The design and implementation of 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 describes the design and implementation 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]. The reader is assumed to be familiar with the two taxonomies and with advanced C++ programming techniques.

The toolkit is implemented largely in an object-oriented style, with finite automata and regular expressions being defined as classes. All of the classes and functions in the toolkit are presented in the same format. For each class (or function) the format includes a short description of its behaviour, details of its implementation, and techniques for improving its performance.

Contents

1 Introduction

The **FIRE engine** is a class library implementing finite automata and regular expression algorithms. The algorithms implemented are all of the algorithms presented in the taxonomies of finite automata algorithms [Wat93a, Wat93b]. This paper outlines the design and implementation of the class library, which consists of 9000 lines of C++ code. A number of other toolkits (mostly serving different purposes) are available; they are detailed in [Wat94], which also contains an introduction to the **FIRE engine**.

In order to understand the implementation of the **FIRE engine**, the reader is assumed to be familiar with the taxonomies [Wat93a, Wat93b], and with C++ terminology and programming techniques. An introduction to the C++ language can be found in [Lip91, Str91] (the language reference appears in [Str91, SE90]), while an introduction to C++ programming techniques can be found in [Bud94]. Advanced C++ programming techniques are presented in [Cop92, MeS92, Mur93]. Object-oriented programming, in general, is discussed in [Boo94, Bud91, MeB88].

The most effective way to understand the implementation of the class library is to read the code and the comments embedded within the code. The explanations given in this paper are intended as an outline, to assist in understanding the code.

This paper is structured as follows:

- Part I provides descriptions of all of the foundations classes — those classes used in the implementations of other classes.
- Part II gives descriptions of all classes related to regular expressions and Σ-algebras.
- Part III gives descriptions of all classes implementing finite automata.
- Part IV presents the functions implementing deterministic finite automata minimization algorithms.

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 Coding conventions and performance issues

The toolkit is coded in a style which makes heavy use of the *assert* function. For all except the simplest classes, a member function called *class_invariant* is defined; this member returns 1 if the object is structurally sound, and 0 otherwise. All member functions which are visible to clients should only be invoked on structurally sound objects and must leave the object in a structurally sound condition. The function call *assert(class_invariant())* is explicitly invoked at the begining and end of all such member functions (except, of course, the function *class_invariant*). The definition of the class invariant is a useful starting point for understanding the structure and implementation of a particular class.

Additional assertions are used in places where the incorrect values of variables (or relations between variables) will cause catastrophic results (i.e. the machine is likely to crash). Assertions are also used in places where they may be of use in understanding the surrounding code. As mentioned below, *assert* can be disabled by compiling with the macro `NDEBUG` defined —meaning that there is no assertion overhead in a production version of the **FIRE engine**.

Almost all of the classes have an *ostream* insertion operator defined (an output operator). For implementation classes the output members are mainly for debugging purposes.

Many classes provide a member function called *reincarnate*. This function clears the object, leaving it in the same condition as it would be in after the argumentless constructor for the object. The only difference is that any memory already allocated by the members of the object is retained for future use.

1.1.1 Performance tuning

There are some general techniques that can be used to improve the performance of the various parts of the **FIRE engine** toolkit. The easiest method of improving the performance is to compile the toolkit with the macro `NDEBUG` defined. This will disable the *assert* function call, and should only be done when the program is functioning correctly and the safety of assertions and class invariants is no longer needed. Before hand-optimizing the code itself, an important first step is to profile the execution of the toolkit. Some preliminary profiling shows that a large amount of time is spent copying and allocating memory. Much of this time can be eliminated by using two techniques:

- Define and use special versions of the **new** and **delete** operators for the classes that make extensive use of memory management.
- "Use-count" all classes which copy large amounts of data in their copy constructors or assignment operators. (See any of [Cop92, MeS92, Mur93] for explanations of "use-counting".)

1.2 Presentation conventions

Each class is presented in a standard format. The following items appear in the description of a class:

1. The name of the class, and whether it is a user level class (a class intended for use by program components internal and external to the **FIRE engine**), or an implementation level class (one used only by other classes within the **FIRE engine**).
2. The **Files** clause lists the files in which the class declaration and definition are stored. The names of the files accociated with a class are usually the class name, followed by .h for interface (or header) files and .cpp for definition files. All file names are short enough for use under MS-DOS.
3. The **Uses** clause lists the classes of which this class is an immediate client or from which this class inherits directly. (Most classes use themselves; this is not mentioned.)
4. The **Description** clause gives a brief description of the purpose of the class.
5. The .h file (if there is one) corresponding to the class is listed.
6. The optional **Implementation** clause outlines the implementation of the class.
7. The optional **Performance** clause gives some suggestions on possible performance improvements to the implementation of the class.
8. The .cpp file (if there is one) corresponding to the class is listed.
9. The description ends with the □ symbol.

Part I
Foundation classes

2 Character ranges

The header file `extras.h` contains inline functions for computing minima and maxima of two numbers. The functions are overloaded to provide versions for types **char** and **int**.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:01 $
#ifndef EXTRAS_H
#define EXTRAS_H

inline int min( const int a, const int b ) {
        return( a < b ? a : b );
}

inline int max( const int a, const int b ) {
        return( a < b ? b : a );
}

#endif
```

User class: *CharRange*

Files: `charrang.h`, `charrang.cpp`

Uses:

Description: Sets of characters (denoted by a *CharRange*) can be used as automata transition labels and as regular expression atoms (unlike most other systems, which use single characters). The sets of characters are limited to a range of characters in the execution character set (no particular character set is assumed, and ASCII or EBCDIC can both be used). The ranges are specified by their upper and lower (inclusive) bounds. The denoted range may not be empty (allowing them to be empty greatly complicates the algorithms). Many member functions are provided, including functions determining if two *CharRanges* have a character in common, if one is entirely contained in another, or if they denote sets that are adjacent to one another (in the execution character set).

```
/* (c) Copyright 1994 by Bruce Watson */
// $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 );

private:
        char lo, hi;
};

// class_invariant() is not satisfied.
inline CharRange::CharRange() :
                lo( 1 ),
                hi( 0 ) {}

inline CharRange::CharRange( const CharRange& r ) :
                lo( r.lo ),
                hi( r.hi ) {
        assert( class_invariant() );
}
```

```
inline CharRange::CharRange( const char l, const char h ) :
                lo( l ),
                hi( h ) {
        assert( class_invariant() );
}

inline CharRange::CharRange( const char a ) :
                lo( a ),
                hi( a ) {
        assert( class_invariant() );
}

inline int CharRange::contains( const char a ) const {
        assert( class_invariant() );
        return( (lo <= a) && (a <= hi) );
}

inline char CharRange::low() const {
        assert( class_invariant() );
        return( lo );
}

inline char CharRange::high() const {
        assert( class_invariant() );
        return( hi );
}

inline int CharRange::operator==( const CharRange& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( (lo == r.lo) && (hi == r.hi) );
}

inline int CharRange::operator!=( const CharRange& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( !operator==( r ) );
}

inline int CharRange::operator<=( const CharRange& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( r.lo <= lo && hi <= r.hi );
}

inline int CharRange::operator>=( const CharRange& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( lo <= r.lo && r.hi <= hi );
}

inline int CharRange::not_disjoint( const CharRange& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( (lo <= r.hi) && (hi >= r.lo) );
}

inline int CharRange::overlap_or_adjacent( const CharRange& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( not_disjoint( r )
                || (r.lo == hi + 1)
                || (r.hi == lo - 1) );
}

inline CharRange& CharRange::merge( const CharRange& r ) {
```

```
        assert( overlap_or_adjacent( r ) );
        lo = (char)min( lo, r.lo );
        hi = (char)max( hi, r.hi );
        return( *this );
}

inline CharRange& CharRange::intersection( const CharRange& r ) {
        lo = (char)max( lo, r.lo );
        hi = (char)min( hi, r.hi );
        return( *this );
}

inline CharRange CharRange::left_excess( const CharRange& r ) const {
        auto CharRange ret( min( lo, r.lo ), max( lo, r.lo ) - 1 );
        return( ret );
}

inline CharRange CharRange::right_excess( const CharRange& r ) const {
        auto CharRange ret( min( hi, r.hi ) + 1, max( hi, r.hi ) );
        return( ret );
}

inline int CharRange::class_invariant() const {
        // This could possibly be wrong if something goes weird with char being
        // signed or unsigned.
        return( lo <= hi );
}

#endif
```

Implementation: *CharRanges* are represented as a pair of characters: the upper and lower (inclusive) bounds. There is a potential for problems caused by the fact that the language standard does not specify whether **char** is **signed** or **unsigned**.

Performance: In most functions that take a *CharRange*, the parameter passing is by value (since a pair of characters fit into the parameter passing registers on most processors). On some machines, performance can be improved by passing a **const** reference.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:55:11 $
#include "charrang.h"

int CharRange::ordering( const CharRange& r ) const {
        if( *this == r ) {
                return( 0 );
        } else if( lo == r.lo ) {
                return( hi - r.hi );
        } else {
                return( lo - r.lo );
        }
}

ostream& operator<<( ostream& os, const CharRange r ) {
        if( r.hi == r.lo ) {
                return( os << '\'' << r.lo << '\'' );
        } else {
                return( os << '[' << '\'' << r.lo << '\''
                        << '\'' << r.hi << '\'' << ']' );
        }
}
```

□

Implementation class: *CRSet*

Files: `crset.h`, `crset.cpp`

Uses: *CharRange*

Description: A *CRSet* is a set of *CharRanges*, with a restriction: all *CharRanges* in a *CRSet* are pairwise disjoint. When a new *CharRange* is added to a *CRSet*, some *CharRanges* may be split to ensure that the disjointness property is preserved. Members are provided to check if two *CRSets* cover the same characters, and to combine two *CRSets*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:55:39 $
#ifndef CRSET_H
#define CRSET_H

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


// A data-structure to hold sets of CharRange's. The CharRange's are kept pair-
// wise disjoint.
// These are generally used in the construction of DFA's, and in DFA minimization.

class CRSet {
public:
        // Some basic constructors.
        // Create the emptyset:
        inline CRSet();

        // The singleton set:
        CRSet( const CharRange a );

        CRSet( const CRSet& r );

        // Also need a destructor and an operator=
        inline ~CRSet();

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


        // Some basic operations.

        // Add a CharRange to this set, splitting it as required.
        // It's not possible to add the empty CharRange to a CRSet!
        inline CRSet& add( CharRange a );

        // Append a CharRange onto the end of this set, knowing that a is disjoint with
        // any others in the set.
        inline CRSet& append( const CharRange a );

        // Combine two CRSet's, splitting CharRange's to maintain their pairwise
        // disjointness.
        CRSet& combine( const CRSet& s );

        // Does *this and r cover the same set of characters?
        int equivalent_cover( const CRSet& r ) const;

        // How many CharRanges are in *this?
        inline int size() const;

        // Some iterators:
        // Fetch the it'th CharRange in *this.
        inline const CharRange& iterator( const int it ) const;
```

```
        // Is there even an it'th CharRange?
        inline int iter_end( const int it ) const;


        // Some special members:
        inline int class_invariant() const;

        friend ostream& operator<<( ostream& os, const CRSet& r );

private:
        // A simple helper.
        void ensure_min_size( int s );

        // Add a CharRange to this set, splitting it as required.
        // Adding begins at i. This is used in CRSet::add().
        void add_at( CharRange a, int i );

        // Is r covered by this set?
        int covered( const CharRange r ) const;

        // A class constant:
        //          must be at least 1.
        enum { expansion_size = 5 };

        // Some implementation details:
        int howmany;
        int in_use;
        CharRange *data;
};

inline CRSet::CRSet() :
                howmany( 0 ),
                in_use( 0 ),
                data( 0 ) {
        assert( class_invariant() );
}

inline CRSet::~CRSet() {
        assert( class_invariant() );
        delete [] data;
}

// Insert a without checking to see if it overlaps with any of them already.
inline CRSet& CRSet::append( const CharRange a ) {
        assert( class_invariant() );
        assert( a.class_invariant() );
        ensure_min_size( in_use + 1 );
        data[in_use++] = a;
        assert( class_invariant() );
        return( *this );
}

// Add a CharRange to this set, splitting it as required.
// It's not possible to add the empty CharRange to a CRSet!
inline CRSet& CRSet::add( CharRange a ) {
        assert( class_invariant() );
        add_at( a, 0 );
        return( *this );
}

inline int CRSet::size() const {
        assert( class_invariant() );
        return( in_use );
}

inline const CharRange& CRSet::iterator( const int it ) const {
        assert( class_invariant() );
```

```
	assert( !iter_end( it ) );
	return( data[it] );
}

inline int CRSet::iter_end( const int it ) const {
	assert( class_invariant() );
	assert( 0 <= it );
	return( it >= in_use );
}

inline int CRSet::class_invariant() const {
	auto int result;
	result =  0 <= howmany
		&& 0 <= in_use
		&& in_use <= howmany
		&& ( howmany == 0 ? data == 0 : data != 0 )
		&& 1 <= expansion_size;

	// Make sure that all of data[] are pairwise disjoint.
	auto int i;
	for( i = 0; (i < in_use) && result; i++ ) {
		auto int j;
		for( j = i + 1; (j < in_use) && result; j++ ) {
			result = !data[i].not_disjoint( data[j] );
		}
	}
	return( result );
}

#endif
```

Implementation: A *CRSet* contains a pointer to a dynamically allocated array of *CharRanges*, an integer representing the size of the array, and an integer representing the number of elements in the array that are in use. The member *add_at* is a good candidate for optimization.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/03 12:18:49 $
#include "crset.h"


CRSet::CRSet( const CharRange a ) :
		howmany( expansion_size ),
		in_use( 1 ),
		data( new CharRange [expansion_size] ) {
	data[0] = a;
	assert( class_invariant() );
}

CRSet::CRSet( const CRSet& r ) :
		howmany( r.in_use + expansion_size ),
		in_use( r.in_use ),
		data( new CharRange [r.in_use + expansion_size] ) {
	assert( r.class_invariant() );
	auto int i;
	for( i = 0; i < in_use; i++ ) {
		data[i] = r.data[i];
	}
	assert( class_invariant() );
}

const CRSet& CRSet::operator=( const CRSet& r ) {
	assert( r.class_invariant() );
	// May need to allocate some more memory.
	if( r.in_use > howmany ) {
```

```
		auto CharRange *d( new CharRange [howmany = r.in_use + expansion_size] );
		delete [] data;
		data = d;
	}
	assert( r.in_use <= howmany );

	auto int i;
	for( i = 0; i < r.in_use; i++ ) {
		data[i] = r.data[i];
	}
	in_use = r.in_use;

	assert( class_invariant() );
	return( *this );
}

void CRSet::add_at( CharRange a, int i ) {
	assert( class_invariant() );
	assert( a.class_invariant() );

	// Go through data[ ] to see if a can be coalesced with data[i].
	for( ; i < in_use && !data[i].not_disjoint( a ); i++ );

	// Perhaps this is a coalescing candidate.
	// There are several possibilities, namely:
	//	a and data[i] are disjoint
	//	a <= data[i] (a is entirely contained in data[i]
	//	data[i] <= a
	//	a and data[i] are not disjoint (but no containment holds).
	//		In this case, we have three pieces: the intersection,
	//		and a slice each from a, data[i].

	if( i < in_use ) {
		assert( data[i].not_disjoint( a ) );
		if( data[i] == a ) {
			// Nothing to do.
		} else {
			assert( data[i].not_disjoint( a ) && (data[i] != a) );

			// a needs some splitting. We know that the excess
			// does not fall into data[0. .i], we begin adding the excess
			// at data[i+1].
			// For efficiency (not implemented here), we could just
			// append any part of data[i] that doesn't fall within
			// a, since data[i] is pairwise disjoint with all other
			// elements in array data[ ].

			auto CharRange old( data[i] );
			data[i].intersection( a );
			//	data[i] now contains old intersection a.

			// Now deal with the left and right excesses of old with a.
			// There are a few possibilities:
			if( a.low() != old.low() ) {
				// Then there is a left excess
				add_at( a.left_excess( old ), i + 1 );
			}
			if( a.high() != old.high() ) {
				// Then there is a right excess
				add_at( a.right_excess( old ), i + 1 );
			}
		}
	} else {
		assert( i >= in_use );

		// There may still be something left to put in data[ ].
		append( a );
```

```
        }

        assert( class_invariant() );
        return;
}

CRSet& CRSet::combine( const CRSet& r ) {
        assert( r.class_invariant() );
        assert( class_invariant() );
        auto int i;
        for( i = 0; i < r.in_use; i++ ) {
                add( r.data[i] );
        }
        assert( class_invariant() );
        return( *this );
}

// Does *this and r cover the same set of letters?
int CRSet::equivalent_cover( const CRSet& r ) const {
        assert( class_invariant() );
        auto int i;
        auto int result( 1 );
        for( i = 0; (i < in_use) && result; i++ ) {
                result = r.covered( data[i] );
        }
        return( result );
}

// Is r covered by this set?
int CRSet::covered( const CharRange r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );

        auto int i;

        for( i = 0; (i < in_use) && !r.not_disjoint( data[i] ); i++ );

        if( i < in_use ) {
                assert( r.not_disjoint( data[i] ) );
                // They have something in common.
                if( r <= data[i] ) {
                        // r is completely covered by data[i]
                        return( 1 );
                } else {
                        // Now, figure out if the excesses are
                        // also covered.
                        // This could be made more efficient!
                        auto int ret1( 1 );
                        auto int ret2( 1 );
                        if( r.low() != data[i].low() ) {
                                // There actually is a left excess:
                                ret1 = covered( r.left_excess( data[i] ) );
                        }
                        if( r.high() != data[i].high() ) {
                                // There actually is a right excess:
                                ret2 = covered( r.right_excess( data[i] ) );
                        }
                        return( ret1 && ret2 );
                }
        } else {
                assert( i >= in_use );
                // r is not covered by *this.
                return( 0 );
        }
}

// Make sure that data is at least s (adjusting howmany, but not in_use).
```

```
void CRSet::ensure_min_size( int s ) {
        assert( class_invariant() );
        // May need to allocate some new memory.
        if( s > howmany ) {
                auto CharRange *d( new CharRange [howmany = s + expansion_size] );
                auto int i;
                for( i = 0; i < in_use; i++ ) {
                        d[i] = data[i];
                }
                delete [] data;
                data = d;
        }

        assert( class_invariant() );
}

ostream& operator<<( ostream& os, const CRSet& r ) {
        os << '{';
        auto int i;
        for( i = 0; i < r.in_use; i++ ) {
                os << ' ' << r.data[i] << ' ';
        }
        return( os << '}' );
}
```

□

3 Bit vectors

Implementation class: *BitVec*

Files: `bitvec.h`, `bitvec.cpp`

Uses: `extras.h`

Description: Bit vectors wider than a single word are used in implementing sets without a compiled-in size bound. In addition to many of the standard bit level operators, iterators are also provided. The vectors have a width that is adjustable through a member function. Binary operators on vectors must have operands of the same width, and such width management is left to the client. When a vector is widened, the newly allocated bits are assumed to be zero.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:55:06 $
#ifndef BITVEC_H
#define BITVEC_H

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

// Implement a bit-vector. This class can be replaced with any standard library class that
// provides the same functionality. Many of the normal bit-wise operators are available on
// BitVec's.
// BitVec's clients are StateSet and the item set's (ISImpl).

class BitVec {
public:
        // Constructors, destructors, operator=:

        // Default operator creates the emptyset, with little effort.
        // This assumes that delete [] 0 is okay (when it is deleted later).
        inline BitVec();

        // Normal copy constructor.
        BitVec( const BitVec& r );

        // Normal non-virtual destructor.
        inline ~BitVec();

        // Needs a special operator=.
        const BitVec& operator=( const BitVec& r );


        // Tests:

        // A special equality operator.
        int operator==( const BitVec& r ) const;

        // Inequality operator.
        inline int operator!=( const BitVec& r ) const;

        // Is there a bit set?
        int something_set() const;

        // Howmany bits are set?
        int bits_set() const;

        // Bit vector operators:

        // set a bit
        inline BitVec& set_bit( const int r );
```

```
        // unset a bit
        inline BitVec& unset_bit( const int r );

        BitVec& bitwise_or( const BitVec& r );

        BitVec& bitwise_and( const BitVec& r );

        BitVec& bitwise_unset( const BitVec& r );

        BitVec& bitwise_complement();

        // Bitwise containment.
        int contains( const BitVec& r ) const;

        // Is a bit set?
        inline int contains( const int r ) const;

        // Does *this have a bit in common with r?
        int something_common( const BitVec& r ) const;

        // Make *this the empty vector.
        void clear();

        // What is the set bit with the smallest index in *this?
        int smallest() const;


        // Some width related members:

        inline int width() const;

        void set_width( const int r );

        // Recycle this BitVec.
        inline void reincarnate();

        // Shift *this left, zero filling.
        BitVec& left_shift( const int r );

        // Append another bit-vector.
        BitVec& append( const BitVec& r );


        // Iterators:

        // Place the index of the first set bit in the iteration in reference r.
        // r == -1 if there is no first one.
        inline int iter_start( int& r ) const;

        // Is r the last set bit in an iteration sequence.
        inline int iter_end( int r ) const;

        // Place the next set bit, after r (in the iteration sequence), in reference r.
        int iter_next( int& r ) const;


        // Other special members:

        friend ostream& operator<<( ostream& os, const BitVec& r );

        // Structural invariant on the BitVec.
        inline int class_invariant() const;

private:
        // Some trivial helpers:

        // Compute the number of words required to hold st bits.
```

```
        inline int words_required( const int st ) const;

        // Compute the word index of a particular bit r.
        inline int word_index( const int r ) const;

        // Compute the bit index (from the LSB) of a bit r.
        inline int bit_index( const int r ) const;


        // Actual representation:
        int bits_in_use;
        int words;
        unsigned int *data;

        // A class constant, used for bit-vector width.
        enum { bits_per_word = (sizeof( unsigned int ) * 8) };
};



// Now for the inline versions of the member functions:
inline BitVec::BitVec() :
                bits_in_use( 0 ),
                words( 0 ),
                data( 0 ) {
        assert( class_invariant() );
}

inline BitVec::~BitVec() {
        assert( class_invariant() );
        delete [] data;
}

inline void BitVec::reincarnate() {
        assert( class_invariant() );
        clear();
        bits_in_use = 0;
        assert( class_invariant() );
}

inline int BitVec::words_required( const int st ) const {
        // This may be a little more complicated than it has to be.
        return( (st / bits_per_word) + ((st % bits_per_word) != 0 ? 1 : 0) );
}

inline int BitVec::word_index( const int r ) const {
        return( r / bits_per_word );
}

inline int BitVec::bit_index( const int r ) const {
        return( r % bits_per_word );
}

inline int BitVec::operator!=( const BitVec& r ) const {
        return( !operator==( r ) );
}

inline int BitVec::width() const {
        return( bits_in_use );
}

inline BitVec& BitVec::set_bit( const int r ) {
        assert( (0 <= r) && (r < width()) );
        assert( class_invariant() );
        data[word_index( r )] |= (1U << bit_index( r ));
        return( *this );
}
```

```
inline BitVec& BitVec::unset_bit( const int r ) {
        // Unset the r'th bit.
        assert( (0 <= r) && (r < width()) );
        assert( class_invariant() );
        data[word_index( r )] &= ~(1U << bit_index( r ));
        return( *this );
}

inline int BitVec::contains( const int r ) const {
        // Check if the r'th bit is set.
        assert( r < width() );
        assert( class_invariant() );
        return( (data[word_index( r )] & (1U << bit_index( r ))) != 0U );
}

inline int BitVec::iter_start( int& r ) const {
        assert( class_invariant() );
        return( r = smallest() );
}

inline int BitVec::iter_end( int r ) const {
        return( r == -1 );
}

inline int BitVec::class_invariant() const {
        return( (width() <= words * bits_per_word)
                && (width() >= 0)
                && (words >= 0)
                && (words == 0 ? data == 0 : data != 0) );
}

#endif
```

Implementation: The implementation is largely replaceable by any standard library providing the same functionality. The width is stored as an integer in the object, along with the number of words allocated. The vector itself is stored as a pointer to an allocated block of **unsigned int**. The width can only grow (except in the case of a *reincarnate* member function call; it will then shrink to zero).

Performance: The low level bit operations could benefit from being written in assembler. The copying of the data area that occurs in the copy constructor and assignment operator can be avoided by use-counting. The left shift operator and the iteration operators could be implemented more efficiently.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:54:59 $
#include "bitvec.h"
#include "extras.h"

BitVec::BitVec( const BitVec& r ) :
                bits_in_use( r.bits_in_use ),
                words( r.words ),
                data( new unsigned int [r.words] ) {
        auto int i;
        for( i = 0; i < words; i++ ) {
                data[i] = r.data[i];
        }
        assert( class_invariant() );
}

const BitVec& BitVec::operator=( const BitVec& r ) {
```

```
	assert( r.class_invariant() );
	// Check against assignment to self.
	if( this != &r ) {
		// The data area may need to be grown.
		if( words < words_required( r.width() ) ) {
			words = words_required( r.width() );
			// Wipe out previously used memory.
			// Assume delete [] 0 is okay.
			delete [] data;
			// Could probably benefit from a specialized operator new.
			data = new unsigned int [words];
		}
		assert( words >= words_required( r.width() ) );

		bits_in_use = r.bits_in_use;
		auto int i;
		for( i = 0; i < words_required( r.width() ); i++ ) {
			data[i] = r.data[i];
		}
		// Make sure that nothing extraneous is in the vector.
		for( ; i < words; i++ ) {
			data[i] = 0U;
		}
	}
	assert( class_invariant() );
	return( *this );
}

int BitVec::operator==( const BitVec& r ) const {
	assert( class_invariant() && r.class_invariant() );
	// Can't be equal if bit counts aren't the same.
	assert( width() == r.width() );
	// Compare all of the bits.
	auto int i;
	// There's no real need to check beyond words_required( bits_in_use ).
	for( i = 0; i < min( words, r.words ); i++ ) {
		if( data[i] != r.data[i] ) {
			// Return at the earliest possible moment.
			return( 0 );
		}
	}
	return( 1 );
}

int BitVec::bits_set() const {
	assert( class_invariant() );
	auto int s( 0 );
	auto int st;
	for( st = 0; st < width(); st++ ) {
		if( contains( st ) ) {
			s++;
		}
	}
	return( s );
}

void BitVec::clear() {
	// Zero-out all of the words.
	auto int i;
	for( i = 0; i < words; i++ ) {
		data[i] = 0U;
	}
}

BitVec& BitVec::bitwise_or( const BitVec& r ) {
	assert( class_invariant() && r.class_invariant() );
	assert( width() == r.width() );
```

```
	// General-purpose counter.
	auto int i;
	for( i = 0; i < min( words, r.words ); i++ ) {
		data[i] |= r.data[i];
	}
	return( *this );
}

BitVec& BitVec::bitwise_and( const BitVec& r ) {
	assert( class_invariant() && r.class_invariant() );
	assert( width() == r.width() );
	auto int i;
	for( i = 0; i < min( words, r.words ); i++ ) {
		data[i] &= r.data[i];
	}
	return( *this );
}

BitVec& BitVec::bitwise_unset( const BitVec& r ) {
	assert( class_invariant() && r.class_invariant() );
	assert( width() == r.width() );
	auto int i;
	for( i = 0; i < min( words, r.words ); i++ ) {
		data[i] &= ~r.data[i];
	}
	return( *this );
}

BitVec& BitVec::bitwise_complement() {
	assert( class_invariant() );

	// Do a bitwise complement of all of data that's in use.
	auto int i;
	auto int w( words_required( bits_in_use ) );
	for( i = 0; i < w; i++ ) {
		data[i] = ~data[i];
	}
	// Now make sure that only bits that the only 1 bits are those that fall
	// within [0,bits_in_use)
	if( bit_index( bits_in_use ) != 0 ) {
		// There are some bit's that are not used in data[w-1].
		// ``and'' the unused ones away.
		data[w - 1] &= ((1U << bit_index( bits_in_use )) - 1);
	}

	assert( class_invariant() );
	return( *this );
}

int BitVec::contains( const BitVec& r ) const {
	assert( class_invariant() && r.class_invariant() );
	assert( width() == r.width() );
	auto int i;
	for( i = 0; i < min( words, r.words ); i++ ) {
		if( r.data[i] != (r.data[i] & data[i]) ) {
			return( 0 );
		}
	}
	return( 1 );
}

int BitVec::something_common( const BitVec& r ) const {
	assert( class_invariant() && r.class_invariant() );
	assert( width() == r.width() );
	// Check if there's a bit in common.
	auto int i;
	auto unsigned int result( 0U );
```

```
	for( i = 0; i < min( words, r.words ); i++ ) {
		result |= (data[i] & r.data[i]);
	}
	return( result != 0U );
}

int BitVec::something_set() const {
	assert( class_invariant() );
	// Check if all bits are unset.
	auto int i;
	auto unsigned int result( 0U );
	for( i = 0; i < words; i++ ) {
		result |= data[i];
	}
	return( result != 0U );
}

BitVec& BitVec::left_shift( const int r ) {
	// Don't do anything needless:
	if( r != 0 ) {
		words = words_required( width() + r );
		auto unsigned int *d( new unsigned int [words]);
		auto int i;
		for( i = 0; i < words; i++ ) {
			d[i] = 0U;
		}
		auto int wi = word_index( r );
		auto int bi = bit_index( r );
		// (wi,bi) index where the first bit of data[ ] must go in d[ ]
		if( bi == 0 ) {
			// Word aligned.
			for( i = 0; i < words_required( width() ); i++ ) {
				d[i + wi] = data[i];
			}
		} else {
			// Deal with the unaligned case.

			// Mask has the (bi) least-significant bits on.
			auto unsigned int rmask = (1U << bi) - 1;
			for( i = 0; i < words_required( width() ); i++ ) {
				// move data[i] into d[i+wi] and d[i+wi+1]

				//		d[i + wi] = (d[i + wi] & rmask) | ((data[i] << bi) & ~rmask);
				// Assume that zero-filling occurs on the LSB.
				d[i + wi] = (d[i + wi] & rmask) | (data[i] << bi);
				d[i + wi + 1] = (data[i] >> (bits_per_word - bi)) & rmask;
			}
		}
		delete [] data;
		data = d;
		bits_in_use += r;
	}
	return( *this );
}

void BitVec::set_width( const int r ) {
	assert( class_invariant() );
	// Make sure that things are not shrinking.
	assert( r >= width() );
	// The data[ ] area may need some growing.
	if( r > words * bits_per_word ) {
		auto int w( words_required( r ) );
		auto unsigned int *d( new unsigned int [w] );
		auto int i;
		// Copy the useful stuff over to the new area.
		for( i = 0; i < words; i++ ) {
			d[i] = data[i];
```

```
                }
                // Make sure that now additional bits are set.
                for( ; i < w; i++ ) {
                        d[i] = 0U;
                }
                delete [] data;
                data = d;
                words = w;
        }
        bits_in_use = r;
        assert( class_invariant() );
}

// The following could be made a lot more efficient by not constructing the local BitVec.
BitVec& BitVec::append( const BitVec& r ) {
        assert( class_invariant() && r.class_invariant() );
        auto BitVec t( r );
        // Make room for *this.

        t.left_shift( width() );
        assert( t.width() == width() + r.width() );
        set_width( width() + r.width() );
        assert( t.width() == width() );

        bitwise_or( t );
        assert( class_invariant() );
        return( *this );
}

int BitVec::iter_next( int& r ) const {
        assert( class_invariant() );
        // Find the next bit that is set.
        for( r++ ; r < width(); r++ ) {
                if( contains( r ) ) {
                        return( r );
                }
        }
        return( r = -1 );
}

int BitVec::smallest() const {
        assert( class_invariant() );
        // Find the first non-empty word.
        auto int r;
        auto int howfar( words_required( width() ) );
        for( r = 0; (r <  howfar) && (data[r] == 0); r++ );

        // Could be that there isn't one:
        if( r == howfar ) {
                return( -1 );
        }

        // Find the first non-zero bit.
        auto int bitnum( r * bits_per_word );
        auto unsigned int d( data[r] );
        // Keep eating bits until the least significant bit is 1.
        //        We're sure that there will be a bit.
        assert( d != 0U );
        while( d % 2 == 0 ) {
                bitnum++;
                d = d >> 1;
        }
        return( bitnum );
}

ostream& operator<<( ostream& os, const BitVec& r ) {
        assert( r.class_invariant() );
```

```
        // Output the BitVec as a space-separated sequence of set bits.
        auto int st;
        os << "{";
        for( r.iter_start( st ); !r.iter_end( st ); r.iter_next( st ) ) {
                os << " " << st << " ";
        }
        return( os << "}" );
}
```

□

4 States and related classes

Implementation class: *State*

Files: `state.h`

Uses:

Description: In the **FIRE engine**, all finite automaton states are encoded as non-negative integers. The special value *Invalid* is used to flag an invalid state.

Implementation: Not making *State* a class has the effect that no extra information can be stored in a state, and a *State* can be silently converted to an **int**.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:44 $
#ifndef STATE_H
#define STATE_H

// Encode automata states as integers.
typedef signed int State;
// Invalid states mean something bad is about to happen.
const State Invalid = -1;

#endif
```

□

Implementation class: *StatePool*

Files: `st-pool.h`

Uses: *State*

Description: A *StatePool* is a dispenser of *States* (beginning with 0). A member function *size()* returns the number of *States* allocated so far. Two *StatePools* can be combined into one, with the result that the *StatePool* being combined into ***this** must be renamed to not clash with the *States* already allocated from ***this**.

Implementation: An integer counter is maintained, indicating the next *State* to be allocated. The class is really too simple to have a class invariant.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:41 $
#ifndef STATEPOOL_H
#define STATEPOOL_H

#include "state.h"
#include <iostream.h>

// All states in an automaton are allocated from a StatePool. StatePool's can be
// merged together to form a larger one. (Care must be taken to rename any relations
// or sets (during merging) that depend on the one StatePool.)

class StatePool {
public:
        // By default, start allocating states at 0:
        inline StatePool();

        // A copy-constructed StatePool starts at the same point.
        inline StatePool( const StatePool& r );
```

```
        // Default destructor and operator= are okay.

        // How many states are already allocated (one more than that last allocated one,
        // since it begins at 0).
        inline int size() const;

        // Incorporate another StatePool by making it disjoint. Anything
        // associated with the other pool must be renamed explicitly.
        inline StatePool& incorporate( const StatePool r );

        // Allocate a new state.
        inline State allocate();

        // Clear all states that were allocated; start again at 0.
        inline StatePool& reincarnate();

        // Does this StatePool contain State r.
        inline int contains( const State r ) const;

        // Output a StatePool.
        friend ostream& operator<<( ostream& os, const StatePool r );

private:
        // The next one to be allocated.
        int next;
};

inline StatePool::StatePool() :
                next( 0 ) {}

inline StatePool::StatePool( const StatePool& r ) :
                next( r.next ) {}

inline int StatePool::size() const {
        return( next );
}

inline StatePool& StatePool::incorporate( const StatePool r ) {
        next += r.size();
        return( *this );
}

inline State StatePool::allocate() {
        return( next++ );
}

inline StatePool& StatePool::reincarnate() {
        next = 0;
        return( *this );
}

inline int StatePool::contains( const State r ) const {
        return( 0 <= r && r < next );
}

inline ostream& operator<<( ostream& os, const StatePool r ) {
        return( os << "[0," << r.next << ')' );
}

#endif
```

□

Implementation class: *StateSet*

Files: `stateset.h`, `stateset.cpp`

Uses: *BitVec*, *State*

Description: A *StateSet* is a set of *States* with a certain capacity. The capacity (called the *domain*) is one more than the greatest *State* that the set can contain. It can easily be enlarged, but it can only be shrunk (to zero) through the *reincarnate* member function. The maintainance of the domain is the responsibility of the client. Normal set operations are available, including iterators. Most member functions taking another *StateSet* expect it to have the same *domain* as ***this** (with catastrophic results if this is not true). The *States* in the set can also be renamed, so that they do not clash with the *States* of a certain *StatePool*. It is possible to take the union of two *StateSets* containing *States* from different *StatePools*. In this case, the *States* in the incoming *StateSet* are renamed during the union; this is done with the *disjointing_union* member function.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:52 $
#ifndef STATESET_H
#define STATESET_H

#include "bitvec.h"
#include "state.h"
#include <iostream.h>
#include <assert.h>

// Implement a set of State's. Normal set operations are available. The StateSet
// is normally associated (implicitly) with a particular StatePool; whenever a StateSet
// is to interact with another (from a different StatePool), its States must be renamed
// (to avoid a name clash). The capacity of a StateSet must be explicitly managed; many
// set operations are not bounds-checked when assert() is turned off.

class StateSet : protected BitVec {
public:
        // Constructors, destructors, operator=:

        // Default constructor creates the emptyset, with little effort.
        // This assumes that delete [] 0 is okay.
        inline StateSet();

        // Normal copy constructor.
        inline StateSet( const StateSet& r );

        // Normal non-virtual destructor, default will call the base.

        // Needs a special operator=.
        inline const StateSet& operator=( const StateSet& r );


        // Tests:

        // A special equality operator.
        inline int operator==( const StateSet& r ) const;

        // Inequality operator.
        inline int operator!=( const StateSet& r ) const;

        // Is this set empty?
        inline int empty() const;

        // What is the size of this set (cardinality)?
        inline int size() const;

        // Set operators (may affect *this):
```

```
// complement *this.
inline StateSet& complement();

// inserts a State.
inline StateSet& add( const State r );

// remove a State from the set.
inline StateSet& remove( const State r );

// Set union.
inline StateSet& set_union( const StateSet& r );

// Set intersection.
inline StateSet& intersection( const StateSet& r );

// Set difference.
inline StateSet& remove( const StateSet& r );

// Set containment.
inline int contains( const StateSet& r ) const;

// Is a State in the set?
inline int contains( const State r ) const;

// Does this set have something in common with r?
inline int not_disjoint( const StateSet& r ) const;

// Make this set the emptyset.
inline void clear();

// What is the smallest element of *this?
inline State smallest() const;


// Some domain related members:

// How many States can this set contain?
//        [0,domain()) can be contained in *this.
inline int domain() const;

inline void set_domain( const int r );

// Recycle this StateSet.
inline void reincarnate();

// Rename the elements of this StateSet so that they don't fall within
// StatePool r.
inline StateSet& st_rename( const int r );

// Include another StateSet into this one, renaming all the States.
inline StateSet& disjointing_union( const StateSet& r );


// Iterators:

// Place the first State in the iteration in reference r.
// r == Invalid if there is no first one.
inline State iter_start( State& r ) const;

// Is r the last State in an iteration sequence.
inline int iter_end( State r ) const;

// Place the next State, after r (in the iteration sequence), in reference r.
inline State iter_next( State& r ) const;


// Other special members:
```

```
	friend ostream& operator<<( ostream& os, const StateSet& r );

	// Structural invariant on the StateSet, just the class invariant of the base.
	inline int class_invariant() const;
};

inline StateSet::StateSet() :
		BitVec() {}

inline StateSet::StateSet( const StateSet& r ) :
		BitVec( r ) {}

inline const StateSet& StateSet::operator=( const StateSet& r ) {
	BitVec::operator=( r );
	return( *this );
}

inline int StateSet::operator==( const StateSet& r ) const {
	return( BitVec::operator==( r ) );
}

inline int StateSet::operator!=( const StateSet& r ) const {
	return( !operator==( r ) );
}

inline int StateSet::empty() const {
	return( !BitVec::something_set() );
}

inline int StateSet::size() const {
	return( BitVec::bits_set() );
}

inline StateSet& StateSet::complement() {
	assert( class_invariant() );
	BitVec::bitwise_complement();
	assert( class_invariant() );
	return( *this );
}

inline StateSet& StateSet::add( const State r ) {
	BitVec::set_bit( r );
	return( *this );
}

inline StateSet& StateSet::remove( const State r ) {
	BitVec::unset_bit( r );
	return( *this );
}

inline StateSet& StateSet::set_union( const StateSet& r ) {
	BitVec::bitwise_or( r );
	return( *this );
}

inline StateSet& StateSet::intersection( const StateSet& r ) {
	BitVec::bitwise_and( r );
	return( *this );
}

inline StateSet& StateSet::remove( const StateSet& r ) {
	BitVec::bitwise_unset( r );
	return( *this );
}

inline int StateSet::contains( const StateSet& r ) const {
```

```
	return( BitVec::contains( r ) );
}

inline int StateSet::contains( const State r ) const {
	return( BitVec::contains( r ) );
}

inline int StateSet::not_disjoint( const StateSet& r ) const {
	return( BitVec::something_common( r ) );
}

inline void StateSet::clear() {
	BitVec::clear();
}

inline State StateSet::smallest() const {
	return( BitVec::smallest() );
}

inline int StateSet::domain() const {
	return( BitVec::width() );
}

inline void StateSet::set_domain( const int r ) {
	assert( r >= BitVec::width() );
	BitVec::set_width( r );
}

inline void StateSet::reincarnate() {
	BitVec::reincarnate();
}

inline StateSet& StateSet::st_rename( const int r ) {
	BitVec::left_shift( r );
	return( *this );
}

inline StateSet& StateSet::disjointing_union( const StateSet& r ) {
	BitVec::append( r );
	return( *this );
}

inline State StateSet::iter_start( State& r ) const {
	return( BitVec::iter_start( r ) );
}

inline int StateSet::iter_end( State r ) const {
	return( BitVec::iter_end( r ) );
}

inline State StateSet::iter_next( State& r ) const {
	return( BitVec::iter_next( r ) );
}

inline int StateSet::class_invariant() const {
	// The Invalid stuff is a required definition.
	return( BitVec::class_invariant() && (Invalid == -1) );
}

#endif
```

Implementation: *StateSet* inherits from *BitVec* for implementation. Most of the member functions are inline dummies, calling the *BitVec* members.

Performance: *StateSet* would benefit from use-counting in *BitVec*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:50 $
#include "stateset.h"

ostream& operator<<( ostream& os, const StateSet& r ) {
        return( os << (BitVec&)r );
}
```

□

Implementation class: *StateTo*

Files: `stateto.h`

Uses: *State*

Description: *StateTo* is a template class implementing mappings from *States* to some class *T* (the template type parameter). It is used extensively in transition relations and binary relations on states. Two methods of mapping a *State* are provided:

- Member function *map* takes a *State* and returns a non-**const** reference to the associated *T*. This is most often used to set up the map, by changing the associated *T*.
- Member function *lookup* takes a *State* and returns a **const** reference to the associated *T*. This is intended for use in clients that only lookup information, without changing the map.

StateTo includes a *reincarnate* member function, which does not reincarnate the *T's*. As with *StateSets*, the domain of the *StateTo* must be explicitly maintained by the client. A special member function *disjointing_union* is available which performs a normal union after the domain of the incoming *StateTo* has been renamed to avoid clashing with the domain of ***this**.

Implementation: Each *StateTo* contains a pointer to a dynamically allocated array of *T*, an integer indicating the size of the array, and an integer indicating how many of the array elements are in use. Whenever the array of *T* is enlarged, a bit extra is allocated to avoid repeated calls to **new**. Class *T* must have a default constructor, an assignment operator and an insertion (output) operator.

Performance: *StateTo* would benefit from use-counting in the class *T*. Use-counting in *StateTo* itself would likely add too much overhead if it were already done in *T*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:54 $
#ifndef STATETO_H
#define STATETO_H

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

// Map State's to class T's. Used to implement transitions (TransRel and DTransRel),
// and State relations (StateRel and StateEqRel).

template<class T>
class StateTo {
// Cannot be protected because the function in dfaseed.h (construct_components)
// needs access.
```

```
public:
        // Constructors, destructors, operator=:

        // Default is to not map anything.
        StateTo();

        // Copying can be costly. Use-counts could make this cheaper.
        StateTo( const StateTo<T>& r );

        // Assume delete [] 0 is okay.
        virtual ~StateTo();

        // Use counts could make this cheaper.
        const StateTo<T>& operator=( const StateTo<T>& r );

        // First, a const lookup operator.
        inline const T& lookup( const State r ) const;

        // Used to associate a State and a T in the mapping. Note: not const.
        inline T& map( const State r );


        // Some domain members:

        // How many States can *this map?
        inline int domain() const;

        // Set a new domain.
        void set_domain( const int r );

        // Note the reincarnate() doesn't reincarnate the T's.
        inline void reincarnate();


        // Allow two mappings to be combined; the domain of *this remains the same, while the
        // domain of r is renamed to not clash with the domain of *this.
        StateTo<T>& disjointing_union( const StateTo<T>& r );


        // Some extras:

        friend ostream& operator<<( ostream& os, const StateTo<T>& r );

        // Assert that everything's okay.
        inline int class_invariant() const;


private:
        // Represent the map as a dynamically allocated array of T's.
        int howmany;
        int in_use;
        T *data;

        // When the array is grown by a certain amount, it also grows by an extra
        // buffer amount for efficiency.
        enum { expansion_extra = 5 };
};

template<class T>
StateTo<T>::StateTo() :
                howmany( 0 ),
                in_use( 0 ),
                data( 0 ) {
        assert( class_invariant() );
}

// Copying can be costly. Use-counts could make this cheaper.
```

```
template<class T>
StateTo<T>::StateTo( const StateTo<T>& r ) :
                howmany( r.in_use + expansion_extra ),
                in_use( r.in_use ),
                data( new T [r.in_use + expansion_extra] ) {
        assert( r.class_invariant() );
        auto int i;
        for( i = 0; i < r.in_use; i++ ) {
                data[i] = r.data[i];
        }
        assert( class_invariant() );
}

// Assume there is nothing wrong with delete [] 0.
template<class T>
StateTo<T>::~StateTo() {
        delete [] data;
}

// Use-counts could make this cheaper.
template<class T>
const StateTo<T>& StateTo<T>::operator=( const StateTo<T>& r ) {
        assert( r.class_invariant() );
        // Prevent assignment to self.
        if( this != &r ) {
                // Is there enough memory to contain the assignment?
                if( howmany < r.in_use ) {
                        delete [] data;
                        data = new T [howmany = r.in_use + expansion_extra];
                }
                in_use = r.in_use;
                assert( in_use <= howmany );
                auto int i;
                for( i = 0; i < in_use; i++ ) {
                        data[i] = r.data[i];
                }
        }
        assert( class_invariant() );
        return( *this );
}

// The actual mapping functions:
template<class T>
inline const T& StateTo<T>::lookup( const State r ) const {
        assert( class_invariant() );
        // First check that it's in bounds.
        assert( 0 <= r && r < domain() );
        return( data[r] );
}

// Used to associate a State and a T in the mapping. Note: not const.
template<class T>
inline T& StateTo<T>::map( const State r ) {
        assert( class_invariant() );
        // First check that it's in bounds.
        assert( 0 <= r && r < domain() );
        return( data[r] );
}

// How many States can *this map?
template<class T>
inline int StateTo<T>::domain() const {
        return( in_use );
}

// Set a new domain.
template<class T>
```

```
void StateTo<T>::set_domain( const int r ) {
	assert( class_invariant() );
	// Make sure that things aren't shrinking.
	assert( r >= in_use );
	if( r > howmany ) {
		auto int i;
		auto T *d( new T [howmany = r + expansion_extra] );
		for( i = 0; i < in_use; i++ ) {
			d[i] = data[i];
		}
		delete [] data;
		data = d;
		// Note: the newly created T's are only default constructed.
	}
	in_use = r;
	assert( class_invariant() );
}

// NB: the reincarnate() doesn't reincarnate the T's.
template<class T>
inline void StateTo<T>::reincarnate() {
	in_use = 0;
}

// Allow two mappings to be combined; the domain of *this remains the same, while the
// domain of r is renamed to avoid this->domain().
template<class T>
StateTo<T>& StateTo<T>::disjointing_union( const StateTo<T>& r ) {
	assert( class_invariant() && r.class_invariant() );
	// Don't do anything that's needless.
	if( r.domain() != 0 ) {
		auto int olddom( domain() );
		set_domain( domain() + r.domain() );
		assert( domain() == olddom + r.domain() );

		// Copy r into the new location.
		auto int i;
		for( i = 0; i < r.domain(); i++ ) {
			data[i + olddom] = r.data[i];
		}
	}
	assert( class_invariant() );
	return( *this );
}

// T is assumed to have an operator<<
template<class T>
inline ostream& operator<<( ostream& os, const StateTo<T>& r ) {
	assert( r.class_invariant() );
	auto State i;
	for( i = 0; i < r.in_use; i++ ) {
		os << i << "->" << r.data[i] << endl;
	}
	return( os );
}

// Assert that everything's okay.
template<class T>
inline int StateTo<T>::class_invariant() const {
	return( 0 <= in_use
		&& in_use <= howmany
		// So 0 <= howmany by transitivity.
		&& (howmany != 0 ? data != 0 : data == 0) );
}


#endif
```

□

5 Transitions and related classes

Implementation class: *TransPair*

Files: `tr-pair.h`

Uses: *CharRange, State*

Description: A *TransPair* is a transition, a structure consisting of a label (a *CharRange*) and a destination *State*. The two are contained in the *TransPair* structure.

Implementation: A **struct** is defined, without member functions.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 16:00:40 $
#ifndef TR_PAIRS_H
#define TR_PAIRS_H
#include "charrang.h"
#include "state.h"

// Implement a single transition, for use in classes TransImpl, Trans (trans.h
// and trans.cpp) and class DTrans (dtrans.h and dtrans.cpp).

struct TransPair {
        CharRange transition_label;
        State transition_destination;
};

#endif
```

□

Implementation class: *TransImpl*

Files: `tr-impl.h`, `tr-impl.cpp`

Uses: *CharRange, CRSet, State, StateSet, TransPair*

Description: A *TransImpl* is a set of *TransPairs*. It is used as a base class for *Trans* and *DTrans*, which are used to implement transition relations. *TransPairs* can be added, but not removed from the set. When a new *TransPair* is added, if its *CharRange* is adjacent to the *CharRange* of a *TransPair* already in the set, and the destination *State* of the two *TransPairs* is the same, then the two *TransPairs* are merged into one.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 16:00:38 $
#ifndef TRANSIMPL_H
#define TRANSIMPL_H

#include "charrang.h"
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "tr-pair.h"
#include <assert.h>
#include <iostreams.h>

// Implement the basics of a char to StateSet map (a set of TransPair's).
// It is used by DTrans and Trans.

class TransImpl {
```

```
protected:
        // Constructors, destructors, operator=:

        // Assume that delete [] 0 is okay.
        inline TransImpl();

        // Copy constructor allocates more memory. Should use (use-counting)
        // for efficiency.
        TransImpl( TransImpl& const r );

        // Destructor is virtual, simple.
        virtual ~TransImpl();

        // operator=() must copy the memory.
        const TransImpl& operator=( const TransImpl& r );


        // Some member functions for making transitions:


        // What are all of the labels on transitions in *this?
        CRSet out_labels() const;

        // What are all transition labels with destination in r?
        CRSet labels_into( const StateSet& r ) const;


        // Some special member functions:

        // Clear out all previous transitions, and zero the domain.
        inline void reincarnate();


        // Add a transition to the set.
        TransImpl& add_transition( const CharRange a, const State q );

        // Allow classes that inherit from TransImpl to have access to the real data:
        inline TransPair& transitions( const int i ) const;

        // Output the transitions.
        friend ostream& operator<<( ostream& os, const TransImpl& r );

        // Maintain the class invariant.
        inline int class_invariant() const;


        // Helpers:
        void ensure_min_size( int w );


        // Implementation details:

        // How many transitions are there.
        int howmany;
        int in_use;

private:
        // A dynamically allocated array of (CharRange,State) pairs (transitions).
        TransPair *data;

        // For efficiency of the expansion helper function.
        enum { expansion_extra = 5 };
};


// Some inline members:
```

```
// Assume that delete [] 0 is okay.
inline TransImpl::TransImpl() :
                howmany( 0 ),
                in_use( 0 ),
                data( 0 ) {
        assert( class_invariant() );
}

inline void TransImpl::reincarnate() {
        assert( class_invariant() );
        in_use = 0;
        assert( class_invariant() );
}

inline TransPair& TransImpl::transitions( const int i ) const {
        assert( class_invariant() );
        // i can be greater than in_use in some weird cases where the
        // transitions are still being set up.
        assert( 0 <= i );
        return( data[i] );
}

// Maintain the class invariant.
inline int TransImpl::class_invariant() const {
        auto int ret( 1 );
        auto int i;
        for( i = 0; (i < in_use) && ret; i++ ) {
                ret = (data[i].transition_destination >= 0);
        }
        ret = ret && (in_use <= howmany);
        return( ret );
}

#endif
```

Implementation: A *TransImpl* contains a pointer to a dynamically allocated array of *TransPair*, an integer indicating the size of the array, and an integer indicating how much of the array is in use.

Performance: Use-counting this class would bring a great benefit to *Trans*, *DTrans*, *TransRel*, and *DTransRel*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 16:00:35 $
#include "tr-impl.h"

// Copy constructor allocates more memory. Should use copy-on-write (use-
// counting) for efficiency.
TransImpl::TransImpl( TransImpl& const r ) :
                howmany( r.in_use + expansion_extra ),
                in_use( r.in_use ),
                data( new TransPair [r.in_use + expansion_extra] ) {
        assert( r.class_invariant() );
        auto int i;
        for( i = 0; i < in_use; i++ ) {
                data[i] = r.data[i];
        }
        assert( class_invariant() );
}

// Destructor is virtual, simple.
virtual TransImpl::~TransImpl() {
        assert( class_invariant() );
        delete [] data;
```

```
}

// Standard dynamic memory copy:
const TransImpl& TransImpl::operator=( const TransImpl& r ) {
	assert( class_invariant() && r.class_invariant() );
	if( this != &r ) {
		// Don't use ensure_min_size() here due to excessive copying.
		if( howmany < r.in_use ) {
			auto TransPair *d( new TransPair [howmany = r.in_use
						+ expansion_extra] );
			delete data;
			data = d;
		}
		assert( howmany >= r.in_use );
		auto int i;
		for( i = 0; i < r.in_use; i++ ) {
			data[i] = r.data[i];
		}
		in_use = r.in_use;
	}
	assert( class_invariant() );
	return( *this );
}

CRSet TransImpl::out_labels() const {
	auto int i;
	auto CRSet a;
	for( i = 0; i < in_use; i++ ) {
		a.add( data[i].transition_label );
	}
	return( a );
}

CRSet TransImpl::labels_into( const StateSet& r ) const {
	assert( class_invariant() );
	assert( r.class_invariant() );
	auto int i;
	auto CRSet a;
	for( i = 0; i < in_use; i++ ) {
		assert( data[i].transition_destination <= r.domain() );
		if( r.contains( data[i].transition_destination ) ) {
			a.add( data[i].transition_label );
		}
	}
	assert( class_invariant() );
	return( a );
}

TransImpl& TransImpl::add_transition( const CharRange a, const State q ) {
	assert( 0 <= q );
	assert( a.class_invariant() );

	// Can the transition be merged with an already existing one.
	auto int i;
	for( i = 0;  i < in_use; i++ ) {
		// They're mergeable if they go to the same place
		// and the labels are adjacent.
		if( (data[i].transition_destination == q)
				&& (data[i].transition_label.overlap_or_adjacent( a )) ) {
			data[i].transition_label.merge( a );
			return( *this );
		}
	}
	// The transition really must be added, since it wasn't merged.

	// Is there a need to grow the data[].
	ensure_min_size( in_use + 1 );
```

```
        assert( in_use < howmany );
        data[in_use].transition_label = a;
        data[in_use++].transition_destination = q;

        return( *this );
}

void TransImpl::ensure_min_size( int w ) {
        assert( class_invariant() );
        assert( 0 <= w );
        if( howmany < w ) {
                auto TransPair *d( new TransPair [howmany = w + expansion_extra] );
                auto int i;
                for( i = 0; i < in_use; i++ ) {
                        d[i] = data[i];
                }
                // Assume that delete [] 0 is okay.
                delete [] data;
                data = d;
        }
        assert( howmany >= w );
        assert( class_invariant() );
}

ostream& operator<<( ostream& os, const TransImpl& r ) {
        assert( r.class_invariant() );
        os << '{';
        auto int i;
        for( i = 0; i < r.in_use; i++ ) {
                os << ' ' << r.data[i].transition_label << "->"
                        << r.data[i].transition_destination << ' ';
        }
        return( os << '}' );
}
```

□

Implementation class: *Trans*

Files: `trans.h`, `trans.cpp`

Uses: *CharRange, CRSet, State, StateSet, TransImpl, TransPair*

Description: A *Trans* is a set of *TransPairs* (transitions) for use in transition relations. Transitions can be added to the set, but not removed. Member functions are provided to compute the destination of the transition set, on a particular character or on a *CharRange* (the destination is returned as a *StateSet*). Since a *StateSet* is returned, *Trans* has a range (which is one more than the maximum *State* that can occur as a transition destination) which is managed by the client and is used to determine the domain of the returned *StateSet*. Another member function computes the set of all transition labels, returning a *CRSet*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 16:00:44 $
#ifndef TRANS_H
#define TRANS_H

#include "charrang.h"
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "tr-pair.h"
#include "tr-impl.h"
#include <assert.h>
```

```
#include <iostreams.h>

// Map char to StateSet, using TransImpl.
// Implement the transitions for one state, or for the inverse of Qmap
// (in RFA's, LBFA's, and RBFA's).

class Trans : protected TransImpl {
public:
        // Constructors, destructors, operator=:

        // Assume that delete [] 0 is okay.
        inline Trans();

        // Copy constructor allocates more memory. Should use (use-counting for
        // efficiency.
        inline Trans( const Trans& r );


        // operator=() must copy the memory.
        inline const Trans& operator=( const Trans& r );


        // Some member functions for making transitions:

        // Map a char to the corresponding StateSet.
        StateSet operator[]( const char a ) const;

        // Map a CharRange to the corresponding StateSet
        //        assuming that the CharRange is entirely contained in the label
        //        of a transition.
        StateSet range_transition( const CharRange a ) const;

        // What are all of the transitions in *this?
        inline CRSet out_labels() const;

        // What are all transition labels into StateSet r?
        inline CRSet labels_into( const StateSet& r ) const;


        // Some special member functions:

        // Clear out all prev. transitions, and zero the domain.
        inline void reincarnate();

        // The range of States that can be transitioned to.
        inline int range() const;

        // Change the range of States that can be transitioned to.
        //        This is used in determining the domain() of the StateSet's (which
        //        are destinations of transitions).
        inline void set_range( const int r );


        // Add one more transition to the set.
        inline Trans& add_transition( const CharRange a, const State q );

        // Do normal set union.
        Trans& set_union( const Trans& r );

        // Incorporate another Trans, while renaming all of the states.
        Trans& disjointing_union( const Trans& r );

        // Rename all of the States (that are destinations of some transition)
        // such that none of them fall in the range [0,r).
        Trans& st_rename( const int r );

        // Output the transitions.
```

```
        friend ostream& operator<<( ostream& os, const Trans& r );

        // Maintain the class invariant.
        inline int class_invariant() const;


private:
        // Implementation details:
        // The domain() of the associated StatePool.
        int destination_range;
};


// Some inline members:
inline Trans::Trans() :
                TransImpl(),
                destination_range( 0 ) {
        assert( class_invariant() );
}

inline Trans::Trans( const Trans& r ) :
                TransImpl( (TransImpl&)r ),
                destination_range( r.destination_range ) {
        assert( class_invariant() );
}

inline const Trans& Trans::operator=( const Trans& r ) {
        assert( class_invariant() );
        assert( r.class_invariant() );

        TransImpl::operator=( (TransImpl&)r );
        destination_range = r.destination_range;
        assert( class_invariant() );
        return( *this );
}

inline CRSet Trans::out_labels() const {
        assert( class_invariant() );
        return( TransImpl::out_labels() );
}

inline CRSet Trans::labels_into( const StateSet& r ) const {
        assert( class_invariant() );
        return( TransImpl::labels_into( r ) );
}

inline void Trans::reincarnate() {
        assert( class_invariant() );
        TransImpl::reincarnate();
        assert( class_invariant() );
}

inline int Trans::range() const {
        assert( class_invariant() );
        return( destination_range );
}

inline void Trans::set_range( const int r ) {
        assert( class_invariant() );
        destination_range = r;
        assert( class_invariant() );
}

inline Trans& Trans::add_transition( const CharRange a, const State q ) {
        assert( class_invariant() );
        TransImpl::add_transition( a, q );
        assert( class_invariant() );
```

```
	return( *this );
}

// Maintain the class invariant.
inline int Trans::class_invariant() const {
	auto int ret( 1 );
	auto int i;
	for( i = 0; (i < in_use) && ret; i++ ) {
		ret = (0 <= transitions( i ).transition_destination)
			&& (transitions( i ).transition_destination < destination_range);
	}
	return( ret );
}

#endif
```

Implementation: *Trans* inherits from *TransImpl* for implementation. It also maintains the range, as an integer. Most member functions are simply calls to the corresponding *TransImpl* member.

Performance: See the base class *TransImpl*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 16:00:42 $
#include "trans.h"


StateSet Trans::operator[]( const char a ) const {
	assert( class_invariant() );
	auto int i;
	auto StateSet result;
	result.set_domain( range() );
	for( i = 0; i < in_use; i++ ) {
		if( transitions( i ).transition_label.contains( a ) ) {
			result.add( transitions( i ).transition_destination );
		}
	}
	return( result );
}

StateSet Trans::range_transition( const CharRange a ) const {
	assert( class_invariant() );
	auto int i;
	auto StateSet result;
	result.set_domain( range() );
	for( i = 0; i < in_use; i++ ) {
		if( a <= transitions( i ).transition_label ) {
			result.add( transitions( i ).transition_destination );
		}
	}
	return( result );
}

// The following member does not change the range.
Trans& Trans::set_union( const Trans& r ) {
	assert( class_invariant() && r.class_invariant() );
	assert( range() == r.range() );
	if( r.in_use != 0 ) {
		// May need to expand here:
		ensure_min_size( in_use + r.in_use );

		assert( howmany >= in_use + r.in_use );
		auto int i;
		for( i = 0; i < r.in_use; i++ ) {
```

```
                transitions(i + in_use) = r.transitions( i );
            }
            in_use += r.in_use;
        }
        assert( class_invariant() );
        return( *this );
}

Trans& Trans::disjointing_union( const Trans& r ) {
        assert( class_invariant() );
        // Don't do any useless work.
        if( r.in_use != 0 ) {
                ensure_min_size( in_use + r.in_use );

                assert( howmany >= in_use + r.in_use );
                auto int i;
                for( i = 0; i < r.in_use; i++ ) {
                        transitions( i + in_use ).transition_label =
                                r.transitions( i ).transition_label;
                        transitions( i + in_use ).transition_destination =
                                r.transitions( i ).transition_destination + destination_range;
                }
                in_use += r.in_use;
                destination_range += r.destination_range;
        }
        assert( class_invariant() );
        return( *this );
}

// Rename all states in this Trans, so that they don't clash with
// those in the range [0,r).
Trans& Trans::st_rename( const int r ) {
        assert( class_invariant() );
        // Don't do anything useless:
        if( r != 0 ) {
                auto int i;
                for( i = 0; i < in_use; i++ ) {
                        transitions( i ).transition_destination += r;
                }
                destination_range += r;
        }
        assert( class_invariant() );
        return( *this );
}

ostream& operator<<( ostream& os, const Trans& r ) {
        assert( r.class_invariant() );
        return( os << (TransImpl&)r );
}
```

□

Implementation class: *DTrans*

Files: `dtrans.h`, `dtrans.cpp`

Uses: *CharRange, CRSet, State, StateSet, TransImpl, TransPair*

Description: A *DTrans* is a set of *TransPairs*, with one restriction: the transition labels (*CharRanges*) are pairwise disjoint. (They are pairwise disjoint, since *DTrans* is used to implement deterministic transition functions.) The client must ensure that only transitions with disjoint labels are added. Member functions are provided to compute the destination of a transition, on a particular character or *CharRange*, returning a *State* (which is *Invalid* if there is no applicable transition). A *State* is returned (instead of a *StateSet*), since *DTrans* implement deterministic transition sets. A member function computes the set of all transition labels, returning a *CRSet*. Another member takes a *StateSet* and returns the set of all transition labels on transitions with the destination in the *StateSet*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:55 $
#ifndef DTRANS_H
#define DTRANS_H

#include "charrang.h"
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "tr-pair.h"
#include "tr-impl.h"
#include <assert.h>
#include <iostreams.h>

// Implement a DFA's transition function for one State.
// Map a char to a unique next State, using TransImpl

class DTrans : protected TransImpl {
public:
	// Constructors, destructors, operator=:

	// By default, don't introduce any transitions.
	inline DTrans();

	// Copy constructor does a dynamic memory copy.
	inline DTrans( const DTrans& r );

	inline const DTrans& operator=( const DTrans& r );


	// Normal member functions:

	// Map a char to the unique next state.
	State operator[]( const char a ) const;

	// Map a CharRange to the corresponding State
	//	assuming that the CharRange is entirely contained in the label
	//	of a transition.
	State range_transition( const CharRange a ) const;

	// What are the labels of transitions out of *this.
	inline CRSet out_labels() const;

	// What are all transition labels into StateSet r?
	inline CRSet labels_into( const StateSet& r ) const;

	// Is there a valid out-transition on a?
	int valid_out_transition( const CharRange a ) const;

	// What is the range (States) of this map?
	//	return a StateSet with domain dom.
	StateSet range( int dom ) const;


	// Special member functions:

	// Recycle this entire structure.
	inline void reincarnate();

	// Create a new out-transition.
	inline DTrans& add_transition( const CharRange a, const State q );

	friend ostream& operator<<( ostream& os, const DTrans& r );
```

```
	inline int class_invariant() const;
};


// Inlines (mostly calling the base class):
inline DTrans::DTrans() :
		TransImpl() {
	assert( class_invariant() );
}

inline DTrans::DTrans( const DTrans& r ) :
		TransImpl( (TransImpl&)r ) {
	assert( class_invariant() );
}

inline const DTrans& DTrans::operator=( const DTrans& r ) {
	assert( class_invariant() );
	assert( r.class_invariant() );
	TransImpl::operator=( (TransImpl&)r );
	assert( class_invariant() );
	return( *this );
}

inline CRSet DTrans::out_labels() const {
	assert( class_invariant() );
	return( TransImpl::out_labels() );
}

inline CRSet DTrans::labels_into( const StateSet& r ) const {
	assert( class_invariant() );
	return( TransImpl::labels_into( r ) );
}

inline void DTrans::reincarnate() {
	assert( class_invariant() );
	TransImpl::reincarnate();
}

inline DTrans& DTrans::add_transition( const CharRange a, const State q ) {
	assert( class_invariant() );
	TransImpl::add_transition( a, q );
	return( *this );
}

inline int DTrans::class_invariant() const {
	// Should also check that all transitions are on different symbols.
	return( TransImpl::class_invariant() );
}

#endif
```

Implementation: *DTrans* inherits from *TransImpl* for implementation. Most member functions are simply calls to the corresponding *TransImpl* member.

Performance: See the base class *TransImpl*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:53 $
#include "dtrans.h"


// Map the char to the next State.
State DTrans::operator[]( const char a ) const {
	assert( class_invariant() );
```

```
        // Take first valid transition.
        auto int i;
        for( i = 0; i < in_use; i++ ) {
                if( transitions( i ).transition_label.contains( a ) ) {
                        return( transitions( i ).transition_destination );
                }
        }
        assert( class_invariant() );
        return( Invalid );
}

State DTrans::range_transition( const CharRange a ) const {
        assert( class_invariant() );
        // Take first valid transition.
        auto int i;
        for( i = 0; i < in_use; i++ ) {
                if( a <= transitions( i ).transition_label ) {
                        return( transitions( i ).transition_destination );
                }
        }
        assert( class_invariant() );
        return( Invalid );
}

int DTrans::valid_out_transition( const CharRange a ) const {
        assert( class_invariant() );
        auto int i;
        for( i = 0; i < in_use; i++ ) {
                if( a <= transitions( i ).transition_label ) {
                        return( 1 );
                }
        }
        assert( class_invariant() );
        return( 0 );
}

StateSet DTrans::range( int dom ) const {
        assert( class_invariant() );
        auto StateSet result;
        result.set_domain( dom );

        auto int i;
        for( i = 0; i < in_use; i++ ) {
                result.add( transitions( i ).transition_destination );
        }
        assert( class_invariant() );
        return( result );
}

ostream& operator<<( ostream& os, const DTrans& r ) {
        assert( r.class_invariant() );
        return( os << (TransImpl&)r );
}
```

□

6 Relations

Implementation class: *StateRel*

Files: `staterel.h`, `staterel.cpp`

Uses: *State, StateSet, StateTo*

Description: A *StateRel* is a binary relation on *States*. The class is used in the implementation of the ϵ-transition relation for class *FA* [Wat93a, Definition 2.1] and the *follow* relation [Wat93a, Definition 4.24] in classes *RFA, LBFA* and *RBFA*. Many relation operators are available, including taking the image of a *State* or a *StateSet*, and the star-closure of a *StateSet*. In order to easily construct *StateRels*, a member function *union_cross* is provided, which performs the following operation (where l, r are either *States* or *StateSets*):

$$*\textbf{this} := *\textbf{this} \cup (l \times r)$$

Standard union is available, as is *disjointing_union* which operates analogously to the member function in *StateSet* and *StateTo* (incoming *States* are renamed to avoid a collision). As with *StateSet*, the domain must be explicitly maintained by the client. Those member functions which take a *StateSet* or *StateRel* as parameter expect the parameter to have the same domain as ***this**.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:47 $
#ifndef STATEREL_H
#define STATEREL_H

#include "state.h"
#include "stateset.h"
#include "stateto.h"
#include <assert.h>
#include <iostream.h>

// Implement binary relations on States. This is most often used for epsilon transitions
// and follow relations.

class StateRel : protected StateTo<StateSet> {
public:
        // Constructors, destructors, operator=:

        // Default argument-less constructor calls to the base class.
        inline StateRel();

        // Copy constructor does too.
        inline StateRel( const StateRel& r );

        // Default destructor is okay

        inline const StateRel& operator=( const StateRel& r );


        // Some relational operators:

        // Compute the image of r under *this.
        StateSet image( const StateSet& r ) const;

        // Or, compute the image of a single State.
        inline const StateSet& image( const State r ) const;


        // Compute the reflexive and transitive closure of r under *this.
        StateSet closure( const StateSet& r ) const;
```

```
        // Some functions updating *this:

        // Member functions union_cross(A,B) makes *this the union (relation-
        // wise) of *this with A times B (Cartesian cross product).
        inline StateRel& union_cross( State p, State q );
        inline StateRel& union_cross( State st, const StateSet& S );
        StateRel& union_cross( const StateSet& S, State st );
        StateRel& union_cross( const StateSet& A, const StateSet& B );

        // Remove a pair of States from the relation.
        inline StateRel& remove_pair( const State p, const State q );
        StateRel& remove_pairs( const StateSet& P, const StateSet& Q );

        // Clear out this relation, without changing the domain.
        void clear();

        // Perform normal union of two relations.
        StateRel& set_union( const StateRel& r );


        // Some domain members:

        // What is the domain of this relation.
        inline int domain() const;

        // Change the domain of this relation.
        void set_domain( const int r );

        // Recycle this entire relation.
        void reincarnate();


        // Union relation r into *this, while adjusting r.
        StateRel& disjointing_union( const StateRel& r );


        // Some special members:

        friend ostream& operator<<( ostream& os, const StateRel& r );

        int class_invariant() const;
};

inline StateRel::StateRel() :
                StateTo<StateSet>() {
        assert( class_invariant() );
}

inline StateRel::StateRel( const StateRel& r ) :
                StateTo<StateSet>( (StateTo<StateSet> &)r ) {
        assert( class_invariant() );
}

inline const StateRel& StateRel::operator=( const StateRel& r ) {
        assert( r.class_invariant() );
        // call back to the base class.
        StateTo<StateSet>::operator=( (const StateTo<StateSet>&) r );
        assert( class_invariant() );
        return( *this );
}

inline const StateSet& StateRel::image( const State r ) const {
        assert( class_invariant() );
        assert( 0 <= r && r < domain() );
        // call back to the base class.
        return( lookup( r ) );
```

```
}

inline int StateRel::domain() const {
        return( StateTo<StateSet>::domain() );
}

inline StateRel& StateRel::union_cross( State p, State q ) {
        assert( class_invariant() );
        assert( (0 <= p) && (p < domain()) );
        assert( (0 <= q) && (q < domain()) );
        map( p ).add( q );
        assert( class_invariant() );
        return( *this );
}

// Map a st also to StateSet S.
inline StateRel& StateRel::union_cross( State st, const StateSet& S ) {
        assert( class_invariant() );
        assert( S.class_invariant() );
        assert( (0 <= st) && (st < domain()) && (S.domain() == domain()) );
        map( st ).set_union( S );
        assert( class_invariant() );
        return( *this );
}

inline StateRel& StateRel::remove_pair( const State p, const State q ) {
        assert( class_invariant() );
        assert( (0 <= p) && (p < domain()) );
        assert( (0 <= q) && (q < domain()) );
        map( p ).remove( q );
        map( q ).remove( p );
        assert( class_invariant() );
        return( *this );
}

inline int StateRel::class_invariant() const {
        // First, we must satisfy the base class_invariant().
        auto int ret( StateTo<StateSet>::class_invariant() );
        auto int i;
        for( i = 0; (i < domain()) && ret; i++ ) {
                ret = ret && lookup( i ).class_invariant()
                        && (lookup( i ).domain() == domain());
        }
        return( ret );
}

#endif
```

Implementation: *StateRel* inherits for implementation from *StateTo<StateSet>*. Many member functions are simply calls to the corresponding member of *StateTo*. Member functions such as *reincarnate* first reincarnates all of the *StateSets*, followed by the *StateTo*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:46 $
#include "staterel.h"

StateSet StateRel::image( const StateSet& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        assert( r.domain() == domain() );
        // result is, by default, the emptyset.
        auto StateSet result;
        result.set_domain( domain() );
        auto State st;
```

```
	for( r.iter_start( st ); !r.iter_end( st ); r.iter_next( st ) ) {
		result.set_union( lookup( st ) );
	}
	return( result );
}

StateSet StateRel::closure( const StateSet& r ) const {
	assert( class_invariant() );
	assert( r.class_invariant() );
	assert( r.domain() == domain() );

	// Use an iterative method to compute the Kleene closure.
	auto StateSet result( r );

	auto StateSet intermediate( image( r ) );
	assert( intermediate.domain() == result.domain() );
	while( !result.contains( intermediate ) ) {
		result.set_union( intermediate );
		intermediate = image( result );
	}
	return( result );
}

// Map all members of S to st as well.
StateRel& StateRel::union_cross( const StateSet& S, State st ) {
	assert( class_invariant() );
	assert( S.class_invariant() );
	assert( (S.domain() == domain()) && (0 <= st) && (st < domain()) );
	auto State i;
	for( S.iter_start( i ); !S.iter_end( i ); S.iter_next( i ) ) {
		map( i ).add( st );
	}
	assert( class_invariant() );
	return( *this );
}

// This could probably have made use of union_cross(State,StateSet&).
StateRel& StateRel::union_cross( const StateSet& A, const StateSet& B ) {
	assert( class_invariant() );
	assert( A.class_invariant() && B.class_invariant() );
	assert( (A.domain() == domain()) && (B.domain() == domain()) );
	auto State i;
	for( A.iter_start( i ); !A.iter_end( i ); A.iter_next( i ) ) {
		map( i ).set_union( B );
	}
	assert( class_invariant() );
	return( *this );
}

StateRel& StateRel::remove_pairs( const StateSet& P, const StateSet& Q ) {
	assert( class_invariant() );
	assert( P.domain() == domain() );
	assert( Q.domain() == domain() );

	auto State i;
	for( P.iter_start( i ); !P.iter_end( i ); P.iter_next( i ) ) {
		map( i ).remove( Q );
	}
	assert( class_invariant() );
	return( *this );
}

StateRel& StateRel::set_union( const StateRel& r ) {
	assert( class_invariant() && r.class_invariant() );
	assert( domain() == r.domain() );
	auto State i;
	for( i = 0; i < r.domain(); i++ ) {
```

```
                map( i ).set_union( r.lookup( i ) );
                // Could also have been done with
                //          union_cross( i, r.lookup( i ) );
        }
        assert( class_invariant() );
        return( *this );
}

void StateRel::set_domain( const int r ) {
        assert( class_invariant() );
        assert( r >= domain() );
        StateTo<StateSet>::set_domain( r );
        auto int i;
        for( i = 0; i < domain(); i++ ) {
                map( i ).set_domain( r );
        }
        assert( class_invariant() );
}

void StateRel::clear() {
        assert( class_invariant() );
        auto int i;
        for( i = 0; i < domain(); i++ ) {
                map( i ).clear();
        }
        assert( class_invariant() );
}

void StateRel::reincarnate() {
        assert( class_invariant() );
        auto int i;
        for( i = 0; i < domain(); i++ ) {
                map( i ).reincarnate();
        }
        StateTo<StateSet>::reincarnate();
        assert( class_invariant() );
}

StateRel& StateRel::disjointing_union( const StateRel& r ) {
        assert( class_invariant() && r.class_invariant() );
        auto int olddom( domain() );
        StateTo<StateSet>::disjointing_union( r );
        auto int i;
        // domain() is the new one, olddom is the old one:
        assert( domain() == olddom + r.domain() );

        // Just adjust the domains of those StateSet's that were already in *this.
        for( i = 0; i < olddom; i++ ) {
                map( i ).set_domain( domain() );
        }
        // Got to rename the incoming ones.
        for( ; i < domain(); i++ ) {
                map( i ).st_rename( olddom );
        }
        assert( class_invariant() );
        return( *this );
}

ostream& operator<<( ostream& os, const StateRel& r ) {
        assert( r.class_invariant() );
        return( os << (const StateTo<StateSet>&)r );
}
```

□

Implementation class: *SymRel*

Files: `symrel.h`, `symrel.cpp`

Uses: *State, StateSet, StateTo*

Description: A *SymRel* is a binary symmetrical relation on *States*. They are used mainly in *DFA* minimization algorithms. Member functions are provided to add and remove pairs of *States* from the relation, as well as to test for membership in the relation.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 16:00:01 $
#ifndef SYMREL_H
#define SYMREL_H

#include "state.h"
#include "stateset.h"
#include "staterel.h"
#include <assert.h>
#include <iostream.h>

// Implement symmetrical binary relations on States. Class SymRel inherits from
// StateRel for implementation.

class SymRel : protected StateRel {
public:
	// Constructors etc.
	inline SymRel();

	inline SymRel( const SymRel& r );

	inline const SymRel& operator=( const SymRel& r );

	// Some relation operators.

	// Make *this into the identity relation.
	SymRel& identity();

	// What is the image of a State under *this
	inline const StateSet& image( const State p ) const;

	// Add a pair of States.
	inline SymRel& add_pair( const State p, const State q );
	inline SymRel& add_pairs( const StateSet& P, const StateSet& Q );

	// Remove a pair of States.
	inline SymRel& remove_pair( const State p, const State q );
	inline SymRel& remove_pairs( const StateSet& P, const StateSet& Q );

	// Are a pair of States present?
	inline int contains_pair( const State p, const State q ) const;

	// Complement *this.
	SymRel& complement();


	// Some domain related members.
	inline int domain() const;

	inline void set_domain( const int r );

	// Special members.
	friend ostream& operator<<( ostream& os, const SymRel& r );

	inline int class_invariant() const;
};
```

```
inline SymRel::SymRel() :
                StateRel() {
        assert( class_invariant() );
}

inline SymRel::SymRel( const SymRel& r ) :
                StateRel( (const StateRel &)r ) {
        assert( class_invariant() );
}

inline const SymRel& SymRel::operator=( const SymRel& r ) {
        assert( r.class_invariant() );
        StateRel::operator=( (const StateRel &)r );
        assert( class_invariant() );
        return( *this );
}

inline const StateSet& SymRel::image( const State p ) const {
        assert( class_invariant() );
        return( StateRel::image( p ) );
}

inline SymRel& SymRel::add_pair( const State p, const State q ) {
        assert( class_invariant() );
        StateRel::union_cross( p, q );
        StateRel::union_cross( q, p );
        assert( class_invariant() );
        return( *this );
}

inline SymRel& SymRel::add_pairs( const StateSet& P, const StateSet& Q ) {
        assert( class_invariant() );
        StateRel::union_cross( P, Q );
        StateRel::union_cross( Q, P );
        assert( class_invariant() );
        return( *this );
}

inline SymRel& SymRel::remove_pair( const State p, const State q ) {
        assert( class_invariant() );
        StateRel::remove_pair( p, q );
        StateRel::remove_pair( q, p );
        assert( class_invariant() );
        return( *this );
}

inline SymRel& SymRel::remove_pairs( const StateSet& P, const StateSet& Q ) {
        assert( class_invariant() );
        StateRel::remove_pairs( P, Q );
        StateRel::remove_pairs( Q, P );
        assert( class_invariant() );
        return( *this );
}

inline int SymRel::contains_pair( const State p, const State q ) const {
        assert( class_invariant() );
        return( StateRel::image( p ).contains( q ) );
}

inline int SymRel::domain() const {
        assert( class_invariant() );
        return( StateRel::domain() );
}

inline void SymRel::set_domain( const int r ) {
        StateRel::set_domain( r );
        assert( class_invariant() );
```

```
}

inline int SymRel::class_invariant() const {
        // This should also assert symmetry of the relation.
        return( StateRel::class_invariant() );
}

#endif
```

Implementation: *SymRel* inherits for implementation from *StateRel*. All of the member functions simply call through to *StateRel*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:58 $
#include "symrel.h"

SymRel& SymRel::identity() {
        assert( class_invariant() );
        // Iterate over the States, setting them to the identity.
        auto State p;
        for( p = 0; p < domain(); p++ ) {
                StateTo<StateSet>::map( p ).clear();
                StateTo<StateSet>::map( p ).add( p );
        }
        assert( class_invariant() );
        return( *this );
}

SymRel& SymRel::complement() {
        assert( class_invariant() );
        // Go through all of the StateSets and complement them.
        auto State p;
        for( p = 0; p < domain(); p++ ) {
                StateTo<StateSet>::map( p ).complement();
        }
        assert( class_invariant() );
        return( *this );
}

ostream& operator<<( ostream& os, const SymRel& r ) {
        assert( r.class_invariant() );
        return( os << (const StateRel &)r );
}
```

□

Implementation class: *StateEqRel*

Files: `st-eqrel.h`, `st-eqrel.cpp`

Uses: *State*, *StateSet*, *StateTo*

Description: A *StateEqRel* is a binary equivalence relation on *States*. They are used mainly in *DFA* minimization algorithms. The argumentless constructor leaves the *StateEqRel* as the total relation. The *identity* member function makes the *StateEqRel* the identity relation. As with many other classes related to sets of *States*, the domain of the *StateEqRel* must be managed by the client. Member functions are available for determining if two *States* are equivalent, for splitting an equivalence class, and for determining the equivalence class (as a *StateSet*) of a *State*. Each equivalence class has a unique representative; if an equivalence class is split, one of the (at most two) resulting classes is guaranteed to have, as its unique representative, the representative of the original class. Iterator member functions can be used to iterate over the set of representatives, thereby iterating over the set of equivalence classes.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:39 $
#ifndef STATEEQREL_H
#define STATEEQREL_H

#include "stateset.h"
#include "stateto.h"
#include <assert.h>
#include <iostream.h>


// Implement an equivalence relation on State's. These are used mainly in the DFA
// minimization algorithms. As with other State relations, the domain must be
// maintained explicitly.

class StateEqRel : protected StateTo<StateSet *> {
public:
	// Some constructors etc:
	//	Construct the total eq. relation of domain r.
	StateEqRel( const int r );

	inline StateEqRel( const StateEqRel& r );

	inline const StateEqRel& operator=( const StateEqRel& r );


	// Some members for changing the relation:

	// Make two States equivalent:
	StateEqRel& equivalize( const State p, const State q );

	// Split an equivalence class into two (assuming that r is entirely contained
	// in a class):
	StateEqRel& split( const StateSet& r );

	// Make *this the identity relation:
	StateEqRel& identity();


	// Basic access members:

	// Are States p and q equivalent?
	inline int equivalent( const State p, const State q ) const;

	// What is the equivalence class [p]?
	inline const StateSet& equiv_class( const State p ) const;

	// What is the unique representative of eq. class [p]?
	inline State eq_class_representative( const State p ) const;

	// What is the set of representatives of equivalence classes of *this?
	StateSet representatives() const;


	// Special members:

	// Domain setting stuff:
	void set_domain( int r );
	inline int domain() const;


	friend ostream& operator<<( ostream& os, const StateEqRel& r );
```

```
        inline int class_invariant() const;
};

inline StateEqRel::StateEqRel( const StateEqRel& r ) :
                StateTo<StateSet *>( r ) {
        assert( class_invariant() );
}

inline const StateEqRel& StateEqRel::operator=( const StateEqRel& r ) {
        assert( class_invariant() );
        assert( r.class_invariant() );
        StateTo<StateSet *>::operator=( r );
        assert( class_invariant() );
        return( *this );
}

inline int StateEqRel::equivalent( const State p, const State q ) const {
        assert( class_invariant() );
        assert( (0 <= p) && (p < domain()) );
        assert( (0 <= q) && (q < domain()) );
        return( lookup( p ) == lookup( q ) );
}

inline const StateSet& StateEqRel::equiv_class( const State p ) const {
        assert( class_invariant() );
        assert( (0 <= p) && (p < domain()) );
        return( *lookup( p ) );
}

inline State StateEqRel::eq_class_representative( const State p ) const {
        assert( class_invariant() );
        assert( (0 <= p) && (p < domain()) );
        return( lookup( p )->smallest() );
}

inline int StateEqRel::domain() const {
        return( StateTo<StateSet *>::domain() );
}

inline int StateEqRel::class_invariant() const {
        auto int result( 1 );
        auto State i;
        for( i = 0; i < domain() && result; i++ ) {
                result = (domain() == lookup( i )->domain());
                result = result && lookup( i )->class_invariant();
                // i's eq. class should contain i itself.
                result = result && lookup( i )->contains( i );
                // And that's all that lookup( i ) should contain:
                //        Iterate over lookup( i ) and check that.
                auto State j;
                for( lookup( i )->iter_start( j );
                                !lookup( i )->iter_end( j ) && result;
                                lookup( i )->iter_next( j ) ) {
                        result = (lookup( i ) == lookup( j ));
                }
        }
        return( result );
}

#endif
```

Implementation: *StateEqRel* inherits for implementation from *StateTo<StateSet *>*. Two equivalent *States* are mapped by the *StateTo* to pointers to the same *StateSet*.

```
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:37 $
#include "st-eqrel.h"

// Construct the total eq. relation of domain r.
StateEqRel::StateEqRel( const int r ) :
		StateTo<StateSet *>() {
	StateTo<StateSet *>::set_domain( r );

	// Now construct a StateSet representing the total relation.
	auto StateSet *const t( new StateSet );
	t->set_domain( r );
	t->complement();

	auto State p;
	for( p = 0; p < r; p++ ) {
		StateTo<StateSet *>::map( p ) = t;
	}

	// Do not delete t!!
	assert( class_invariant() );
}

StateEqRel& StateEqRel::equivalize( const State p, const State q ) {
	assert( class_invariant() );
	// Don't do anything needless:
	if( !equivalent( p, q ) ) {
		// This is dangerous if they point to the same thing.
		assert( lookup( p ) != lookup( q ) );

		// Include q's eq. class in p's
		map( p )->set_union( *lookup( q ) );

		// Get q's eq. class relating to the p's.
		auto StateSet *oldq( lookup( q ) );
		auto StateSet *newp( lookup( p ) );
		assert( oldq != newp );

		auto State i;
		for( oldq->iter_start( i ); !oldq->iter_end( i ); oldq->iter_next( i ) ) {
			map( i ) = newp;
		}
		assert( equivalent( p, q ) );
		delete oldq;
		assert( lookup( p ) == lookup( q ) );
	}

	assert( class_invariant() );
	return( *this );
}

StateEqRel& StateEqRel::split( const StateSet& r ) {
	assert( class_invariant() );
	assert( domain() == r.domain() );

	// Only split it if there's actually some splitting to do.
	//	Choose first State in r for starters.
	auto State i;
	r.iter_start( i );
	// Only do the splitting if r is wholly contained in the eq. class of it's
	// smallest element, and if the splitting will actually produce a new eq.
	// class.
	if( lookup( i )->contains( r ) && *lookup( i ) != r ) {
		// Split it.
		//	Make all of not( r )'s States equivalent.
		map( i )->remove( r );
		// Make all of r's States equivalent.
```

```
			//	Build a new StateSet for the new class.
			auto StateSet *n( new StateSet( r ) );
			// i is already the first State in r.
			for( ; !r.iter_end( i ); r.iter_next( i ) ) {
				map( i ) = n;
			}
		}

		assert( class_invariant() );
		return( *this );
}

StateEqRel& StateEqRel::identity() {
		assert( class_invariant() );

		// We'll need the representatives later.
		auto StateSet repr( representatives() );

		// Wipe out all previous stuff.
		auto State st;
		for( st = 0; st < domain(); st++ ) {
			// There may be something to wipe out here.
			// To avoid double delete [] of the same memory, only do the
			// deletion when st is the representative of it's eq. class.
			if( repr.contains( st ) ) {
			}
			// Assign the new StateSet.
			map( st ) = new StateSet;
			// It's the emptyset, and set the domain.
			lookup( st )->set_domain( domain() );
			assert( lookup( st )->empty() );
			lookup( st )->add( st );
		}
		assert( class_invariant() );
		return( *this );
}

StateSet StateEqRel::representatives() const {
		assert( class_invariant() );
		// Keep track of the States to consider.
		auto StateSet consider;
		consider.set_domain( domain() );
		consider.complement();

		auto StateSet ret;
		ret.set_domain( domain() );

		auto State st;
		// Iterate over all States that must be considered.
		//	starting with all of the States.
		for( consider.iter_start( st ); !consider.iter_end( st ); consider.iter_next( st ) ) {
			ret.add( eq_class_representative( st ) );
			// No need to look at the rest of eq. class [st].
			consider.remove( equiv_class( st ) );
		}

		assert( class_invariant() );
		return( ret );
}

void StateEqRel::set_domain( int r ) {
		assert( class_invariant() );
		StateTo<StateSet *>::set_domain( r );

		assert( class_invariant() );
}
```

```
ostream& operator<<( ostream& os, const StateEqRel& r ) {
        assert( r.class_invariant() );
        os << "\nStateEqRel\n";
        // Print out all of the equivalence classes (each only once).
        auto State st;
        for( st = 0; st < r.domain(); st++ ) {
                // If i is the representative of its eq. class, then print the class.
                if( st == r.eq_class_representative( st ) ) {
                        os << *r.lookup( st ) << '\n';
                }
        }
        return( os );
}
```

□

Implementation class: *TransRel*

Files: `transrel.h`, `transrel.cpp`

Uses: *CharRange, CRSet, State, StateSet, StateTo, Trans*

Description: A *TransRel* maps a *StateSet* and a character (or a *CharRange*) to a *StateSet*. It is used to implement the transition relation in the class *FA* (see [Wat93a, Definition 2.1]). The responsibility of maintaining the domain of the relation lies with the client. Transitions can be added, but not removed.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 16:00:48 $
#ifndef TRANSREL_H
#define TRANSREL_H

#include "state.h"
#include "stateset.h"
#include "stateto.h"
#include "charrang.h"
#include "crset.h"
#include "trans.h"
#include <iostream.h>

// Implement a transition relation, using a function from States to Trans (which, in turn
// are char -> State).
// This is used for transition relations in normal FA's.

class TransRel : protected StateTo<Trans> {
public:
        // Constructors, destructors, operator=:

        // Default argument-less constructor is okay.
        inline TransRel();

        inline TransRel( const TransRel& r );

        // Default destructor is okay

        inline const TransRel& operator=( const TransRel& r );


        // Some relational operators:

        // Compute the image of r, and a under *this.
        StateSet image( const StateSet& r, const char a ) const;
```

```
        // Transitioning on a CharRange? (Same idea as above.)
        StateSet transition_on_range( const StateSet& r, const CharRange a ) const;

        // On which labels can we transition?
        CRSet out_labels( const StateSet& r ) const;


        // Some functions updating *this:

        inline TransRel& add_transition( const State p, const CharRange r, const State q );


        // Some domain related members:

        // What is the domain of this relation.
        inline int domain() const;

        // Change the domain of this relation.
        void set_domain( const int r );

        // Recycle this entire structure.
        void reincarnate();

        // Union relation r into *this, while adjusting r.
        TransRel& disjointing_union( const TransRel& r );


        // Some special members:

        friend ostream& operator<<( ostream& os, const TransRel& r );

        inline int class_invariant() const;
};

inline TransRel::TransRel() :
                StateTo<Trans>() {
        assert( class_invariant() );
}

inline TransRel::TransRel( const TransRel& r ) :
                StateTo<Trans>( (StateTo<Trans> &)r ) {
        assert( class_invariant() );
}

inline const TransRel& TransRel::operator=( const TransRel& r ) {
        assert( r.class_invariant() );
        // Call back to the base class.
        StateTo<Trans>::operator=( (const StateTo<Trans>&) r );
        assert( class_invariant() );
        return( *this );
}

inline int TransRel::domain() const {
        return( StateTo<Trans>::domain() );
}

inline TransRel& TransRel::add_transition( const State p, const CharRange r, const State q ) {
        assert( class_invariant() );
        assert( (0 <= p) && (p < domain()) );
        assert( (0 <= q) && (q < domain()) );
        map( p ).add_transition( r, q );
        assert( class_invariant() );
        return( *this );
}

inline int TransRel::class_invariant() const {
        // First, we must satisfy the base class_invariant().
```

```
        auto int res( StateTo<Trans>::class_invariant() );
        auto int i;
        for( i = 0; (i < domain()) && res; i++ ) {
                res = lookup( i ).class_invariant() && (lookup( i ).range() == domain());
        }
        return( res );
}

#endif
```

Implementation: *TransRel* inherits for implementation from *StateTo<Trans>*. Many of the member functions simply call the corresponding members functions of *StateTo* or *Trans*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 16:00:46 $
#include "transrel.h"

StateSet TransRel::image( const StateSet& r, const char a ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        assert( r.domain() == domain() );
        // result is, by default, the emptyset.
        auto StateSet result;
        result.set_domain( domain() );
        auto State st;
        for( r.iter_start( st ); !r.iter_end( st ); r.iter_next( st ) ) {
                // Here we assume that the Trans in the TransRel have the same domain too.
                result.set_union( lookup( st )[a] );
        }
        return( result );
}

StateSet TransRel::transition_on_range( const StateSet& r, const CharRange a ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        assert( r.domain() == domain() );

        auto StateSet result;
        result.set_domain( domain() );
        // Go through all of the States to see which transitions happen.
        auto State st;
        for( r.iter_start( st ); !r.iter_end( st ); r.iter_next( st ) ) {
                result.set_union( lookup( st ).range_transition( a ) );
        }
        return( result );
}

CRSet TransRel::out_labels( const StateSet& r ) const {
        assert( class_invariant() );

        auto CRSet result;
        auto State st;
        // Go through all of r's States.
        for( r.iter_start( st ); !r.iter_end( st ); r.iter_next( st ) ) {
                result.combine( lookup( st ).out_labels() );
        }
        return( result );
}

void TransRel::set_domain( const int r ) {
        assert( class_invariant() );
        StateTo<Trans>::set_domain( r );
        auto int i;
        for( i = 0; i < domain(); i++ ) {
```

```
                map( i ).set_range( r );
        }
        assert( class_invariant() );
}

void TransRel::reincarnate() {
        assert( class_invariant() );
        auto int i;
        // First, reincarnate() all of the components before *this.
        for( i = 0; i < domain(); i++ ) {
                map( i ).reincarnate();
        }
        StateTo<Trans>::reincarnate();
        assert( class_invariant() );
}

TransRel& TransRel::disjointing_union( const TransRel& r ) {
        assert( class_invariant() && r.class_invariant() );
        auto int olddom( domain() );
        StateTo<Trans>::disjointing_union( r );
        assert( StateTo<Trans>::domain() == r.domain() + olddom );
        auto int i;
        for( i = 0; i < olddom; i++ ) {
                map( i ).set_range( domain() );
        }
        for( ; i < domain(); i++ ) {
                map( i ).st_rename( olddom );
        }
        assert( class_invariant() );
        return( *this );
}

ostream& operator<<( ostream& os, const TransRel& r ) {
        assert( r.class_invariant() );
        return( os << (const StateTo<Trans>&)r );
}
```

□

Implementation class: *DTransRel*

Files: `dtransre.h`, `dtransre.cpp`

Uses: *CharRange, CRSet, DTrans, State, StateSet, StateTo*

Description: A *DTransRel* maps a *State* and a character (or a *CharRange*) to a *State*. It is used to implement the deterministic transition relation in the class *DFA* (see [Wat93a, Property 2.25]). The domain of the relation must be maintained explicitly by the client. Transitions can be added to the relation, but not removed. The client must ensure that the relation remains deterministic.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:59 $
#ifndef DTRANSREL_H
#define DTRANSREL_H

#include "state.h"
#include "stateset.h"
#include "stateto.h"
#include "dtrans.h"
#include "crset.h"
#include <assert.h>
#include <iostream.h>
```

```
// Implement a deterministic transition relation, as a function from States time
// char to State. This is used for transition relations in DFA's.

class DTransRel : protected StateTo<DTrans> {
public:
        // Constructors, destructors, operator=:

        // Argument-less constructor:
        inline DTransRel();

        inline DTransRel( const DTransRel& r );

        // Default destructor is okay:

        inline const DTransRel& operator=( const DTransRel& r );


        // Some relational operators:

        // Compute the image of r, and a under *this.
        inline State image( const State r, const char a ) const;

        // Compute the image of r, and CharRange a under *this.
        inline State transition_on_range( const State r, const CharRange a ) const;

        // Compute the reverse transition on a range under *this:
        StateSet reverse_transition( const State r, const CharRange a ) const;

        // What are the transition labels between some State r and StateSet s?
        inline CRSet labels_between( const State r, const StateSet& s ) const;

        // Given State r, what are the labels on transitions out of r?
        inline CRSet out_labels( const State r ) const;

        // What are all States reverse reachable from r?
        StateSet reverse_closure( const StateSet& r ) const;


        // Some functions updating *this:

        inline DTransRel& add_transition( const State p, const CharRange a, const State q );



        // Some domain members:

        // What is the domain of this relation.
        inline int domain() const;

        // Change the domain of this relation.
        inline void set_domain( const int r );

        // Recycle this entire structure.
        void reincarnate();


        // Some special members:

        friend ostream& operator<<( ostream& os, const DTransRel& r );

        inline int class_invariant() const;
};

inline DTransRel::DTransRel() :
                StateTo<DTrans>() {
        assert( class_invariant() );
}
```

```
inline DTransRel::DTransRel( const DTransRel& r ) :
            StateTo<DTrans>( (StateTo<DTrans> &)r ) {
      assert( class_invariant() );
}

inline const DTransRel& DTransRel::operator=( const DTransRel& r ) {
      assert( r.class_invariant() );
      // Call back to the base class.
      StateTo<DTrans>::operator=( (const StateTo<DTrans>&) r );
      assert( class_invariant() );
      return( *this );
}

inline State DTransRel::image( const State r, const char a ) const {
      assert( class_invariant() );
      assert( 0 <= r && r < domain() );

      return( lookup( r )[a] );
}

inline State DTransRel::transition_on_range( const State r, const CharRange a ) const {
      assert( class_invariant() );
      assert( 0 <= r && r < domain() );

      return( lookup( r ).range_transition( a ) );
}

inline CRSet DTransRel::labels_between( const State r, const StateSet& s ) const {
      assert( class_invariant() );
      assert( s.class_invariant() );
      assert( domain() == s.domain() );
      assert( 0 <= r && r <= domain() );
      return( lookup( r ).labels_into( s ) );
}

inline CRSet DTransRel::out_labels( const State r ) const {
      assert( class_invariant() );
      return( lookup( r ).out_labels() );
}

inline DTransRel& DTransRel::add_transition( const State p,
            const CharRange a, const State q ) {
      assert( class_invariant() );
      map( p ).add_transition( a, q );
      assert( class_invariant() );
      return( *this );
}

inline int DTransRel::domain() const {
      return( StateTo<DTrans>::domain() );
}

inline void DTransRel::set_domain( const int r ) {
      assert( class_invariant() );
      StateTo<DTrans>::set_domain( r );
      assert( class_invariant() );
}

inline int DTransRel::class_invariant() const {
      // First, we must satisfy the base class_invariant().
      auto int i( StateTo<DTrans>::class_invariant() );
      auto int k;
      for( k = 0; (k < domain()) && i; k++ ) {
            i = i && lookup( k ).class_invariant();
      }
      return( i );
```

```
}

inline ostream& operator<<( ostream& os, const DTransRel& r ) {
	assert( r.class_invariant() );
	return( operator<<( os, (const StateTo<DTrans>&)r ) );
}

#endif
```

Implementation: *DTransRel* inherits for implementation from *StateTo<DTrans>*. Many of the member functions are simply calls to the members of *StateTo* or *DTrans*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:57 $
#include "dtransre.h"

StateSet DTransRel::reverse_transition( const State r, const CharRange a ) const {
	assert( class_invariant() );
	assert( (0 <= r) && (r < domain()) );

	auto StateSet ret;
	ret.set_domain( domain() );
	auto State p;
	// Iterate over all of the States, and see which have transitions to r.
	for( p = 0; p < domain(); p++ ) {
		if( transition_on_range( p, a ) == r ) {
			ret.add( p );
		}
	}

	return( ret );
}

StateSet DTransRel::reverse_closure( const StateSet& r ) const {
	assert( class_invariant() );
	assert( r.class_invariant() );
	assert( domain() == r.domain() );

	auto StateSet result( r );
	auto StateSet intermediate;
	intermediate.set_domain( domain() );
	assert( intermediate.empty() );

	do {
		result.set_union( intermediate );
		intermediate.clear();

		// Go through the State's to see which can reach set result on
		// some transition.
		auto State st;
		for( st = 0; st < domain(); st++ ) {
			if( result.not_disjoint( lookup( st ).range( domain() ) ) ) {
				intermediate.add( st );
			}
		}
	} while( !result.contains( intermediate ) );

	return( result );
}

void DTransRel::reincarnate() {
	assert( class_invariant() );
	// Don't reset the domain()
	auto int i;
```

```
        for( i = 0; i < domain(); i++ ) {
                map( i ).reincarnate();
        }
        StateTo<DTrans>::reincarnate();
        assert( class_invariant() );
}
```

60

□

Part II
Regular expressions and Σ-algebras

7 Regular expressions

Implementation class: *REops*

Files: `reops.h`

Uses:

Description: The regular operators (of the Σ-algebra — see class *Reg* or [Wat93a, Definition 3.12]) are encoded as elements of an enumeration. These are usually stored in a type-field in regular expressions. They play an important role in computing the homomorphic (Σ-algebra) image of a regular expression.

Implementation: *REops* is defined as an **enum**.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:25 $
#ifndef REOPS_H
#define REOPS_H


// Define the regular operators as an enumerated type. These correspond to the
// Sigma algebra operators (see Definition 3.12 of the Taxonomy, and Sigma.h).

enum REops {
        EPSILON,
        EMPTY,
        SYMBOL,
        OR,
        CONCAT,
        STAR,
        PLUS,
        QUESTION
};

#endif
```

□

Implementation class: *RE*

Files: `re.h`, `re.cpp`

Uses: *CharRange, CRSet, REops*

Description: An *RE* is a regular expression (see [Wat93a, Section 3] for the definition of regular expressions). The argumentless constructor constructs the regular expression $\emptyset$ (the regular expression denoting the empty language). It is not possible to construct more complex regular expressions with the member functions (see the template class *Reg*<*RE*> for information on constructing regular expressions). Class *RE* is the implementation of the carrier set of the Σ-term algebra (an *RE* is a term in the Σ-term algebra).

A member function returns the operator type of the main (or root) operator of the regular expression. (The operator types are enumerated in *REops*.) Other member functions can be used to determine additional information about the main (root) operator of the regular expression: the associated *CharRange* (for a *SYMBOL* basis operator), and the subexpressions (for non-basis, binary and unary, operators). A very basic *istream* extraction (input) operator is provided, expecting the input to be a regular expression in prefix notation; the operator does little error checking.

```
/* (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;

protected:
	// Some protected helpers, mainly for Reg<RE>:

	inline void set_root_operator( REops r );
	inline void set_symbol( const CharRange r );

	// Make a copy of *this in *r
	void shallow_copy( RE *const r ) const;

	void reincarnate();

	// Put *this into OR normal form (ONF) (used in the snf functions)
	RE& onf();

	// Destroy node *r, moving its data into *this.
	void assimilate_node( RE *const r );


	// Some implementation details.
	REops op;
	RE *left, *right;
	CharRange sym;
};

inline RE::RE() : op( EMPTY ),
		left( 0 ),
		right( 0 ) {
	assert( class_invariant() );
}

// Some inline operators:

inline REops RE::root_operator() const {
	return( op );
```

```
}

inline CharRange RE::symbol() const {
        assert( op == SYMBOL );
        return( sym );
}

// Assume that *this is a higher (non-basis) regular operator.
inline const RE& RE::left_subexpr() const {
        assert( (op != EMPTY) && (op != EPSILON) && (op != SYMBOL) );
        return( *left );
}

// Assume that *this is a binary regular operator.
inline const RE& RE::right_subexpr() const {
        assert( (op == OR) || (op == CONCAT) );
        return( *right );
}

inline void RE::set_root_operator( REops r ) {
        // No class assertion here, since *this is still being constructed.
        op = r;
}

inline void RE::set_symbol( const CharRange r ) {
        assert( op == SYMBOL );
        sym = r;
}

inline int RE::operator==( const RE& r ) const {
        return( ordering( r ) == 0 );
}

inline int RE::operator!=( const RE& r ) const {
        return( ordering( r ) != 0 );
}

inline int RE::operator<( const RE& r ) const {
        return( ordering( r ) < 0 );
}

inline int RE::operator>=( const RE& r ) const {
        return( ordering( r ) >= 0 );
}

inline int RE::operator>( const RE& r ) const {
        return( ordering( r ) > 0 );
}

inline int RE::operator<=( const RE& r ) const {
        return( ordering( r ) <= 0 );
}

inline int RE::class_invariant() const {
        switch( op ) {
        case EPSILON:
        case EMPTY:
                return( left == 0 && right == 0 );
        case SYMBOL:
                return( left == 0 && right == 0 );
        case OR:
        case CONCAT:
                return( left != 0 && right != 0
                        && left_subexpr().class_invariant()
                        && right_subexpr().class_invariant() );
        case STAR:
        case PLUS:
```

```
        case QUESTION:
                return( left != 0 && right == 0
                        && left_subexpr().class_invariant() );
        default:
                return( 0 );
        }
}

#endif
```

Implementation: Regular expressions are implemented as expression trees. The comparison of regular expressions is defined recursively and uses a depth-first traversal.

Performance: Use-counting could avoid the deep-tree copies that are performed. An *RE* could be stored as a prefix-form string, as in [RW93]; this could give some problems with the heavy use of members such as *left_subexpr* and *right_subexpr*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:19 $
#include "re.h"
#include "dsre.h"
#include "dsre-opt.h"
#include "dfaseed.h"


RE::RE( const RE& e ) :
                op( e.op ),
                right( 0 ) {        // Only the right on purpose.
        assert( e.class_invariant() );

        switch( e.root_operator() ) {
        case EPSILON:
        case EMPTY:
                left = 0;
                break;
        case SYMBOL:
                left = 0;
                // Now it's safe to copy the sym.
                sym = e.sym;
                break;
        case OR:
        case CONCAT:
                right = new RE( e.right_subexpr() );
        case STAR:
        case PLUS:
        case QUESTION:
                left = new RE( e.left_subexpr() );
                break;
        default:
                // Should throw() something!
                break;
        }

        assert( class_invariant() );
}


RE::~RE() {
        delete left;
        delete right;
}

const RE& RE::operator=( const RE& e ) {
```

```
        assert( e.class_invariant() );

        op = e.op;
        right = 0;

        switch( e.root_operator() ) {
        case EPSILON:
        case EMPTY:
                left = 0;
                break;
        case SYMBOL:
                left = 0;
                // In no other case is it safe to copy the sym.
                sym = e.sym;
                break;
        case OR:
        case CONCAT:
                delete right;
                right = new RE( e.right_subexpr() );
        case STAR:
        case PLUS:
        case QUESTION:
                delete left;
                left = new RE( e.left_subexpr() );
                break;
        }

        assert( class_invariant() );
        return( *this );
}

// How many symbols are there in *this RE?
int RE::num_symbols() const {
        assert( class_invariant() );

        auto int ret;
        switch( op ) {
        case EPSILON:
        case EMPTY:
                ret = 0;
                break;
        case SYMBOL:
                ret = 1;
                break;
        case OR:
        case CONCAT:
                ret = left->num_symbols() + right->num_symbols();
                break;
        case STAR:
        case PLUS:
        case QUESTION:
                ret = left->num_symbols();
                break;
        }
        return( ret );
}

// How many operators are there in *this RE?
int RE::num_operators() const {
        assert( class_invariant() );

        auto int ret;
        switch( op ) {
        case EPSILON:
        case EMPTY:
        case SYMBOL:
                ret = 1;
```

```
                break;
        case OR:
        case CONCAT:
                ret = left->num_operators() + right->num_operators() + 1;
                break;
        case STAR:
        case PLUS:
        case QUESTION:
                ret = left->num_operators() + 1;
                break;
        }
        return( ret );
}

// Copy *this into *r
void RE::shallow_copy( RE *const r ) const {
        r->op = op;
        r->left = left;
        r->right = right;
        r->sym = sym;
}

void RE::reincarnate() {
        if( left != 0 ) {
                // This is a non-basic operator for reincarnation.
                // Delete what used to be there.
                delete left;
                left = 0;
                if( right != 0 ) {
                        // This is a binary operator for reincarnation.
                        delete right;
                        right = 0;
                }
        }
        op = EMPTY;
        assert( class_invariant() );
}

// Output in prefix notation.
// Part of this could be REops, if enums could be used for overload resolution.
ostream& operator<<( ostream& os, const RE& r ) {
        switch( r.root_operator() ) {
        case EMPTY:
                os << '0';
                break;
        case EPSILON:
                os << '1';
                break;
        case SYMBOL:
                os << r.symbol();
                break;
        case OR:
                os << "| " << r.left_subexpr() << ' ' << r.right_subexpr();
                break;
        case CONCAT:
                os << ". " << r.left_subexpr() << ' ' << r.right_subexpr();
                break;
        case STAR:
                os << "* " << r.left_subexpr();
                break;
        case PLUS:
                os << "+ " << r.left_subexpr();
                break;
        case QUESTION:
                os << "? " << r.left_subexpr();
                break;
        }
```

```
	return( os );
}

// Input, in prefix notation. Not much error checking is done!
istream& operator>>( istream& is, RE& r ) {
	// Wipe out anything that used to be in r.
	delete r.left;
	delete r.right;
	// If something goes wrong try to leave r as the EMPTY regular expression.
	r.left = r.right = 0;
	r.op = EMPTY;

	auto char c;
	// It could be that there is nothing in th input.
	if( !(is >> c) ) {
		is.clear( ios::badbit | is.rdstate() );
	} else {
		switch( c ) {
		case '1':
			r.op = EPSILON;
			break;
		case '0':
			r.op = EMPTY;
			break;
		case '[':
			// This stuff should really be as the extraction operator
			// in class CharRange.
			is >> c;
			if( c != '\'' ) {
				cerr << "First CharRange incorrect\n";
				is.clear( ios::badbit | is.rdstate() );
			} else {
				// Input first character of the range.
				is >> c;
				auto char c2;
				is >> c2;
				if( c2 != '\'' ) {
					cerr << "First CharRange not terminated properly\n";
					is.clear( ios::badbit | is.rdstate() );
				} else {
					is >> c2;
					if( c2 != '\'' ) {
						cerr << "Second CharRange incorrect\n";
						is.clear( ios::badbit | is.rdstate() );
					} else {
						// Input first char. of second range.
						is >> c2;
						r.sym = CharRange( c, c2 );
						r.op = SYMBOL;
						is >> c;
						if( c != '\'' ) {
							cerr << "Second CharRange not"
								" terminated properly\n";
							is.clear( ios::badbit | is.rdstate() );
						}
						is >> c;
						if( c != ']' ) {
							cerr << "Close block not found\n";
							is.clear( ios::badbit | is.rdstate() );
						}
					}
				}
			}
			break;
		case '\'':
			is >> c;
			r.sym = CharRange( c );
```

```
                r.op = SYMBOL;
                is >> c;
                if( c != '\'' ) {
                        cerr << "Single character terminated incorrectly\n";
                        is.clear( ios::badbit | is.rdstate() );
                }
                break;
        case '|':
                r.op = OR;
                r.left = new RE;
                r.right = new RE;
                is >> *r.left;
                is >> *r.right;
                break;
        case '.':
                r.op = CONCAT;
                r.left = new RE;
                r.right = new RE;
                is >> *r.left;
                is >> *r.right;
                break;
        case '*':
        case '+':
        case '?':
                if( c == '*' ) {
                        r.op = STAR;
                } else if( c == '+' ) {
                        r.op = PLUS;
                } else if( c == '?' ) {
                        r.op = QUESTION;
                }
                r.left = new RE;
                is >> *r.left;
                break;
        default:
                cerr << "Something went wrong\n";
                is.clear( ios::badbit | is.rdstate() );
        }
        assert( r.class_invariant() );
    }
    return( is );
}
```

□

7.1 Similarity of *REs*

Similarity is an equivalence relation on regular expressions, defined in [Wat93a, Definition 5.26]. A pair of similar regular expressions denote the same language, while their expression trees may not be identical. Similarity is used extensively in Brzozowski's deterministic finite automata construction [Wat93a, Construction 5.34].

The normal *RE* equality operator can be used to determine the similarity of two regular expressions if they are both in similarity normal form (SNF). In order to put an *RE* into SNF, we define a total ordering (which we will also write as $\leq$) on regular expressions; the member function *ordering* takes two regular expressions and returns a negative integer, zero, or a positive integer depending upon their relative position in the total ordering. (The ordering is essentially the lexicographic ordering of the prefix-form of the regular expression.) An *RE* is in SNF if and only if all of its subexpressions are in SNF, and all *OR* subexpressions (subexpressions with *OR* as the root operator[1]) are in *OR* normal form (ONF). An *OR* regular expression is in ONF if and only if:

[1] operator *OR* corresponds to the $\cup$ operator in [Wat93a, Section 3], and we sometimes write $\cup$ instead of *OR*

- All of its *OR* operators are left-associative; for example $(E_0 \cup E_1) \cup E_2$ is left-associative, while $E_0 \cup (E_1 \cup E_2)$ is not.
- The *OR* regular expression is in one of two forms (where $\leq$ is the total ordering on regular expressions): either $(E_0 \cup E_1) \cup E_2$ where $E_1 < E_2$ and $(E_0 \cup E_1)$ is in ONF, or $E_0 \cup E_1$ where $E_0 < E_1$. (The use of $<$ instead of $\leq$ implies that if the subexpressions of an *OR* expression are equal (under the total ordering), then they are collapsed into one expression, implementing the idempotence of *OR*.)

The ONF is used to provide the equivalent of a multi-ary *OR* operator, implementing the associativity, commutativity and idempotence of the *OR* operator, as detailed in [Wat93a, Definition 5.26].

The member function *in_snf* returns 1 if the *RE* is in SNF, and 0 otherwise. The member *snf* puts the *RE* in SNF, using function *onf* as a helper. Member function *reduce* optimizes an *RE*, removing any redundant subexpressions. It implements rules 1 to 5 of [Wat93a, Remark 5.32]. The function performs a bottom-up traversal of the regular expression, making such transformations as $\emptyset \cdot E \longrightarrow \emptyset$ and $\emptyset^* \longrightarrow \epsilon$.

All of these members (including the total ordering) are defined in file `deriv.cpp`.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:55:57 $
#include "re.h"
#include "sigma.h"


// Does *this accept epsilon?
// Implement Definition 3.20 and Property 3.21
int RE::Null() const {
	assert( class_invariant() );

	auto int ret;
	switch( op ) {
	case EPSILON:
		ret = 1;
		break;
	case EMPTY:
	case SYMBOL:
		ret = 0;
		break;
	case OR:
		ret = left->Null() || right->Null();
		break;
	case CONCAT:
		ret = left->Null() && right->Null();
		break;
	case STAR:
		ret = 1;
		break;
	case PLUS:
		ret = left->Null();
		break;
	case QUESTION:
		ret = 1;
		break;
	}
	return( ret );
}

// What is the First of *this?
// Implement Definition 4.60
CRSet RE::First() const {
	assert( class_invariant() );
```

```
	auto CRSet a;
	switch( op ) {
	case EPSILON:
	case EMPTY:
		break;
	case SYMBOL:
		a.append( sym );
		break;
	case OR:
		a = left->First().combine( right->First() );
		break;
	case CONCAT:
		a = left->First();
		if( left->Null() ) {
			a.combine( right->First() );
		}
		break;
	case STAR:
	case PLUS:
	case QUESTION:
		a = left->First();
		break;
	}
	return( a );
}

// Is *this in similarity normal form (SNF)?
int RE::in_snf() const {
	assert( class_invariant() );

	auto int ret;
	// SNF means that all OR nodes are left-associative, and
	// in the two cases:
	//	A OR B that A < B, or
	//	(A OR B) OR C that B < C and (A OR B) is in SNF.
	switch( op ) {
	case EPSILON:
	case EMPTY:
	case SYMBOL:
		// The bases are always in SNF.
		ret = 1;
		break;
	case OR:
		if( (right->op == OR) || !right->in_snf() ) {
			// This can't be in SNF.
			ret = 0;
		} else {
			if( left->op == OR ) {
				// Second case.
				ret = (*(left->right) < *right) && left->in_snf();
			} else {
				// First case.
				ret = (*left < *right) && left->in_snf();
			}
		}
		break;
	case CONCAT:
		// A CONCAT is in SNF if the subexpressions are in SNF.
		ret = left->in_snf() && right->in_snf();
		break;
	case STAR:
	case PLUS:
	case QUESTION:
		// A unary subexpr. is in SNF if it's subexpressions are too.
		ret = left->in_snf();
		break;
	}
```

```
	return( ret );
}

// Put *this into similarity normal form.
RE& RE::snf() {
	assert( class_invariant() );

	switch( op ) {
	case EPSILON:
	case EMPTY:
	case SYMBOL:
		// Already in SNF.
		break;
	case OR:
		// While the right is an OR, rotate them.
		// Note: this is not a true rotation, since we get
		// E | (F | G) -> (F | E) | G instead of (E | F) | G
		// but, by commutativity it's okay.
		while( right->op == OR ) {
			auto RE *temp( left );
			left = right;
			right = left->right;
			left->right = temp;
			assert( class_invariant() );
		}
		assert( right->op != OR );
		left->snf();
		assert( class_invariant() );
		assert( left->in_snf() );
		right->snf();
		assert( class_invariant() );
		assert( right->in_snf() );
		onf();
		assert( class_invariant() );
		break;
	case CONCAT:
		left->snf();
		assert( class_invariant() );
		assert( left->in_snf() );
		right->snf();
		assert( class_invariant() );
		assert( right->in_snf() );
		break;
	case STAR:
	case PLUS:
	case QUESTION:
		left->snf();
		assert( class_invariant() );
		assert( left->in_snf() );
		break;
	}
	assert( in_snf() );
	assert( class_invariant() );
	return( *this );
}

// What is the derivative of *this?
// Implement Definition 5.23
RE RE::derivative( const CharRange& r ) const {
	assert( class_invariant() );

	// Duplicate some of the knowledge appearing in sig-re.cpp
	auto RE e;
	switch( op ) {
	case EPSILON:
	case EMPTY:
		assert( e.op == EMPTY );
```

```
			break;
		case SYMBOL:
			if( r <= sym ) {
				e.op = EPSILON;
				assert( e.class_invariant() );
			} else {
				// else, it should be EMPTY.
				assert( e.op == EMPTY );
			}
			break;
		case OR:
			e.op = OR;
			e.left = new RE( left->derivative( r ) );
			e.right = new RE( right->derivative( r ) );
			assert( e.class_invariant() );
			break;
		case CONCAT:
			if( left->Null() ) {
				e.op = OR;
				e.right = new RE( right->derivative( r ) );

				e.left = new RE;
				e.left->op = CONCAT;
				e.left->left = new RE( left->derivative( r ) );
				e.left->right = new RE( *right );
				assert( e.class_invariant() );
			} else {
				e.op = CONCAT;
				e.left = new RE( left->derivative( r ) );
				e.right = new RE( *right );
				assert( e.class_invariant() );
			}
			break;
		case STAR:
			e.op = CONCAT;
			e.left = new RE( left->derivative( r ) );
			e.right = new RE( *this );
			assert( e.class_invariant() );
			break;
		case PLUS:
			e.op = CONCAT;
			e.left = new RE( left->derivative( r ) );

			e.right = new RE;
			e.right->op = STAR;
			e.right->right = 0;
			e.right->left = new RE( *left );

			assert( e.class_invariant() );
			break;
		case QUESTION:
			e = left->derivative( r );
			break;
		// By the class_invariant(), the default: cannot occur.
		}
		assert( e.class_invariant() );
		return( (RE)e );
}

// Implement rules 1-4 of Remark 5.32 of the Taxonomy.
RE& RE::reduce() {
		assert( class_invariant() );
		switch( op ) {
		case EPSILON:
		case EMPTY:
		case SYMBOL:
			// These basis operators cannot be reduced further.
```

```
        break;
case OR:
        left->reduce();
        assert( class_invariant() );
        right->reduce();
        assert( class_invariant() );
        if( left->op == EMPTY ) {
                // EMPTY OR E == E
                delete left;
                assimilate_node( right );
                assert( class_invariant() );
        } else if( right->op == EMPTY ) {
                delete right;
                assimilate_node( left );
                assert( class_invariant() );
        }
        // The case where left and right are both EMPTY is handled correctly.
        break;
case CONCAT:
        left->reduce();
        assert( class_invariant() );
        right->reduce();
        assert( class_invariant() );
        if( left->op == EMPTY ) {
                // EMPTY CONCAT E == EMPTY
                delete right;
                assimilate_node( left );
                assert( class_invariant() );
        } else if( right->op == EMPTY ) {
                delete left;
                assimilate_node( right );
                assert( class_invariant() );
        } else if( left->op == EPSILON ) {
                // EPSILON CONCAT E == E
                delete left;
                assimilate_node( right );
                assert( class_invariant() );
        } else if( right->op == EPSILON ) {
                delete right;
                assimilate_node( left );
                assert( class_invariant() );
        }
        break;
case STAR:
        left->reduce();
        assert( class_invariant() );
        if( left->op == EPSILON ) {
                // EPSILON * == EPSILON
                assimilate_node( left );
                assert( class_invariant() );
        } else if( left->op == EMPTY ) {
                // EMPTY * == EPSILON
                delete left;
                op = EPSILON;
                left = right = 0;
                assert( class_invariant() );
        }
        break;
case PLUS:
        left->reduce();
        assert( class_invariant() );
        if( left->op == EPSILON ) {
                // EPSILON + == EPSILON
                assimilate_node( left );
                assert( class_invariant() );
        } else if( left->op == EMPTY ) {
                // EMPTY + == EMPTY
```

```
                        assimilate_node( left );
                        assert( class_invariant() );
                }
                break;
        case QUESTION:
                left->reduce();
                assert( class_invariant() );
                if( left->op == EPSILON ) {
                        // EPSILON ? == EPSILON
                        assimilate_node( left );
                        assert( class_invariant() );
                } else if( left->op == EMPTY ) {
                        // EMPTY ? == EPSILON
                        delete left;
                        op = EPSILON;
                        left = right = 0;
                        assert( class_invariant() );
                }
                break;
        }
        assert( class_invariant() );
        return( *this );
}

// Put *this into OR normal form (ONF) (used in the snf functions)
RE& RE::onf() {
        assert( class_invariant() );
        assert( op == OR );
        assert( right->op != OR );

        // Two possibilities: (A OR B) OR C, A OR B.
        if( left->op == OR ) {
                // First one.
                // Figure out which order B,C should be in:
                auto int ord( left->right->ordering( *right ) );
                if( ord == 0 ) {
                        // B == C
                        // destroy C.
                        delete right;
                        assimilate_node( left );
                } else if( ord > 0 ) {
                        // B > C
                        // Need to swap B,C
                        // Just do it with pointers:
                        auto RE *const temp( right );
                        right = left->right;
                        left->right = temp;

                        // Also make sure that (what's now) (A OR C)
                        // is also in ONF.
                        left->onf();
                }
                // else, it's already in OR NF.
        } else {
                // Second case.
                // Figure out the order of A,B:
                auto int ord( left->ordering( *right ) );
                if( ord == 0 ) {
                        // A == B
                        // Wipe out one of the two nodes, and make *this
                        // equal to the remaining one.
                        delete right;
                        assimilate_node( left );
                } else if( ord > 0 ) {
                        // A > B
                        // Need to swap A,B (left and right):
                        auto RE *const temp( right );
```

```
                        right = left;
                        left = temp;
                }
                // else, it's already in OR NF
        }

        assert( class_invariant() );
        assert( in_snf() );
        return( *this );
}

// Some ordering members (largely used in similarity)
// (This is a kind of lexicographic ordering.)
int RE::ordering( const RE& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        if( op != r.op ) {
                return( op - r.op );
        } else {
                assert( op == r.op );
                auto int ret;
                switch( op ) {
                case EPSILON:
                case EMPTY:
                        // These two are unordered (equal).
                        ret = 0;
                        break;
                case SYMBOL:
                        // Both are symbols, so order on the sym.
                        ret = sym.ordering( r.sym );
                        break;
                case OR:
                case CONCAT:
                        ret = left->ordering( r.left_subexpr() );
                        if( ret != 0 ) {
                                // Do nothing.
                        } else {
                                // Compare the right subexprs, since the left
                                // ones are equal.
                                ret = right->ordering( r.right_subexpr() );
                        }
                        break;
                case STAR:
                case PLUS:
                case QUESTION:
                        ret = left->ordering( r.left_subexpr() );
                        break;
                }
                return( ret );
        }
}

// Assimilate *r into *this, and wipe out *r.
void RE::assimilate_node( RE *const r ) {
        // We can't assert( class_invariant() );
        //         since we can't be sure that *this is still good.
        //         This just delays catching any potential invariant violations.
        assert( r->class_invariant() );
        op = r->op;
        left = r->left;
        right = r->right;
        sym = r->sym;
        assert( class_invariant() );

        r->op = EMPTY;
        r->left = r->right = 0;
        delete r;
```

```
}
```

7.2 Derivatives

Two functions on regular expressions, *Null* and *First* (see [Wat93a, Example 3.20, Definition 4.60] — both used in computing derivatives of regular expressions), are defined as recursive member functions in file `deriv.cpp`. Function *Null* returns 1 if the empty word (ϵ) is in the language denoted by the regular expression, and 0 otherwise. Function *First* returns a *CRSet* denoting the characters which can occur as the first character of a word in the language denoted by the regular expression.

Member function *derivative* takes a *CharRange* and returns the derivative of the regular expression with respect to the *CharRange*. The derivative is computed recursively according to the definition given in [Wat93a, Definition 5.23]. It is used in the implementation of Brzozowski's construction.

8 The Σ-algebra

The Σ-algebra (as defined in [Wat93a, Section 3]) is defined as a template class.

Implementation class: *Reg*

Files: `sigma.h`

Uses: *CharRange, RE, REops*

Description: *Reg* implements the Σ-algebra as defined in [Wat93a, Section 3]. The single type (or generic) parameter *T* is used to specify the carrier set of the Σ-algebra. The template *Reg* is an abstract template, in the sense that it is only used to force a common interface between the Σ-algebras. Most of the member functions are left undefined, with versions specific to the carrier set being defined separately (see files `sig-re.cpp`, `sig-fa.cpp`, and `sig-rfa.cpp`). Regular expressions (class *RE*) are taken as the Σ-term algebra. A special member function is defined, taking a regular expression, and constructing the homomorphic image of the *RE*, in the Σ-algebra of *T*. The Σ-algebra is the only way to construct regular expressions, and it is particularly useful for constructing parts of a finite automaton separately. *Reg*<*T*> inherits publicly from a *T*, and since a *Reg*<*T*> contains nothing extra, it can be cast to a *T* safely (no slicing occurs, and no information is lost). Presently, only *Reg*<*RE*>, *Reg*<*FA*>, and *Reg*<*RFA*> are available. (Each of *RE*, *FA*, and *RFA* also have constructors taking a **const** reference to *RE*; for practical purposes, these constructors are much faster than constructing the homomorphic image using *Reg*.)

Implementation: The template is used to force a common interface for Σ-algebras with different carrier sets. Forcing an interface is commonly done using abstract base classes, although this is not possible in this case, as the return types of many of the member functions would not be correct.

Performance: The homomorphic image member function is recursive, and keeps partially constructed results in local variables. Use-counting in the carrier class can improve performance.

```
/* (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();


protected:
        // Helper for constructing the homomorphic image of a regular expression.
        //        (Can also be used as a type of copy constructor for RE's.)
        inline void homomorphic_image( const RE& r );
};


// Helper for constructing the homomorphic image of a regular expression.
//        (Can also be used as a type of copy constructor for RE's.)
template<class T>
void Reg<T>::homomorphic_image( const RE& r ) {
        assert( r.class_invariant() );

        // Construct the homomorphic image of r in *this.
        switch( r.root_operator() ) {
        case EMPTY:
                // Default constructor T() should already leave *this accepting the
                // empty language.
                //        The following call is redundant:
                empty();
                break;
        case EPSILON:
                // Make *this accept epsilon.
                epsilon();
                break;
        case SYMBOL:
                // Make *this accept the specified symbol.
                symbol( r.symbol() );
                break;
        case OR:
```

```
                // First make *this accept the language of the left side of the OR
                // by constructing a T for the right subexpression.
                homomorphic_image( r.left_subexpr() );
                // Make *this accept the (left or right) language.
                or( Reg<T>( r.right_subexpr() ) );
                break;
        case CONCAT:
                // See OR case.
                homomorphic_image( r.left_subexpr() );
                concat( Reg<T>( r.right_subexpr() ) );
                break;
        case STAR:
                // See OR case.
                homomorphic_image( r.left_subexpr() );
                star();
                break;
        case PLUS:
                // See OR case.
                homomorphic_image( r.left_subexpr() );
                plus();
                break;
        case QUESTION:
                // See OR case.
                homomorphic_image( r.left_subexpr() );
                question();
                break;
        }
}

#endif
```

□

8.1 The Σ-term algebra *Reg<RE>*

Implementation class: *Reg<RE>*

Files: `sig-re.cpp`

Uses: *CharRange, Reg, RE, REops*

Description: The template instantiation *Reg<RE>* can be used to construct regular expressions (see the documentation on template class *Reg*).

Implementation: The implementation consists of straight-forward expression tree manipulations. The implementation of *RE* is declared **protected** to give *Reg<RE>* access to it.

Performance: These member functions would benefit from use-counting of class *RE*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:31 $
#include "charrang.h"
#include "reops.h"
#include "re.h"
#include "sigma.h"

// Implementations of the Sigma-algebra operators, with RE as the carrier.
// This is the Sigma-term algebra (see the Taxonomy, Section 3).

Reg<RE>& Reg<RE>::epsilon() {
        // This may have been something in a previous life.
        reincarnate();
```

```
        set_root_operator( EPSILON );
        assert( class_invariant() );
        return( *this );
}

Reg<RE>& Reg<RE>::empty() {
        // See epsilon() case.
        reincarnate();

        set_root_operator( EMPTY );
        assert( class_invariant() );
        return( *this );
}

Reg<RE>& Reg<RE>::symbol( const CharRange r ) {
        // See epsilon case.
        reincarnate();

        set_root_operator( SYMBOL );
        set_symbol( r );
        assert( class_invariant() );
        return( *this );
}

Reg<RE>& Reg<RE>::or( const Reg<RE>& r ) {
        assert( class_invariant() );
        assert( r.class_invariant() );

        auto RE *const lft( new RE );

        // Make a copy of *this.
        shallow_copy( lft );
        auto RE *const rt( new RE( r ) );
        op = OR;
        left = lft;
        right = rt;

        assert( class_invariant() );
        return( *this );
}

Reg<RE>& Reg<RE>::concat( const Reg<RE>& r ) {
        assert( class_invariant() );
        assert( r.class_invariant() );

        auto RE *const lft( new RE );
        shallow_copy( lft );
        auto RE *const rt( new RE( r ) );
        op = CONCAT;
        left = lft;
        right = rt;

        assert( class_invariant() );
        return( *this );
}

Reg<RE>& Reg<RE>::star() {
        assert( class_invariant() );

        auto RE *const d( new RE );
        shallow_copy( d );
        op = STAR;
        left = d;
        right = 0;

        assert( class_invariant() );
        return( *this );
```

```
}

Reg<RE>& Reg<RE>::plus() {
        assert( class_invariant() );

        auto RE *const d( new RE );
        shallow_copy( d );
        op = PLUS;
        left = d;
        right = 0;

        assert( class_invariant() );
        return( *this );
}

Reg<RE>& Reg<RE>::question() {
        assert( class_invariant() );

        auto RE *const d( new RE );
        shallow_copy( d );
        op = QUESTION;
        left = d;
        right = 0;

        assert( class_invariant() );
        return( *this );
}
```

□

Part III
Automata and their components

9 Abstract finite automata

Implementation class: *FAabs*

Files: `faabs.h`

Uses: *DFA* (mentioned)

Description: *FAabs* is an abstract class, used to provide a common interface to the different types of finite automata (which are *FA*, *DFA*, *RFA*, *LBFA*, and *RBFA*). The following operations are provided via member functions:

1. determine how many states there are in the concrete finite automaton;
2. restart the finite automaton in its start states;
3. advance to a new set of states (make a transition) on a character;
4. determine if the automaton is in accepting states;
5. determine if the automaton is stuck (unable to make further transitions);
6. determine if a string is in the language of the automaton, and
7. return a deterministic finite automaton (a *DFA*) which accepts the same language.

By convention, the argumentless constructor of a class inheriting from *FAabs* must construct an object of that class that accepts the empty language (i.e. the object does not accept anything).

Implementation: Since it is an abstract base class (pure virtual), all except one of the member functions are pure virtual; the *acceptable* member function is defined.

```
/* (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
```

□

10 Deterministic *FAs*

Implementation class: *DFA*

Files: `dfa.h`, `dfa.cpp`

Uses: *DFA_components, DSDFARev, DTransRel, FAabs, State, StateSet, StateEqRel, SymRel*

Description: A *DFA* is a deterministic finite automaton (see [Wat93a, Property 2.25] for the definition of *DFA*). It implements the interface required by the abstract class *FAabs*. Additionally, it also provides member functions to reverse the automaton, and to minimize it (the minimization functions are described later). A special constructor is provided, which takes a structure containing the essential components of a *DFA*, and uses those components to construct the automaton; this constructor is used in several *DFA* constructions involving the subset construction (see [Wat93a, p. 12–13] for the subset construction).

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:56:25 $
#ifndef DFA_H
#define DFA_H

#include "faabs.h"
#include "state.h"
#include "stateset.h"
#include "dtransre.h"
#include "dfacomp.h"
#include "st-eqrel.h"
#include "symrel.h"
#include "dsdfarev.h"
#include <assert.h>
#include <iostream.h>

// Deterministic finite automata: can be constructed from non-det. finite
// automata, can be minimized.

class DFA : virtual public FAabs {
public:
	// Constructors, destructors, operator=:

	// Default copy constr, destr, operator= are okay.
	DFA();

	// A special constructor used for subset construction etc.
	DFA( const DFA_components& r );

	// Some member functions:


	// A special one, to reconstruct a DFA, given new components, etc.
	inline DFA& reconstruct( const DFA_components& r );

	// Member functions required by class FAabs:

	virtual int num_states() const;
	virtual void restart();
	virtual void advance( char a );
	virtual int in_final() const;
	virtual int stuck();
	virtual DFA determinism() const;


	// A function to reverse *this.
	DFA& reverse();
```

```
        // Some minimization algorithms:

        inline DFA& min_Brzozowski();
        DFA& min_HopcroftUllman();
        DFA& min_dragon();
        DFA& min_Hopcroft();
        DFA& min_Watson();


        // Some member functions relating to useful State's.

        // Can all States reach a final State?
        int Usefulf() const;

        // Remove any States that cannot reach a final State.
        DFA& usefulf();


        // Special member functions:

        friend ostream& operator<<( ostream& os, const DFA& r );

        inline int class_invariant() const;

protected:
        // Given a minimizing equivalence relation, shrink the DFA.
        DFA& compress( const StateEqRel& r );
        DFA& compress( const SymRel& r );

        // Attempt to split the eq. class [p]_P w.r.t. [q]_P
        // (Return 1 if it was split, 0 otherwise.)
        State split( const State p, const State q, const CharRange a, StateEqRel& P ) const;

        // A helper for min_Watson.
        int are_eq( State p, State q, SymRel& S, const StateEqRel& H, const SymRel& Z ) const;

        // Implementation details:

        StatePool Q;
        // S must be a singleton set, or empty.
        StateSet S;
        StateSet F;
        DTransRel T;

        // Some simulation details:

        State current;
};

// Create a DFA with no State's, accepting nothing.
inline DFA::DFA() :
                Q(),
                S(),
                F(),
                T(),
                current( Invalid ) {
        assert( class_invariant() );
}

// A special constructor used for subset construction etc.
//        A DFA constructed this way will always have a start State.
inline DFA::DFA( const DFA_components& r ) :
                Q( r.Q ),
                S( r.S ),
                T( r.T ),
                F( r.F ) {
        current = Invalid;
```

```
        assert( class_invariant() );
}

inline DFA& DFA::reconstruct( const DFA_components& r ) {
        Q = r.Q;
        S = r.S;
        T = r.T;
        F = r.F;
        current = Invalid;
        assert( class_invariant() );
        return( *this );
}

// See Section 2 (p. 5) of the minimization Taxonomy.
inline DFA& DFA::min_Brzozowski() {
        assert( class_invariant() );
        return( reverse().reverse() );
}

inline int DFA::class_invariant() const {
        return( Q.size() == S.domain()
                && Q.size() == F.domain()
                && Q.size() == T.domain()
                && current < Q.size()
                && S.size() <= 1 );
}

#endif
```

Implementation: All of the member functions are straight-forward implementations, with the exception of the minimization and reversal member functions. Those are further described later (the minimization member functions are discussed in Part IV). The constructor which takes a *DFA_components* object is only provided because most C++ compilers do not support template member functions yet. When these are fully supported, template function *construct_components* will be inlined in the constructor.

Performance: Use-counting the classes used in the automata (such as *DTransRel* and *StateSet*) would improve performance on copy construction and assignment.

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:56:20 $
#include "charrang.h"
#include "crset.h"
#include "stateto.h"
#include "dfa.h"
#include "dfaseed.h"
#include <assert.h>
#include <iostream.h>


int DFA::num_states() const {
        assert( class_invariant() );
        return( Q.size() );
}

void DFA::restart() {
        assert( class_invariant() );
        // There really should be only one state in S.
        //        this is ensured by the class invariant.
        current = S.smallest();
}

void DFA::advance( char a ) {
```

```
        assert( class_invariant() );
        assert( a != Invalid );
        current = T.image( current, a );
}

int DFA::in_final() const {
        assert( class_invariant() );
        return( F.contains( current ) );
}

int DFA::stuck() {
        assert( class_invariant() );
        return( current == Invalid );
}

DFA DFA::determinism() const {
        assert( class_invariant() );
        // Keep it really simple!
        return( *this );
}

DFA& DFA::reverse() {
        // Make sure that *this is structurally sound.
        assert( class_invariant() );
        // Now construct the DFA components from an abstract DFA state.
        reconstruct( construct_components( DSDFARev( F, &T, &S ) ) );
        return( *this );
}

DFA& DFA::compress( const StateEqRel& r ) {
        assert( class_invariant() );

        // All of the components will be constructed into this structure:
        auto DFA_components ret;

        // Give each eq. class of r a State name:
        //         newnames maps States to their new names after compression.
        auto StateTo<State> newnames;
        newnames.set_domain( Q.size() );

        auto State st;
        for( st = 0; st < Q.size(); st++ ) {
                // If st is the representative of its class allocate a new name.
                auto State strep( r.eq_class_representative( st ) );
                if( st == strep ) {
                        newnames.map( st ) = ret.Q.allocate();
                } else {
                        newnames.map( st ) = newnames.lookup( strep );
                }
        }

        // Construct the new transition relation.
        ret.T.set_domain( ret.Q.size() );
        ret.F.set_domain( ret.Q.size() );

        auto CRSet a;

        for( st = 0; st < Q.size(); st++ ) {
                // If st is the representative, construct the transition.
                if( st == r.eq_class_representative( st ) ) {
                        auto State stprime( newnames.lookup( st ) );

                        // What are st's out-transitions?
                        auto CharRange b;

                        a = T.out_labels( st );
                        // The out-labels of any other element of [st]_r could have
```

```
                        // been used instead. Some other choice may, indeed, lead
                        // to a smaller DFA. This approach is used for simplicity.
                        auto int it;
                        // Iterate over the labels, constructing the transitions.
                        for( it = 0; !a.iter_end( it ); it++ ) {
                                b = a.iterator( it );
                                ret.T.add_transition( stprime, b,
                                        newnames.lookup( T.transition_on_range( st, b ) ) );
                        }

                        // st's eq. class may be final.
                        if( F.contains( st ) ) {
                                ret.F.add( stprime );
                        }
                }
        }

        // Set up the new start state.
        ret.S.set_domain( ret.Q.size() );
        ret.S.add( newnames.lookup( S.smallest() ) );

        reconstruct( ret );
        assert( class_invariant() );
        return( *this );
}

DFA& DFA::compress( const SymRel& r ) {
        assert( class_invariant() );

        // All of the components will be constructed into this structure:
        auto DFA_components ret;

        // Give each eq. class of r a State name:
        //      newnames maps States to their new names after compression.
        auto StateTo<State> newnames;
        newnames.set_domain( Q.size() );

        auto StateSet consider;
        consider.set_domain( Q.size() );
        consider.complement();

        // Build the set of representatives.
        auto StateSet repr;
        repr.set_domain( Q.size() );

        auto State st;
        for( consider.iter_start( st ); !consider.iter_end( st ); consider.iter_next( st ) ) {
                // st will always be the representative of its class.
                assert( st == r.image( st ).smallest() );
                repr.add( st );

                // give st the new name
                auto State n( ret.Q.allocate() );
                newnames.map( st ) = n;

                // Go over [st], and give them all the new name.
                auto State z;
                for( r.image( st ).iter_start( z ); !r.image( st ).iter_end( z );
                                r.image( st ).iter_next( z ) ) {
                        newnames.map( z ) = n;
                }

                // Now mark [st] as having been done already.
                consider.remove( r.image( st ) );
                // The outer iterator should still work okay.
        }
```

```
        // Construct the new transition relation.
        ret.T.set_domain( ret.Q.size() );
        ret.F.set_domain( ret.Q.size() );

        auto CRSet a;
        // Go over all of the representatives (eq. classes), constructing the
        // transitions.
        for( repr.iter_start( st ); !repr.iter_end( st ); repr.iter_next( st ) ) {
                auto State stprime( newnames.lookup( st ) );

                // What are st's out-transitions?
                auto CharRange b;

                a = T.out_labels( st );
                // The out-labels of any other element of [st]_r could have
                // been used instead. Some other choice may, indeed, lead
                // to a smaller DFA. This approach is used for simplicity.
                auto int it;
                // Iterate over the labels, constructing the transitions.
                for( it = 0; !a.iter_end( it ); it++ ) {
                        b = a.iterator( it );
                        ret.T.add_transition( stprime, b,
                                newnames.lookup( T.transition_on_range( st, b ) ) );
                }

                // st's eq. class may be final.
                if( F.contains( st ) ) {
                        ret.F.add( stprime );
                }
        }

        // Set up the new start state.
        ret.S.set_domain( ret.Q.size() );
        ret.S.add( newnames.lookup( S.smallest() ) );

        reconstruct( ret );
        assert( class_invariant() );
        return( *this );
}

// Can all of the State's in *this reach a final State?
// Implement Definition 2.23
int DFA::Usefulf() const {
        assert( class_invariant() );
        auto StateSet r( T.reverse_closure( F ) );
        return( r.complement().empty() );
}

// Remove any State's from *this that cannot reach a final State. (This is a last
// step in minimization, since some of the min. algorithms may yield a DFA with a
// sink state.)
// Implement Remark 2.39
DFA& DFA::usefulf() {
        assert( class_invariant() );
        auto StateSet freachable( T.reverse_closure( F ) );

        auto StateTo<State> newnames;
        newnames.set_domain( Q.size() );

        // All components will be constructed into a special structure:
        auto DFA_components ret;

        auto State st;
        for( st = 0; st < Q.size(); st++ ) {
                // If this is a Usefulf State, carry it over by giving it a name
                // in the new DFA.
                if( freachable.contains( st ) ) {
```

```
                newnames.map( st ) = ret.Q.allocate();
            }
        }

        // It is possible that nothing needs to be done (ie. the all States were
        // already F useful).
        if( Q.size() != ret.Q.size() ) {
            ret.T.set_domain( ret.Q.size() );
            ret.F.set_domain( ret.Q.size() );

            auto CRSet a;
            for( st = 0; st < Q.size(); st++ ) {
                // Only construct the transitions if st is final reachable.
                if( freachable.contains( st ) ) {
                    a = T.out_labels( st );
                    auto State stprime( newnames.lookup( st ) );

                    auto CharRange b;
                    auto int it;
                    // Construct the transitions.
                    for( it = 0; !a.iter_end( it ); it++ ) {
                        b = a.iterator( it );
                        ret.T.add_transition( stprime, b,
                            newnames.lookup(
                                T.transition_on_range( st, b ) ) );
                    }
                    // This may be a final State.
                    if( F.contains( st ) ) {
                        ret.F.add( stprime );
                    }
                }
            }

            ret.S.set_domain( ret.Q.size() );

            // Add a start State only if the original one was final reachable.
            if( S.not_disjoint( freachable ) ) {
                ret.S.add( newnames.lookup( S.smallest() ) );
            }
            reconstruct( ret );
        }

        assert( class_invariant() );
        return( *this );
}

ostream& operator<<( ostream& os, const DFA& r ) {
        os << "\nDFA\n";
        os << "Q = " << r.Q << '\n';
        os << "S = " << r.S << '\n';
        os << "F = " << r.F << '\n';
        os << "Transitions = \n" << r.T << '\n';
        os << "current = " << r.current << '\n';
        return( os );
}
```

□

11 Constructing a *DFA*

Implementation class: *DFA_components*

Files: `dfacomp.h`

Uses: *DTransRel, StatePool, StateSet*

Description: A *DFA_components* is a structure containing the essential parts of a *DFA*. Class *DFA* has a constructor taking a *DFA_components*. It is used in the subset construction, as the return type of template function *construct_components*.

Implementation: The class is implemented as a **struct**.

Performance: The template function *construct_components* should really be a constructor template of *DFA*. This would avoid needing *DFA_components*, and the overhead of passing the structure by value.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:56:29 $
#ifndef DFACOMP_H
#define DFACOMP_H

#include "st-pool.h"
#include "stateset.h"
#include "dtransre.h"


struct DFA_components {
        StatePool Q;
        StateSet S;
        DTransRel T;
        StateSet F;
};

#endif
```

□

11.1 Abstract states and the subset construction

An abstract state is a certain type of object used in the construction of a *DFA*. They are essentially states in the subset construction (see [Wat93a, p. 12–13]) with some extra information. The template function *construct_components* uses the extra information to construct a *DFA*. All abstract states in **FIRE engine** are implemented using classes with names beginning with *DS*, such as *DSDFARev*. An abstract state must have the following member functions:

- an argumentless constructor, a copy constructor, an assignment operator, and equality and inequality operators;
- *final* — returns 1 if the abstract state is an accepting (final) state, 0 otherwise;
- *out_labels* — returns a *CRSet* representing the set of labels of out-transitions from the abstract state; and
- *out_transition* — takes a *CharRange* (which is an element of the *CRSet* returned by *out_labels*), and returns the abstract state resulting from a transition on the *CharRange*.

Implementation function: *construct_components*

Files: `dfaseed.h`

Uses: *CharRange, CRSet, DFA_components, State, StateTo*

Description: Template function *construct_components* implements the subset construction, with useless state removal [Wat93a, p. 12–13]. It takes an "abstract state" and constructs the components of a *DFA* (using the abstract state argument as the start state), returning them in a *DFA_components* structure. It does this by encoding the abstract states as *States*, using a breadth-first traversal of the start-reachable abstract states. During the traversal, the deterministic transition relation (a *DTransRel*) is also constructed. As a result of using a traversal starting at the start state, only reachable *States* and the reachable part of the deterministic transition relation are constructed.

Implementation: This template function should really appear as a template constructor of class *DFA*. It is implemented separately, as template member functions have only recently been incorporated into the draft of the C++ language standard. The function implements the subset construction (see [Wat93a, pp. 12, 13]). Each abstract state is assigned a *State*, using a *StateTo* to keep track of the names assigned. The out-transitions from each *State* are constructed, using the *StateTo* to lookup the encoding of an abstract state. The abstract state classes are to be written in such a way that there are only finitely many of them, thereby ensuring termination of the function. The useless state removal is hidden in the fact that, by starting with the start state, only reachable *States* are constructed.

Performance: The performance could be greatly increased by making this template function a template constructor of class *DFA*, avoiding by-value structure passing.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:56:35 $
#ifndef DFASEED_H
#define DFASEED_H

#include "state.h"
#include "stateto.h"
#include "charrang.h"
#include "crset.h"
#include "dfacomp.h"
#include <assert.h>


// A template function, representing an abstract DFA, used in the construction
// of real DFA's.
// It is assumed that class T is a DS??? class (constrained genericity). See
// dsfa.h or dsdfarev.h for examples of a DS??? class.

// Takes a T (an abstract start State), and constructs a DFA.
template<class T>
DFA_components construct_components( T& abs_start ) {

        auto DFA_components ret;

        // Map each of the State's to a unique T, and vice-versa.
        auto StateTo<T> names;

        // Allocate a name for the new start State, and insert it into the namer.
        auto State s( ret.Q.allocate() );

        names.set_domain( ret.Q.size() );
        names.map( s ) = abs_start;

        ret.T.set_domain( ret.Q.size() );
        ret.F.set_domain( ret.Q.size() );
        // As invariant:
        //        ret.T.domain() == ret.Q.size() && ret.F.domain() == ret.Q.size()
        //        and all States < current are already done.
```

```
	auto State current;
	// Now construct the transitions, and finalness of each of the States.
	//		ret.Q grows as we go.
	for( current = 0; current < ret.Q.size(); current++ ) {
		// If the abstract state associated with current says it's final, then
		// current is too.
		if( names.lookup( current ).final() ) {
			ret.F.add( current );
		}

		// Now go through all of the out-transitions of the abstract
		// state associated with current (ie. names.lookup( current )).
		auto CRSet a( names.lookup( current ).out_labels() );

		auto CharRange b;
		auto int it;
		for( it = 0; !a.iter_end( it ); it++ ) {
			b = a.iterator( it );

			// Do something with the destination of the transition.
			auto T dest( names.lookup( current ).out_transition( b ) );

			// See if dest already has a name (by linear search).
			auto State i;
			for( i = 0; i < ret.Q.size() && (names.lookup( i ) != dest); i++ ) {
			}

			// The abstract state may not have a name yet, so we may need
			// to allocate one.
			if( i == ret.Q.size() ) {
				// Associate i with dest.
				// And maintain the invariant.
				auto State j( ret.Q.allocate() );

				assert( i == j );
				names.set_domain( ret.Q.size() );
				ret.T.set_domain( ret.Q.size() );
				ret.F.set_domain( ret.Q.size() );

				names.map( i ) = dest;
			}

			// Now i is the name of dest.
			assert( dest == names.lookup( i ) );
			// Create the actual transition.
			ret.T.add_transition( current, b, i );
		}
		// That's all for current.
	}

	// Time to resynchronize the domain() of ret.S with ret.Q.size():
	ret.S.set_domain( ret.Q.size() );
	// Add the start State now.
	ret.S.add( s );

	// Return, constructing the DFA on the way out.
	return( ret );
}

#endif
```

□

As an example of an abstract state, consider class *DSDFARev*.

Implementation class: *DSDFARev*

Files: `dsdfarev.h`, `dsdfarev.cpp`

Uses: *CharRange, CRSet, DTransRel, StateSet*

Description: Class *DSDFARev* is used in the reversal of a *DFA*. It is an abstract state representing a set of states, in the reversal of the deterministic automaton (the reversal of a deterministic finite automaton is not necessarily deterministic). A *DSDFARev* is constructed in the *DFA* member function *reverse*. Member functions support all of the required interface of an abstract state.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:08 $
#ifndef DSDFAREV_H
#define DSDFAREV_H

#include "charrang.h"
#include "crset.h"
#include "stateset.h"
#include "dtransre.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It is used in the reversal of a DFA.
// Objects of the class represent States in the subset construction of the
// reverse of a DFA.

class DSDFARev {
public:
	// Must always have an argument-less constructor.
	inline DSDFARev();

	inline DSDFARev( const DSDFARev& r );

	// A special constructor:
	inline DSDFARev( const StateSet& rq,
			const DTransRel *rT,
			const StateSet *rS );

	inline const DSDFARev& operator=( const DSDFARev& r );

	// The required member functions:
	inline int final() const;
	CRSet out_labels() const;
	DSDFARev out_transition( const CharRange a ) const;
	inline int operator==( const DSDFARev& r ) const;
	inline int operator!=( const DSDFARev& r ) const;

	friend ostream& operator<<( ostream& os, const DSDFARev& r );

	inline int class_invariant() const;

private:
	StateSet which;
	const DTransRel *T;
	const StateSet *S;
};

// Must always have an argument-less constructor.
inline DSDFARev::DSDFARev() :
		which(),
		T( 0 ),
		S( 0 ) {}
```

```
inline DSDFARev::DSDFARev( const DSDFARev& r ) :
                which( r.which ),
                T( r.T ),
                S( r.S ) {
        assert( class_invariant() );
}

// A special constructor:
inline DSDFARev::DSDFARev( const StateSet& rq,
                const DTransRel *rT,
                const StateSet *rS ) :
                        which( rq ),
                        T( rT ),
                        S( rS ) {
        assert( class_invariant() );
}

inline const DSDFARev& DSDFARev::operator=( const DSDFARev& r ) {
        assert( r.class_invariant() );
        which = r.which;
        T = r.T;
        S = r.S;
        assert( class_invariant() );
        return( *this );
}

inline int DSDFARev::final() const {
        assert( class_invariant() );
        return( which.not_disjoint( *S ) );
}

inline int DSDFARev::operator==( const DSDFARev& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        assert( T == r.T && S == r.S );
        return( which == r.which );
}

inline int DSDFARev::operator!=( const DSDFARev& r ) const {
        return( !operator==( r ) );
}

inline int DSDFARev::class_invariant() const {
        return( T != 0
                && S != 0
                && which.domain() == T->domain()
                && which.domain() == S->domain() );
}

#endif
```

Implementation: The set of *States* in the reversed *DFA* is represented as a *StateSet*. A *DSDFARev* maintains pointers to the components of the *DFA* from which it was created. Two *DSDFARevs* are equal if their *StateSets* are equal.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:02 $
#include "dsdfarev.h"

CRSet DSDFARev::out_labels() const {
        assert( class_invariant() );
        auto CRSet a;

        // Go through all of the States to see which have transitions
        // into a State represented by this->which.
```

```
	auto State i;
	for( i = 0; i < which.domain(); i++ ) {
		a.combine( T->labels_between( i, which ) );
	}
	return( a );
}

DSDFARev DSDFARev::out_transition( const CharRange a ) const {
	assert( class_invariant() );

	auto StateSet result;
	// Set the correct domain.
	result.set_domain( which.domain() );

	// Find the reverse transitions by going through all State's.
	auto State i;
	for( i = 0; i < which.domain(); i++ ) {
		auto State j( T->transition_on_range( i, a ) );
		// We know that there's a transition from i to j on a.
		if( which.contains( j ) ) {
			// Then we want to add the reverse transition.
			result.add( i );
		}
	}

	assert( class_invariant() );
	// Construct the abstract State on the way out of here:
	return( DSDFARev( result, T, S ) );
}

ostream& operator<<( ostream& os, const DSDFARev& r ) {
	os << "\nDSDFARev\n" << r.which << *r.T << *r.S << '\n';
	return( os );
}
```

□

11.2 Derivatives of regular expressions

Implementation class: *DSRE*

Files: `dsre.h`, `dsre.cpp`

Uses: *CharRange*, *CRSet*, *RE*

Description: A *DSRE* represents a derivative of a regular expression. A derivative of a regular expression is again a regular expression; a *DSRE* simply provides an abstract state interface to the derivatives.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:44 $
#ifndef DSRE_H
#define DSRE_H

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


// This class is used to represent abstract States in a DFA that is still under
// construction. It is used in the construction of a DFA from an RE (using
// Brzozowski's construction). Most member functions are calls through to the
```

```
// corresponding RE members.
// DSRE inherits from RE for implementation.

class DSRE : protected RE {
public:
        // Must always have an argument-less constructor.
        inline DSRE();

        // A special constructor:
        inline DSRE( const RE& r );

        inline const DSRE& operator=( const DSRE& r );

        // The required member functions:
        inline int final() const;
        inline CRSet out_labels() const;
        inline DSRE out_transition( const CharRange a ) const;
        inline int operator==( const DSRE& r ) const;
        inline int operator!=( const DSRE& r ) const;


        friend ostream& operator<<( ostream& os, const DSRE& r );

        inline int class_invariant() const;
};

// Must always have an argument-less constructor.
inline DSRE::DSRE() :
                RE() {
        assert( class_invariant() );
}

inline DSRE::DSRE( const RE& r ) :
                RE( r ) {
        assert( r.class_invariant() );
        // They must always be in similarity normal form SNF (see deriv.cpp)
        RE::snf();
        assert( class_invariant() );
}

inline const DSRE& DSRE::operator=( const DSRE& r ) {
        assert( r.class_invariant() );
        RE::operator=( (const RE&)r );
        assert( class_invariant() );
        return( *this );
}

inline int DSRE::final() const {
        assert( class_invariant() );
        return( RE::Null() );
}

inline CRSet DSRE::out_labels() const {
        assert( class_invariant() );
        return( RE::First() );
}

inline DSRE DSRE::out_transition( const CharRange a ) const {
        assert( class_invariant() );
        return( RE::derivative( a ) );
}

inline int DSRE::operator==( const DSRE& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( RE::operator==( r ) );
}
```

```
inline int DSRE::operator!=( const DSRE& r ) const {
        return( !operator==( r ) );
}

inline int DSRE::class_invariant() const {
        return( RE::class_invariant() && RE::in_snf() );
}

#endif
```

Implementation: A *DSRE* inherits from *RE*. The interface required of abstract states (such as member functions *final*, *out_labels*, and *out_transition*) is provided through the use of the *RE* member functions such as *Null*, *First*, and *derivative* (respectively). When a new *DSRE* is constructed, it is put into similarity normal form, to ensure that there will only be finitely many *DSREs* (see [Wat93a, Theorem 5.31]).

Performance: *RE* member function *derivative* returns an *RE* by value, which is then passed to the *DSRE* constructor. Use-counting *RE* could eliminate the overhead of copy constructor and destructor calls.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:43 $
#include "dsre.h"

ostream& operator<<( ostream& os, const DSRE& r ) {
        assert( r.class_invariant() );
        return( os << "\nDSRE\n" << (const RE&)r );
}
```

□

Implementation class: *DSREopt*

Files: `dsre-opt.h`, `dsre-opt.cpp`

Uses: *CharRange, CRSet, RE*

Description: This class is identical to *DSRE*, with one addition: the underlying regular expression (the derivative) is reduced — redundant subexpressions are eliminated. This makes it more likely that two derivatives (denoting the same language) will be found to be equal (using the *DSREopt* equality operator) in template function *construct_components*, thus giving a *DFA* with fewer *States*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:41 $
#ifndef DSREOPT_H
#define DSREOPT_H

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


// This class is used to represent abstract States in a DFA that is still under
// construction. It is used in the construction of a DFA from an RE (using
// Brzozowski's optimized construction). It differs from DSRE in that the derivatives
// are optimized (redundant information is removed – see deriv.cpp)
```

```
// Most member functions are calls through to the corresponding RE members.
// DSREopt inherits from RE for implementation.

class DSREopt : protected RE {
public:
        // Must always have an argument-less constructor.
        inline DSREopt();

        // A special constructor:
        inline DSREopt( const RE& r );

        inline const DSREopt& operator=( const DSREopt& r );

        // The required member functions:
        inline int final() const;
        inline CRSet out_labels() const;
        inline DSREopt out_transition( const CharRange a ) const;
        inline int operator==( const DSREopt& r ) const;
        inline int operator!=( const DSREopt& r ) const;


        friend ostream& operator<<( ostream& os, const DSREopt& r );

        inline int class_invariant() const;
};

// Must always have an argument-less constructor.
inline DSREopt::DSREopt() :
                RE() {
        assert( class_invariant() );
}

inline DSREopt::DSREopt( const RE& r ) :
                RE( r ) {
        assert( r.class_invariant() );
        // They must always be in similarity normal form SNF (see deriv.cpp)
        RE::snf();
        // They must also be reduced.
        RE::reduce();
        assert( class_invariant() );
}

inline const DSREopt& DSREopt::operator=( const DSREopt& r ) {
        assert( r.class_invariant() );
        RE::operator=( (const RE&)r );
        assert( class_invariant() );
        return( *this );
}

inline int DSREopt::final() const {
        assert( class_invariant() );
        return( RE::Null() );
}

inline CRSet DSREopt::out_labels() const {
        assert( class_invariant() );
        return( RE::First() );
}

inline DSREopt DSREopt::out_transition( const CharRange a ) const {
        assert( class_invariant() );
        return( RE::derivative( a ) );
}

inline int DSREopt::operator==( const DSREopt& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
```

```
        return( RE::operator==( r ) );
}

inline int DSREopt::operator!=( const DSREopt& r ) const {
        return( !operator==( r ) );
}

inline int DSREopt::class_invariant() const {
        return( RE::class_invariant() && RE::in_snf() );
}

#endif
```

Implementation: Redundant information is eliminated by calling *RE* member function *reduce* in the constructor from an *RE*.

Performance: See *DSRE*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:39 $
#include "dsre-opt.h"

ostream& operator<<( ostream& os, const DSREopt& r ) {
        assert( r.class_invariant() );
        return( os << "\nDSRE\n" << (const RE&)r );
}
```

□

11.3 Item sets

Implementation class: *ISImpl*

Files: `is-impl.h`, `is-impl.cpp`

Uses: *BitVec, CharRange, CRSet, RE*

Description: *ISImpl* is a base class implementing an item set [Wat93a, Definition 5.58]. It is used as a base class in abstract state classes *DSIS*, *DSDeRemer*, and *DSIS_opt*. It includes a constructor from a pointer to *RE* (the regular expression associated with the item set). The pointed-to *RE* is assumed to remain in existence while the *DFA* components are constructed by template function *construct_components*. Additionally, the abstract state members *final* and *out_labels* are defined. The member function *move_dots* takes a *CharRange* and provides the functionality of "derivatives of item sets" (as defined in [Wat93a, Definition 5.61]), while member function *D* implements the dot closure relation $\mathcal{D}$ defined in [Wat93a, Definition 5.63].

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:50 $
#ifndef ISIMPL_H
#define ISIMPL_H

#include "charrang.h"
#include "crset.h"
#include "re.h"
#include "bitvec.h"
#include <assert.h>
#include <iostream.h>
```

```
// This class is used to represent abstract States in a DFA that is still under
// construction. It is used as the base class in representing item sets, for the
// item set construction, DeRemer's construction, and the optimized item set
// construction. This implements Section 5.5 of the Taxonomy.

class ISImpl {
protected:
        // Must always have an argument-less constructor.
        inline ISImpl();

        // A special constructor:
        ISImpl( const RE *r );

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

        // The required member functions:
        inline int final() const;
        CRSet out_labels() const;

        inline int operator==( const ISImpl& r ) const;
        inline int operator!=( const ISImpl& r ) const;

        // Move the dots across sym nodes.
        inline void move_dots( const CharRange& r );


        friend ostream& operator<<( ostream& os, const ISImpl& r );

        inline int class_invariant() const;

        // Some implementation details:
        // Which RE is this with-respect-to:
        const RE *e;

        // Where are the dots:
        //        The indices in the BitVec's are the pre-order traversal orders of
        //        the nodes in *e.
        //        before indicates the nodes with a dot before them;
        //        after indicates the nodes with a dot after them.
        BitVec before;
        BitVec after;

private:
        // Some helpers:

        // Move the dots as far as possible.
        //        This is named according to Definition 5.63 of the Taxonomy.
        void D( const RE& r, int& node_num );

        // Fetch the labels throughout the tree.
        // Helper to out_labels().
        CRSet traverse_labels( const RE& r, int& node_num ) const;

        // Move the dots throughout the RE tree:
        // Helper to move_dots().
        void ISImpl::traverse_move_dots( const RE& r, const CharRange& a, int& node_num );
};

// Must always have an argument-less constructor.
//        The resulting ISImpl won't satisfy the class_invariant()
inline ISImpl::ISImpl() :
                e( 0 ) {
}

inline int ISImpl::final() const {
```

```
	assert( class_invariant() );
	// It will be final if there is a dot after the root (which is node 0
	// in the traversal order).
	// See construction 5.69 of the Taxonomy.
	return( after.contains( 0 ) );
}

inline CRSet ISImpl::out_labels() const {
	assert( class_invariant() );
	// Start at the root, node 0.
	auto int node( 0 );
	return( traverse_labels( *e, node ) );
}

inline int ISImpl::operator==( const ISImpl& r ) const {
	assert( class_invariant() );
	assert( r.class_invariant() );
	// *this and r must be relative to the same RE.
	assert( e == r.e );
	// All dots must be in the same places.
	return( (before == r.before) && (after == r.after) );
}

inline int ISImpl::operator!=( const ISImpl& r ) const {
	return( !operator==( r ) );
}

// Move the dots across sym nodes.
inline void ISImpl::move_dots( const CharRange& r ) {
	assert( class_invariant() );
	// Traverse the RE tree.
	// i maintains the node number.
	auto int i( 0 );
	// Start without dots after the nodes.
	after.clear();
	traverse_move_dots( *e, r, i );
	assert( i == e->num_operators() );

	// Now go back to the root, and do the closure.
	i = 0;
	D( *e, i );
	assert( i == e->num_operators() );
}

inline int ISImpl::class_invariant() const {
	return( e->class_invariant()
		&& before.width() == e->num_operators()
		&& after.width() == before.width() );
}

#endif
```

Implementation: As explained in [Wat93a, Section 5.5], a dot (an item in an item set) can occur either before or after a regular expression, or any of its subexpressions. The subexpressions of the associated regular expression (including the expression itself) are encoded by their node number (starting at 0) in a prefix traversal of the *RE* expression tree. The dots before subexpressions are stored as bits in a *BitVec* called *before*, while the dots after subexpressions are store as bits in a *BitVec*, called *after* (in both cases, the bits indicate if the corresponding dot is present). Two *ISImpls* are equal if these two *BitVecs* are equal, i.e. their dots before and after are equal. The member functions *move_dots* and *D* both make prefix traversals of the *RE* expression tree, keeping track of the prefix number of each subexpression.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
```

```
// $Date: 1994/05/02 15:58:47 $
#include "is-impl.h"

ISImpl::ISImpl( const RE *r ) :
        e( r ) {
    assert( r->class_invariant() );

    auto int i( r->num_operators() );
    before.set_width( i );
    after.set_width( i );
    assert( !(before.something_set() || after.something_set()) );

    // Now put a dot before the root of *e.
    before.set_bit( 0 );

    // Now propagate the dots.
    auto int nodes( 0 );
    D( *r, nodes );
    assert( nodes == r->num_operators() );
    assert( class_invariant() );
}

const ISImpl& ISImpl::operator=( const ISImpl& r ) {
    assert( r.class_invariant() );
    e = r.e;
    before = r.before;
    after = r.after;
    assert( class_invariant() );
    return( *this );
}

// Move the dots throughout the RE tree:
// Only dots in front a SYMBOL node, matching CharRange a, can advance.
void ISImpl::traverse_move_dots( const RE& r, const CharRange& a, int& node_num ) {
    assert( r.class_invariant() );

    // node_num is the preorder number of the associated regular expression.
    switch( r.root_operator() ) {
    case EPSILON:
    case EMPTY:
        // Remove any dot that may be before.
        before.unset_bit( node_num );
        node_num++;
        break;
    case SYMBOL:
        // if there's a match, advance the bit.
        if( (a <= r.symbol()) && before.contains( node_num ) ) {
            after.set_bit( node_num );
        }
        before.unset_bit( node_num );
        node_num++;
        break;
    case OR:
    case CONCAT:
        before.unset_bit( node_num );
        node_num++;
        traverse_move_dots( r.left_subexpr(), a, node_num );
        traverse_move_dots( r.right_subexpr(), a, node_num );
        break;
    case STAR:
    case PLUS:
    case QUESTION:
        before.unset_bit( node_num );
        node_num++;
        traverse_move_dots( r.left_subexpr(), a, node_num );
        break;
    }
```

```
}

// Move the dots as far as possible.
//        This is named according to Definition 5.63 of the Taxonomy.
// node_num is the preorder number of r (in the associated RE).
void ISImpl::D( const RE& r, int& node_num ) {
        // Need to remember if there was a dot in front of this node, and what the preorder
        // number of this node is.
        auto int b;
        auto int nn;
        auto int node_right;
        auto int sub_parity;

        switch( r.root_operator() ) {
        case EPSILON:
                // The dot always moves through an EPSILON.
                if( before.contains( node_num ) ) {
                        after.set_bit( node_num );
                }
                node_num++;
                break;
        case EMPTY:
        case SYMBOL:
                // The dot never moves through EMPTY or SYMBOL.
                // Just advance to the next preorder number.
                node_num++;
                break;
        case OR:
                // Is there one before this node?
                b = before.contains( node_num );
                nn = node_num++;
                // If b, the propagate the dot inside the (left of the) OR too.
                if( b ) {
                        // node_num is now the node num of the left subexpr.
                        before.set_bit( node_num );
                }
                // Propagate through the left subexpression.
                D( r.left_subexpr(), node_num );
                // If the dot is after the left subexpr, then propagate out of the OR.
                if( after.contains( nn + 1 ) ) {
                        after.set_bit( nn );
                }

                // If b, propagate the dot to the right of the OR.
                if( b ) {
                        before.set_bit( node_num );
                }
                // Save the node number of the right subexpr.
                node_right = node_num;
                // Propagate through the right subexpr.
                D( r.right_subexpr(), node_num );
                // If the dot is after the right, then propagate it out of the OR.
                if( after.contains( node_right ) ) {
                after.set_bit( nn );
                }
                break;
        case CONCAT:
                nn = node_num++;
                // If there's a dot before the CONCAT, propagate it into the left subexpr.
                if( before.contains( nn ) ) {
                        before.set_bit( node_num );
                }
                // Propagate through the left subexpr.
                D( r.left_subexpr(), node_num );
                // If the dot is after the left subexpr, move it before the right one.
                if( after.contains( nn + 1 ) ) {
                        before.set_bit( node_num );
```

```
				}

				// Save the node number of the right subexpr.
				node_right = node_num;

				// Propagate through the right.
				D( r.right_subexpr(), node_num );
				// If the dot is after the right subexpr, move it behind the CONCAT.
				if( after.contains( node_right ) ) {
					after.set_bit( nn );
				}
				break;
		case STAR:
				// A dot before the expression means that it moves after too.
				if( before.contains( node_num ) ) {
					after.set_bit( node_num );
				}
				// Fall through here.
		case PLUS:
				// The fall through from the STAR is correct!
				nn = node_num++;
				// If there's a dot before the STAR/PLUS and into the subexpr.
				if( before.contains( nn ) ) {
					before.set_bit( node_num );
				}
				// Remember if there's already a dot before the subexpr.
				sub_parity = before.contains( node_num );

				// Propagate through the subexpr.
				D( r.left_subexpr(), node_num );

				// Check if a dot is after the subexpr.
				if( after.contains( nn + 1 ) ) {
					// Move the dot out.
					after.set_bit( nn );
					// and move the bit around before the subexpr.
					// if it wasn't already there before.
					if( !sub_parity ) {
						// Re-propagate the dots.
						//         nn + 1 is the node num of the subexpr.
						before.set_bit( ++nn );
						D( r.left_subexpr(), nn );
					}
				}
				break;
		case QUESTION:
				nn = node_num++;
				// If there's a dot before the QUESTION, move it after, and into the subexpr.
				if( before.contains( nn ) ) {
					after.set_bit( nn );
					before.set_bit( node_num );
				}
				// Propagate through the subexpr.
				D( r.left_subexpr(), node_num );

				// If there's a dot after the subexpr, propagate it out.
				if( after.contains( nn + 1 ) ) {
					after.set_bit( nn );
				}
				break;
		}
}

// Fetch the labels throughout the tree.
CRSet ISImpl::traverse_labels( const RE& r, int& node_num ) const {
		// assume that the root of r is node number node_num.
		assert( class_invariant() );
```

```
        assert( r.class_invariant() );

        auto CRSet b;
        switch( r.root_operator() ) {
        case EPSILON:
        case EMPTY:
                node_num++;
                break;
        case SYMBOL:
                // Check if this node has a dot before it.
                if( before.contains( node_num ) ) {
                        b.append( r.symbol() );
                }
                node_num++;
                break;
        case OR:
        case CONCAT:
                node_num++;
                b = traverse_labels( r.left_subexpr(), node_num );
                b.combine( traverse_labels( r.right_subexpr(), node_num ) );
                break;
        case STAR:
        case PLUS:
        case QUESTION:
                node_num++;
                b = traverse_labels( r.left_subexpr(), node_num );
        }
        return( b );
}

ostream& operator<<( ostream& os, const ISImpl& r ) {
        assert( r.class_invariant() );
        return( os << "\nISImpl\n" << *(r.e) << "\nbefore: " << r.before
                << "\nafter: " << r.after << '\n' );
}
```

□

Implementation class: *DSIS*

Files: `dsis.h`, `dsis.cpp`

Uses: *CharRange, CRSet, ISImpl, RE*

Description: Class *DSIS* provides the full abstract state interface of an item set (see class *ISImpl*). *DSIS* inherits from *ISImpl* for implementation. It only adds the *out_transition* member function. It is used in the item set construction [Wat93a, Construction 5.69].

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:31 $
#ifndef DSIS_H
#define DSIS_H

#include "charrang.h"
#include "crset.h"
#include "re.h"
#include "is-impl.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It represents the item sets in the Iconstr construction.
// See Construction 5.69
// DSIS inherits from ISImpl for implementation.
```

```
class DSIS : protected ISImpl {
public:
        // Must always have an argument-less constructor.
        inline DSIS();

        // A special constructor:
        inline DSIS( const RE *r );

        inline const DSIS& operator=( const DSIS& r );

        // The required member functions:
        inline int final() const;
        inline CRSet out_labels() const;
        DSIS out_transition( const CharRange a ) const;
        inline int operator==( const DSIS& r ) const;
        inline int operator!=( const DSIS& r ) const;


        friend ostream& operator<<( ostream& os, const DSIS& r );

        inline int class_invariant() const;
};

// The default constructor won't leave *this in a condition satisfying the
// class invariant.
inline DSIS::DSIS() :
                ISImpl() {
}

inline DSIS::DSIS( const RE *r ) :
                ISImpl( r ) {
        assert( r->class_invariant() );
        assert( class_invariant() );
}

inline const DSIS& DSIS::operator=( const DSIS& r ) {
        assert( r.class_invariant() );
        ISImpl::operator=( (const ISImpl&)r );
        // *this may not satisfy the invariant until after the assignment.
        assert( class_invariant() );
        return( *this );
}

inline int DSIS::final() const {
        assert( class_invariant() );
        return( ISImpl::final() );
}

inline CRSet DSIS::out_labels() const {
        assert( class_invariant() );
        return( ISImpl::out_labels() );
}

inline int DSIS::operator==( const DSIS& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( ISImpl::operator==( r ) );
}

inline int DSIS::operator!=( const DSIS& r ) const {
        return( !operator==( r ) );
}

inline int DSIS::class_invariant() const {
        // Should really check that before and after are fully closed.
        return( ISImpl::class_invariant() );
```

```
}

#endif
```

Implementation: Most of the implementation is provided by class *ISImpl*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:29 $
#include "dsis.h"

DSIS DSIS::out_transition( const CharRange a ) const {
	assert( class_invariant() );
	auto DSIS ret( *this );
	ret.move_dots( a );
	return( ret );
}

ostream& operator<<( ostream& os, const DSIS& r ) {
	assert( r.class_invariant() );
	return( os << "\nDSIS\n" << (const ISImpl&)r );
}
```

□

Implementation class: *DSDeRemer*

Files: `dsderem.h`, `dsderem.cpp`

Uses: *CharRange, CRSet, ISImpl, RE*

Description: Class *DSDeRemer* provides the abstract state interface of an item set (see class *ISImpl*). *DSDeRemer* inherits from *ISImpl* for implementation. It is used in DeRemer's construction [Wat93a, Construction 5.75]. DeRemer's construction can yield smaller *DFAs* than the item set construction.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:56:54 $
#ifndef DSDEREMER_H
#define DSDEREMER_H

#include "charrang.h"
#include "crset.h"
#include "re.h"
#include "is-impl.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It represents the item sets in the DeRemer construction.
// See Construction 5.75
// DSDeRemer inherits from ISImpl for implementation.

class DSDeRemer : protected ISImpl {
public:
	// Must always have an argument-less constructor.
	inline DSDeRemer();

	// A special constructor:
	inline DSDeRemer( const RE *r );
```

```
        inline const DSDeRemer& operator=( const DSDeRemer& r );

        // The required member functions:
        inline int final() const;
        inline CRSet out_labels() const;
        DSDeRemer out_transition( const CharRange a ) const;
        inline int operator==( const DSDeRemer& r ) const;
        inline int operator!=( const DSDeRemer& r ) const;


        friend ostream& operator<<( ostream& os, const DSDeRemer& r );

        inline int class_invariant() const;

private:
        // Helpers: the optimization function of the DeRemer construction.
        inline void opt();
        void traverse_opt( const RE& r, int& i );
};

inline DSDeRemer::DSDeRemer() :
                ISImpl() {
}

inline DSDeRemer::DSDeRemer( const RE *r ) :
                ISImpl( r ) {
        assert( r->class_invariant() );
        // The ISImpl constructor will have closed before/after, but not optimized yet.
        opt();
        assert( class_invariant() );
}

inline const DSDeRemer& DSDeRemer::operator=( const DSDeRemer& r ) {
        assert( r.class_invariant() );
        ISImpl::operator=( (const ISImpl&)r );
        assert( class_invariant() );
        return( *this );
}

inline int DSDeRemer::final() const {
        assert( class_invariant() );
        return( ISImpl::final() );
}

inline CRSet DSDeRemer::out_labels() const {
        assert( class_invariant() );
        return( ISImpl::out_labels() );
}

inline int DSDeRemer::operator==( const DSDeRemer& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        return( ISImpl::operator==( r ) );
}

inline int DSDeRemer::operator!=( const DSDeRemer& r ) const {
        return( !operator==( r ) );
}

inline void DSDeRemer::opt() {
        auto int i( 0 );
        traverse_opt( *e, i );
        assert( i == (e->num_operators()) );
}

inline int DSDeRemer::class_invariant() const {
        // Should really check that before and after are fully closed,
```

```
        // and that they are optimized.
        return( ISImpl::class_invariant() );
}

#endif
```

Implementation: The implementation is largely provided by the base class *ISImpl*. The member function *opt* (with helper *traverse_opt*) implements DeRemer's dot filter function $\mathcal{X}$ as defined in [Wat93a, Definition 5.72]. The filter function makes a single prefix traversal of the *RE* expression tree. By applying the filter, it becomes more likely that template function *construct_components* will find two item sets (denoting the same language) to be equal (this will result in fewer states in the constructed *DFA*).

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:56:46 $
#include "dsderem.h"

DSDeRemer DSDeRemer::out_transition( const CharRange a ) const {
        assert( class_invariant() );
        auto DSDeRemer ret( *this );
        ret.move_dots( a );
        ret.opt();
        return( ret );
}

void DSDeRemer::traverse_opt( const RE& r, int& node_num ) {
        assert( class_invariant() );
        assert( r.class_invariant() );
        // Recursively traverse r and apply the optimizations of Defn. 5.72
        // (function X) of the Taxonomy.
        switch( r.root_operator() ) {
        case EPSILON:
        case EMPTY:
        case SYMBOL:
                node_num++;
                break;
        case OR:
                before.unset_bit( node_num );
                // Fall through is good.
        case CONCAT:
                node_num++;
                traverse_opt( r.left_subexpr(), node_num );
                traverse_opt( r.right_subexpr(), node_num );
                break;
        case STAR:
                before.unset_bit( node_num );
                node_num++;
                after.unset_bit( node_num );
                traverse_opt( r.left_subexpr(), node_num );
                break;
        case PLUS:
        case QUESTION:
                node_num++;
                traverse_opt( r.left_subexpr(), node_num );
                break;
        }
}

ostream& operator<<( ostream& os, const DSDeRemer& r ) {
        assert( r.class_invariant() );
        return( os << "\nDSDeRemer\n" << (const ISImpl&)r );
}
```

□

Implementation class: *DSISopt*

Files: `dsis-opt.h`, `dsis-opt.cpp`

Uses: *CharRange, CRSet, ISImpl, RE*

Description: Class *DSIS_opt* provides the abstract state interface of an item set (see class *ISImpl*). *DSIS_opt* inherits from *ISImpl* for implementation. It is used in the optimized item set construction [Wat93a, Construction 5.82]. The optimized item set construction (called *Oconstr* in [Wat93a]) can yield much smaller *DFAs* than either the normal item set construction or DeRemer's construction.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:27 $
#ifndef DSIS_OPT_H
#define DSIS_OPT_H

#include "charrang.h"
#include "crset.h"
#include "re.h"
#include "is-impl.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It represents the item sets in the DeRemer construction.
// See Construction 5.75
// DSIS_opt inherits from ISImpl for implementation.

class DSIS_opt : protected ISImpl {
public:
	// Must always have an argument-less constructor.
	inline DSIS_opt();

	// A special constructor:
	inline DSIS_opt( const RE *r );

	inline const DSIS_opt& operator=( const DSIS_opt& r );

	// The required member functions:
	inline int final() const;
	inline CRSet out_labels() const;
	DSIS_opt out_transition( const CharRange a ) const;
	inline int operator==( const DSIS_opt& r ) const;
	inline int operator!=( const DSIS_opt& r ) const;


	friend ostream& operator<<( ostream& os, const DSIS_opt& r );

	inline int class_invariant() const;

private:
	// Helpers: the optimization function of the DeRemer construction.
	inline void opt();
	void traverse_opt( const RE& r, int& i );
};

inline DSIS_opt::DSIS_opt() :
		ISImpl() {
}

inline DSIS_opt::DSIS_opt( const RE *r ) :
```

```
        ISImpl( r ) {
    assert( r->class_invariant() );
    // The ISImpl constructor will close before/after, but not optimize.
    opt();
    assert( class_invariant() );
}

inline const DSIS_opt& DSIS_opt::operator=( const DSIS_opt& r ) {
    assert( r.class_invariant() );
    ISImpl::operator=( (const ISImpl&)r );
    assert( class_invariant() );
    return( *this );
}

inline int DSIS_opt::final() const {
    assert( class_invariant() );
    return( ISImpl::final() );
}

inline CRSet DSIS_opt::out_labels() const {
    assert( class_invariant() );
    return( ISImpl::out_labels() );
}

inline int DSIS_opt::operator==( const DSIS_opt& r ) const {
    assert( class_invariant() );
    assert( r.class_invariant() );
    return( ISImpl::operator==( r ) );
}

inline int DSIS_opt::operator!=( const DSIS_opt& r ) const {
    return( !operator==( r ) );
}

inline void DSIS_opt::opt() {
    // Remember if there's a dot after the root.
    auto int par( after.contains( 0 ) );
    after.clear();
    auto int i( 0 );
    traverse_opt( *e, i );
    assert( i == (e->num_operators()) );
    // The dot after the root may have been removed, so it can be put back.
    if( par ) {
        after.set_bit( 0 );
    }
}

inline int DSIS_opt::class_invariant() const {
    // Should really check that before and after are fully closed,
    // and that they are optimized.
    return( ISImpl::class_invariant() );
}

#endif
```

Implementation: The implementation is largely provided by the base class *ISImpl*. The member function *opt* (with helper *traverse_opt*) implements the optimized dot filter function $\mathcal{Y}$ as defined in [Wat93a, Definition 5.79]. The filter function makes a single prefix traversal of the *RE* expression tree. By applying the filter, it becomes more likely that template function *construct_components* will find two item sets (denoting the same language) to be equal. The reason why the optimized item set construction yields smaller *DFAs* than DeRemer's construction is that the $\mathcal{Y}$ filter is more effective at removing redundant information than the $\mathcal{X}$ filter used in DeRemer's construction (see [Wat93a, Remark 5.81]).

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:25 $
#include "dsis-opt.h"

DSIS_opt DSIS_opt::out_transition( const CharRange a ) const {
	assert( class_invariant() );
	auto DSIS_opt ret( *this );
	ret.move_dots( a );
	ret.opt();
	return( ret );
}

// Implement Definition 5.79
void DSIS_opt::traverse_opt( const RE& r, int& node_num ) {
	assert( class_invariant() );
	assert( r.class_invariant() );

	// Recursively traverse r and apply the optimizations of Defn. 5.79
	// (function Y) of the Taxonomy.
	// After will already have been cleared by DSIS_opt::opt()
	switch( r.root_operator() ) {
	case EPSILON:
	case EMPTY:
		before.unset_bit( node_num );
		node_num++;
		break;
	case SYMBOL:
		// Don't remove a bit before — see Defintion 5.79
		node_num++;
		break;
	case OR:
	case CONCAT:
		before.unset_bit( node_num );
		node_num++;
		traverse_opt( r.left_subexpr(), node_num );
		traverse_opt( r.right_subexpr(), node_num );
		break;
	case STAR:
	case PLUS:
	case QUESTION:
		before.unset_bit( node_num );
		node_num++;
		traverse_opt( r.left_subexpr(), node_num );
		break;
	}
}

ostream& operator<<( ostream& os, const DSIS_opt& r ) {
	assert( r.class_invariant() );
	return( os << "\nDSIS_opt\n" << (const ISImpl&)r );
}
```

□

12 Finite automata

Implementation class: *FA*

Files: `fa.h`, `fa.cpp`

Uses: *FAabs, RE, State, StatePool, StateRel, StateSet, TransRel*

Description: Class *FA* implements finite automata as defined in [Wat93a, Definition 2.1]. It inherits from *FAabs*, and implements the interface defined there. A constructor taking an *RE* is provided, implementing Thompson's top-down construction [Wat93a, Construction 4.5]. This constructor can be more efficient than using the Σ-algebra *Reg<FA>*.

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

#include "faabs.h"
#include "st-pool.h"
#include "stateset.h"
#include "staterel.h"
#include "transrel.h"
#include "re.h"
#include <assert.h>
#include <iostream.h>

// Implement general finite automata (Definition 2.1 of the Taxonomy).
// These can be constructed by the Sigma-algebra (see sig-fa.cpp) or using
// the special constructor from RE's.

class FA : virtual public FAabs {
public:
        // Constructors, destructors, operator=:
        FA();

        // Provide a copy constructor:
        FA( const FA& r );

        // A special constructor, implementing the Thompson's top-down construction
        // (see Construction 4.5).
        FA( const RE& e );

        // Default operator= is okay.

        // Basic FAabs member functions (without overriding acceptable()):

        virtual int num_states() const;
        virtual void restart();
        virtual void advance( char a );
        virtual int in_final() const;
        virtual int stuck();
        virtual DFA determinism() const;


        // Special member functions:

        friend ostream& operator<<( ostream& os, const FA& r );

        inline int class_invariant() const;

protected:
        // Functions states_reqd, td (see Construction 4.5) for use in the constructor from RE.
        int states_reqd( const RE& e );
        void td( const State s, const RE& e, const State f );
```

```
        // recycle this FA:
        void reincarnate();

        // Implementation details:

        StatePool Q;
        StateSet S, F;

        // Transitions maps each State to its out-transitions.
        TransRel Transitions;
        // E is the epsilon transition relation.
        StateRel E;

        // Simulation stuff:
        StateSet current;
};

inline int FA::class_invariant() const {
        // Should also check that current is E-closed.
        return( Q.size() == S.domain()
                && Q.size() == F.domain()
                && Q.size() == Transitions.domain()
                && Q.size() == E.domain()
                && Q.size() == current.domain()
                && S.class_invariant()
                && F.class_invariant()
                && Transitions.class_invariant()
                && E.class_invariant()
                && current.class_invariant() );
}

#endif
```

Implementation: Most of the member functions are implemented in a straight-forward manner. The constructor from an *RE* makes a traversal of the *RE* to determine the number of *States* to allocate before-hand.

Performance: Since Thompson's construction yields finite automata with certain structural properties (see [Wat93a, end of Definition 4.1]), a special "Thompson's construction" automata class could be efficiently implemented, instead of the general finite automaton structure defined in *FA*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:02 $
#include "fa.h"
#include "dfaseed.h"
#include "dsfa.h"
#include "dfa.h"

// Construct an FA accepting nothing.
FA::FA() : Q() {
        auto State s( Q.allocate() );
        auto State f( Q.allocate() );
        S.set_domain( Q.size() );
        S.add( s );

        F.set_domain( Q.size() );
        F.add( f );

        Transitions.set_domain( Q.size() );
        E.set_domain( Q.size() );
```

```
        current.set_domain( Q.size() );
        assert( class_invariant() );
}

FA::FA( const FA& r ) :
                Q( r.Q ),
                S( r.S ),
                F( r.F ),
                Transitions( r.Transitions ),
                E( r.E ),
                current( r.current ) {
        assert( class_invariant() );
}

FA::FA( const RE& e ) {
        // First, allocate enough states by setting up the domains.
        // See Section 4.1.1 of the Taxonomy.
        auto int states_required( states_reqd( e ) );
        S.set_domain( states_required );
        F.set_domain( states_required );
        Transitions.set_domain( states_required );
        E.set_domain( states_required );
        current.set_domain( states_required );

        auto State s( Q.allocate() );
        auto State f( Q.allocate() );
        S.add( s );
        F.add( f );
        td( s, e, f );
        assert( class_invariant() );
}

// The following follows directly from inspecting Thompson's construction
// (Definition 4.1).
int FA::states_reqd( const RE& e ) {
        assert( e.class_invariant() );

        auto int ret;
        switch( e.root_operator() ) {
        case EPSILON:
        case EMPTY:
        case SYMBOL:
                ret = 2;
                break;
        case OR:
                ret = 2 + states_reqd( e.left_subexpr() ) + states_reqd( e.right_subexpr() );
                break;
        case CONCAT:
                ret = states_reqd( e.left_subexpr() ) + states_reqd( e.right_subexpr() );
                break;
        case STAR:
        case PLUS:
        case QUESTION:
                ret = 2 + states_reqd( e.left_subexpr() );
                break;
        }
        return( ret );
}

int FA::num_states() const {
        return( Q.size() );
}

// The simulation details follow directly from Section 2 of the Taxonomy.
void FA::restart() {
        assert( class_invariant() );
        current = E.closure( S );
```

```
        assert( class_invariant() );
}

void FA::advance( char a ) {
        assert( class_invariant() );
        // Compute the epsilon-closure.
        current = E.closure( Transitions.image( current, a ) );
        assert( current == E.closure( current ) );
        assert( class_invariant() );
}

int FA::in_final() const {
        assert( class_invariant() );
        return( current.not_disjoint( F ) );
}

int FA::stuck() {
        assert( class_invariant() );
        return( current.empty() );
}

DFA FA::determinism() const {
        // Make sure that *this is structurally sound.
        assert( class_invariant() );
        // Now construct the DFA components.
        return( DFA( construct_components( DSFA( S, &Transitions, &E, &F ) ) ) );
}

ostream& operator<<( ostream& os, const FA& r ) {
        assert( r.class_invariant() );
        os << "\nFA\n";
        os << "Q = " << r.Q << '\n';
        os << "S = " << r.S << '\n';
        os << "F = " << r.F << '\n';
        os << "Transitions = \n" << r.Transitions << '\n';
        os << "E = \n" << r.E << '\n';
        os << "current = " << r.current << '\n';
        return( os );
}

void FA::td( const State s, const RE& e, const State f ) {
        assert( e.class_invariant() );
        // Implement function td (of Construction 4.5).
        // Construct an FA to accept the language of e, between s and f.

        switch( e.root_operator() ) {
        case EPSILON:
                E.union_cross( s, f );
                break;
        case EMPTY:
                break;
        case SYMBOL:
                Transitions.add_transition( s, e.symbol(), f );
                break;
        case OR:
                {
                        auto State p( Q.allocate() );
                        auto State q( Q.allocate() );
                        td( p, e.left_subexpr(), q );
                        auto State r( Q.allocate() );
                        auto State t( Q.allocate() );
                        td( r, e.right_subexpr(), t );

                        E.union_cross( s, p );
                        E.union_cross( s, r );

                        E.union_cross( q, f );
```

```
                        E.union_cross( t, f );
                }
                break;
        case CONCAT:
                {
                        auto State p( Q.allocate() );
                        auto State q( Q.allocate() );
                        td( s, e.left_subexpr(), p );
                        td( q, e.right_subexpr(), f );

                        E.union_cross( p, q );
                }
                break;
        case STAR:
                {
                        auto State p( Q.allocate() );
                        auto State q( Q.allocate() );
                        td( p, e.left_subexpr(), q );

                        E.union_cross( s, p );
                        E.union_cross( q, p );
                        E.union_cross( q, f );
                        E.union_cross( s, f );
                }
                break;
        case PLUS:
                {
                        auto State p( Q.allocate() );
                        auto State q( Q.allocate() );
                        td( p, e.left_subexpr(), q );

                        E.union_cross( s, p );
                        E.union_cross( q, p );
                        E.union_cross( q, f );
                }
                break;
        case QUESTION:
                {
                        auto State p( Q.allocate() );
                        auto State q( Q.allocate() );
                        td( p, e.left_subexpr(), q );

                        E.union_cross( s, p );
                        E.union_cross( q, f );
                        E.union_cross( s, f );
                }
                break;
        }
}

void FA::reincarnate() {
        Q.reincarnate();
        S.reincarnate();
        F.reincarnate();
        Transitions.reincarnate();
        E.reincarnate();
        current.reincarnate();
}
```

□

Implementation class: *DSFA*

Files: `dsfa.h`, `dsfa.cpp`

Uses: *CharRange, CRSet, StateRel, StateSet, TransRel*

Description: Class *DSFA* implements the abstract class interface required to construct a *DFA* from an *FA*. A *DSFA* is constructed in the *FA* member function *determinism*, and then passed to template function *construct_components*, which constructs the *DFA* components.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:23 $
#ifndef DSFA_H
#define DSFA_H

#include "charrang.h"
#include "crset.h"
#include "stateset.h"
#include "staterel.h"
#include "transrel.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It is used in the construction of a DFA from an FA. The behaviour
// of a DSFA follows the simulation behaviour of an FA as in the subset construction.

class DSFA {
public:
	// Must always have an argument-less constructor.
	inline DSFA();

	inline DSFA( const DSFA& r );

	// A special constructor:
	DSFA( const StateSet& rq,
			const TransRel *rT,
			const StateRel *rE,
			const StateSet *rF );

	inline const DSFA& operator=( const DSFA& r );

	// The required member functions:
	inline int final() const;
	CRSet out_labels() const;
	DSFA out_transition( const CharRange a ) const;
	inline int operator==( const DSFA& r ) const;
	inline int operator!=( const DSFA& r ) const;


	friend ostream& operator<<( ostream& os, const DSFA& r );

	inline int class_invariant() const;

private:
	StateSet which;
	const TransRel *T;
	const StateRel *E;
	const StateSet *F;
};

// Must always have an argument-less constructor.
inline DSFA::DSFA() :
		which(),
		T( 0 ),
		E( 0 ),
		F( 0 ) {
	// No assertion of the class invariant, since it won't qualify.
}

inline DSFA::DSFA( const DSFA& r ) :
```

```
            which( r.which ),
            T( r.T ),
            E( r.E ),
            F( r.F ) {
      assert( class_invariant() );
}

inline const DSFA& DSFA::operator=( const DSFA& r ) {
      assert( r.class_invariant() );
      // *this may not satisfy its invariant yet.
      which = r.which;
      T = r.T;
      E = r.E;
      F = r.F;
      assert( class_invariant() );
      return( *this );
}

inline int DSFA::final() const {
      assert( class_invariant() );
      return( which.not_disjoint( *F ) );
}

inline int DSFA::operator==( const DSFA& r ) const {
      assert( class_invariant() );
      assert( r.class_invariant() );
      assert( T == r.T && E == r.E && F == r.F );
      return( which == r.which );
}

inline int DSFA::operator!=( const DSFA& r ) const {
      return( !operator==( r ) );
}

inline int DSFA::class_invariant() const {
      return( T != 0
            && E != 0
            && F != 0
            && which.domain() == T->domain()
            && which.domain() == E->domain()
            && which.domain() == F->domain()
            && which == E->closure( which ) );
}

#endif
```

Implementation: A *DSFA* contains a *StateSet*, representing the set of *FA States* that make up a *DFA* state in the subset construction (see [Wat93a, p. 12–13] for an explanation of the subset construction). The implementation follows directly from the way in which an *FA* processes an input string.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:16 $
#include "dsfa.h"

// A special constructor:
DSFA::DSFA( const StateSet& rq,
            const TransRel *rT,
            const StateRel *rE,
            const StateSet *rF ) :
                  which( rE->closure( rq ) ),
                  T( rT ),
                  E( rE ),
                  F( rF ) {
```

```
        assert( class_invariant() );
}

CRSet DSFA::out_labels() const {
        assert( class_invariant() );
        return( T->out_labels( which ) );
}

DSFA DSFA::out_transition( const CharRange a ) const {
        assert( class_invariant() );
        // Construct the abstract State on the way out of here:
        //      Assume that which is already E closed, then transition on CharRange
        //      a; the StateSet gets E closed again in the constructor.
        return( DSFA( T->transition_on_range( which, a ), T, E, F ) );
}

ostream& operator<<( ostream& os, const DSFA& r ) {
        os << "\nDSFA\n" << r.which << *r.T << *r.E << *r.F << '\n';
        return( os );
}
```

□

12.1 The Σ-algebra *Reg<FA>*

Implementation class: *Reg<FA>*

Files: `sig-fa.cpp`

Uses: *CharRange, FA, Reg, State*

Description: The template instantiation *Reg<FA>* implements Thompson's Σ-algebra of finite automata, as defined in [Wat93a, Definition 4.1]. The operators can be used directly to construct complex finite automata, or the *Reg<FA>* constructor from *RE* (the constructor defined in class *Reg*) can be used to construct the homomorphic image of the regular expression (the constructor uses these operators indirectly).

Implementation: The implementation follows directly from [Wat93a, Definition 4.1]. In some places it is complicated by the management of domains of *StateSets*, *StateRels*, and *TransRels*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:30 $
#include "charrang.h"
#include "sigma.h"
#include "fa.h"

// Implement the Sigma-algebra operators (Definition 4.29 of the Taxonomy).
Reg<FA>& Reg<FA>::epsilon() {
        // *this may have been something in a previous life.
        // Wipe out all previous components;
        reincarnate();

        auto State s( Q.allocate() );
        auto State f( Q.allocate() );

        S.set_domain( Q.size() );
        S.add( s );

        F.set_domain( Q.size() );
        F.add( f );

        Transitions.set_domain( Q.size() );
```

```
    E.set_domain( Q.size() );
    E.union_cross( s, f );

    current.set_domain( Q.size() );

    assert( class_invariant() );
    return( *this );
}

Reg<FA>& Reg<FA>::empty() {
    // See epsilon case.
    reincarnate();

    auto State s( Q.allocate() );
    auto State f( Q.allocate() );

    S.set_domain( Q.size() );
    S.add( s );

    F.set_domain( Q.size() );
    F.add( f );

    Transitions.set_domain( Q.size() );

    E.set_domain( Q.size() );

    current.set_domain( Q.size() );

    assert( class_invariant() );
    return( *this );
}

Reg<FA>& Reg<FA>::symbol( const CharRange r ) {
    // See epsilon case.
    reincarnate();

    auto State s( Q.allocate() );
    auto State f( Q.allocate() );

    S.set_domain( Q.size() );
    S.add( s );

    F.set_domain( Q.size() );
    F.add( f );

    Transitions.set_domain( Q.size() );
    Transitions.add_transition( s, r, f );

    E.set_domain( Q.size() );

    current.set_domain( Q.size() );

    assert( class_invariant() );
    return( *this );
}

Reg<FA>& Reg<FA>::or( const Reg<FA>& r ) {
    assert( class_invariant() );
    assert( r.class_invariant() );
    // All state-related stuff in r must be adjusted.

    Q.incorporate( r.Q );
    auto State s( Q.allocate() );
    auto State f( Q.allocate() );

    S.disjointing_union( r.S );
```

```
        S.set_domain( Q.size() );

        F.disjointing_union( r.F );
        F.set_domain( Q.size() );

        Transitions.disjointing_union( r.Transitions );
        Transitions.set_domain( Q.size() );

        E.disjointing_union( r.E );
        E.set_domain( Q.size() );

        E.union_cross( s, S );
        E.union_cross( F, f );

        S.clear();
        S.add( s );

        F.clear();
        F.add( f );

        current.set_domain( Q.size() );

        assert( class_invariant() );
        return( *this );
}

Reg<FA>& Reg<FA>::concat( const Reg<FA>& r ) {
        assert( class_invariant() );
        assert( r.class_invariant() );
        // See the or operator.
        // All state-related stuff in r must be adjusted.

        // Save the old domain().
        auto int olddom( Q.size() );

        // Incorporate the other StatePool.
        Q.incorporate( r.Q );

        // Adjust the domain of the Start states.
        S.set_domain( Q.size() );

        // Transitions are just unioned.
        Transitions.disjointing_union( r.Transitions );

        // The epsilon trans. are unioned and F times r.S are added.
        E.disjointing_union( r.E );
        F.set_domain( Q.size() );

        // r.S will be needed for epsilon transitions.
        auto StateSet S1( r.S );
        // rename it to the new StatePool size.
        S1.st_rename( olddom );

        E.union_cross( F, S1 );

        // F remains the Final states.
        F = r.F;
        F.st_rename( olddom );

        current.set_domain( Q.size() );

        assert( class_invariant() );
        return( *this );
}

Reg<FA>& Reg<FA>::star() {
        assert( class_invariant() );
```

```
        // Create some new States, and adjust the domains.
        auto State s( Q.allocate() );
        auto State f( Q.allocate() );

        S.set_domain( Q.size() );
        F.set_domain( Q.size() );
        Transitions.set_domain( Q.size() );
        E.set_domain( Q.size() );
        current.set_domain( Q.size() );

        E.union_cross( s, S );
        E.union_cross( s, f );
        E.union_cross( F, S );
        E.union_cross( F, f );

        S.clear();
        S.add( s );
        F.clear();
        F.add( f );

        assert( class_invariant() );
        return( *this );
}

Reg<FA>& Reg<FA>::plus() {
        assert( class_invariant() );

        // Create some new States, and adjust the domains.
        auto State s( Q.allocate() );
        auto State f( Q.allocate() );

        S.set_domain( Q.size() );
        F.set_domain( Q.size() );
        Transitions.set_domain( Q.size() );
        E.set_domain( Q.size() );
        current.set_domain( Q.size() );

        E.union_cross( s, S );
        E.union_cross( F, S );
        E.union_cross( F, f );

        S.clear();
        S.add( s );
        F.clear();
        F.add( f );

        assert( class_invariant() );
        return( *this );
}

Reg<FA>& Reg<FA>::question() {
        assert( class_invariant() );

        // Create some new States, and adjust the domains.
        auto State s( Q.allocate() );
        auto State f( Q.allocate() );

        S.set_domain( Q.size() );
        F.set_domain( Q.size() );
        Transitions.set_domain( Q.size() );
        E.set_domain( Q.size() );
        current.set_domain( Q.size() );

        E.union_cross( s, S );
        E.union_cross( s, f );
        E.union_cross( F, f );
```

```
        S.clear();
        S.add( s );
        F.clear();
        F.add( f );
                                                                        230
        assert( class_invariant() );
        return( *this );
}
```

□

13 Reduced finite automata

Implementation class: *RFA*

Files: `rfa.h`, `rfa.cpp`

Uses: *DFA, FAabs, LBFA, RBFA, RE, StatePool, StateRel, StateSet, Trans*

Description: Class *RFA* implements reduced finite automata, as defined in [Wat93a, Definition 4.24]. It inherits from *FAabs*, and implements the interface defined by abstract base *FAabs*. A constructor taking an *RE* provides a very efficient top-down implementation of homomorphism *rfa* defined in [Wat93a, Definition 4.30].

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:28 $
#ifndef RFA_H
#define RFA_H

#include "faabs.h"
#include "st-pool.h"
#include "stateset.h"
#include "staterel.h"
#include "trans.h"
#include "re.h"
#include "dfa.h"
#include <assert.h>
#include <iostream.h>

// The existence of LBFA and RBFA are needed, for friendship.
// Can't include lbfa.h or rbfa.h due to circularity.
class LBFA;
class RBFA;

// Implement RFA's (reduced finite automata – Definition 4.24 of the Taxonomy)
// as a type of FAabs.

class RFA : virtual public FAabs {
public:
	// Some constructors:

	RFA();

	// Need a copy constructor:
	RFA( const RFA& r );

	// The Sigma-homomorphism constructor (see Definition 4.30):
	RFA( const RE& e );

	// Default destr, operator= are okay.

	// Normal FAabs member functions (don't override acceptable()):

	virtual int num_states() const;
	virtual void restart();
	virtual void advance( char a );
	virtual int in_final() const;
	virtual int stuck();
	virtual DFA determinism() const;


	// An alternate implementation of determinism() using the LBFA interpretation.
	virtual DFA determinism2() const;


	// Special members:
```

```
        friend ostream& operator<<( ostream& os, const RFA& r );

        // These two must be friends so that they can construct themselves properly.
        friend class LBFA;
        friend class RBFA;

        inline int class_invariant() const;

protected:
        // A helper: compute all of the components into place.
        //          for use in the constructor.
        void rfa_into( const RE& e,
                StatePool& Qp,
                StateSet& f,
                StateSet& l,
                Trans& Qm,
                StateRel& foll,
                int& N );

        // Recycle *this:
        void reincarnate();

        // Implementation details, protected for Reg<RFA> use.
        StatePool Q;
        StateSet first, last;
        Trans Qmap_inverse;
        StateRel follow;
        int Nullable;

        // Some simulation stuff.
        int final;
        StateSet current;
};

inline int RFA::class_invariant() const {
        return( Q.size() == first.domain()
                && Q.size() == last.domain()
                && Q.size() == Qmap_inverse.range()
                && Q.size() == follow.domain()
                && Q.size() == current.domain()
                && first.class_invariant()
                && last.class_invariant()
                && Qmap_inverse.class_invariant()
                && follow.class_invariant()
                && current.class_invariant() );
}

#endif
```

Implementation: Most of the member functions are implemented in a straight-forward manner. In implementing *RFA*, a choice must be made between the two interpretations of *States*: the left-biased or the right-biased interpretation (see the bottom of p. 73 in [Wat93a] for a brief explanation of the different interpretations). The interpretation chosen here is the right-biased one, since it appears to be the most efficient one in practice.

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:26 $
#include "rfa.h"
#include "dsrfa.h"
#include "dsrfa2.h"
#include "dfaseed.h"
```

```
RFA::RFA() :
                Nullable( 0 ) {
        assert( class_invariant() );
}

RFA::RFA( const RFA& r ) :
                Q( r.Q ),
                first( r.first ),
                last( r.last ),
                Qmap_inverse( r.Qmap_inverse ),
                follow( r.follow ),
                Nullable( r.Nullable ),
                final( r.final ),
                current( r.current ) {
        assert( class_invariant() );
}

// The Sigma-homomorphism constructor (see Definition 4.30):
RFA::RFA( const RE& e ) {
        assert( e.class_invariant() );

        auto int size( e.num_symbols() );
        first.set_domain( size );
        last.set_domain( size );
        Qmap_inverse.set_range( size );
        follow.set_domain( size );

        // Now compute all of the components into place in *this.
        rfa_into( e, Q, first, last, Qmap_inverse, follow, Nullable );

        current.set_domain( size );

        assert( class_invariant() );
}

// Provide implementation for all of the FAabs required members.
int RFA::num_states() const {
        return( Q.size() );
}

// The following functions make use of the RBFA interpretation of states
// (Section 4.3 of the Taxonomy).
void RFA::restart() {
        assert( class_invariant() );
        // Ready to see any char corresponding to a first.
        current = first;
        // Only acceptable if *this accepts epsilon (ie. Nullable is true).
        final = Nullable;
        assert( class_invariant() );
}

void RFA::advance( char a ) {
        assert( class_invariant() );

        // Figure out which are valid out-transitions.
        current.intersection( Qmap_inverse[a] );

        // current is the set of States we can really be in to see a.
        // We're accepting if at least one of the current States is also a last.
        final = current.not_disjoint( last );
        current = follow.image( current );
        assert( class_invariant() );
}

int RFA::in_final() const {
        assert( class_invariant() );
        return( final );
```

```
}

int RFA::stuck() {
	assert( class_invariant() );
	return( current.empty() );
}

DFA RFA::determinism() const {
	assert( class_invariant() );

	// Now construct the DFA components.
	return( construct_components( DSRFA( first,
			&Qmap_inverse,
			&follow,
			&first,
			&last,
			Nullable ) ) );
}

DFA RFA::determinism2() const {
	assert( class_invariant() );
	// Now construct, using the right-biased interpretation.
	return( construct_components( DSRFA2( &Qmap_inverse,
			&follow,
			&first,
			&last,
			Nullable ) ) );
}

// Implement a top down version of Sigma homomorphism rfa (Definition 4.30)
void RFA::rfa_into( const RE& e,
		StatePool& Qp,
		StateSet& f,
		StateSet& l,
		Trans& Qm,
		StateRel& foll,
		int& N ) {

	assert( e.class_invariant() );

	switch( e.root_operator() ) {
	case EPSILON:
		// Nothing to do, but make it nullable.
		N = 1;
		return;
	case EMPTY:
		N = 0;
		return;
	case SYMBOL:
		// There is a symbol here, so give it a State.
		{
			auto State q( Qp.allocate() );
			f.add( q );
			l.add( q );
			Qm.add_transition( e.symbol(), q );
			// Nothing to do to follow.
			N = 0;
		}
		return;
	case OR:
	case CONCAT:
		// Lots of common code, so do the two together.
		{
			// Compute the rfa of e.left_subexpr() into *this.
			rfa_into( e.left_subexpr(), Qp, f, l, Qm, foll, N );

			// Use some locals to store the pieces of the RFA of
```

```
                // the right subexpression.
                auto StateSet fr;
                fr.set_domain( f.domain() );
                auto StateSet lr;
                lr.set_domain( l.domain() );
                auto Trans Qmr;
                Qmr.set_range( Qm.range() );
                auto StateRel follr;
                follr.set_domain( foll.domain() );
                auto int Nr;

                // Now compute the RFA components corresponding to the right subexpr.
                //      Note: states are still taken from Qp.
                rfa_into( e.right_subexpr(), Qp, fr, lr, Qmr, follr, Nr );

                Qm.set_union( Qmr );
                if( e.root_operator() == OR ) {
                        f.set_union( fr );
                        l.set_union( lr );
                        foll.set_union( follr );
                        N = N || Nr;
                } else {
                        // This is a CONCAT.
                        assert( e.root_operator() == CONCAT );
                        foll.set_union( follr );
                        foll.union_cross( l, fr );

                        if( N ) {
                                f.set_union( fr );
                        }
                        if( Nr ) {
                                l.set_union( lr );
                        } else {
                                l = lr;
                        }
                        N = N && Nr;
                }
        }
        return;
case STAR:
case PLUS:
        // First, do the subexpression.
        rfa_into( e.left_subexpr(), Qp, f, l, Qm, foll, N );
        foll.union_cross( l, f );
        if( e.root_operator() == STAR ) {
                N = 1;
        } else {
                assert( e.root_operator() == PLUS );
        }
        return;
case QUESTION:
        rfa_into( e.left_subexpr(), Qp, f, l, Qm, foll, N );
        N = 1;
        return;
}
}

void RFA::reincarnate() {
        Q.reincarnate();
        first.reincarnate();
        last.reincarnate();
        Qmap_inverse.reincarnate();
        follow.reincarnate();
        Nullable = final = 0;
        current.reincarnate();
}
```

```
ostream& operator<<( ostream& os, const RFA& r ) {
	assert( r.class_invariant() );
	os << "\nRFA\n";
	os << "Q = " << r.Q << '\n';
	os << "first = " << r.first << '\n';
	os << "last = " << r.last << '\n';
	os << "Qmap_inverse = " << r.Qmap_inverse << '\n';
	os << "follow = \n" << r.follow << '\n';
	os << "Nullable = " << (r.Nullable ? "true" : "false");
	os << "\nfinal = " << (r.final ? "true" : "false");
	os << "\ncurrent = " << r.current << '\n';
	return( os );
}
```

□

Implementation class: *DSRFA*

Files: `dsrfa.h`, `dsrfa.cpp`

Uses: *CharRange, CRSet, StateRel, StateSet, Trans*

Description: Class *DSRFA* implements the abstract class interface required to construct a *DFA* from an *RFA*. A *DSRFA* is constructed in the *RFA* member function *determinism*, and then passed to template function *construct_components*, which constructs the components of the *DFA*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:48 $
#ifndef DSRFA_H
#define DSRFA_H

#include "charrang.h"
#include "crset.h"
#include "stateset.h"
#include "staterel.h"
#include "transrel.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It is used in the construction of a DFA from an RFA. The behaviour
// of a DSRFA follows the simulation behaviour of an RFA.

class DSRFA {
public:
	// Must always have an argument-less constructor.
	inline DSRFA();

	inline DSRFA( const DSRFA& r );

	// A special constructor:
	DSRFA( const StateSet& rq,
			const Trans *rQmap_inverse,
			const StateRel *rfollow,
			const StateSet *rfirst,
			const StateSet *rlast,
			const int rfinalness );

	inline const DSRFA& operator=( const DSRFA& r );

	// The required member functions:
	inline int final() const;
	CRSet out_labels() const;
```

```
	DSRFA out_transition( const CharRange a ) const;
	inline int operator==( const DSRFA& r ) const;
	inline int operator!=( const DSRFA& r ) const;


	friend ostream& operator<<( ostream& os, const DSRFA& r );

	inline int class_invariant() const;

private:
	// This stuff may vary from object to object.
	StateSet which;
	int finalness;

	// This stuff should be the same for all objects corresponding
	// to a particular RFA.
	const Trans *Qmap_inverse;
	const StateRel *follow;
	const StateSet *first;
	const StateSet *last;
};

// Must always have an argument-less constructor.
inline DSRFA::DSRFA() :
		which(),
		Qmap_inverse( 0 ),
		follow( 0 ),
		first( 0 ),
		last( 0 ),
		finalness( 0 ) {
	// No assert since it won't satisfy the class invariant.
}

inline DSRFA::DSRFA( const DSRFA& r ) :
		which( r.which ),
		Qmap_inverse( r.Qmap_inverse ),
		follow( r.follow ),
		first( r.first ),
		last( r.last ),
		finalness( r.finalness ) {
	assert( class_invariant() );
}

inline const DSRFA& DSRFA::operator=( const DSRFA& r ) {
	assert( r.class_invariant() );
	// *this may not satisfy the class invariant yet.
	which = r.which;
	Qmap_inverse = r.Qmap_inverse;
	follow = r.follow;
	first = r.first;
	last = r.last;
	finalness = r.finalness;
	assert( class_invariant() );
	return( *this );
}

inline int DSRFA::final() const {
	assert( class_invariant() );
	return( finalness );
}

inline int DSRFA::operator==( const DSRFA& r ) const {
	assert( class_invariant() );
	assert( r.class_invariant() );
	assert( Qmap_inverse == r.Qmap_inverse
		&& follow == r.follow
		&& first == r.first
```

```
                && last == r.last );
        return( (which == r.which) && (finalness == r.finalness) );
}

inline int DSRFA::operator!=( const DSRFA& r ) const {
        return( !operator==( r ) );
}

inline int DSRFA::class_invariant() const {
        return( Qmap_inverse != 0
                && follow != 0
                && first != 0
                && last != 0
                && which.domain() == Qmap_inverse->range()
                && which.domain() == follow->domain()
                && which.domain() == first->domain()
                && which.domain() == last->domain() );
}

#endif
```

Implementation: A *DSRFA* contains a *StateSet*, representing the set of *RFA States* that make up a *DFA* state in the subset construction (under the right-biased interpretation of states). It also contains an integer indicating if the abstract state is a final one.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:46 $
#include "dsrfa.h"

// Use the RBFA interpretation of State's (Section 4.3 of the Taxonomy).

// A special constructor:
DSRFA::DSRFA( const StateSet& rq,
                const Trans *rQmap_inverse,
                const StateRel *rfollow,
                const StateSet *rfirst,
                const StateSet *rlast,
                const int rfinalness ) :
                        which( rq ),
                        Qmap_inverse( rQmap_inverse ),
                        follow( rfollow ),
                        first( rfirst ),
                        last( rlast ),
                        finalness( rfinalness ) {
        assert( class_invariant() );
}

CRSet DSRFA::out_labels() const {
        assert( class_invariant() );
        return( Qmap_inverse->labels_into( which ) );
}

// Pretty much follow the stuff in rbfa.cpp
DSRFA DSRFA::out_transition( const CharRange a ) const {
        assert( class_invariant() );
        auto StateSet t( Qmap_inverse->range_transition( a ).intersection( which ) );
        return( DSRFA( follow->image( t ),
                        Qmap_inverse,
                        follow,
                        first,
                        last,
                        t.not_disjoint( *last ) ) );
}

ostream& operator<<( ostream& os, const DSRFA& r ) {
```

```
        os << "\nDSRFA\n" << r.which << *r.Qmap_inverse << *r.follow
                << *r.first << *r.last << r.finalness << '\n';
        return( os );
}
```

□

Implementation class: *DSRFA2*

Files: dsrfa2.h, dsrfa2.cpp

Uses: *CharRange, CRSet, StateRel, StateSet, Trans*

Description: Class *DSRFA2* implements the abstract class interface required to construct a *DFA* from an *RFA*. *DSRFA2* is an alternative to *DSRFA*, using the left-biased interpretation of *RFA States*, instead of the right-biased interpretation used in *DSRFA*. A *DSRFA2* is constructed in the *RFA* member function *determinism2*, and then passed to template function *construct_components*, which constructs the components of the *DFA*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:52 $
#ifndef DSRFA2_H
#define DSRFA2_H

#include "charrang.h"
#include "crset.h"
#include "stateset.h"
#include "staterel.h"
#include "transrel.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It is used in the construction of a DFA from an RFA. The behaviour
// of a DSRFA2 follows the left-biased interpretation of RFA's (unlike DSRFA, which
// follows the right-biased interpretation). See the last item on p. 73 of the
// Taxonomy.

class DSRFA2 {
public:
        // Must always have an argument-less constructor.
        inline DSRFA2();

        inline DSRFA2( const DSRFA2& r );

        // A special constructor (for use in constructing the start state):
        DSRFA2( const Trans *rQmap_inverse,
                        const StateRel *rfollow,
                        const StateSet *rfirst,
                        const StateSet *rlast,
                        const int rfinalness );

        inline const DSRFA2& operator=( const DSRFA2& r );

        // The required member functions:
        inline int final() const;
        CRSet out_labels() const;
        DSRFA2 out_transition( const CharRange a ) const;
        inline int operator==( const DSRFA2& r ) const;
        inline int operator!=( const DSRFA2& r ) const;


        friend ostream& operator<<( ostream& os, const DSRFA2& r );
```

```
	inline int class_invariant() const;

private:
	// Another special constructor, for use in out_transitions():
	DSRFA2( const StateSet& rq,
			const Trans *rQmap_inverse,
			const StateRel *rfollow,
			const StateSet *rfirst,
			const StateSet *rlast );

	// This stuff may vary from object to object.
	StateSet which;
	int finalness;

	// This stuff should be the same for all objects corresponding
	// to a particular RFA.
	const Trans *Qmap_inverse;
	const StateRel *follow;
	const StateSet *first;
	const StateSet *last;
};

// Must always have an argument-less constructor.
inline DSRFA2::DSRFA2() :
		which(),
		Qmap_inverse( 0 ),
		follow( 0 ),
		first( 0 ),
		last( 0 ),
		finalness( 0 ) {
	// No assert since it won't satisfy the class invariant.
}

inline DSRFA2::DSRFA2( const DSRFA2& r ) :
		which( r.which ),
		Qmap_inverse( r.Qmap_inverse ),
		follow( r.follow ),
		first( r.first ),
		last( r.last ),
		finalness( r.finalness ) {
	assert( class_invariant() );
}

inline const DSRFA2& DSRFA2::operator=( const DSRFA2& r ) {
	assert( r.class_invariant() );
	// *this may not satisfy the class invariant yet.
	which = r.which;
	Qmap_inverse = r.Qmap_inverse;
	follow = r.follow;
	first = r.first;
	last = r.last;
	finalness = r.finalness;
	assert( class_invariant() );
	return( *this );
}

inline int DSRFA2::final() const {
	assert( class_invariant() );
	return( finalness );
}

inline int DSRFA2::operator==( const DSRFA2& r ) const {
	assert( class_invariant() );
	assert( r.class_invariant() );
	assert( Qmap_inverse == r.Qmap_inverse
		&& follow == r.follow
```

```
                && first == r.first
                && last == r.last );
        return( (which == r.which) && (finalness == r.finalness) );
}

inline int DSRFA2::operator!=( const DSRFA2& r ) const {
        return( !operator==( r ) );
}

inline int DSRFA2::class_invariant() const {
        return( Qmap_inverse != 0
                && follow != 0
                && first != 0
                && last != 0
                && which.domain() == Qmap_inverse->range()
                && which.domain() == follow->domain()
                && which.domain() == first->domain()
                && which.domain() == last->domain() );
}

#endif
```

Implementation: A *DSRFA2* contains a *StateSet*, representing the set of *RFA States* that make up a *DFA* state in the subset construction (under the left-biased interpretation of states). It also contains an integer indicating if the abstract state is a final one.

Performance: For higher performance in constructing a *DFA*, use class *DSRFA* instead.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:50 $
#include "dsrfa2.h"

// Use the LBFA interpretation of State's (Section 4.2 of the Taxonomy).

// A special constructor (for use in constructing the start state):
DSRFA2::DSRFA2( const Trans *rQmap_inverse,
                const StateRel *rfollow,
                const StateSet *rfirst,
                const StateSet *rlast,
                const int rfinalness ) :
                        Qmap_inverse( rQmap_inverse ),
                        follow( rfollow ),
                        first( rfirst ),
                        last( rlast ),
                        finalness( rfinalness ) {
        // We assume that we receive the finalness magically as a parameter.

        which.set_domain( follow->domain() );
        // The emptiness of which is the indicator of the start state.
        assert( which.empty() );
        assert( class_invariant() );
}

// Another special constructor, for use in out_transitions():
DSRFA2::DSRFA2( const StateSet& rq,
                const Trans *rQmap_inverse,
                const StateRel *rfollow,
                const StateSet *rfirst,
                const StateSet *rlast ) :
                        which( rq ),
                        Qmap_inverse( rQmap_inverse ),
                        follow( rfollow ),
                        first( rfirst ),
                        last( rlast ) {
        assert( !rq.empty() );
```

```
        finalness = rq.not_disjoint( *last );
        assert( class_invariant() );
}

CRSet DSRFA2::out_labels() const {
        assert( class_invariant() );
        // There are two cases to deal with, it's either the start state, or not.
        //        emptiness of which is indicator of start state.
        if( which.empty() ) {
                return( Qmap_inverse->labels_into( *first ) );
        } else {
                return( Qmap_inverse->labels_into( follow->image( which ) ) );
        }
}

// Pretty much follow the stuff in rbfa.cpp
DSRFA2 DSRFA2::out_transition( const CharRange a ) const {
        assert( class_invariant() );
        // Again, two cases:
        auto StateSet t( which.empty() ? *first : follow->image( which ) );

        assert( !t.empty() );
        return( DSRFA2( Qmap_inverse->range_transition( a ).intersection( t ),
                        Qmap_inverse,
                        follow,
                        first,
                        last ) );
}

ostream& operator<<( ostream& os, const DSRFA2& r ) {
        os << "\nDSRFA2\n" << r.which << *r.Qmap_inverse << *r.follow
                << *r.first << *r.last << r.finalness << '\n';
        return( os );
}
```

□

13.1 The Σ-algebra *Reg<RFA>*

Implementation class: *Reg<RFA>*

Files: `sig-rfa.cpp`

Uses: *CharRange, Reg, RFA*

Description: The template instantiation *Reg<RFA>* implements the Σ-algebra of reduced finite automata, as defined in [Wat93a, Definition 4.29]. The operators can be used directly to construct complex *RFAs*, or the *Reg<RFA>* constructor from *RE* (which is defined in template class *Reg*) can be used to construct the homomorphic image of the regular expression.

Implementation: The implementation follows directly from [Wat93a, Definition 4.29].

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:33 $
#include "charrang.h"
#include "rfa.h"
#include "sigma.h"

// Implement the Sigma-algebra operators (Definition 4.29 of the Taxonomy).
Reg<RFA>& Reg<RFA>::epsilon() {
        // This RFA may have been something in a previous life.
        // Wipe out all previous info in this structure.
        reincarnate();
```

```
        Nullable = 1;

        assert( class_invariant() );
        return( *this );
}

Reg<RFA>& Reg<RFA>::empty() {
        // See epsilon case.
        reincarnate();

        assert( Nullable == 0 );

        assert( class_invariant() );
        return( *this );
}

Reg<RFA>& Reg<RFA>::symbol( const CharRange r ) {
        // See epsilon case.
        reincarnate();

        auto State q( Q.allocate() );

        first.set_domain( Q.size() );
        first.add( q );

        last.set_domain( Q.size() );
        last.add( q );

        Qmap_inverse.set_range( Q.size() );
        Qmap_inverse.add_transition( r, q );

        follow.set_domain( Q.size() );
        // Nothing to add to follow.

        Nullable = 0;

        current.set_domain( Q.size() );

        assert( class_invariant() );
        return( *this );
}

Reg<RFA>& Reg<RFA>::or( const Reg<RFA>& r ) {
        assert( class_invariant() );
        assert( r.class_invariant() );
        // All state-related stuff in r must be adjusted.

        Q.incorporate( r.Q );

        first.disjointing_union( r.first );

        last.disjointing_union( r.last );

        Qmap_inverse.disjointing_union( r.Qmap_inverse );

        follow.disjointing_union( r.follow );

        Nullable = Nullable || r.Nullable;

        current.set_domain( Q.size() );
        assert( class_invariant() );
        return( *this );
}

Reg<RFA>& Reg<RFA>::concat( const Reg<RFA>& r ) {
        assert( class_invariant() );
        assert( r.class_invariant() );
```

```
        // See the or operator.
        // All state-related stuff in r must be adjusted.
        // First, incorporate the follow sets.

        follow.disjointing_union( r.follow );
        // Rename the incoming StateSet's first and last StateSets.
        auto StateSet fi1( r.first );
        fi1.st_rename( Q.size() );
        auto StateSet la1( r.last );
        la1.st_rename( Q.size() );

        Q.incorporate( r.Q );
        // Adjust last as well.
        last.set_domain( Q.size() );
        follow.union_cross( last, fi1 );

        first.set_domain( Q.size() );
        if( Nullable ) {
                first.set_union( fi1 );
        };

        if( r.Nullable ) {
                last.set_union( la1 );
        } else {
                last = la1;
        }

        Qmap_inverse.disjointing_union( r.Qmap_inverse );

        Nullable = Nullable && r.Nullable;

        current.set_domain( Q.size() );
        assert( class_invariant() );
        return( *this );
}

Reg<RFA>& Reg<RFA>::star() {
        assert( class_invariant() );

        // Nothing to do to Q, first, last, Qmap_inverse.
        follow.union_cross( last, first );
        Nullable = 1;
        assert( class_invariant() );
        return( *this );
}

Reg<RFA>& Reg<RFA>::plus() {
        assert( class_invariant() );

        // Don't change Q, first, last, Qmap_inverse, Nullable.
        follow.union_cross( last, first );
        assert( class_invariant() );
        return( *this );
}

Reg<RFA>& Reg<RFA>::question() {
        assert( class_invariant() );

        // Don't change Q, first, last, Qmap_inverse, follow.
        Nullable = 1;
        assert( class_invariant() );
        return( *this );
}
```

□

14 Left-biased finite automata

Implementation class: *LBFA*

Files: `lbfa.h`, `lbfa.cpp`

Uses: *DFA, FAabs, RFA, State, StatePool, StateRel, StateSet, Trans*

Description: Class *LBFA* implements left-biased finite automata, as defined in [Wat93a, Definition 4.20]. It inherits from *FAabs*, and implements the interface defined by abstract base *FAabs*. A constructor taking an *RFA* provides a method of (indirectly) constructing an *LBFA* from regular expressions. The constructor implements isomorphism *decode* as defined in [Wat93a, Definition 4.28]. A special member function takes an an *RFA* and implements the non-isomorphic mapping *convert* defined in [Wat93a, Definition 4.35]. This mapping provides an alternative way of constructing an *LBFA* from an *RFA* (see [Wat93a, p. 36–37]).

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:54 $
#ifndef LBFA_H
#define LBFA_H

#include "state.h"
#include "faabs.h"
#include "st-pool.h"
#include "stateset.h"
#include "staterel.h"
#include "trans.h"
#include "rfa.h"
#include <assert.h>
#include <iostream.h>

// Implement left-biased FA's (see Definition 4.20 of the Taxonomy).

class LBFA : virtual public FAabs {
public:
        // Constructors, destructors, operator=.

        // Default constructor must build the empty language acceptor.
        LBFA();

        // Default copy constructor, destr, operator= are okay.

        // Specially construct from an RFA (see Definition 4.28).
        LBFA( const RFA& r );


        // Standard FAabs operators. Don't override acceptable():

        virtual int num_states() const;
        virtual void restart();
        virtual void advance( char a );
        virtual int in_final() const;
        virtual int stuck();
        virtual DFA determinism() const;


        // Some Sigma-algebra conversions:
        //          taken from Definitions 4.28 and 4.35
        virtual LBFA& decode( const RFA& r );
        virtual LBFA& convert( const RFA& r );


        // Special member functions:
```

```
        friend ostream& operator<<( ostream& os, const LBFA& r );

        inline int class_invariant() const;

protected:
        // Implementation stuff, protected for Reg<LBFA>:

        StatePool Q;
        // Single start state.
        State s;
        StateSet F;
        // Qmap (see Definition 4.24) stored as its inverse.
        Trans Qmap_inverse;
        StateRel follow;

        // Simulation stuff:
        StateSet current;
};

inline int LBFA::class_invariant() const {
        return( Q.contains( s )
                && Q.size() == F.domain()
                && Q.size() == Qmap_inverse.range()
                && Q.size() == follow.domain()
                && Q.size() == current.domain()
                && F.class_invariant()
                && Qmap_inverse.class_invariant()
                && follow.class_invariant()
                && current.class_invariant() );
}

#endif
```

Implementation: Class *LBFA* is a **friend** of class *RFA*. Most of the member functions are implemented in a straight-forward manner.

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:52 $
#include "lbfa.h"
#include "dfaseed.h"
#include "dslbfa.h"


LBFA::LBFA() {
        // Ensure that *this has the LBFA properties.
        //        a single start State, with no in-transitions.
        s = Q.allocate();
        F.set_domain( Q.size() );
        Qmap_inverse.set_range( Q.size() );
        follow.set_domain( Q.size() );
        current.set_domain( Q.size());
        assert( class_invariant() );
}

// Implement homomorphism decode (Definition 4.28).
LBFA::LBFA( const RFA& r ) :
                Q( r.Q ),
                F( r.last ),
                Qmap_inverse( r.Qmap_inverse ),
                follow( r.follow ) {
        assert( r.class_invariant() );

        // The new start state:
        s = Q.allocate();
        F.set_domain( Q.size() );
```

```
        Qmap_inverse.set_range( Q.size() );
        follow.set_domain( Q.size() );
        current.set_domain( Q.size() );

        // ...which is final if the RFA is nullable.
        if( r.Nullable ) {
                F.add( s );
        }
        auto StateSet fst( r.first );
        fst.set_domain( Q.size() );
        follow.union_cross( s, fst );
        assert( r.class_invariant() );
}

int LBFA::num_states() const {
        assert( class_invariant() );
        return( Q.size() );
}

void LBFA::restart() {
        assert( class_invariant() );
        current.clear();
        current.add( s );
        assert( class_invariant() );
}

void LBFA::advance( char a ) {
        assert( class_invariant() );

// There are two possible ways to implement this. It's not clear which is
// more efficient.
//        current = Qmap_inverse[a].intersection( follow.image( current ) );
        current = follow.image( current ).intersection( Qmap_inverse[a] );
        assert( class_invariant() );
}

int LBFA::in_final() const {
        return( current.not_disjoint( F ) );
}

int LBFA::stuck() {
        return( current.empty() );
}

DFA LBFA::determinism() const {
        // Make sure that *this is structurally sound.
        assert( class_invariant() );

        // Need s as a singleton StateSet.
        auto StateSet S;
        S.set_domain( Q.size() );
        S.add( s );
        // Now construct the DFA components.
        return( construct_components( DSLBFA( S, &Qmap_inverse, &follow, &F ) ) );
}

// Implement homomorphism decode (Definition 4.28).
LBFA& LBFA::decode( const RFA& r ) {
        // Implement Definition 4.28 of the Taxonomy.
        assert( r.class_invariant() );

        Q = r.Q;
        F = r.last;
        Qmap_inverse = r.Qmap_inverse;
        follow = r.follow;

        // The new start state:
```

```
        s = Q.allocate();
        F.set_domain( Q.size() );
        Qmap_inverse.set_range( Q.size() );
        follow.set_domain( Q.size() );
        current.set_domain( Q.size() );

        // ...which is final if the RFA is nullable.
        if( r.Nullable ) {
                F.add( s );
        }
        auto StateSet fst( r.first );
        fst.set_domain( Q.size() );
        follow.union_cross( s, fst );

        assert( class_invariant() );
        return( *this );
}

// Implement non-homomorphic RFA->LBFA mapping convert (Defn. 4.35).
LBFA& LBFA::convert( const RFA& r ) {
        // Implement Definition 4.35 of the Taxonomy.
        assert( r.class_invariant() );

        Q = r.Q;
        // r.first must be a singleton set for this to work properly!
        // (See Property 4.37):
        assert( r.first.size() == 1 );
        s = r.first.smallest();

        F = r.last;
        Qmap_inverse = r.Qmap_inverse;
        follow = r.follow;

        current = r.current;

        assert( class_invariant() );
        return( *this );
}

ostream& operator<<( ostream& os, const LBFA& r ) {
        assert( r.class_invariant() );
        os << "\nLBFA\n";
        os << "Q = " << r.Q << '\n';
        os << "s = " << r.s << '\n';
        os << "F = " << r.F << '\n';
        os << "Qmap_inverse = " << r.Qmap_inverse << '\n';
        os << "follow = \n" << r.follow << '\n';
        os << "current = " << r.current << '\n';
        return( os );
}
```

□

Implementation class: *DSLBFA*

Files: `dslbfa.h`, `dslbfa.cpp`

Uses: *CharRange, CRSet, StateRel, StateSet, Trans*

Description: Class *DSLBFA* implements the abstract class interface required to construct a *DFA* from an *LBFA*. A *DSLBFA* is constructed in the *LBFA* member function *determinism*, and then passed to template function *construct_components*, which constructs the components of the *DFA*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
```

```
// $Date: 1994/05/02 15:57:34 $
#ifndef DSLBFA_H
#define DSLBFA_H

#include "charrang.h"
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "staterel.h"
#include "transrel.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It is used in the construction of a DFA from an LBFA. The behaviour
// of a DSLBFA follows the simulation behaviour of an LBFA.

class DSLBFA {
public:
        // Must always have an argument-less constructor.
        inline DSLBFA();

        inline DSLBFA( const DSLBFA& r );

        // A special constructor:
        DSLBFA( const StateSet& rq,
                        const Trans *rQmap_inverse,
                        const StateRel *rfollow,
                        const StateSet *rF );

        inline const DSLBFA& operator=( const DSLBFA& r );

        // The required member functions:
        inline int final() const;
        CRSet out_labels() const;
        DSLBFA out_transition( const CharRange a ) const;
        inline int operator==( const DSLBFA& r ) const;
        inline int operator!=( const DSLBFA& r ) const;


        friend ostream& operator<<( ostream& os, const DSLBFA& r );

        inline int class_invariant() const;

private:
        StateSet which;
        const Trans *Qmap_inverse;
        const StateRel *follow;
        const StateSet *F;
};

// Must always have an argument-less constructor.
inline DSLBFA::DSLBFA() :
                which(),
                Qmap_inverse( 0 ),
                follow( 0 ),
                F( 0 ) {
        // No assertion of the class invariant since it isn't satisfied yet.
}

inline DSLBFA::DSLBFA( const DSLBFA& r ) :
                which( r.which ),
                Qmap_inverse( r.Qmap_inverse ),
                follow( r.follow ),
                F( r.F ) {
        assert( class_invariant() );
```

```
}

inline const DSLBFA& DSLBFA::operator=( const DSLBFA& r ) {
        assert( r.class_invariant() );
        // *this does not satisfy the class invariant yet.
        which = r.which;
        Qmap_inverse = r.Qmap_inverse;
        follow = r.follow;
        F = r.F;
        assert( class_invariant() );
        return( *this );
}

inline int DSLBFA::final() const {
        assert( class_invariant() );
        return( which.not_disjoint( *F ) );
}

inline int DSLBFA::operator==( const DSLBFA& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        assert( Qmap_inverse == r.Qmap_inverse && follow == r.follow && F == r.F );
        return( which == r.which );
}

inline int DSLBFA::operator!=( const DSLBFA& r ) const {
        return( !operator==( r ) );
}

inline int DSLBFA::class_invariant() const {
        return( Qmap_inverse != 0
                && follow != 0
                && F != 0
                && which.domain() == Qmap_inverse->range()
                && which.domain() == follow->domain()
                && which.domain() == F->domain() );
}

#endif
```

Implementation: A *DSLBFA* contains a *StateSet*, representing the set of *LBFA States* that make up a *DFA* state in the subset construction. It also includes pointers to some components of the *LBFA* that constructed it.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:33 $
#include "dslbfa.h"

// A special constructor:
DSLBFA::DSLBFA( const StateSet& rq,
                const Trans *rQmap_inverse,
                const StateRel *rfollow,
                const StateSet *rF ) :
                        which( rq ),
                        Qmap_inverse( rQmap_inverse ),
                        follow( rfollow ),
                        F( rF ) {
        assert( class_invariant() );
}

CRSet DSLBFA::out_labels() const {
        assert( class_invariant() );
        // *Qmap_inverse contains the transitions _into_ State's.
        // So, we take the image of which under follow to get which's out-transitions.
        return( Qmap_inverse->labels_into( follow->image( which ) ) );
```

```
}

// Pretty much follow the stuff in lbfa.cpp
DSLBFA DSLBFA::out_transition( const CharRange a ) const {
        assert( class_invariant() );
        return( DSLBFA( follow->image( which ).intersection(
                        Qmap_inverse->range_transition( a ) ),
                        Qmap_inverse,
                        follow,
                        F ) );
}

ostream& operator<<( ostream& os, const DSLBFA& r ) {
        os << "\nDSLBFA\n" << r.which << *r.Qmap_inverse << *r.follow << *r.F << '\n';
        return( os );
}
```

□

15 Right-biased finite automata

Implementation class: *RBFA*

Files: `rbfa.h`, `rbfa.cpp`

Uses: *DFA, FAabs, RFA, State, StatePool, StateRel, Trans*

Description: Class *RBFA* implements right-biased finite automata, as defined in [Wat93a, Construction 4.43]. It inherits from *FAabs*, and implements the interface defined by abstract base class *FAabs*. A constructor taking an *RFA* provides a method of (indirectly) constructing an *RBFA* from regular expressions. The constructor implements isomorphism $R \circ \mathit{decode} \circ R$ as defined in [Wat93a, p. 41]. A special member function takes a reference to an *RFA* and implements the non-isomorphic mapping $R \circ \mathit{convert} \circ R$ defined in [Wat93a, p. 43]. This mapping provides an alternative way of constructing an *RBFA* from an *RFA* (see [Wat93a, p. 42–43]).

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:18 $
#ifndef RBFA_H
#define RBFA_H

#include "faabs.h"
#include "st-pool.h"
#include "stateset.h"
#include "staterel.h"
#include "trans.h"
#include "rfa.h"
#include <assert.h>
#include <iostream.h>

// Implement right-biased FA's (Construction 4.43 in the Taxonomy).

class RBFA : virtual public FAabs {
public:
        // Constructors, destructors, operator=:

        // Default constructor builds the empty language acceptor.
        RBFA();

        // Default constr, copy constr, destr, operator= are okay.

        // Special constructor (from RFA) implementing R; decode; R (see
        // Construction 4.45 of the Taxonomy):
        RBFA( const RFA& r );


        // Standard FAabs member functions. Don't override acceptable():

        virtual int num_states() const;
        virtual void restart();
        virtual void advance( char a );
        virtual int in_final() const;
        virtual int stuck();
        virtual DFA determinism() const;


        // Some Sigma-algebra conversions:
        //         as the duals of those given in Definitions 4.28 and 4.35
        virtual RBFA& decode( const RFA& r );
        virtual RBFA& convert( const RFA& r );


        // Special member functions:
```

```
        friend ostream& operator<<( ostream& os, const RBFA& r );

        inline int class_invariant() const;

protected:
        // Implementation stuff, protected for Reg<RBFA>:

        StatePool Q;
        StateSet S;
        // Single final state as required (Construction 4.43):
        State f;
        // See rfa.h and lbfa.h for explanation of Qmap_inverse:
        Trans Qmap_inverse;
        StateRel follow;

        // Simulation stuff:
        StateSet current;
};

inline int RBFA::class_invariant() const {
        return( Q.contains( f )
                && Q.size() == S.domain()
                && Q.size() == Qmap_inverse.range()
                && Q.size() == follow.domain()
                && Q.size() == current.domain()
                && S.class_invariant()
                && Qmap_inverse.class_invariant()
                && follow.class_invariant()
                && current.class_invariant() );
}

#endif
```

Implementation: Class *RBFA* is a **friend** of class *RFA*. Most of the member functions are implemented in a straight-forward manner.

```
/* (c) Copyright 1994 by Bruce Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:16 $
#include "rbfa.h"
#include "dsrbfa.h"
#include "dfaseed.h"


RBFA::RBFA() {
        // Give *this the properties required of Construction 4.43
        f = Q.allocate();
        S.set_domain( Q.size() );
        Qmap_inverse.set_range( Q.size() );
        follow.set_domain( Q.size() );
        current.set_domain( Q.size() );

        assert( class_invariant() );
}

// Implement R; decode; R (Construction 4.45 of the Taxonomy):
RBFA::RBFA( const RFA& r ) :
                Q( r.Q ),
                S( r.first ),
                Qmap_inverse( r.Qmap_inverse ),
                follow( r.follow ) {
        assert( r.class_invariant() );

        // Create the new final state.
        f = Q.allocate();
```

```
        S.set_domain( Q.size() );
        Qmap_inverse.set_range( Q.size() );
        follow.set_domain( Q.size() );
        current.set_domain( Q.size() );

        if( r.Nullable ) {
                S.add( f );
        }
        auto StateSet la( r.last );
        la.set_domain( Q.size() );
        follow.union_cross( la, f );
        assert( class_invariant() );
}

int RBFA::num_states() const {
        assert( class_invariant() );
        return( Q.size() );
}

void RBFA::restart() {
        assert( class_invariant() );
        current = S;
        assert( class_invariant() );
}

void RBFA::advance( char a ) {
        // Compute which states we can even transition out of, then
        // transition out of them.
        assert( class_invariant() );
        current = follow.image( current.intersection( Qmap_inverse[a] ) );
        assert( class_invariant() );
}

int RBFA::in_final() const {
        assert( class_invariant() );
        return( current.contains( f ) );
}

int RBFA::stuck() {
        assert( class_invariant() );
        return( current.empty() );
}

DFA RBFA::determinism() const {
        // Make sure that *this is structurally sound.
        assert( class_invariant() );
        // Now construct the DFA components.
        return( construct_components( DSRBFA( S, &Qmap_inverse, &follow, f ) ) );
}

// Implement homomorphism R ; decode ; R (Construction 4.45):
RBFA& RBFA::decode( const RFA& r ) {
        // Implement R ; decode ; R of the Taxonomy (p. 41).
        assert( r.class_invariant() );

        Q = r.Q;
        S = r.first;
        Qmap_inverse = r.Qmap_inverse;
        follow = r.follow;

        // Allocate the new final state.
        f = Q.allocate();
        S.set_domain( Q.size() );
        Qmap_inverse.set_range( Q.size() );
        follow.set_domain( Q.size() );

        current.set_domain( Q.size() );
```

```
        current.clear();

        if( r.Nullable ) {
                S.add( f );
        }
        auto StateSet la( r.last );
        la.set_domain( Q.size() );
        follow.union_cross( la, f );

        assert( class_invariant() );
        return( *this );
}

// Implement non-homomorphic RFA->RBFA mapping R ; conver ; R (see
// page 43 of the Taxonomy):
RBFA& RBFA::convert( const RFA& r ) {
        // Implement R ; convert ; R of the Taxonomy (p. 43).
        assert( r.class_invariant() );

        Q = r.Q;
        // r.last must be a singleton for this to work.
        // (See Property 4.37 of the Taxonomy):
        f = r.last.smallest();
        assert( r.last.size() == 1 );

        S = r.first;
        Qmap_inverse = r.Qmap_inverse;
        follow = r.follow;

        current = r.current;

        assert( class_invariant() );
        return( *this );
}

ostream& operator<<( ostream& os, const RBFA& r ) {
        assert( r.class_invariant() );
        os << "\nRBFA\n";
        os << "Q = " << r.Q << '\n';
        os << "S = " << r.S << '\n';
        os << "f = " << r.f << '\n';
        os << "Qmap_inverse = " << r.Qmap_inverse << '\n';
        os << "follow = \n" << r.follow << '\n';
        os << "current = " << r.current << '\n';
        return( os );
}
```

□

Implementation class: *DSRBFA*

Files: `dsrbfa.h`, `dsrbfa.cpp`

Uses: *CharRange, CRSet, StateRel, StateSet, Trans*

Description: Class *DSRBFA* implements the abstract class interface required to construct a *DFA* from an *RBFA*. A *DSRBFA* is constructed in the *RBFA* member function *determinism*, and then passed to template function *construct_components*, which constructs the components of the *DFA*.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:38 $
#ifndef DSRBFA_H
#define DSRBFA_H
```

```
#include "charrang.h"
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "staterel.h"
#include "transrel.h"
#include <assert.h>
#include <iostream.h>


// This class is used to represent abstract States in a DFA that is still under
// construction. It is used in the construction of a DFA from an RBFA. The behaviour
// of a DSRBFA follows the simulation behaviour of an RBFA.

class DSRBFA {
public:
        // Must always have an argument-less constructor.
        inline DSRBFA();

        inline DSRBFA( const DSRBFA& r );

        // A special constructor:
        DSRBFA( const StateSet& rq,
                        const Trans *rQmap_inverse,
                        const StateRel *rfollow,
                        const State rf );

        inline const DSRBFA& operator=( const DSRBFA& r );

        // The required member functions:
        inline int final() const;
        CRSet out_labels() const;
        DSRBFA out_transition( const CharRange a ) const;
        inline int operator==( const DSRBFA& r ) const;
        inline int operator!=( const DSRBFA& r ) const;


        friend ostream& operator<<( ostream& os, const DSRBFA& r );

        inline int class_invariant() const;

private:
        StateSet which;
        const Trans *Qmap_inverse;
        const StateRel *follow;
        State f;
};

// Must always have an argument-less constructor.
inline DSRBFA::DSRBFA() :
                which(),
                Qmap_inverse( 0 ),
                follow( 0 ),
                f( Invalid ) {
        // *this will not yet satisfy the class invariant.
}

inline DSRBFA::DSRBFA( const DSRBFA& r ) :
                which( r.which ),
                Qmap_inverse( r.Qmap_inverse ),
                follow( r.follow ),
                f( r.f ) {
        assert( class_invariant() );
}

inline const DSRBFA& DSRBFA::operator=( const DSRBFA& r ) {
        assert( r.class_invariant() );
```

```
        // *this may not satisfy the invariant yet.
        which = r.which;
        Qmap_inverse = r.Qmap_inverse;
        follow = r.follow;
        f = r.f;
        assert( class_invariant() );
        return( *this );
}

inline int DSRBFA::final() const {
        assert( class_invariant() );
        return( which.contains( f ) );
}

inline int DSRBFA::operator==( const DSRBFA& r ) const {
        assert( class_invariant() );
        assert( r.class_invariant() );
        assert( Qmap_inverse == r.Qmap_inverse && follow == r.follow && f == r.f );
        return( which == r.which );
}

inline int DSRBFA::operator!=( const DSRBFA& r ) const {
        return( !operator==( r ) );
}

inline int DSRBFA::class_invariant() const {
        return( Qmap_inverse != 0
                && follow != 0
                && 0 <= f
                && f < follow->domain()
                && which.domain() == Qmap_inverse->range()
                && which.domain() == follow->domain() );
}

#endif
```

Implementation: A *DSRBFA* contains a *StateSet*, representing the set of *RBFA States* that make up a *DFA* state in the subset construction. It also includes pointers to some components of the *RBFA* which constructed it.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:57:36 $
#include "dsrbfa.h"

// A special constructor:
DSRBFA::DSRBFA( const StateSet& rq,
                const Trans *rQmap_inverse,
                const StateRel *rfollow,
                const State rf ) :
                        which( rq ),
                        Qmap_inverse( rQmap_inverse ),
                        follow( rfollow ),
                        f( rf ) {
        assert( class_invariant() );
}

CRSet DSRBFA::out_labels() const {
        assert( class_invariant() );
        return( Qmap_inverse->labels_into( which ) );
}

// Pretty much follow the stuff in rbfa.cpp
DSRBFA DSRBFA::out_transition( const CharRange a ) const {
        assert( class_invariant() );
        return( DSRBFA( follow->image( Qmap_inverse->range_transition( a )
```

```
                    .intersection( which ) ),
                Qmap_inverse,
                follow,
                f ) );
}

ostream& operator<<( ostream& os, const DSRBFA& r ) {
    os << "\nDSRBFA\n" << r.which << *r.Qmap_inverse << *r.follow << r.f << '\n';
    return( os );
}
```

□

16 Automata constructions

The finite automata construction functions are functions which take an *RE* (or a pointer to an *RE*) and return an automaton of some type (one of *FA*, *RFA*, *LBFA*, *RBFA*, or *DFA*). Functions implementing Brzozowski's construction [Wat93a, Section 5.3] are given in `dconstrs.cpp` and `dconstrs.h`. Functions implementing the item set constructions [Wat93a, Section 5.5] are given in `iconstrs.cpp` and `iconstrs.h`. Files `constrs.h` and `constrs.cpp` contain all of the remaining functions (from [Wat93a, Section 4]).

All of the function declarations (or prototypes) are accompanied by their corresponding reference in [Wat93a].

```
/* (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 15:55:23 $
#include "constrs.h"

LBFA BS_variation( const RE& r ) {
        assert( r.class_invariant() );

        // Construct the homomorphic image of r.
        auto RFA l( r );

        // Construct the begin-marker.
        auto Reg<RFA> m;
        m.symbol( '$' );

        // Attach the begin-marker.
        m.concat( (const Reg<RFA>&)l );

        // Construct the dummy LBFA, for use with the non-homomorphic mapping
        // convert (Defn. 4.35 of the Taxonomy)
        auto LBFA LB;

        return( LB.convert( m ) );
}
```

```
RBFA BS_variation_dual( const RE& r ) {
        // See above.
        assert( r.class_invariant() );

        auto RFA l( r );

        // Construct the end-marker.
        auto Reg<RFA> m;
        m.symbol( '$' );

        // Attach the end-marker.
        ((Reg<RFA>&)l).concat( m );

        auto RBFA RB;

        return( RB.convert( l ) );
}
```

```
/* (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 15:55:45 $
#include "dconstrs.h"
#include "dsre.h"
#include "dsre-opt.h"
#include "dfaseed.h"


DFA Brz( const RE& r ) {
        assert( r.class_invariant() );
        return( construct_components( DSRE( r ) ) );
}

DFA Brz_optimized( const RE& r ) {
        assert( r.class_invariant() );
        return( construct_components( DSREopt( r ) ) );
}
```

```
/* (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
```

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:39 $
#include "dconstrs.h"
#include "dsis.h"
#include "dsderem.h"
#include "dsis-opt.h"
#include "dfaseed.h"


DFA Iconstr( const RE *r ) {
        assert( r->class_invariant() );
        return( construct_components( DSIS( r ) ) );
}

DFA DeRemer( const RE *r ) {
        assert( r->class_invariant() );
        return( construct_components( DSDeRemer( r ) ) );
}
DFA Oconstr( const RE *r ) {
        assert( r->class_invariant() );
        return( construct_components( DSIS_opt( r ) ) );
}
```

Part IV
Minimizing *DFAs*

An interesting *DFA* minimization function is the one due to Brzozowski (appearing in [Wat93b, Section 2]).

User function: *DFA::min_Brzozowski*

Files: `dfa.h`

Uses: see the entry for class *DFA*

Description: Brzozowski's minimization algorithm differs from all of the other minimization algorithms, and appears as inline *DFA* member function *min_Brzozowski*. The member function simply minimizes the *DFA*.

Implementation: The implementation appears in file `dfa.h`. The implementation follows directly from [Wat93b, Section 2].

□

The remainder of Part IV deals with the other minimization functions, based upon their derivations in [Wat93b].

17 Helper functions

Two member functions of class *DFA* are used as helpers in those minimization functions which are based upon an equivalence relation (see [Wat93b, Section 3]).

Implementation function: *DFA::compress*

Files: `dfa.cpp`

Uses: see the entry for class *DFA*

Description: Member function *compress* is overloaded to take either a *StateEqRel* (an equivalence relation on *States*), or a *SymRel* (a symmetrical relation on *States*). The relation is assumed to be a refinement of relation E [Wat93b, Definition 3.3]. The member function implements function *merge* defined in [Wat93b, Transformation 3.1].

Implementation: The implementation appears in file `dfa.cpp`. The implementation follows directly from [Wat93b, Transformation 3.1].

□

Implementation function: *DFA::split*

Files: `min.cpp`

Uses: *CharRange, DFA, State, StateEqRel, StateSet*

Description: Member function *split* takes two *States* (p and q), a *CharRange* a, and a *StateEqRel*. The two *States* are assumed to be representatives in the *StateEqRel*. The equivalence class of p is then split into those *States* (equivalent to p) which transition on a to a *State* equivalent to q, and those that do not. If in fact there was a split (the equivalence class of p is split into two equivalence classes), p will still be the unique representative of one of the two new equivalence classes; in this case, the (unique) representative of the other equivalence class is returned. If no such split occurred, function *split* returns *Invalid*.

Implementation: The implementation is a simple, iterative one. It implements the equivalent of the assignment to Q_0' in Algorithm 4.4 of [Wat93b].

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:06 $
#include "charrang.h"
#include "state.h"
#include "stateset.h"
#include "st-eqrel.h"
#include "dfa.h"

State DFA::split( const State p, const State q, const CharRange a, StateEqRel& P ) const {
	assert( class_invariant() );
	assert( P.class_invariant() );

	// p and q must be representatives of their eq. classes.
	assert( p == P.eq_class_representative( p ) );
	assert( q == P.eq_class_representative( q ) );

	// Split [p] with respect to [q] and CharRange a.
	auto StateSet part;
	part.set_domain( Q.size() );

	// Iterate over [p], and see whether each member transitions into [q]
	// on CharRange a.
	auto State st;
	for( P.equiv_class( p ).iter_start( st );
```

```
                !P.equiv_class( p ).iter_end( st );
                P.equiv_class( p ).iter_next( st ) ) {
            auto State dest( T.transition_on_range( st, a ) );
            // It could be that dest == Invalid.
            //          if not, check if dest is in [q].
            if( dest != Invalid && P.equivalent( dest, q ) ) {
                part.add( st );
            }
        }

        // The following containment must hold after the splitting.
        assert( P.equiv_class( p ).contains( part ) );

        // Return non-zero if something was split.
        if( (part != P.equiv_class( p )) && !part.empty() ) {
            // Now figure out what the other piece is.
            auto StateSet otherpiece( P.equiv_class( p ) );
            otherpiece.remove( part );
            assert( !otherpiece.empty() );

            P.split( part );
            assert( class_invariant() );

            // Now, we must return the representative of the newly created
            // equivalence class.
            auto State x( P.eq_class_representative( otherpiece.smallest() ) );
            assert( x != Invalid );
            // It could be that p is not in part.
            auto State y( P.eq_class_representative( part.smallest() ) );
            assert( y != Invalid );
            assert( x != y );

            if( p == x ) {
                // If p is the representative of 'otherpiece', then
                // return the representative of 'part'.
                assert( otherpiece.contains( p ) );
                return( y );
            } else {
                // If p is the representative of 'part', then return the
                // representative of 'otherpiece'.
                assert( part.contains( p ) );
                return( x );
            }
        } else {
            assert( (part == P.equiv_class( p )) || (part.empty()) );
            // No splitting to be done.
            return( Invalid );
        }
}
```

□

18 Aho, Sethi, and Ullman's algorithm

User function: *DFA::min_dragon*

Files: `min-asu.cpp`

Uses: *CRSet, DFA, State, StateEqRel, StateSet*

Description: Member function *min_dragon* is an implementation of Algorithm 4.6 in [Wat93b]. It is named the "dragon" function after Aho, Sethi, and Ullman's "dragon book" [ASU86].

Implementation: The algorithm begins with the approximation E_0 to the *State* equivalence relation E. It then repeatedly partitions the equivalence classes until the greatest fixed point (E) is reached.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:59 $
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "st-eqrel.h"
#include "dfa.h"

// Implement Algorithm 4.6 of the Taxonomy.
DFA& DFA::min_dragon() {
	assert( class_invariant() );
	assert( Usefulf() );

	// We'll need the combination of all of the out-transitions of all of the
	// States, when splitting equivalence classes.
	auto CRSet C;
	auto State st;
	for( st = 0; st < Q.size(); st++ ) {
		C.combine( T.out_labels( st ) );
	}

	// P is the equivalence relation approximation to E. It is initialized
	// to the total relation with domain Q.size().
	auto StateEqRel P( Q.size() );
	// We now initialize it to E_0.
	P.split( F );

	// reps is the set of representatives of P.
	auto StateSet reps( P.representatives() );

	// [st] is going to be split w.r.t. something.

	// The following is slightly convoluted.
	reps.iter_start( st );
	while( !reps.iter_end( st ) ) {
		// Try to split [st] w.r.t. every class [q].
		auto State q;

		// Keep track of whether something was indeed split.
		//	Having to use this variable could be avoided with a goto
		auto int something_split( 0 );

		// Iterate over all q, and try to split [st] w.r.t. all other
		// equivalence classes [q].
		// Stop as early as possible.
		for( reps.iter_start( q ); !reps.iter_end( q )
				&& !something_split; reps.iter_next( q ) ) {
			// Iterate over the possible transition labels, and
			// do a split if possible.
			auto int i;
```

```
                for( i = 0; !C.iter_end( i ) && !something_split; i++ ) {
                        something_split = (split( st, q, C.iterator( i ), P ) != Invalid);
                }
            }

            // If something was split, restart the outer repetition.
            if( something_split ) {
                // The set of representatives will have changed due to
                // the split.
                reps = P.representatives();
                reps.iter_start( st );
                // Now continue the outer repetition with the restarted
                // representatives.
            } else {
                // Just go on as usual.
                reps.iter_next( st );
            }
        }

        compress( P );
        assert( class_invariant() );

        return( *this );
}
```

□

19 Hopcroft and Ullman's algorithm

User function: *DFA::min_HopcroftUllman*

Files: `min-hu.cpp`

Uses: *CRSet, DFA, State, StateSet, SymRel*

Description: This member function provides a very convoluted implementation of Algorithm 4.7 of [Wat93b]. It computes the distinguishability relation (relation D in [Wat93b]).

Implementation: The algorithm is not quite the same as that presented in [Wat93b, Algorithm 4.7]. The member function computes the equivalence relation (on *States*) by first computing distinguishability relation D. Initially, the pairs of distinguishable *States* are those pairs p, q where one is a final *State* and the other is not. Pairs of *States* that reach p, q are then also distinguishable (according to the efficiency improvement property given in [Wat93b, Section 4.7, p. 16]). Iteration terminates when all distinguishable *States* have been considered.

Performance: This algorithm can be expected to run quite slowly, since it makes use of reverse transitions in the deterministic transition relation (*DTransRel*). The transition relations (*TransRel* and *DTransRel*) are optimized for forward transitions.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:05 $
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "symrel.h"
#include "dfa.h"

// Implement (a version of) Algorithm 4.7 of the minimization Taxonomy.
DFA& DFA::min_HopcroftUllman() {
	assert( class_invariant() );
	assert( Usefulf() );

	// We need the combination of all transition labels, to iterate over
	// transitions.
	auto State st;
	auto CRSet C;
	for( st = 0; st < Q.size(); st++ ) {
		C.combine( T.out_labels( st ) );
	}

	// First, figure out which are the non-final States.
	auto StateSet nonfinal( F );
	nonfinal.complement();

	// Use Z to keep track of the pairs that still need to be considered for distinguishedness.
	auto SymRel Z;
	Z.set_domain( Q.size() );

	// We begin with those pairs that are definitely distinguished.
	//	this includes pairs with different parity.
	Z.add_pairs( F, nonfinal );

	// It also includes pairs with differing out-transitions.
	//	iterate over C (the CRSet) and check the haves and have-nots
	auto StateSet have;
	have.set_domain( Q.size() );
	auto StateSet havenot;
	havenot.set_domain( Q.size() );

	auto int it;
```

```
for( it = 0; !C.iter_end( it ); it++ ) {
        // have and havenot must be empty for the update to be correct.
        assert( have.empty() && havenot.empty() );
        auto State p;
        // Iterate over the States and check which have a transition.
        for( p = 0; p < Q.size(); p++ ) {
                // Does p have the transition?
                if( T.transition_on_range( p, C.iterator( it ) ) != Invalid ) {
                        have.add( p );
                } else {
                        havenot.add( p );
                }
        }
        // have and havenot are distinguished from one another.
        //         (under the assumption that Usefulf() holds)
        Z.add_pairs( have, havenot );

        have.clear();
        havenot.clear();
}

// G will be use to accumulate the (distinguishability) relation D.
auto SymRel G;
G.set_domain( Q.size() );

// Now consider all of the pairs until nothing changes.
while( 1 ) {
        auto State p;
        // Go looking for the next pair to do.
        for( p = 0; p < Q.size() && Z.image( p ).empty(); p++ );

        // There may be nothing left to do.
        if( p == Q.size() ) {
                break;
        } else {
                assert( !Z.image( p ).empty() );
                // Choose q such that {p,q} is in Z.
                //         We know that such a q exists.
                auto State q( Z.image( p ).smallest() );
                assert( q != Invalid );

                // Move {p,q} from Z across to G.
                Z.remove_pair( p, q );
                G.add_pair( p, q );

                // Now, check out the reverse transitions from p and q.
                auto int i;
                // Iterate over all of the labels.
                for( i = 0; !C.iter_end( i ); i++ ) {
                        auto StateSet pprime( T.reverse_transition( p,
                                        C.iterator( i ) ) );
                        auto StateSet qprime( T.reverse_transition( q,
                                        C.iterator( i ) ) );
                        // pprime and qprime are all distinguished.
                        // Iterate over both sets and flag them as distinguished.
                        auto State pp;
                        for( pprime.iter_start( pp ); !pprime.iter_end( pp );
                                        pprime.iter_next( pp ) ) {
                                auto State qp;
                                for( qprime.iter_start( qp );
                                                !qprime.iter_end( qp );
                                                qprime.iter_next( qp ) ) {
                                        // Mark pp, qp for processing if they
                                        // haven't already been done.
                                        if( !G.contains_pair( pp, qp ) ) {
                                                Z.add_pair( pp, qp );
                                        } // if
```

```
                        } // for
                    } // for
                } // for
            } // if
        } // while

        // Now, compute E from D
        G.complement();
        // And use it to compress the DFA.
        compress( G );

        assert( class_invariant() );
        return( *this );
}
```

□

20 Hopcroft's algorithm

User function: *DFA::min_Hopcroft*

Files: `min-hop.cpp`

Uses: *CRSet, DFA, State, StateEqRel, StateSet*

Description: Member function *min_Hopcroft* implements Hopcroft's $n \log n$ minimization algorithm, as presented in [Wat93b, Algorithm 4.8].

Implementation: The member function uses some encoding tricks to effectively implement the abstract algorithm. The combination of the out-transitions of all of the *States* is stored in a *CRSet* C. Set L from the abstract algorithm is implemented as a mapping from *States* to **int** (an array of **int** is used). Array L should be interpreted as follows: if *State* q a representative, then the following pairs still require processing (are still in abstract set L):

$$([q], C_0), \ldots, ([q], C_{L(q)-1})$$

The remaining pairs do not require processing:

$$([q], C_{L(q)}), \ldots, ([q], C_{|C|-1})$$

This implementation facilitates quick scanning of L for the next valid *State-CharRange* pair.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:03 $
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "st-eqrel.h"
#include "dfa.h"

// Implement Algorithm 4.8 (Hopcroft's O(n log n) algorithm).
DFA& DFA::min_Hopcroft() {
        assert( class_invariant() );

        // This algorithm requires that the DFA not have any final unreachable
        // States.
        assert( Usefulf() );

        auto State q;

        // Keep track of the combination of all of the out labels of State's.
        auto CRSet C;
        for( q = 0; q < Q.size(); q++ ) {
                C.combine( T.out_labels( q ) );
        }

        // Encode set L as a mapping from State to [0,|C|] where:
        //        if q is a representative of a class in the partition P, then
        //        L (the abstract list) contains
        //                ([q],C_0), ([q],C_1), ..., ([q],C_(L(q)-1))
        //        but not
        //                ([q],C_(L(q))), ..., ([q],C_(|C|-1))
        auto int *const L( new int [Q.size()] );
        for( q = 0; q < Q.size(); q++ ) {
                L[q] = 0;
        }

        // Initialize P to be the total equivalence relation.
        auto StateEqRel P( Q.size() );
        // Now set P to be E_0.
```

```
	P.split( F );

	// Now, build the set of representatives and initialize L.
	auto StateSet repr( P.representatives() );
	if( F.size() <= (Q.size() - F.size()) ) {
		repr.intersection( F );
	} else {
		repr.remove( F );
	}

	// Do the final set up of L
	for( repr.iter_start( q ); !repr.iter_end( q ); repr.iter_next( q ) ) {
		L[q] = C.size();
	}

	// Use a break to get of this loop.
	while( 1 ) {
		// Find the first pair in L that still needs processing.
		for( q = 0; q < Q.size() && !L[q]; q++ );

		// It may be that we're at the end of the processing.
		if( q == Q.size() ) {
			break;
		} else {
			// Mark this element of L as processed.
			L[q]--;

			// Iterate over all eq. classes, and try to split them.
			auto State p;
			repr = P.representatives();
			for( repr.iter_start( p ); !repr.iter_end( p );
					repr.iter_next( p ) ) {
				// Now split [p] w.r.t. (q,C_(L[q]))
				auto State r( split( p, q, C.iterator( L[q] ), P ) );
				// r is the representative of the new split of the
				// eq. class that was represented by p.

				if( r != Invalid ) {
					// p and r are the new representatives.
					// Now update L with the smallest of
					// [p] and [r].
					if( P.equiv_class( p ).size()
							<= P.equiv_class( r ).size() ) {
						L[r] = L[p];
						L[p] = C.size();
					} else {
						L[r] = C.size();
					} // if
				} // if
			} // for
		} // if
	} // while

	// L is no longer needed.
	delete L;

	// We can now use P to compress the DFA.
	compress( P );

	assert( class_invariant() );
	return( *this );
}
```

□

21 A new minimization algorithm

User function: *DFA::min_Watson*

Files: `min-bww.cpp`

Uses: *CRSet, DFA, State, StateEqRel, SymRel*

Description: Member function *min_Watson* implements the new minimization algorithm appearing in [Wat93b, Sections 4.6–4.7]. The algorithm computes the equivalence relation E (on states) from below. The importance of this is explained in [Wat93b, p. 16].

Implementation: Function *min_Watson* is an implementation of Algorithm 4.10 (of [Wat93b]). The helper function *are_eq* is an implementation of the second algorithm appearing in Section 4.6 of [Wat93b]. Function *are_eq* takes two parameters more than the version appearing in [Wat93b]: H (a *StateEqRel*) and Z (a *SymRel*). These parameters are used to discover equivalence (or distinguishability) of *States* earlier than the abstract algorithm would.

Performance: Memoization in function *are_eq* would provide a great speedup.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:01 $
#include "state.h"
#include "crset.h"
#include "symrel.h"
#include "st-eqrel.h"
#include "dfa.h"
#include <assert.h>

// The following is function equiv from Section 4.6 of the min. taxonomy.
int DFA::are_eq( State p, State q, SymRel& S, const StateEqRel& H, const SymRel& Z ) const {
	if( S.contains_pair( p, q ) || H.equivalent( p, q ) ) {
		// p and q are definitely equivalent.
		return( 1 );
	} else if( !Z.contains_pair( p, q ) ) {
		assert( !S.contains_pair( p, q ) );
		assert( !H.equivalent( p, q ) );
		// p and q were already compared (in the past) and found to be
		// inequivalent.
		return( 0 );
	} else {
		assert( !H.equivalent( p, q ) );
		assert( Z.contains_pair( p, q ) );

		// Now figure out all of the valid out-transitions from p and q.
		auto CRSet C( T.out_labels( p ) );
		C.combine( T.out_labels( q ) );

		// Just make sure that p and q have the same out-transitions.
		auto int it;
		for( it = 0; !C.iter_end( it ); it++ ) {
			if( (T.transition_on_range( p, C.iterator( it ) ) == Invalid )
					|| (T.transition_on_range( q, C.iterator( it ) )
						== Invalid ) ) {
				// There's something that one transitions on, but
				// the other doesn't
				return( 0 );
			} // if
		} // for

		// p and q have out-transitions on equivalent labels.
		// Keep track of whether checking needs to continue.
		S.add_pair( p, q );
		for( it = 0; !C.iter_end( it ); it++ ) {
```

```
                        auto State pdest( T.transition_on_range( p, C.iterator( it ) ) );
                        auto State qdest( T.transition_on_range( q, C.iterator( it ) ) );
                        if( !are_eq( pdest, qdest, S, H, Z ) ) {
                                // p and q have been found distinguished.
                                S.remove_pair( p, q );
                                return( 0 );
                        }
                } // for

                // p and q have been found equivalent.
                S.remove_pair( p, q );
                return( 1 );
        } // if
}

DFA& DFA::min_Watson() {
        assert( class_invariant() );
        assert( Usefulf() );

        // (Symmetrical) State relation S is from p.14 of the min. taxonomy.
        auto SymRel S;
        S.set_domain( Q.size() );

        // H is used to accumulate equivalence relation E.
        auto StateEqRel H( Q.size() );
        // Start with the identity since this is approximation from below
        // w.r.t. refinement.
        H.identity();

        // Z is a SymRel containing pairs of States still to be considered.
        auto SymRel Z;
        Z.set_domain( Q.size() );
        Z.identity();
        // We will need the set of non-final States to initialize Z.
        auto StateSet nonfinal( F );
        nonfinal.complement();
        Z.add_pairs( F, nonfinal );
        // Z now contains those pairs that definitely do not need comparison.

        Z.complement();
        // Z initialized properly now.

        auto State p;
        // Consider each p.
        for( p = 0; p < Q.size(); p++ ) {
                auto State q;
                // Consider each q that p still needs to be compared to.
                for( Z.image( p ).iter_start( q ); !Z.image( p ).iter_end( q );
                                Z.image( p ).iter_next( q ) ) {
                        // Now compare p and q.
                        if( are_eq( p, q, S, H, Z ) ) {
                                // p and q are equivalent.
                                H.equivalize( p, q );
                        } else {
                                // Don't do anything since we aren't computing
                                // distinguishability explicitly.
                        }

                        // Comparing q to p is the same as comparing [q] and [p]
                        // all at once.
                        // Mark them as such.
                        Z.remove_pairs( H.equiv_class( p ), H.equiv_class( q ) );
                } // for
        } // for

        compress( H );
```

```
        assert( class_invariant() );
        return( *this );
}
```

□

References

[ASU86] Aho, A.V., R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools,* Addison-Wesley Publishing Co., Reading, MA, 1986.

[Boo94] Booch, G. *Object oriented analysis and design, with applications,* (second edition) The Benjamin/Cummings Publishing Co., Redwood City, CA, 1994.

[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.

[HN92] Henricson, M. and E. Nyquist. "Programming in C++: rules and recommendations," Technical reports M90 0118 Uen, Ellemtel Telecommunication Systems Laboratories, Alvsjo, Sweden, 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.

[MeB88] Meyer, B. *Object oriented software construction,* Prentice Hall International, New York, NY, 1988.

[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.

[SE90] Stroustrup, B. and M. Ellis. *The annotated C++ reference manual,* Addison-Wesley, Reading, MA, 1990.

[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. "An introduction to the **FIRE engine**: A C++ toolkit for FInite automata and Regular Expressions," Computing Science Note 94/21, Eindhoven University of Technology, The Netherlands, 1994. Available by ftp from ftp.win.tue.nl in pub/techreports/pi.