersection, we make (p, q) an accepting state if both p is an accepting state in M_1 and q is an accepting state in M_2. To get the union, we make (p, q) an accepting state if either p is an accepting state in M_1 or q is an accepting state in M_2. The algorithm for building a product DFA for intersection is described more formally in Hopcroft, Motwani and Ullman, 2001 (p. 135).

State minimization is the most complex of the DFA modification methods. The standard textbook algorithm is employed to determine which pairs of states can be distinguished

```
class DFA extends Automaton
{
    boolean certifiedclean = false;
// constructors
    public DFA();
    public DFA(Alphabet a);
// methods
    public Object clone();
    public void changeState()
    public void run();
    public boolean accepts (String inword);
    public void addState(String sname);
    public void deleteState(String sname);
    public DFA cleanCopy();
    public void makeClean();
    public void makeComplete();
    public void renameStates();
    public DFA standardCopy();
    public boolean isStandard();
    public void standardize();
    public DFA complement();
    public DFA intersect(DFA thatdfa);
    public DFA union(DFA thatdfa);
    public DFA minimize();
}
```

Figure 4: The DFA class.

from each other, and those states which are indistinguishable are merged (see Hopcroft, Motwani and Ullman, 2001, p. 159). A helping class called DFAMinimizer is defined to encapsulate building the table of state distinctions. Once the table of distinctions has been created, the state set and rule set are rebuilt to eliminate the unneeded states.

The Graphical Interface

The LanguageLab class provides a graphical interface for experimenting with formal languages. General Information about the language currently being viewed or developed is displayed in the top panel of the LanguageLab frame. The remainder of the frame is divided into two panels, one dedicated to testing strings for membership in the language (using an acceptor), and the other dedicated to generating a list of strings in the language (using a generator). The LanguageLab frame has a menu bar that can be used to invoke pop-up frames for creating and editing Alphabets, Acceptors, and Generators. One of the options for creating an acceptor invokes the DFA editor.

The editing interface for DFAs is shown in Figure 5. It has an information panel at the top that contains the name of the automaton, the alphabet, and the number of states. It has a central editing panel where rules can be created and edited. This panel displays the set of states and the list of rules. States can be selected from the list to be designated as final states or as the start state. The panel at the bottom of the frame allows step-by-step testing of an input string, and the panel at the right side of the frame shows a list of DFAs

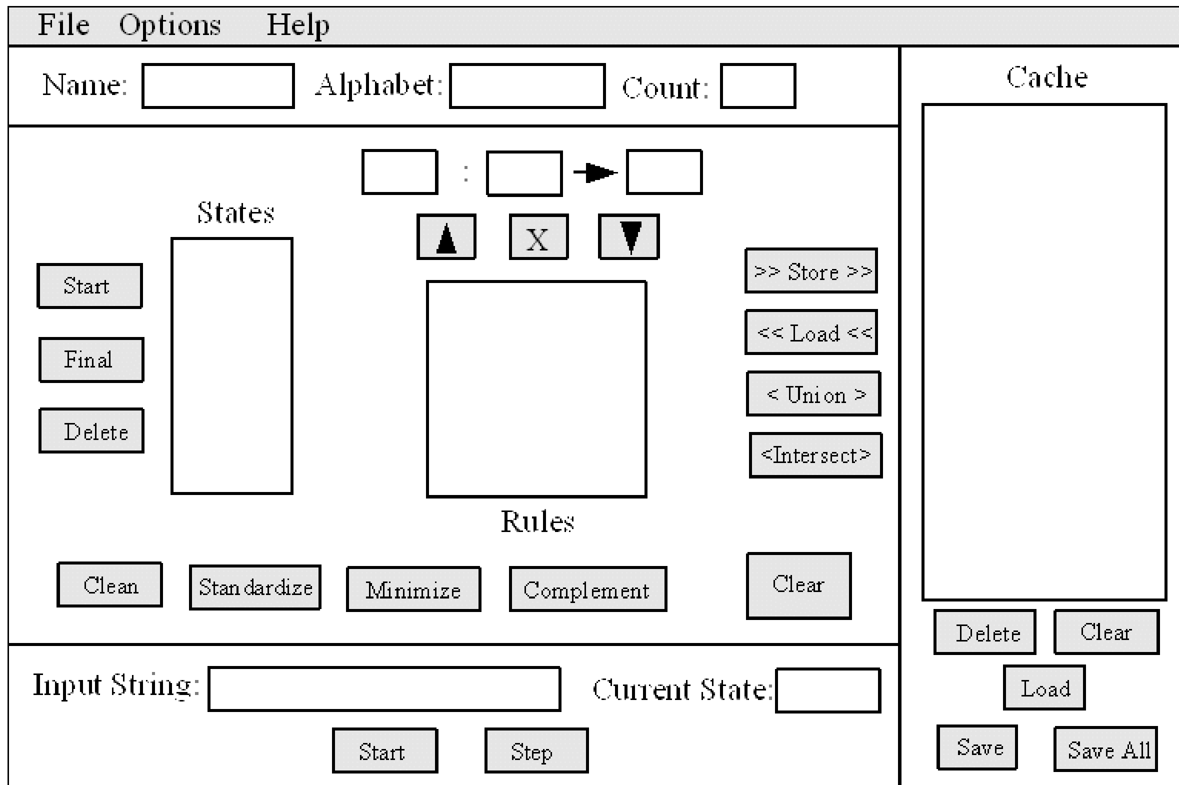

Figure 5: DFA editing interface.

that have been stored in the Cache. The central editing panel has controls for moving the current DFA to the cache, loading a DFA from the cache, intersecting the current DFA with a selected DFA from the cache, and forming the union of the current DFA with a selected DFA from the cache. The editing panel also has buttons for cleaning, standardizing, minimizing, and complementing the current DFA.

The menu bar on the DFA frame has a "File" menu which offers "Open" and "Save" options for loading a DFA from a file or storing a DFA to a file. It also has an "Apply" option to create a new FormalLanguage with the current DFA as its acceptor. When the "Apply" option is selected, control is returned to the LanguageLab frame, where further acceptance and generation tests can be conducted.

Conclusion

The DFA extensions to the LanguageLab provide a powerful set of tools for creating finite automata. Students can see textbook algorithms at work in a graphical environment that supports the development of complex specifications for regular languages. The toolset is robust and efficient enough for research and production tasks involving lexical analysis, pattern matching, and numerous other common applications of finite automata. LanguageLab can be used to provide a laboratory component for courses in formal languages and automata, compilers, and software engineering.

References

Flanagan, D. (1999). *Java in a Nutshell: A Desktop Quick Reference*. Sebastopol, CA: O'Reilly and Associates.

Hopcroft, J., R. Motwani, and J. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. Boston, MA: Addison-Wesley.

Kinber, E., and C. Smith (2001). *Theory of Computing: A Gentle Introduction*. New Saddle River, NJ: Prentice Hall.

O'Neil, T. E. (2003). "A Development Environment for Formal Languages." *Proceedings of the 36th Midwest Instruction and Computing Symposium*, Duluth, MN.