An introduction to the FIRE engine: A C++ toolkit for FInite automata and Regular Expressions

Bruce W. Watson
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven
The Netherlands
email: `watson@win.tue.nl`

May 17, 1994

Abstract

This paper is an introduction to the programmer's interface of version 1.1 of the **FIRE engine**. The **FIRE engine** is a C++ class library implementing finite automata and regular expression algorithms. The algorithms implemented in the toolkit are almost all of those presented in the taxonomies of finite automata algorithms [Wat93a, Wat93b]. None of the implementation details of the library are discussed — such design and implementation details are given in [Wat94].

The toolkit is unique in providing implementations of all of the known algorithms for constructing finite automata. The implementations, which were developed with efficiency in mind, are intended for use in production quality applications. No shell or graphical user-interface is provided, as the toolkit is intended for integration into applications. The implementations of the algorithms follow directly from the abstract algorithms appearing in [Wat93a, Wat93b]. As such, the toolkit also serves as an educational example of the implementation of abstract algorithms.

Contents

1 Introduction

The **FIRE engine** is a C++ class library implementing finite automata and regular expression algorithms. The algorithms implemented in the toolkit are almost all of those presented in the taxonomies of finite automata algorithms [Wat93a, Wat93b]. This paper serves as an introduction to the interface of version 1.1 of the **FIRE engine**. None of the implementation details of the library are discussed — such design and implementation details are given in [Wat94].

The toolkit is a computing engine, providing classes and algorithms of a low enough level that they can be used in most applications requiring finite automata or regular expressions. Unlike most of the other toolkits [RW93, CH91, JPTW90], the **FIRE engine** does not have an interactive user-interface. The toolkit (which consists of approximately 9000 lines of C++ code) is implemented as an object-oriented class library, as opposed to a procedural library, in order to reuse code as much as possible, and to minimize the cost of maintaining the code. C++ was chosen as the implementation language because compilers for C++ are more widely available than for any other object-oriented language.

The implementation of the **FIRE engine** was driven by a number of aims. Briefly, these aims are:

- To implement the algorithms efficiently enough for use in production quality applications.
- To demonstrate that the abstract algorithms appearing in [Wat93a, Wat93b] could indeed be easily (and efficiently) implemented.
- To provide implementations of all of the known algorithms; this will facilitate performance comparisons between the algorithms, and provide students with a variety of algorithms for educational purposes.
- To implement only a class library as a computing engine, without a user-interface.

The choice of these aims becomes clear when the characteristics of the other existing toolkits are considered.

The following subsections briefly outline other finite automata toolkits, the advantages and characteristics of the **FIRE engine**, future directions for the toolkit, information on obtaining and compiling the toolkit, and an outline of the structure of this paper.

Acknowledgements: I would like to thank the following people for their assistance in the preparation of this paper: Pieter 't Hoen (for implementing two of the algorithms from [Wat93b]), and Kees Hemerik, Erik Poll, Nanette Saes, and Gerard Zwaan (for providing style suggestions on this paper).

1.1 Related toolkits

There are several existing finite automata toolkits. They are:

- The AMORE system, as described in [JPTW90]. The AMORE package is an implementation of the semigroup approach to formal languages. It provides procedures for the manipulation of regular expressions, finite automata, and finite semigroups. The system supports a graphical user-interface on a variety of platforms, allowing the user to interactively and graphically manipulate the finite automata. The program is written (portably) in the C programming language, but it does not provide a programmer's interface. The system is intended to serve two purposes: to support research into language theory and to help explore the efficient implementation of algorithms solving language theoretic problems.
- The AUTOMATE system, as described in [CH91]. AUTOMATE is a package for the symbolic computation on finite automata, extended regular expressions (those with the intersection and complementation operators), and finite semigroups. The system provides a textual

user-interface through which regular expressions and finite automata can be manipulated. A single finite automata construction algorithm (a variant of Thompson's) and a single deterministic finite automata minimization algorithm is provided (Hopcroft's). The system is intended for use in teaching and language theory research. The (monolithic) program is written (portably) in the C programming language, but provides no function library interface for programmers.

- The GRAIL system, as described in [RW93]. GRAIL follows in the tradition of such toolkits as REGPACK [Lei77] and INR [Joh86], which were all developed at the University of Waterloo, Canada. It provides two interfaces:
  - A set of "filter" programs (in the tradition of UNIX). Each of the filters implements an elementary operation on finite automata or regular expressions. Such operations include conversions from regular expressions to finite automata, minimization of finite automata, etc. The filters can be combined as a UNIX "pipe" to create more complex operations; the use of pipes allows the user to examine the intermediate results of complex operations. This interface satisfies the first two (of three) aims of GRAIL [RW93]: to provide a vehicle for research into language theoretic algorithms, and to facilitate teaching of language theory.
  - A raw C++ class library provides a wide variety of language theoretic objects and algorithms for manipulating them. The class library is used directly in the implementation of the filter programs. This interface is intended to satisfy the third aim of GRAIL: an efficient system for use in application software.

The provision of the C++ class interface in GRAIL makes it the only toolkit with aims similar to those of the **FIRE engine**.

1.2 Advantages and characteristics of the FIRE engine

The advantages to using the **FIRE engine**, and the similarities and differences between the **FIRE engine** and the existing toolkits are:

- The **FIRE engine** does not provide a user interface[1]. Many of the other toolkits provide user interfaces for the symbolic manipulation of finite automata and regular expressions. Since the **FIRE engine** is strictly a computing engine, it can be used as the implementation beneath a symbolic computation application.
- The toolkit is implemented for efficiency. Unlike the other toolkits, which are implemented with educational aims, it is intended that the implementations in the **FIRE engine** are efficient enough that they can be used in production quality software.
- Despite the emphasis on efficiency in the **FIRE engine**, the toolkit still has educational value. The toolkit bridges the gap between the easily understood abstract algorithms appearing in [Wat93a, Wat93b] and practical implementation of such algorithms. The C++ implementations of the algorithms display a close resemblance to their abstract counterparts.
- Since the algorithms are direct implementations of those appearing in the taxonomies of abstract algorithms [Wat93a, Wat93b], and correctness proofs are provided for the abstract algorithms in the taxonomies, there is a high degree of confidence in the correctness of the code in the **FIRE engine**.
- Most of the toolkits implement only one of the known algorithms solving any particular problem. For example, AUTOMATE implements only one of the known finite automata construction algorithms. The **FIRE engine** provides implementations of almost all of the known algorithms for all of the problems considered in the taxonomies [Wat93a, Wat93b]. Implementing all of the known algorithms has several advantages:

[1] A rudimentary user interface is included for demonstration purposes.

  - the efficiency of the algorithms (on a given application) can be compared.
  - the algorithms can be studied and compared by those interested in the inner workings of the algorithms;

- The use of Σ-algebras in [Wat93a] can provide great computational efficiency in practice. For example, from regular expressions E_0 and E_1 we can construct finite automata M_0, M_1 (accepting the languages denoted by E_0, E_1 respectively). Assume that we now require a finite automaton accepting the language denoted by $E_0 \cdot E_1$ (their concatenation). With some of the existing toolkits, the new finite automaton would be constructed from scratch. With the **FIRE engine**, a concatenation operator on finite automata is implemented (for two of the different varieties of finite automata), enabling us to compute $M_0 \cdot M_1$ (a finite automaton accepting the desired language). This type of reuse of intermediate results can be a great computational saving.

  The complete Σ-algebra as presented in [Wat93a] is implemented in the **FIRE engine**.

- The class hierarchy in the **FIRE engine** reflects the derivation of the taxonomies in [Wat93a, Wat93b]. In the taxonomies, common parts of algorithm derivations are factored out and presented as one. In the **FIRE engine**, such common parts of derivations are implemented as base classes (thereby implementing the code reuse embodied in the factored derivation).

1.3 Future directions for the toolkit

A number of improvements to the **FIRE engine** will appear in future versions:

- Presently, the **FIRE engine** implements only acceptors. Transducers (as would be required for some types of pattern matching, lexical analysis, and communicating finite automata) will be implemented in a future version.

- The **FIRE engine** only implements Σ-algebras for three carrier sets (regular expressions, finite automata, and reduced finite automata). Future versions of the toolkit will include more Σ-algebras.

- A future version of the toolkit will include support for extended regular expressions, i.e. regular expressions containing intersection or complementation operators.

- Basic regular expressions and automata transition labels are represented by character ranges. A future version of the **FIRE engine** will permit basic regular expressions and transition labels to be built from more complex data-structures. For example, it will be possible to process a string (vector) of structures. (Version 2.0 of GRAIL includes a similar improvement.)

1.4 Obtaining and compiling the toolkit

The **FIRE engine** can be obtained by anonymous ftp from Internet node ftp.win.tue.nl (also known as 131.155.70.100) in directory /pub/techreports/pi/fire.tar.Z

The **FIRE engine** makes use of language features that are defined in version 3.0 of the C++ language. The toolkit has been successfully compiled with Watcom C/C++ 32 version 9.5. The interdependencies between classes is complex, and is fully outlined in [Wat94]. When using only part of the toolkit, instead of determining the dependencies (and thus the required files for compiling and linking), the simplest approach is to compile the entire toolkit, and use a “smart linker” to remove any excess code or data from the executable image.

A makefile compatible with Watcom Make (WMAKE) is included in the distribution.

1.5 Reading this paper

To make use of the **FIRE engine** (and to read this paper), a basic working knowledge of C++ is required. For introductory treatments of C++, see [Bud91, Bud94, Lip91, Str91]. In order to fully understand the **FIRE engine** (by reading [Wat94]), it is necessary to read the taxonomies [Wat93a, Wat93b], and to have an advanced knowledge of C++ techniques as presented in [Cop92, MeS92, Mur93, Str91].

All references to "the taxonomy" appearing int the C++ code are references to [Wat94]. The code given in this paper only includes the **public** parts. For the **private** parts, see [Wat94].

This paper is structured as follows:

- Section 2 gives a description of character ranges. Objects of this type are used as basic regular expressions and as finite automata transition labels.
- Section 3 outlines the class used to implement regular expressions. Because of the nature of Σ-algebras [Wat93a, Section 3], this class does not provide a direct means of constructing regular expressions; such a means is provided by the *Reg* template class, described in Section 4. The regular expression class simply implements the terms of the Σ-term algebra (see [Wat93a, Section 3]).
- Section 4 presents the *Reg* template class, which is used to implement the Σ-algebra given in [Wat93a, Section 3]. The template defines the operators of the Σ-algebra (see [Wat93a, Section 3]).
- Section 5 presents finite automata. An abstract finite automata class is described. All concrete automata classes inherit from this class, which is used to define a common client interface. Concrete (non-abstract) automata classes are also described. These include: finite automata, reduced finite automata, left-biased finite automata, right-biased finite automata, and deterministic finite automata.
- Section 6 gives the function prototypes of the finite automata construction functions (those functions taking a regular expression, and returning some type of finite automata).
- Section 7 gives the function prototypes of the deterministic finite automata minimization functions.

Examples of the use of the classes are spread throughout the paper.

2 Character ranges

In the **FIRE engine**, atomic regular expressions and finite automata transitions can be labeled by sets of characters. The sets of characters allowed are restricted to non-empty ranges of characters in the execution character set (which is usually ASCII or EBCDIC). Class *CharRange* is used to represent such ranges of characters, which are specified by their upper and lower (inclusive) bounds.

The only **public** member functions relevant to the client interface are the constructors and the comparison operators. The argumentless constructor yields a *CharRange*, which denotes the empty range, i.e. one that is invalid. This constructor should not be used by clients. A second constructor takes a **char** and yields the singleton character range (a character range whose upper and lower bounds are equal) containing the **char**. The third constructor takes a pair of **char**s, the upper and lower bounds of a character range, and yields the *CharRange* denoting the range (since the empty range is not permitted, the upper bound must not be less than the lower bound). Constructing and using a *CharRange* denoting the empty range will eventually be caught by an assertion within the **FIRE engine**.

```
// $Revision: 1.1 $
// $Date: 1994/05/02 15:55:17 $
#ifndef CHARRANGE_H
#define CHARRANGE_H

#include "extras.h"
#include <assert.h>
#include <iostream.h>

// Represent a set of characters (in the alphabet) in a subrange of char.
// Used by most classes as the basis of regular expressions and automata.

class CharRange {
public:
        // Some constructors:
        // The default is the empty range of characters.
        //        and this does not satisfy the class invariant.
        inline CharRange();

        // Copy constructor is standard.
        inline CharRange( const CharRange& r );
        inline CharRange( const char l, const char h );
        inline CharRange( const char a );

        // Default operator=, destructor are okay

        // Some normal member functions:
        // Is char a in the range?
        inline int contains( const char a ) const;

        // Access to the representation:
        inline char low() const;
        inline char high() const;


        // Standard comparison operators.
        inline int operator==( const CharRange& r ) const;
        inline int operator!=( const CharRange& r ) const;

        // Containment operators.
        inline int operator<=( const CharRange& r ) const;
        inline int operator>=( const CharRange& r ) const;

        // Is there something in common between *this and r?
        inline int not_disjoint( const CharRange& r ) const;

        // Do *this and r overlap, or are they adjacent?
        inline int overlap_or_adjacent( const CharRange& r ) const;

        // Some operators on CharRanges.
        // Merge two CharRanges if they're adjacent:
        inline CharRange& merge( const CharRange& r );

        // Make *this the intersection of *this and r.
        inline CharRange& intersection( const CharRange& r );

        // Return the left and right excess of *this with r (respectively):
        inline CharRange left_excess( const CharRange& r ) const;
        inline CharRange right_excess( const CharRange& r ) const;


        // Define an ordering on CharRange's, used mainly in RE::ordering().
        int ordering( const CharRange& r ) const;


        // The class structural invariant.
        inline int class_invariant() const;
```

```
        // Some extra stuff.
        friend ostream& operator<<( ostream& os, const CharRange r );
};

#endif
```

3 Regular expressions

Class *RE* is used to represent regular expressions. This class cannot be used to construct regular expressions; to construct them, see Section 4. Class *RE* supports an assignment operator, comparison operators, and operators providing information on the structure of a regular expression (such as the main, or root operator of the regular expression). Most of the operators are used internally by other classes and functions of the *FIRE engine.* The argumentless constructor yields the regular expression denoting the empty language. Class *RE* also has input and output (*iostream* insertion and extraction) operators.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:21 $
#ifndef RE_H
#define RE_H

#include "charrang.h"
#include "crset.h"
#include "reops.h"
#include <assert.h>
#include <iostream.h>

// Regular expressions.
// This class is not too useful for actually constructing them (for that, see
// the Sigma-algebra class in sigma.h). Class RE functions as the terms of the
// Sigma-term algebra. Some members for manipulating them are provided.

class RE {
public:
        // Some constructors, destructors, and operator=:

        // By default, create the RE denoting the empty language.
        RE();

        RE( const RE& r );

        virtual ~RE();

        const RE& operator=( const RE& r );


        // Some basic access member functions:

        // How many SYMBOL nodes in this regular expression?
        //          Mainly for use in constructing RFA's.
        int num_symbols() const;

        // How many operators in this RE?
        //          Used in ISImpl (the item set stuff).
        int num_operators() const;

        // What is the main operator of this regular expression?
        inline REops root_operator() const;

        // What is the CharRange of this RE, if it is a SYMBOL regular expression.
        inline CharRange symbol() const;
```

```
        // What is the left RE of this RE operator (if it is a higher operator).
        inline const RE& left_subexpr() const;

        // What is the right RE of this RE operator (if it a binary operator).
        inline const RE& right_subexpr() const;


        // Some derivatives (Brzozowski's) related member functions:
        // Does *this accept epsilon?
        //         This is from Definition 3.20 and Property 3.21
        int Null() const;

        // What is the First of *this?
        //         This is from Definition 4.60
        CRSet First() const;

        // Is *this in similarity normal form (SNF)?
        int in_snf() const;

        // Put *this into similarity normal form.
        RE& snf();

        // Reduce (optimize) *this by removing useless information.
        RE& reduce();

        // What is the derivative of *this (w.r.t. r)?
        RE derivative( const CharRange& r ) const;

        // Some ordering members (largely used in similarity and comparisons)
        int ordering( const RE& r ) const;

        // Some comparisons:
        inline int operator==( const RE& r ) const;
        inline int operator!=( const RE& r ) const;
        inline int operator<( const RE& r ) const;
        inline int operator>=( const RE& r ) const;
        inline int operator>( const RE& r ) const;
        inline int operator<=( const RE& r ) const;


        // Some extras:

        friend ostream& operator<<( ostream& os, const RE& r );
        friend istream& operator>>( istream& is, RE& r );

        inline int class_invariant() const;
};

#endif
```

---

4 The Σ-algebra

Template class *Reg* is used to define a client interface for the regular Σ-algebra (which is fully defined in [Wat93a, Section 3]). The template takes a single type parameter T which is used as the carrier set of the Σ-algebra. Class *Reg*<*T*> derives publicly from T, and so an *Reg*<*T*> can be used anywhere that a T can. The following member functions are provided:

- An argumentless constructor, a copy constructor, and an assignment operator (assuming that T has all three).

- A constructor from *RE* (a regular expression). Given that regular expressions are the Σ-term algebra, this constructor constructs the homomorphic image of the regular expression in the Σ-algebra of T.
- *epsilon* makes ***this** accept the empty word only.
- *empty* makes ***this** accept the empty language.
- *symbol* takes a *CharRange* and makes ***this** accept the set of *char*s denoted by the *CharRange*.
- *or* takes another *Reg*<*T*>, r, and makes ***this** accept the language of ***this** union the language of r.
- *concat* takes another *Reg*<*T*>, r, and makes ***this** accept the language of ***this** concatenated (on the right) with the language of r.
- *star* makes ***this** accept the Kleene closure of the language of ***this**.
- *plus* makes ***this** accept the plus closure of the language of ***this**.
- *question* adds the empty word, ϵ, to the language accepted by ***this**.

For any two carrier sets T_1 and T_2, the Σ-algebra operator definitions will have very little in common. For this reason, the operators of template class *Reg* are specially defined defined for each carrier set. Presently, the operators are only defined for three type parameters (carrier sets): *Reg*<*RE*>, *Reg*<*FA*>, and *Reg*<*RFA*>.

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:35 $
#ifndef SIGMA_H
#define SIGMA_H

#include "charrang.h"
#include "reops.h"
#include "re.h"

// The signature of the Sigma-algebra is defined as template Reg. It is only
// an interface template, with most of the member functions being filled in as
// specialized versions. Given class T, the class Reg<T> is a Sigma-algebra with
// T as the carrier set. Reg is the only template required, since the regular
// expression Sigma algebra only has one sort: Reg.
// The Reg of a class T is also publicly derived from T, meaning that once
// a Reg<T> is constructed (perhaps using the Sigma-algebra operators), it can
// be used as a regular T (without slicing).
// A special constructor is provided, which constructs the homomorphic image in
// algebra Reg<T> of some regular expression. (Regular expressions are the Sigma-
// term algebra.)

template<class T>
class Reg : public T {
public:
        // Some constructors. Usually pass control back to the base class.
        Reg() : T() {}

        Reg( const Reg<T>& r ) :
                        // Just pass back to the base class.
                        T( (const T &)r ) {}

        // Construct the homomorphic image of regular expression r, using the Sigma-
        // algebra operators (corresponding to the operators of the regular expression).
        Reg( const RE& r ) {
                assert( r.class_invariant() );
```

```
		homomorphic_image( r );
		assert( T::class_invariant() );
	}

	// Default destructor falls through to the ~T()

	inline const Reg<T>& operator=( const Reg<T>& r ) {
		// Call through to the base-class's operator=
		T::operator=( r );
		return( *this );
	}


	// Now the Sigma-algebra signature:

	// Sigma-algebra basis
	Reg<T>& epsilon();
	Reg<T>& empty();
	Reg<T>& symbol( const CharRange r );

	// Sigma-algebra binary ops
	Reg<T>& or( const Reg<T>& r );
	Reg<T>& concat( const Reg<T>& r );

	// Sigma-algebra unary ops
	Reg<T>& star();
	Reg<T>& plus();
	Reg<T>& question();
};

#endif
```

5 Abstract and concrete finite automata

All finite automata in the **FIRE engine** share the same interface. This is enforced by deriving all concrete types of finite automata from the abstract base class *FAabs*, which defines the client interface. The interface includes member functions to restart the automaton (in its start states), advance the automaton by one **char** in the input string, determine if the automaton is in an accepting state, determine if the automaton is stuck (unable to make further transitions), and to determine if a particular string is accepted by the automaton (starting from the current state).

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:07 $
#ifndef FAABS_H
#define FAABS_H

// Just mention DFA here, since circularity would be a problem if we include dfa.h.
class DFA;

// Abstract finite automaton. All other automaton types are derived from this one.
// Basic FA operations are supported.

class FAabs {
public:
	// Return the number of states (or some reasonably close measure).
	virtual int num_states() const = 0;

	// Reset the current state before beginning to process a string.
	// This is not the default condition for most of the derived classes.
	virtual void restart() = 0;

	// Advance the current state by processing a character.
```

```
        virtual void advance( char a ) = 0;

        // Is the current state an accepting (final) one?
        virtual int in_final() const = 0;

        // Is the automaton stuck?
        virtual int stuck() = 0;

        // Is the string w acceptable?
        virtual int acceptable( const char *w ) {
                for( restart(); !stuck() && *w; advance( *(w++) ) ) {}
                // It's acceptable if *this is final, and the whole of w was consumed.
                return( in_final() && !*w );
        }

        // Return a DFA accepting the same language as *this.
        virtual DFA determinism() const = 0;
};

#endif
```

Five concrete finite automata classes are provided in the **FIRE engine**: *FA*, *RFA*, *LBFA*, *RBFA*, and *DFA* (their interfaces are defined in files `fa.h`, `rfa.h`, `lbfa.h`, `rbfa.h`, and `dfa.h` respectively). The classes (respectively) represent general, reduced, left-biased, right-biased, and deterministic finite automata. They are all concrete because they inherit from abstract class *FAabs*, and do not leave any of the member functions as pure virtual member functions (each of the classes implements the client interface defined in class *FAabs*).

Although some of the classes provide some extra member functions, most of these members are for internal use by the **FIRE engine**. A full explanation of these member functions is given in [Wat94]. The construction of any of the concrete finite automata is easily done with the construction functions described in Section 6.

6 Automata constructions

Three header files contain (**inline**) definitions or prototypes of functions that take a regular expression and construct a finite automaton accepting the language of the regular expression. These functions implement all of the construction algorithms given in [Wat93a]. They differ in the type of finite automaton that is created:

- File `constrs.h` contains those construction functions whose operation is based upon the structure of a regular expression (the Σ-algebra functions). These functions correspond to those given in [Wat93a, Section 4].
- File `dconstrs.h` contains functions implementing Brzozowski's construction (and its variants). These functions all produce a *DFA*, and correspond to those given in [Wat93a, Section 5.3].
- File `iconstrs.h` contains the item set construction (and its variants). These functions all produce a *DFA*, and correspond to those given in [Wat93a, Section 5.5].

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:55:29 $
#ifndef CONSTRS_H
#define CONSTRS_H

#include "re.h"
#include "fa.h"
#include "rfa.h"
```

```
#include "lbfa.h"
#include "rbfa.h"
#include "sigma.h"
#include "dfa.h"
#include <assert.h>

// Thompson's construction:
//         use the functions Thompson or Thompson_sigma, or
//         auto FA Th( E );
//         auto Reg<FA> Th2( E );

// Construction 4.5
inline FA Thompson( const RE& r ) {
        return( FA( r ) );
}

// Construction 4.3
inline FA Thompson_sigma( const RE& r ) {
        return( Reg<FA>( r ) );
}

// RFA constructions:
//         use the functions, or
//         auto RFA R( E );
//         auto Reg<RFA> R2( E );
//         To DFA:
//                 R.determinism();
//                 R2.determinism();
//                 R.determinism2();
//                 R2.determinism2();

// Definition 4.30 (variation)
inline RFA rfa( const RE& r ) {
        return( RFA( r ) );
}

// Definition 4.30
inline RFA rfa_sigma( const RE& r ) {
        return( Reg<RFA>( r ) );
}

// Construction 4.39 (variation of McNaughton-Yamada-Glushkov)
inline DFA MYG_shortcut( const RE& r ) {
        return( RFA( r ).determinism2() );
}

// Construction 4.50 (variation of Aho-Sethi-Ullman)
inline DFA ASU_shortcut( const RE& r ) {
        return( RFA( r ).determinism() );
}


// LBFA constructions:
//         using RFA's.

// Construction 4.32
inline LBFA BerrySethi( const RE& r ) {
        return( RFA( r ) );
}

// Construction 4.38
LBFA BS_variation( const RE& r );

// DFA from LBFA construction:
// Construction 4.39
inline DFA MYG( const RE& r ) {
        return( BerrySethi( r ).determinism() );
```

```
}

// RBFA constructions:
//        using RFAs'.

// Construction 4.45
inline RBFA BerrySethi_dual( const RE& r ) {
        return( RFA( r ) );
}

// Construction 4.48
RBFA BS_variation_dual( const RE& r );

// Construction 4.50
inline DFA ASU( const RE& r ) {
        return( BS_variation_dual( r ).determinism() );
}

#endif
```

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:55:51 $
#ifndef DCONSTRS_H
#define DCONSTRS_H

#include "re.h"
#include "dfa.h"
#include <assert.h>

// The numbers in parentheses refer to the Construction number in the Taxonomy.

// DFA constructions:
//        Brzozowski's constructions:
//        Normal (Construction 5.34)
DFA Brz( const RE& r );

//        With strong similarity (Construction 5.34 + Remark 5.32)
DFA Brz_optimized( const RE& r );

#endif
```

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:41 $
#ifndef ICONSTRS_H
#define ICONSTRS_H

#include "re.h"
#include "dfa.h"
#include <assert.h>

// The numbers in parentheses refer to the Construction number in the Taxonomy.

//        Item set constructions:
//        Iconstr (Construction 5.69)
DFA Iconstr( const RE *r );

//        DeRemer's (Construction 5.75)
DFA DeRemer( const RE *r );

//        Oconstr (Construction 5.82)
DFA Oconstr( const RE *r );
```

#endif

---

7 Minimizing *DFAs*

Five deterministic finite automata minimization algorithms are implemented in the toolkit. All of the algorithms are implemented as member functions of class *DFA*, and are therefore declared in file `dfa.h`. The algorithms are:

- Brzozowski's, given in [Wat93b, Section 2]. It appears as member function *min_Brzozowski*.
- Aho, Sethi, and Ullman's, given in [Wat93b, Algorithm 4.6]. It appears as member function *min_dragon*.
- Hopcroft and Ullman's, given in [Wat93b, Algorithm 4.7]. It appears as member function *min_HopcroftUllman*.
- Hopcroft's, given in [Wat93b, Algorithm 4.8]. It appears as member function *min_Hopcroft*.
- A new algorithm, given in [Wat93b, Algorithm 4.9, 4.10]. It appears as member function *min_Watson*.

References

[Bud91] BUDD, T. A. *An introduction to object-oriented programming,* Addison-Wesley, Reading, MA, 1991.

[Bud94] BUDD, T. A. *Classic data structures in C++,* Addison-Wesley, Reading, MA, 1994.

[CH91] CHAMPARNAUD, J. M. AND G. HANSEL. "AUTOMATE: A computing package for automata and finite semigroups," *Journal of Symbolic Computation* 12, p. 197–220, 1991.

[Cop92] COPLIEN, J. O. *Advanced C++: programming styles and idioms,* Addison-Wesley, Reading, MA, 1992.

[JPTW90] JANSEN, V., A. POTTHOF, W. THOMAS, AND U. WERMUTH. "A short guide to the AMORE system," *Aachener Informatik-Berichte* 90(02), Lehrstuhl für Informatik II, Universität Aachen, January 1990.

[Joh86] JOHNSON, J. H. "INR: A program for computing finite automata," Department of Computer Science, University of Waterloo, Waterloo, Canada, January 1986.

[Lei77] LEISS, E. "REGPACK: An interactive package for regular languages and finite automata," Research report CS-77-32, Department of Computer Science, University of Waterloo, Waterloo, Canada, October 1977.

[Lip91] LIPPMAN, S. B. *C++ primer,* (second edition), Addison-Wesley, Reading, MA, 1991.

[MeS92] MEYERS, S. *Effective C++: 50 specific ways to improve your programs,* Addison-Wesley, Reading, MA, 1992.

[Mur93] MURRAY, R. B. *C++ strategies and tactics,* Addison-Wesley, Reading, MA, 1993.

[RW93] RAYMOND, D. R. AND D. WOOD. "The Grail Papers: Version 2.0," Department of Computer Science, University of Waterloo, Canada, January 1994. Available by ftp from CS-archive.uwaterloo.ca.

[Str91] STROUSTRUP, B. *The C++ programming language,* (second edition), Addison-Wesley, Reading, MA, 1991.

[Wat93a] WATSON, B. W. "A taxonomy of finite automata construction algorithms," Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993. Available by ftp from ftp.win.tue.nl in pub/techreports/pi.

[Wat93b] WATSON, B. W. "A taxonomy of finite automata minimization algorithms," Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993. Available by ftp from ftp.win.tue.nl in pub/techreports/pi.

[Wat94] WATSON, B. W. "The design and implementation of the **FIRE engine**: A C++ toolkit for FInite automata and Regular Expressions," Computing Science Note 94/22, Eindhoven University of Technology, The Netherlands, 1994. Available by ftp from ftp.win.tue.nl in pub/techreports/pi.