C++ Classes for the Efficient Manipulation and Storage of Hierarchical Objects

Dean R. E. Long*

UCSC-CRL-93-19
May 27, 1993

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

Many applications must efficiently store and manipulate complex objects. Often sub-objects or entire objects are identical. Memory use can be decreased by the use of object handles which point to shared objects in place of actual objects. If the objects are hierarchical, sub-objects can also be represented with handles, allowing many operations to manipulate handles instead of whole objects. The copy-on-write and object registration techniques presented here reduce the cost of storing, copying, modifying, and matching hierarchical objects. Using object registration, identical objects are detected and shared, allowing objects to be uniquely identified by their location in memory. Copy-on-write object semantics allows increased sharing and reduced copying, while hierarchical copy-on-write objects using handles allows copies to have deep-copy behavior but shallow-copy cost.

*This work, particularly work on the START program, was supported in part by NSF grant CCR-9102635.

Contents

List of Figures

List of Tables

1. Introduction

This document describes the new classes I developed to solve the problems encountered while writing a C++ application. The application needed to quickly copy, modify, and match complex objects. Two of the biggest problems were enormous memory requirements and slow matching performance. By modifying the somewhat monolithic classes slightly so that they were arranged in a hierarchy and by using the new classes, the matching speed and memory efficiency improved dramatically.

To address the above problem of large memory requirements when numerous objects are being operated on, these classes take advantage of the fact that often many objects or sub-objects in memory at any given time are identical. By registering objects, identical objects can be detected and shared. An extra level of indirection is needed to share objects. This is accomplished through the use of handles. Handles also allow copy-on-write semantics to be implemented in a straightforward way. This effectively allows an object copy to have the appearance of a deep copy, but to have the performance of a shallow copy. A deep copy copies the actual object as opposed to just copying the pointer. Changes to one object do not affect the other object. A shallow copy copies the pointer only. Changes to either object affect both objects.

The new classes use C++ templates [Stroustrup and Ellis, 1991], so they are reusable and work for any type object. One base class is used to make registered objects. Another template class is used to create copy-on-write handles to registered objects. The handle template class also requires the use of a hash table template class, which in turn requires the use of a move-to-front list template class. A handle contains a pointer to a registered object, while a registered object contains an object plus a reference count. A registered object could be either a new object that is derived from the registered class, or it could be derived from an existing class and the registered class using multiple inheritance. In either case, a new class can be built that has the same interface as the registered class, but which uses a handle to implement a copy-on-write version of that class. An object referenced through a handle is also unique; object registration is used to make sure that, for each registered class, all identical instances of that class share a single copy of the underlying registered object in memory. This must be done after a handle-referenced object has been modified. Because all non-identical handles will have unique pointers, this allows matching of the object handles to be as simple as comparing the two pointers. The address of an object becomes a unique identifier, limited by the number of objects that can be put in an address space. This is different from computing some n-bit integer identifier for the object, which has to be able to encode *all* possible objects of the type, as only those objects that are actually created during the program's execution have identifiers.

These classes were created in the process of developing a static-analysis program that finds the states of a parallel program. They were essential because the straightforward implementation was too slow and took up too much memory. A new approach was needed to gain the required speed and memory efficiency.

The static-analysis tool can be used to find and analyze the states, including deadlock states, of parallel programs like the Dining Philosophers [Hoare, 1978]. One thing that the program needs to do quickly is to make incremental changes to complex objects and later compare them. By taking advantage of the features provided by the handle class mentioned above, copying, modifying, and comparing of the objects in the program is very efficient.

The large objects are made up of smaller objects, which are themselves made up of even smaller objects. Because of the hierarchical structure of the objects, the handles can share sub-objects as well as entire objects. Thus the amount of space required to store a modified object is only slighter more than the size of the change, not the size of the object. The unmodified sub-objects will remain unchanged.

In the first section the classes are described, along with some examples. The next section describes the application whose requirements motivated the design of these classes. Section 3.1.3 describes the efficiency gains realized when dealing with our main test input, Dining Philosophers. Finally, we conclude with possible ideas for future work.

1.1 Related Work

Copy-on-write virtual memory is widely used in operating systems [Nelson and Ousterhout, 1988]. Ronald White implemented copy-on-write C++ objects using base classes, inheritance, and virtual functions [White, 1991]. This is the method I would have had to use if a C++ compiler with templates was not available. While Mr. White's objects are copy-on-write, multiple copies of the same object can still exist in memory at the same time,[1] leaving opportunities for further memory sharing. My implementation takes advantage of this opportunity with a uniqueness constraint that ensures that identical objects are shared, even after being modified. It also strives to make the job of the programmer easier by providing template classes, which can be simpler to use than writing the necessary virtual functions with the non-template approach.

M. C. Cooper defines a data structure called a *Repnet* that takes advantage of the repetitions in hierarchical objects [Cooper, 1989]. His paper describes the application of these objects to template matching in computer vision algorithms. While a Repnet is a pixel-based structure, generalizing pixels to arbitrary objects gives a structure virtually the same as that produced by the Handle class described here. In fact, the use of the Handle class should lead naturally to implementations of Repnets, quadtrees, octrees, or other self-similar or recursive data structures. An example of a recursive tree data structure is given is Section 2.4.

[1] Objects that do not share a common history cannot be shared.

2. Registered Copy-On-Write Objects

The Handle class described in Section 2.3.2 implements a handle to a shared copy-on-write object. All the shared objects of a particular type are registered in a dictionary data structure. This dictionary is used to determine if a new or modified object matches an existing registered object. This means that each registered object is unique, as defined by its `==` operator. This chapter describes the handles that implement unique copy-on-write (UCOW) objects and how to use them. The first section describes the features of these objects that make them desirable. The second section argues for the use of hierarchical UCOW objects to implement certain types of large objects. The next section describes the Handle class, which implements UCOW objects, in some detail. Finally, a full working example is given that demonstrates the use of these objects in a simple data structure.

2.1 Efficient Operations on Handles

This section explains the operations that Handles perform efficiently. Handles store a pointer to a shared object. Many operations can be performed by manipulating only the pointers stored in handles. The efficiency of operations such as `modify` will be increased further through the use *hierarchical* objects.

2.1.1 Fast Copy

A handle will be copied either by the assignment operator or a copy constructor. For either, the work required to copy a handle is essentially the same. The destination handle will be assigned the pointer of the source handle, and the reference count stored in the object will be incremented. In the case of assignment, the object referenced by the destination handle will have its reference count decremented first.

2.1.2 Match

Any two handles can be quickly compared for equivalence by comparing their respective pointers. Since the pointers are guaranteed to point to unique objects, the objects are different if the pointers are different. If the pointers are the same, the objects they point to are therefore the same. In LISP, two lists can be `EQUAL` but not `EQ`[1] [Winston and Horn, 1981]. With UCOW objects, the distinction between `EQUAL` and `EQ` goes away. If two UCOW objects are `EQUAL` they must also be `EQ`.

2.1.3 Modify

To modify a COW object, a writable copy of the object needs to be constructed by the handle. This requires the handle to make a deep copy of the object if its reference count is greater than one. At first this may seem expensive, but if the object is composed of COW sub-objects, resulting in a hierarchical object, the deep copy will be implemented as shallow copies of the sub-objects. This is possible because the sub-object copy is implemented by its handle. Sub-objects themselves can be built of even smaller COW objects, creating

[1] The former means the lists are identical, the latter means the lists are actually the same list.

a hierarchy. This increases the potential of sub-objects to share storage and extends the depth to which shallow copies can be used. If the hierarchical object is thought of as a tree, a modify operation only affects the smallest sub-tree that contains the modification. In other words, only the dirty bits get copied. If the object was not hierarchical, the smallest portion of the object that could be copied would be the entire object. Therefore, preparing a hierarchical object for writing is faster than preparing a non-hierarchical object. The modification will also create less new memory as long as *all* sub-parts are not modified.

2.1.4 Overhead

If Handles provided just copy-on-write objects, there would be little overhead involved. These handles also provide *unique* copy-on-write objects. The uniqueness guarantee makes comparing objects fast, but a price must be paid for that feature. The implementation must ensure that objects are unique for fast matching and for maximum storage efficiency. To do this, it needs to compare a dirty (potentially modified) object with the other registered objects. To avoid pairwise comparisons between a dirty object and every registered object, a hash table is used. The hash function for an object may be faster than actually comparing two objects. If the dirty object hashes to an empty slot, no objects need be compared. If the slot is not empty, the dirty object needs to be compared with all the objects that hashed to that same slot. If a matching object is found, the dirty object is discarded, and its handle will be directed to point at the existing registered object. If a match is not found, the dirty object is registered, at which point it is considered clean. This pairwise comparison of objects is not necessarily prohibitively expensive. Just like for copying, using hierarchical objects can make comparisons faster even when the top-level handle pointers cannot be compared, in this case because one object is not yet registered. Even though the top-level pointers cannot be directly compared because the dirty object breaks the uniqueness property, we can still reap the benefit of pointer comparisons if all (or even some) of the sub-objects are handles.

In an earlier implementation, the user of a Handle needed to explicitly call a function to unify a dirty object with the set of registered objects. Currently this overhead is performed behind the scenes. The implementation allows one outstanding dirty object, which is unified when necessary. This is actually a performance bonus because it allows a clever implementation to avoid some overhead if the same object is being modified repeatedly. An implementation could also relax the limit on the number of outstanding dirty objects, allowing a small set of dirty objects. However, too large a set of dirty objects could negate the memory savings which is a design goal of the class.

2.2 Hierarchical UCOW Objects

As mentioned in the previous section, designing an object as a hierarchy of UCOW objects can reduce the cost of operations on those objects. The benefit comes from the fact that sub-objects, except for those at the base of the hierarchy, are UCOW objects, so compares and copies only involve pointer manipulations.

There are a few disadvantages to this approach. First, storage for these objects is allocated dynamically, which can be more expensive than static or stack storage, both to allocate and free. Second, the memory locations of an object are probably not close together or contiguous, because each sub-object is accessed through a pointer to dynamically-allocated

memory. This can have adverse affects on locality and thus cache performance. Finally, the extra level of indirection that pointers require can be a significant cost by itself.

Though the approach of using hierarchical UCOW objects to build complex objects has disadvantages, the advantages can outweigh the disadvantages. This will be true if the overhead incurred is offset by realized savings. The deeper the hierarchy, the more likely that a writable object can be created using shallow copies. Also, the percentage of time spent on the overhead of cleaning up dirty objects will decrease as the ratio of copy and compare operations to modify operations increases.

2.3 Descriptions

This section describes the main parameterized types I created in the process of building the static-analysis application. A detailed description of the C++ interfaces to the templates can be found in Appendix B. Throughout this chapter, a capital letter '`T`' is used to represent a type parameter for a parameterized type.

2.3.1 Registered

Some method must be provided to keep track of all the unique objects of a particular type, i.e. the objects must be *registered*. There needs to be exactly one copy of each unique object registered at any given time. A dictionary data structure (the hash table described above) is used to keep track of registered objects. A new object is added to the dictionary only if a matching object cannot be found in the dictionary. The object is immutable while it is registered in the dictionary; changes can only be made after it has been removed from the dictionary or to a copy.

The Registered class is a base class that provides private data needed by registered objects and public member functions necessary to operate on those objects. Type `T` of the `Handle<T>` template must be derived from Registered. Registered has functions `ref` and `unref` that increment and decrement reference counts, respectively. A member function `refs` returns the current reference count. For remembering whether the object is currently registered, there are functions `set_registered`, `unset_registered`, and `is_registered`. The class also declares (but does not define) a function `hash`, which the derived class must provide.[2]

2.3.2 Handle<T>

This class provides a handle for an object of type `T`. Given an object `O` and a handle `H` created from that object, the handle class enforces the invariant that `H` will point to a unique copy of `O`. By unique, we mean that any other handle created with an identical object will share the same copy of the object in memory. When the object referenced by a handle is going to be modified, actions need to be performed to enforce the invariant. The handle class provides functions to do this. Thus, this class allows you to maintain a collection of objects of type `T` such that if there are duplicates, they are efficiently stored, sharing the same memory locations. Each object is referenced through a handle, which

[2] If the derived class does not redefine the hash function, `Registered::hash` should show up at link time as an unresolved symbol. Alternatively, `Registered::hash` could be defined, but print an error that the derived class needs to define a hash function.

contains a pointer to the actual shared object. This class is very useful for manipulating complex data structures, since it facilitates memory sharing and fast comparisons. You can think of it as providing shared copy-on-write objects. Due to the invariant maintained by the class, checking equivalence between two object handles can be very fast if they are both registered because only the addresses of the objects will be compared.

Access to the underlying object is provided by two public member functions, `ro_obj` and `rw_obj`, which return a read-only or read-write reference to the object, respectively. Before modifying an object through its handle, the `modify` private member function will be called on the handle if necessary. If the handle is already "dirty", `rw_obj` can detect this and save some time. This ensures that the changes do not affect any of the other objects that are currently sharing the same storage. After the object has been modified, the `share` private member function will be called eventually, but not necessarily immediately. At the time `share` is called, if the new dirty object matches an existing object, its handle will be changed to reference the existing object. If, however, the modified object does not match any of the currently registered shared objects, it is added to the set, allowing it to be shared in the future.

The assignment operator and copy constructor for a handle are very fast, because they only deal with pointers. A pointer assignment and changes to some reference counts suffices for an assignment. The comparison operator is even simpler. It compares the pointers in the object handles. If the pointers are the same, the handles point to the same object; if the pointers are different, the objects they point to must be different. The private member function `cleanup` will be called first if either of the objects is dirty. Cleaning up a dirty object mostly involves a call to another private member function, `share`. It does all the work of unifying a dirty object with the set of registered object, making sure that identical objects are always shared.

While the Handle class provides a handle for a unique object, it does not and cannot provide the programmer with a new class with the same interface as class `T`. To do so, the programmer should build a new class with the same interface as `T`, but using a `Handle<T>` to operate on the actual object. Functions that may modify the object should use `rw_obj`, and `const` functions should use `ro_obj`. It is possible to just use `rw_obj`, but `ro_obj` is more efficient if the object is not modified.

The current implementation requires that the class `T` provide certain functions. These functions are provided by a class Registered, so all that is needed to use `Handle<T>` is to make sure that class `T` is derived from Registered. Handle uses the generic hash table class of Section 2.3.4 to keep a registry of which objects have been created.

2.3.3 LRUList<T>

LRUList implements a move-to-front doubly-linked list with items of type `T`. `T` should be either a class type or a reference to a class type. Each time an item is accessed it is moved to the front of the list to simulate an LRU (least-recently-used) list. `HashTable` uses this class to store items that hash to the same value. Adding and retrieving items from the list is done through the member functions `add` and `findMatch`. `FindMatch` looks for a matching list item and returns a pointer to the list where it was found or a null pointer if it was not found. The pointer returned by `findMatch` can be passed to `remove` to delete the item from the list.

2.3.4 HashTable<T>

This is a simple hash table class that stores elements of type `T`. It supports enough functions to act as a dictionary data structure[Cormen *et al.*, 1990]. It is similar to the list class in several ways. It has a `findMatch` function like the list class that requires `operator ==` to be defined on the type `T`. This `findMatch` has a second parameter `found` that return whether the element was found. Because the class implements a hash table, type `T` also needs to define a hash function of the form:

```
unsigned hash(void) const;
```

which must return the same value for all objects of type `T` that are equivalent by the `==` operator. Type `T` must define a hash function because elements of type `T` serve as a keys as well as a data elements.

If the specified type parameter `T` is a class, the list will store copies of each element. Like the list class, references are also allowed for the type parameter `T`. In fact, `Handle<T>` uses a `HashTable<T &>` to register objects. Allowing the type parameter `T` to be a reference type is useful, but it complicates the interface because pointers to references and references to references are not allowed in C++. One way to indicate a negative result for `findMatch` is to have the return value be a pointer to an element and return a null pointer if the element is not found. This cannot be done if `T` is a reference type. Also disallowed is having a `T &` as an extra parameter to `findMatch` to return the matched element.

Using exceptions would improve the interface considerably, in particular making the second parameter `found` to `findMatch` unneeded (see appendix). Another solution would be to disallow references, so that a pointer to the element can be returned. If a reference needs to be stored, a class containing a single reference can be defined.

The constructor creates a new hash table of a given size. You can think of the size as a hint to the hash table of how many elements you expect to store, so that the implementation can make a suitable choice for the number of slots. It is possible to store more elements in the table than its "size", but performance will be worse than if it had been created with a larger size. The current implementation uses open hashing with chaining, and does not resize. Each chain uses a move-to-front rule to approximate LRU, which is encapsulated in the `LRUList<T>` parameterized type.

The operations on the hash table are lookup, insert, and delete, with the corresponding member functions being `findMatch`, `add`, and `remove` respectively. The lookup operation is called `findMatch` because its argument is not a key, but an object of type `T`. The hash table will look for a matching object in the table and return it if found. For two objects to *match*, the comparison operator must be defined and return `TRUE`. The member function `add` inserts an object into the table without checking for duplicates, while `remove` takes an object of type `T` as its argument and deletes from the table all objects matching the specified object. See the appendix for more details about the interface for this class.

It is interesting that this class does not distinguish between the data type and the key type. Instead of simply "`HashTable<T>`", a more general hash table class could be specified as "`HashTable<KEY,T>`", where `KEY` is the key type, and `T` is the data type. The hash function and `==` operator would then need to be defined on the `KEY` type, and both the key and data element would be stored in the table. The class `HashTable<T>` instead assumes that the data contains the key (or the data is the key), so type `T` must provide the same interface as a key. Also notice that the class does not have a copy constructor (see appendix). Such a constructor would be straightforward to write, but was not needed for the application being considered.

Usage

There are different ways to make a registered class for use with the Handle template. One way is to inherit from Registered in the interface for the class. If the class is being designed from the beginning as an underlying implementation for a UCOW object, then the class definition would inherit from Registered and include a hash function. This way the constructors only need to be written once.

```
class RegisteredFoo : public Registered {
    unsigned hash(void) const;
    //...
};
```

Another way to get a registered class is to take an existing class and to derive a new class from that class and from `Registered` using multiple inheritance. This requires more work because things like constructors need to be written since they are not inherited, but it can be very useful when the source code to the class is not available. A UCOW version of the library class can share code with the original library class using this method.

```
class Bar;

class RegisteredBar : public Bar, public Registered {
    unsigned hash(void) const;
    //...
};
```

Multiple inheritance is useful but not necessary in this case. An alternative method is given below. Instead of inheriting from the library class, the library class becomes a member of the new class, and corresponding member functions are written to delegate to it. A disadvantage to this approach is the member functions will need to be changed if the library interface changes.

```
class Bar;

class RegisteredBar : public Registered {
public:
    unsigned hash(void) const;
    int foo(int x) { return bar.foo(x); }
    //...
private:
    Bar bar;
};
```

There are also different ways to use the handle class to create an object handle that mimics the behavior of the registered object. You can either inherit from the handle class, or include it as a member of your class and delegate to that member. For classes that contain more than one handle the membership approach is required. Examples of both are shown below.

```
// inheritance
class FooHandle : public Handle<RegisteredFoo> {
    //...
};

// membership
```

```
class BarHandle {
public:
    //...
private:
    Handle<RegisteredBar> left;
    Handle<RegisteredBar> right;
};
```

2.4 Example

This section takes the reader step-by-step through an extended example of how to use the Handle class to build UCOW objects. The data structure used is a binary tree where each node contains presumably large piece of data called a BigNum. The tree will be constructed in such a way that BigNums and subtrees will be shared wherever possible.

2.4.1 UniqueBigNum

Assume BigNum is a class in a library that implements very large integers. The integers might be hundreds of digits long. Now suppose you write a program that creates and compares a large number of these BigNums. Your initial implementation requires too much memory, but you realize that because of the nature of your application, there are many fewer values than numbers, i.e. many of the BigNums have the same value. You decide to change your BigNums to "unique" BigNums using the Handle class described above. This will not be a hierarchical class, but it will illustrate the process of converting a class to be shared and unique. Besides saving memory, comparing objects of the new class UniqueBigNum will be faster, but there is some extra overhead attached to creating and modifying the object handle. The higher the ratio of comparisons to modifications, the greater the performance improvement.

To make a UniqueBigNum, we need to get a handle to a shared object. The Handle template needs some special methods, which we get from Registered:

```
#include <bignum.h>
#include "handle.h"

// Define a registered BigNum
// Use multiple inheritance because BigNum is in a library

class RBigNum : public BigNum, public Registered {
public:
    RBigNum(const BigNum &b) : BigNum(b) {}
    unsigned hash(void) const
    {
        return *this % 63; // int BigNum::operator%(int)
    }
};
```

We supplied a hash function and a copy constructor, the rest is inherited or supplied by the compiler. Now we have something, `RBigNum`, that can be given to `Handle<T>`, giving us `Handle<RBigNum>`:

```
typedef Handle<RBigNum> BigNumHandle;
```

Using a `typedef` is not required, it just keeps us from having to write `Handle<RBigNum>` everywhere. Now we have a handle, but it does not look like a BigNum. Let's use the handle to build a class that looks like a BigNum but also has a value that can be shared. We need to implement all the constructors and arithmetic operators that BigNum had:

```
// Now make a unique BigNum using a handle

class UniqueBigNum {
public:
    // Let the handle's constructors do all the work
    UniqueBigNum(const BigNum &b) : handle(RBigNum(b)) {}
    // The compiler generates an appropriate copy ctor

    unsigned hash(void) const
    {
        return handle.ro_obj().hash();
    }

    UniqueBigNum operator + (const UniqueBigNum &b)
    {
        // Let BigNum class do the work
        return UniqueBigNum(handle.ro_obj() + b.handle.ro_obj());
    }
    UniqueBigNum &operator += (const UniqueBigNum &b)
    {
        // Let BigNum class do the work
        handle.rw_obj() += b.handle.ro_obj();
        return *this;
    }
    // other arithmetic operators follow the same pattern

    int operator == (const UniqueBigNum &b) const
    {
        return handle == b.handle;
    }

    // generated assignment operator does the right thing,
    // so no need to add one

private:
    BigNumHandle handle;
};
```

We now have a new class UniqueBigNum that looks like a BigNum, but only one copy of each distinct value will be stored in memory. Comparing two of these objects costs only a pointer comparison. Next we will see how to build a more complicated data structure that stores shared UniqueBigNums.

2.4.2 BigNumTree

This section describes the building of trees of big numbers. We anticipate that besides numbers in the tree being identical, whole subtrees might also be the same. To take advantage of this, each tree will have shared trees as its children. The root of the tree will also be shared. This is tricky, since the data structure is recursive. The member functions cannot be declared inline because of cyclic dependencies. We define a tree `RBigNumTree` with shared children, but an unshared root, and then use that class to construct a new class `BigNumTree` where all parts are shared. The header file follows.

```
#include "uniquebignum.h"
#include "handle.h"

class RBigNumTree;

typedef Handle<RBigNumTree> BigNumTreeHandle;

class BigNumTree {
public:
    BigNumTree(void);
    BigNumTree(const BigNum &);
    BigNumTree(const BigNum &,
        const BigNumTree &left, const BigNumTree &right);
    BigNumTree(const BigNumTree &);
    unsigned hash(void) const;
    int operator == (const BigNumTree &) const;
private:
    BigNumTreeHandle handle;
};

// I know from the beginning that I will use this class
// with the Handle template, so inherit from Registered here.

class RBigNumTree : public Registered {
public:
    RBigNumTree(const BigNum &);
    RBigNumTree(const BigNum &,
        const BigNumTree &left, const BigNumTree &right);
    RBigNumTree(const RBigNumTree &t);
    unsigned hash(void) const;
    int operator == (const RBigNumTree &b) const;
private:
    UniqueBigNum bignum;
    BigNumTree left, right;
};
```

The source code for the member function definitions follows. All of the functions are small. Most of the work done is calling constructors for data members or delegating calls.

```
#include "bignumtree.h"
```

```
// Let the base class contructors do the work
BigNumTree::BigNumTree(void) : handle() {}

BigNumTree::BigNumTree(const BigNum &n) : handle(RBigNumTree(n)) {}

BigNumTree::BigNumTree(const BigNum &n,
    const BigNumTree &l, const BigNumTree &r
) : handle(RBigNumTree(n, l, r)) {}

BigNumTree::BigNumTree(const BigNumTree &t) : handle(t.handle) {}

unsigned BigNumTree::hash(void) const
{
    return handle.null() ? 0 : handle.ro_obj().hash();
}

int BigNumTree::operator == (const BigNumTree &t) const
{
    return handle == t.handle;
}


// Let everyone else's contructors do the work
RBigNumTree::RBigNumTree(const BigNum &n)
    : bignum(n), left(), right() {}

RBigNumTree::RBigNumTree(const BigNum &n,
    const BigNumTree &l, const BigNumTree &r)
    : bignum(n), left(l), right(r) {}

RBigNumTree::RBigNumTree(const RBigNumTree &t)
    : bignum(t.bignum), left(t.left), right(t.right) {}

int RBigNumTree::operator == (const RBigNumTree &b) const
{
    return bignum == b.bignum && left == b.left && right == b.right;
}

// A fairly simple hash function
unsigned RBigNumTree::hash(void) const
{
    return (left.hash() << 8) ^ (bignum.hash() << 4) ^ right.hash();
}
```

With `BigNumTree` We define the final shared tree using a handle to the registered type `RBigNumTree`. The class mostly defines appropriate constructors for the class. Additional operations to manipulate the tree would normally be added also, but have been omitted for the sake of simplicity. BigNumTree implements a tree data structure where all identical subtrees are shared. As a bonus, the data stored in the tree is also shared. The next

section presents a short program to demonstrate how the sharing works. Though only small numbers are stored in the nodes for simplicity, we will assume the use of BigNums was justified.

2.4.3 Test Program

In this section, a program is presented that shows how BigNumTrees and UniqueBigNums can be shared, with little effort. The first few lines include all the necessary header files, and initialize the static members of the Handle class. Unfortunately this initialization cannot be avoided.[3]

```
#include <iostream.h>
#include "bignumtree.h"

// Declare static members, argument is size of hash table

HandleClassData<RBigNum> Handle<RBigNum>::class_data(7);
HandleClassData<RBigNumTree> Handle<RBigNumTree>::class_data(31);
```

The rest of the program builds three trees, two of which are identical, compares them, and produces some sharing statistics. The first tree it constructs in two steps, the others in one step. The next section describes the output.

```
BigNumTree Build_7502502_Tree_1(void)
{
    // 502 tree
    BigNumTree subtree(BigNum(5),
        BigNumTree(BigNum(0)), BigNumTree(BigNum(2)));

    // 7502502 tree
    return BigNumTree(BigNum(7), subtree, subtree);
}

BigNumTree Build_7502502_Tree_2(void)
{
    return BigNumTree(BigNum(7),
        BigNumTree(BigNum(5),
            BigNumTree(BigNum(0)), BigNumTree(BigNum(2))),
        BigNumTree(BigNum(5),
            BigNumTree(BigNum(0)), BigNumTree(BigNum(2))));
}

BigNumTree Build_7502503_Tree(void)
{
    return BigNumTree(BigNum(7),
        BigNumTree(BigNum(5),
            BigNumTree(BigNum(0)), BigNumTree(BigNum(2))),
        BigNumTree(BigNum(5),
            BigNumTree(BigNum(0)), BigNumTree(BigNum(3))));
```

[3] One alternative would be to provide macros to simplify the programmer's job.

```
}

int main(int, char **)
{
    // 7502502 tree
    BigNumTree tree1 = Build_7502502_Tree_1();

    // another 7502502 tree, from scratch
    BigNumTree tree2 = Build_7502502_Tree_2();

    // 7502503 tree
    BigNumTree tree3 = Build_7502503_Tree();

    if (tree1 == tree2) {
        cout << "Good, they are the same!\n";
    }

    if (!(tree2 == tree3)) {
        cout << "Good, tree2 and tree3 are different.\n";
    }

    cout << "Handle<RBigNumTree>:\n";
    Handle<RBigNumTree>::stats(cout);
    cout << "Handle<RBigNum>:\n";
    Handle<RBigNum>::stats(cout);
}
```

2.4.4 Output

Understanding the output from this program is important. It reveals to what extent objects are being shared. This is the output that the above program produces:

```
Good, tree1 and tree2 are the same!
Good, tree2 and tree3 are different.
Handle<RBigNumTree>:
7 unique objects
17 handles (6 null)
Handle<RBigNum>:
5 unique objects
7 handles
```

As expected, the program prints that `tree1` and `tree2` are the same and that `tree2` and `tree3` are different. The handles for the first two trees will both point to the same shared object. The handle for the third tree will point to a different shared object, since it does not match the other two trees. This is the invariant that the Handle class enforces for identical objects. The statistics for `Handle<RBigNumTree>` show that there are seven unique RBigNumTree objects. Figure 2.1 shows the sharing that is going on. The seven unique trees (counting non-empty subtrees as trees) are (0), (2), (3), (502), (503), (7502502) and (7502503) represented in prefix notation. There are six (0) trees, five (2) trees, one (3) tree, five (502) trees, one (503) tree, two (7502502) trees, and one (7502503) tree represented.

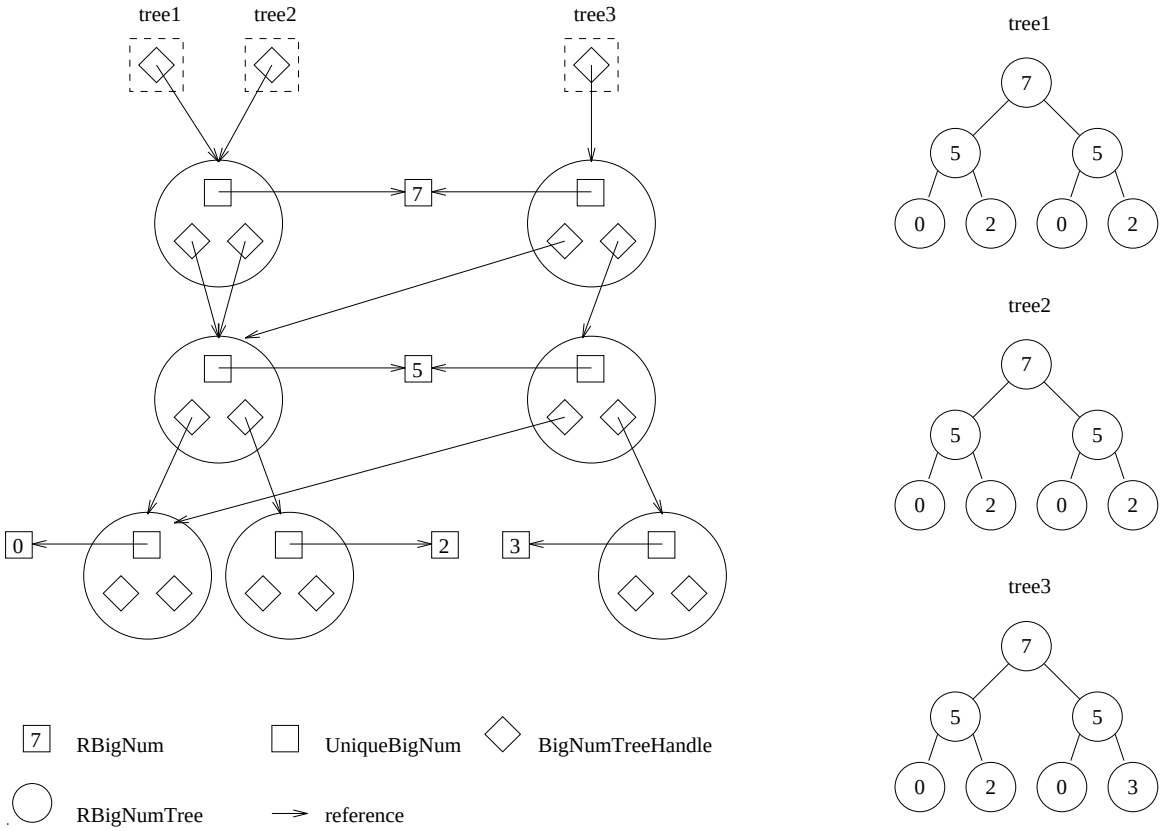

Figure 2.1: Trees of BigNums. The shared data structures for the BigNumTree program are on the left. The trees represented by the program are on the right.

This adds up to twenty-one. But the output also says seventeen handles, six of them null. Let us sort out where these numbers come from.

A null handle is a handle that does not refer to a shared object; its pointer is null. This example represents empty subtrees with null handles. The total number of non-null handles does not always correspond to the total number of trees represented. This is because handles or the objects containing the handles can also be shared. To estimate the amount of memory that would be required without sharing, the total number of trees represented would be a better metric than the total number of handles.

The reason the total number of non-null handles is less than twenty-one has to do with handles being shared. It might seem that the two (7502502) trees would result in four handles for (502) trees. Actually, there are only two. Remember that trees can be shared, and also the contents of the trees, which includes the handles for subtrees. Two handles for (502) trees come from the left and right subtree handles of the shared (7502502) tree object. Sicne both (7502502) trees are using handles, they both share the same tree object (RBigNumTree), so the second tree does not have its own copy of the subtree handles. Analyzing all the trees in this way, we get three (502) handles, one (503) handle, two (0) handles, one (2) handle, one (3) handle, two (7502502) handles, and finally one (7502503) handle (see Figure 2.1). That comes to a total of eleven. If you add the six null handles for the left and right children of the (0), (2) and (3) trees, you get seventeen handles total.

That explains the output for trees. Now how about the BigNums? It should be clear that there are five unique BigNums. All the BigNums are referenced once except for 5 and 7, which are both used twice. There are no null BigNum handles. The reason there are only seven handles is because the trees that store the handles are shared. Instead of having five BigNum handles with value 2, there is only one, and it is in the shared (2) tree. Remember there is only one copy of each tree stored in memory, though there may be many handles pointing to it.

3. Applications

This chapter describes how the classes described earlier have been used in a static analysis tool and a puzzle solver.

3.1 Static Analysis Tool

3.1.1 Introduction

This section describes how the classes described earlier have been used in an application which computes states of parallel programs using static analysis. The application is called START, which stands for Static Anomaly Reporting Tool. The input to the program is a graph representing the flow of control between synchronization events in a parallel program. Given this *Sync Graph*, the application builds a *Concurrency History Graph* or CHG of the concurrency states [Helmbold and McDowell, 1990]. In the process of finding the states and building the CHG, it also detects deadlock states. After computing all the states, they are displayed using the X Window System.

Each node in the CHG is called a *Cstate*. A Cstate is a compressed concurrency state. One Cstate can represent many states when expanded. The major components of a Cstate are a TaskMap and a LabelList. The TaskMap is where all the compression takes place. It represents the tasks, the set of places where the tasks are executing, and other task attributes in a graph. The LabelList contains global attributes for things like synchronization variables. Figure 3.1 shows the decomposition of a Cstate. It is not important to know what all the different classes represent. What should be noted, however, is way they are composed. Appendix D describes all the classes used by a Cstate.

3.1.2 Algorithm

The basic algorithm is not very complicated. The CHG begins with one node: the main program in the *Begin* state. This node is pushed on a stack of nodes to be examined. The main loop pops a node off the stack and generates its successors. Each successor Cstate will be pushed onto the stack and added to the CHG if it is considered "interesting."[1] To generate successors, for each task in the Cstate and for each Action in the task's ActionSet, if the Action it has any successors,[2] the transition is fired and new Cstates are generated by invoking the `exec` virtual function on the successor Action. The algorithm stops when there are no more nodes to examine. Conceptually this is the whole top-level algorithm, but actually it is a little more complicated due to optimizations to maintain the "compression" in compressed concurrency states and due to deadlock detection overhead.

[1] So far, a node is considered interesting if it is different from previously generated nodes. In a future implementation a node might be considered uninteresting if its expansion is a subset of the expansion of another node.

[2] based on the Sync Graph

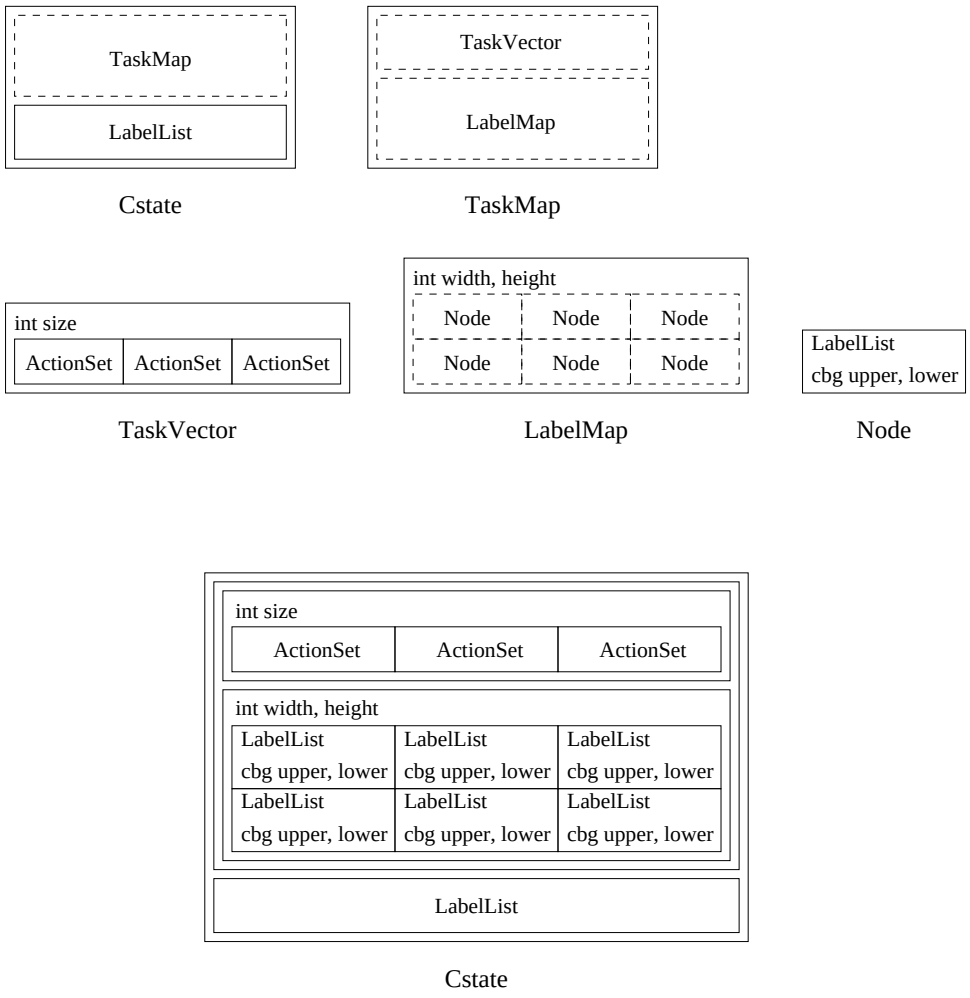

Figure 3.1: Graphical representation of a Cstate

3.1.3 Performance

This section quantifies the performance of UCOW objects by comparing the performance of two implementations of the static analysis tool described earlier. One implementation uses UCOW objects, the other uses normal objects. These normal objects are the same objects to which the handles in the UCOW objects point, so the comparison of the two implementations is as fair as possible. The changes involved to go between these two versions are fairly small. Some header files needed to be changed, but the source code did not because the objects retained the same interface.

The test runs used four different input files, representing simple Dining Philosophers programs. DPHIL5 has five philosophers, and uses a semaphore for each fork. DPHIL5w, DPHIL6w, and DPHIL7w have five, six, and seven philosophers, respectively, but use extra synchronization to reduce the number of states generated.

Table 3.1 shows how many unique objects of different types are stored in memory after the CHG is built. Because each TaskMap contains one TaskVector, and TaskVectors only exist in TaskMaps, each unique TaskVector results in a unique TaskMap. Therefore, the number of unique TaskMaps can be greater, but not less than, the number of unique

Unique Objects in Memory

	TaskMap	LabelList	Label	TaskVector	LabelMap	ActionSet
DPHIL5w	1379	48	16	1379	11	48
DPHIL5	9204	68	15	8331	81	42
DPHIL6w	5796	83	19	5796	13	57
DPHIL7w	24497	150	22	24497	15	66

Table 3.1: The table shows the number of objects in memory at program completion for four input files.

Total Objects Represented

	TaskMap	LabelList	Label	TaskVector	LabelMap	ActionSet
DPHIL5w	1379	17821	18163	1379	1379	13367
DPHIL5	9204	119546	115494	9204	9204	71685
DPHIL6w	5796	86792	90776	5796	5796	65456
DPHIL7w	24497	416252	443978	24497	24497	315084

Table 3.2: The table shows the number of objects stored of the same object types and input files, but with the implementation that does not use UCOW objects. This is the number of objects that would be represented by handles in the UCOW implementation.

Memory in megabytes

	normal	UCOW	ratio
DPHIL5w	2.0	0.61	3.3
DPHIL5	11	2.8	3.9
DPHIL6w	9.3	2.1	4.4
DPHIL7w	44	8.7	5.1

Table 3.3: Dynamic memory requirements for the two versions of START.

TaskVectors. Likewise, each Cstate contains exactly one TaskMap and one LabelList. Because Cstates are the top level objects and the algorithm ensures only one copy of each, entire TaskMaps are rarely shared, and are never shared for these inputs. Two TaskMaps can have the same TaskVector but different LabelMaps. This is the case for DPHIL5. Also notice in this table the small number of LabelMaps for all the inputs. The LabelMap is one of the largest objects used by START, but often several TaskMaps will share the same LabelMap. This demonstrates the fact that a large number of objects can be built from a small number of sub-objects. In comparison to the relatively small object counts in Table 3.1, Table 3.2 shows how many objects of the different types are actually represented. Without handles, this is the count of the number of objects stored in memory. With handles, this is the number of objects represented by the handles. Due to the object sharing that handles perform, this number can be much larger number than the corresponding numbers from Table 3.1.

The running times of the two version of START on the sample programs are given in

	Running time		
	normal	UCOW	speedup
DPHIL5w	35	21	1.7
DPHIL5	213	155	1.4
DPHIL6w	215	128	1.7
DPHIL7w	1285	856	1.5

Table 3.4: Timing measurements in CPU seconds for the two versions of START.

Table 3.4 and the dynamic memory usage is given in Table 3.3. At least in this application, the speed increases due to the use of UCOW objects are not as impressive as the memory savings, but they are still significant. An older version of START did not using hashing to match Cstates in the main loop but instead searched a list. Do to the large number of comparisons required for each Cstate, the UCOW implementation ran much faster than the non-UCOW implementation. The most recent version of START does use hashing to match Cstates, so the performance difference between the UCOW and non-UCOW versions is not so dramatic.

3.2 Puzzle Solver

In addition to research projects, these classes have uses for other applications. One application where these classes have been useful is a simple group theory problem solver. The problem is, given a two-dimensional matrix of numbered cards, find the shortest sequence of rotations of columns or rows that result in the desired transformation. For example, given a 4 by 3 matrix, swap cards (1,1) and (2,1) in the minimal number of moves.

A program was written to solve this problem using bi-directional search [Pohl, 1971]. Queue and hash table template classes were used as well as the UCOW handle class already described. Given the starting configuration and the goal configuration, the algorithm uses a breadth-first search from both configurations and prints the solution if the frontiers of the two searches meet. It simulates a parallel bi-directional search by alternating between the two searches each time a new frontier has been generated. It takes about 15 seconds to run. An earlier implementation that performed a uni-directional search from the initial configuration took several days to run.

4. Conclusion

This thesis presents C++ classes to implement unique copy-on-write objects. These classes will reduce memory usage and running times for appropriate applications. Applications that need to create and subsequently compare large hierarchical objects will benefit from the uniqueness property that allows very fast matching and object sharing. Applications that simply want fast copies and fast copy-and-modify will benefit from the copy-on-write property. Combining the uniqueness property with the copy-on-write property allows UCOW objects to significantly increase memory sharing in many applications. The savings can be dramatic, as we have seen from comparing the two different versions of START on the Dining Philosophers input files.

In addition to being useful, these classes are also easy to use. Using C++ templates allows the code to be more easily reused. Possible future work would be to make the classes more efficient and even easier to use. The hash table class could be changed to resize dynamically. Also, the UCOW handle class could be built on top of a class that only provides copy-on-write handles. This way the user can do without the overhead of unique objects if they are not needed. It might also be interesting to see what the implementation would look like if it uses virtual function polymorphism instead of templates. As in the COW objects described in [White, 1991], implementing UCOW objects without templates would require some sort of virtual `dup` function to copy objects. All the template implementation needs to do is use the copy constructor. At the time of this writing, template support in most C++ compilers is still being refined. Hopefully in the future improved template support will lead to many new useful classes based on templates.

Acknowledgments

This work[1] would not have been possible without the help of several people. I want to thank my thesis reading committee of Prof. David Helmbold, Prof. Charlie McDowell, and Prof. Ira Pohl, for their constructive comments, insights, and expertise. Special thanks go to Charlie for his idea to use C++ and object-oriented design for the project and for introducing me to templates and to Ira for answering even the most esoteric C++ questions without pause. Michelle Abram was an indispensable resource for my LaTeX questions. Daniel Edelson, our resident C++ expert, answered many questions about LaTeX and C++. Prof. John Carroll read an early draft of the thesis and provided useful comments. Finally, Prof. Darrell Long was very helpful in proofreading drafts.

[1] This technical report is a slightly revised version of the M.S. thesis of the same title.

References

[Cooper, 1989] M. C. Cooper. Formal hierarchical object models for fast template matching. *The Computer Journal*, 32(4):351–361, 1989.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Intoduction To Algorithms*. MIT Press and McGraw-Hill, 1990.

[Helmbold and McDowell, 1990] D. P. Helmbold and C. E. McDowell. Computing reachable states of parallel programs. Technical report, U. of Calif. Santa Cruz, UCSC-CRL-90-58, 1990.

[Hoare, 1978] C. A. R. Hoare. Communicating sequential processes. *CACM.*, 21(8):666–77, 1978.

[Nelson and Ousterhout, 1988] Michael Nelson and John Ousterhout. Copy-on-write for Sprite. In *USENIX Conference Proceedings*, pages 187–201. USENIX, 1988.

[Pohl, 1971] Ira Pohl. Bi-directional search. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, chapter 9, pages 127–140. American Elsevier, New York, 1971.

[Stroustrup and Ellis, 1991] Bjarne Stroustrup and Margaret A. Ellis. *The Annotated C++ Reference Manual*. Addison-Wesley, 1991.

[White, 1991] Ronald G. White. Copy-on-write objects for C++. *The C Users Journal*, August 1991.

[Winston and Horn, 1981] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison-Wesley, 1981.

Appendix A. Class Interfaces

A.1 List

```
List<T>::List(void);
```

This is the default constructor, which creates an empty list.

```
List<T>::List(const List &);
```

This is the copy constructor. It creates a copy of the list by making copies of each element using the appropriate copy method.

```
T List<T>::first(void) const;
```

This function returns the first element in the list. Notice that the element is returned as the return value of the function, not by a pointer or reference parameter. The reason is because C++ does not allow references or pointers to references, and the type `T` could be a reference. This function should not be called if the list is empty. It should raise an exception in that case, but few C++ compilers support exceptions at this time.

```
int List<T>::hasData(T elem);
```

This function returns `TRUE` if the list contains an element that matches `elem` and `FALSE` otherwise. A match is detected using the operator `==` between each element and `elem`.

```
T List<T>::findMatch(T elem, int &found);
```

This function is similar to hasData except that it returns the first matching element that it finds. It sets `found` to `TRUE` if it finds a matching element, otherwise `FALSE`. If no element is found, the return value is undefined, but is actually just `elem` to save the time constructing a new `T`. This is another good place to use exceptions.

```
int List<T>::empty(void) const;
```

This functions returns whether the list is empty or not. It should be called before using first and dequeue.

```
void List<T>::prepend(T elem);
```

This functions quickly prepends `elem` to the list.

```
T List<T>::dequeue(void);
```

This functions removes the first element from the list and returns it. It should not be called on an empty list. Exceptions could also be used here.

A.2 ListElem

```
ListElem<T>::ListElem(List<T> &);
```

This is the constructor. Given a list of type `T`, it creates an iterator for elements of type `T`.

```
void ListElem<T>::reset(void);
```

This function resets the iterator, so that the next call to `advance` will set the iterator to the first element. It is needed to iterate over the list multiple times.

```
int ListElem<T>::advance(void);
```

This function advances the iterator to the next element, or the first element if the iterator has just been created or reset. It returns `TRUE` if the iterator is currently referencing an element and `FALSE` if it has been advanced past the last element or the list is empty.

```
ListElem<T>::operator T (void);
```

This functions returns the current element being referenced by the iterator. It should not be called until advance has returned `TRUE`.

A.3 LRU List

```
LRUList<T>::LRUList(void);
```

This is the `LRUList` constructor. It takes no arguments.

```
void LRUList<T>::remove(DListNode<T> *);
```

Given a pointer to a node in the list, this member function removes the node from the list.

```
DListNode<T> *LRUList<T>::findMatch(T);
```

This member function searches the list for the given element. If the element is found, it returns a pointer to its list node, otherwise it returns a null pointer.

```
void LRUList<T>::add(T);
```

Given a data element, this member function adds it to the front of the list.

A.4 Hash Table

```
HashTable<T>::HashTable(int size);
```

This constructor creates a new hash table of the given size. The size is a hint to the hash table of how many elements it expects to store.

```
T HashTable<T>::findMatch(T elem, int &found)
```

This function has the same semantics as the list function of the same name. It sets `found` to whether an element matching `elem` was found, and returns the element if it was found.

```
void HashTable<T>::add(T elem);
```

This function adds the given element to the hash table. It does not check for duplicates. That job is left to the user.

```
int HashTable<T>::remove(T elem);
```

This function will look for an element matching the given element and remove it from the hash table if it finds it. It returns `TRUE` if the element was found, `FALSE` otherwise.

A.5 Registered

```
void Registered::ref(void);
```

Increment the reference count.

```
void Registered::unref(void);
```

Decrement the reference count.

```
int Registered::refs(void) const;
```

Return the reference count;

```
void Registered::set_registered(void);
```

Set the object as being registered.

```
void Registered::unset_registered(void);
```

Set the object as being unregistered.

```
int Registered::is_registered(void) const;
```

Return whether the object is registered.

```
unsigned hash(void) const;
```

This function should be defined by the derived class. It is used by the hash table when objects are registered. Two objects that are equal must return the same value for this function.

A.6 Handle

A.6.1 Public

```
Handle<T>::Handle(void);
```

This is the default Handle constructor. It is necessary for creating a dynamic array of Handles. This constructor creates a null handle. A null handle cannot be accessed because it does not refer to an object; it can only be assigned a new value. Null handles are useful when creating arrays of handles. Extra work is avoided by not creating an object for each handle, since each handle will be assigned a new value anyway.

```
Handle<T>::Handle(const Handle &);
```

This is a fast copy constructor that work much like the assignment operator. Since it is creating a new handle, there is no existing object to destroy. It sets the new handle to point at the object being copied and increments its reference count.

```
Handle<T>::operator = (const Handle &) const;
```

This is a fast assignment operator. It removes a reference from the destination object, destroying it if necessary. Then it adds a reference to the source object, and points the destination handle to it.

```
Handle<T>::operator == (const Handle &) const;
```

This is a fast equivalence operator that compares objects by their addresses, since there is only one copy of each object in memory.[1] This operator can be inherited by a derived class.

```
const T &Handle<T>::ro_obj(void) const;
T &Handle<T>::rw_obj(void);
```

These two functions allow access to the object. The first, `ro_obj`, returns a read-only reference to the shared object. The second, `rw_obj`, returns a writable copy of the object. This copy will not be shared by any other handle, so it can be freely modified.

```
int Handle<T>::null(void) const;
```

This functions tests whether the handle is null or not. The default handle constructor creates a null handle.

A.6.2 Private

```
void Handle<T>::unref(void);
```

This is a private member function that decrements the reference count on the object, and does all the needed cleanup if the reference count goes to zero. It is called by modify and share, but is not needed by the user of this class. A fast assignment operator can be written using `unref`, a pointer assignment, and `ref`.

```
void Handle<T>::modify(void);
```

[1] except dirty objects, which are registered (cleaned up) before the pointers are compared.

This is an important function. It sets up the object handle so that it can be safely modified, which means the reference count needs to be one. It makes a copy of the object if the reference count is greater than one, making the handle point to the new copy.

```
static void Handle<T>::share(T *&obj);
```

This function is the partner of modify. After modify has been called to make an object writable, this function will be called at a later time to cleanup the dirty object by registering it. It will check if the dirty object matches an existing object of that type. If so, the object handle will be changed to share the existing object, otherwise the new object will be added to the list of existing objects.

```
static void Handle<T>::cleanup(void);
```

This function cleans up the dirty object. If there is no current dirty object it does nothing. The private member function `share` is called to cleanup the dirty object.

Appendix B. Source Code – Interfaces

B.1 List

```
#ifndef LIST_H
#define LIST_H

#ifndef NULL
#include <stddef.h>
#endif

template<class T> struct ListNode {
    T data;
    ListNode *nextPtr;

    ListNode(T, ListNode * = NULL);
};

// forward declaration
template<class T> class ListElem;

template<class T> class List {
public:
    List(void);
    List(const List &);
    List &operator = (const List &);

    T first(void) const;
    int hasData(T) const;
    T findMatch(T, int &found) const;
    int empty(void) const;
    void prepend(T);
    T dequeue(void);
    ~List(void);
protected:
    ListNode<T> *firstPtr;
private:
    void release(void);
    void copy(const List &);

    friend ListElem<T>;
};

template<class T> class ListElem {
public:
    ListElem(List<T> &);
    void reset(void);
```

```
    int advance(void);
    operator T (void);
private:
    ListNode<T> *first, *current;
};

// List with remove
template<class T> class RList : public List<T> {
public:
    int remove(T);
};

template<class T> class SortedList : public RList<T> {
public:
    SortedList(void);
    SortedList(const SortedList &);
    // preserve sorted order
    void add(T);
    int remove(T);
    int operator == (const SortedList &) const;
private:
    int length;
};

#endif // LIST_H
```

B.2 Hash Table

```
#ifndef HASH_H
#define HASH_H

#ifndef NULL
#include <stddef.h>
#endif

#include "lru.h"

template<class T> class HashTable {
private:
    unsigned size;
    LRUList<T> *table;
public:
    HashTable(unsigned size);
    T findMatch(T, int &found);
    void add(T);
    int remove(T);
    ~HashTable(void);
```

```
};

#endif // HASH_H
```

B.3 LRU List

```
#ifndef LRU_H
#define LRU_H

#ifndef NULL
#include <stddef.h>
#endif

template<class T> struct DListNode {
    T data;
    DListNode *prevPtr, *nextPtr;

    inline DListNode(T, DListNode *prev = NULL, DListNode *next = NULL);
};

template<class T> class LRUList {
private:
    DListNode<T> *firstPtr, *lastPtr;
public:
    LRUList(void);
    void remove(DListNode<T> *);
    DListNode<T> *findMatch(T);
    void add(T d);
};

#endif // LRU_H
```

B.4 Handle

```
#ifndef HANDLE_H
#define HANDLE_H

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

class Referenced {
public:
    Referenced(void);
    Referenced(const Referenced &);
    void ref(void);
    void unref(void);
    int refs(void) const;
```

```
private:
    int ref_cnt;
    void operator = (const Referenced &);
};

class Registered : public Referenced {
public:
    Registered(void);
    Registered(const Registered &);
    int is_registered(void) const;
    void set_registered(void);
    void unset_registered(void);
    unsigned hash(void) const;
private:
    int registered;
    void operator = (const Registered &);
};

template<class UniqueType> struct HandleClassData {
    unsigned long num_objs;
    unsigned long num_null, num_handles;
    HashTable<UniqueType &> objs;
    UniqueType **dirty_obj;

    HandleClassData(unsigned);
};

template<class UniqueType> class Handle {
public:
    static HandleClassData<UniqueType> class_data;

    Handle(void);
    Handle(const UniqueType &o);
    Handle(const Handle &u);
    static void stats(ostream &cout);
    int null(void) const;
    const UniqueType &ro_obj(void) const;
    UniqueType &rw_obj(void);
    Handle &operator = (Handle &u);
    operator == (const Handle &u) const;
    ~Handle(void);
private:
    UniqueType *obj;

    void unref(void);
    // setup obj for modification, make it writable
    void modify(void);
    // search for identical obj and share it
```

```
    static void share(UniqueType *&obj);
    static void cleanup(void);
};

#endif // HANDLE_H
```

Appendix C. Source Code – Implementations

C.1 List

```
#include <assert.h>
#include "list.h"

template<class T> inline
ListNode<T>::ListNode(T d, ListNode<T> *n) : data(d)
{
    nextPtr = n;
}

template<class T> inline
List<T>::List(void)
{
    firstPtr = NULL;
}

template<class T> inline
List<T>::List(const List<T> &l)
{
    copy(l);
}

template<class T> inline
List<T> &List<T>::operator = (const List<T> &l)
{
    release();
    copy(l);
    return *this;
}

template<class T> inline
T List<T>::first(void) const
{
    return firstPtr->data;
}

template<class T> inline
int List<T>::empty(void) const
{
    return firstPtr == NULL;
}

template<class T> inline
void List<T>::prepend(T d)
```

```
{
    firstPtr = new ListNode<T>(d, firstPtr);
}

template<class T> inline
T List<T>::dequeue(void)
{
    assert(firstPtr != NULL); // should throw exception
    T d = firstPtr->data;
    ListNode<T> *next = firstPtr->nextPtr;
    delete firstPtr;
    firstPtr = next;
    return d;
}

template<class T> inline
List<T>::~List(void)
{
    release();
}

template<class T> inline
ListElem<T>::ListElem(List<T> &l)
{
    first = l.firstPtr;
    current = NULL;
}

template<class T> inline
void ListElem<T>::reset(void)
{
    current = NULL;
}

template<class T> inline
int ListElem<T>::advance(void)
{
    if (current)
        current = current->nextPtr;
    else
        current = first;
    return current != NULL;
}

template<class T> inline
ListElem<T>::operator T (void)
{
    return current->data;
```

```
}

// List with remove

template<class T>
void List<T>::release(void)
{
    ListNode<T> *tmp = firstPtr, *next;
    while (tmp) {
        next = tmp->nextPtr;
        delete tmp;
        tmp = next;
    }
}

template<class T>
void List<T>::copy(const List<T> &l)
{
    ListNode<T> *src = l.firstPtr, **dst = &firstPtr;
    while (src) {
        *dst = new ListNode<T>(src->data);
        dst = &(*dst)->nextPtr;
        src = src->nextPtr;
    }
    *dst = NULL;
}

template<class T>
int List<T>::hasData(T d) const
{
    ListNode<T> *tmp = firstPtr;
    while (tmp && !(d == tmp->data)) {
        tmp = tmp->nextPtr;
    }
    return tmp != NULL;
}

template<class T>
T List<T>::findMatch(T d, int &found) const
{
    ListNode<T> *tmp = firstPtr;
    while (tmp && !(d == tmp->data)) {
        tmp = tmp->nextPtr;
    }
    if (tmp) {
        found = 1;
        return tmp->data;
    } else {
```

```
        found = 0;
        // should throw exception
        return d;
    }
}

// List with remove

template<class T>
int RList<T>::remove(T d)
{
    int count = 0;
    ListNode<T> **dst = &firstPtr;
    while (*dst) {
        if (d == (*dst)->data) {
            ListNode<T> *tmp = (*dst)->nextPtr;
            delete *dst;
            *dst = tmp;
            ++count;
        } else
            dst = &(*dst)->nextPtr;
    }
    return count;
}

template<class T> inline
SortedList<T>::SortedList(void)
{
    length = 0;
}

template<class T> inline
SortedList<T>::SortedList(const SortedList<T> &l) : RList<T>(l)
{
    length = l.length;
}

// preserve sorted order
template<class T>
void SortedList<T>::add(T d)
{
    ListNode<T> **dst = &firstPtr;
    while (*dst && (*dst)->data < d) {
        dst = &(*dst)->nextPtr;
    }
    *dst = new ListNode<T>(d, *dst);
    ++length;
}
```

```
template<class T>
int SortedList<T>::remove(T d)
{
    int count = 0;
    ListNode<T> **dst = &firstPtr;
    while (*dst && (*dst)->data < d)
        dst = &(*dst)->nextPtr;
    while (*dst && d == (*dst)->data) {
        ListNode<T> *tmp = (*dst)->nextPtr;
        delete *dst;
        *dst = tmp;
        ++count;
        --length;
    }
    return count;
}

template<class T>
int SortedList<T>::operator == (const SortedList<T> &s) const
{
    if (length != s.length) return 0;
    ListNode<T> *x = firstPtr, *y = s.firstPtr;
    while (x && x->data == y->data) {
        x = x->nextPtr;
        y = y->nextPtr;
    }
    return x == NULL;
}
```

C.2 Hash Table

```
#include "hash.h"

template<class T>
HashTable<T>::HashTable(unsigned s)
{
    size = s;
    table = new LRUList<T>[size];
}

template<class T>
T HashTable<T>::findMatch(T d, int &found)
{
    DListNode<T> *p = table[d.hash() % size].findMatch(d);
    if (p) {
        found = 1;
```

```
        return p->data;
    } else {
        found = 0;
        return d;
    }
}

template<class T>
void HashTable<T>::add(T d)
{
    table[d.hash() % size].add(d);
}

template<class T>
int HashTable<T>::remove(T d)
{
    LRUList<T> &list = table[d.hash() % size];
    DListNode<T> *p = list.findMatch(d);
    if (p) {
        list.remove(p);
        delete p;
        return 1;
    } else {
        return 0;
    }
}

template<class T>
HashTable<T>::~HashTable(void)
{
    delete[] table;
}
```

C.3 LRU List

```
#include "lru.h"

template<class T>
DListNode<T>::DListNode(T d, DListNode<T> *p, DListNode<T> *n) : data(d)
{
    prevPtr = p; nextPtr = n;
    if (p)
        p->nextPtr = this;
    if (n)
        n->prevPtr = this;
}
```

```
template<class T>
LRUList<T>::LRUList(void)
{
    firstPtr = lastPtr = NULL;
}

template<class T>
void LRUList<T>::remove(DListNode<T> *l)
{
    if (l->prevPtr)
        l->prevPtr->nextPtr = l->nextPtr;
    else
        firstPtr = l->nextPtr;
    if (l->nextPtr)
        l->nextPtr->prevPtr = l->prevPtr;
    else
        lastPtr = l->prevPtr;
    l->prevPtr = l->nextPtr = NULL;
}

template<class T>
DListNode<T> *LRUList<T>::findMatch(T d)
{
    DListNode<T> *p;
    if (p = firstPtr) {
        do {
            if (d == p->data) {
                if (p != firstPtr) {
                    remove(p);
                    p->prevPtr = NULL;
                    p->nextPtr = firstPtr;
                    firstPtr = firstPtr->prevPtr = p;
                }
                return p;
            }
        } while (p = p->nextPtr);
    }
    return NULL;
}

template<class T>
void LRUList<T>::add(T d)
{
    firstPtr = new DListNode<T>(d, NULL, firstPtr);
    if (firstPtr->nextPtr)
        firstPtr->nextPtr->prevPtr = firstPtr;
    if (lastPtr == NULL)
        lastPtr = firstPtr;
```

```
}
```

C.4 Handle

```
#include "handle.h"
#include <assert.h>

Referenced::Referenced(void)
{
    ref_cnt = 1;
}
Referenced::Referenced(const Referenced &)
{
    ref_cnt = 1;
}
void Referenced::ref(void)
{
    ++ref_cnt;
}
void Referenced::unref(void)
{
    --ref_cnt;
    assert(ref_cnt >= 0);
}
int Referenced::refs(void) const
{
    return ref_cnt;
}

Registered::Registered(void)
{
    registered = 0;
}
Registered::Registered(const Registered &)
{
    registered = 0;
}
int Registered::is_registered(void) const
{
    return registered;
}
void Registered::set_registered(void)
{
    registered = 1;
}
void Registered::unset_registered(void)
{
```

```
    registered = 0;
}

template<class UniqueType>
HandleClassData<UniqueType>::HandleClassData(unsigned size) : objs(size)
{
    num_objs = num_handles = num_null = 0;
    dirty_obj = NULL;
}

template<class UniqueType>
Handle<UniqueType>::Handle(void)
{
    ++class_data.num_handles;
    ++class_data.num_null;
    obj = NULL;
}

template<class UniqueType>
Handle<UniqueType>::Handle(const UniqueType &o)
{
    if (class_data.dirty_obj) cleanup();
    obj = new UniqueType(o);
    ++class_data.num_handles;
    class_data.dirty_obj = &obj;
}

template<class UniqueType>
Handle<UniqueType>::Handle(const Handle<UniqueType> &u)
{
    if (class_data.dirty_obj) cleanup();
    if (obj = u.obj) {
        obj->ref();
    } else {
        ++class_data.num_null;
    }
    ++class_data.num_handles;
}

template<class UniqueType>
void Handle<UniqueType>::stats(ostream &cout)
{
    cout << class_data.num_objs << " unique objects\n";
    cout << class_data.num_handles << " handles";
    if (class_data.num_null) {
        cout << " (" << class_data.num_null << " null)\n";
    } else {
        cout << '\n';
```

```
    }
}

template<class UniqueType>
int Handle<UniqueType>::null(void) const
{
    return obj == NULL;
}

template<class UniqueType>
const UniqueType &Handle<UniqueType>::ro_obj(void) const
{
    assert(obj != NULL);
    if (class_data.dirty_obj) cleanup();  // should not be necessary
    return *obj;
}

template<class UniqueType>
UniqueType &Handle<UniqueType>::rw_obj(void)
{
    assert(obj != NULL);
    if (class_data.dirty_obj && class_data.dirty_obj != &obj)
        cleanup();
    if (obj->is_registered())
        modify();
    return *obj;
}

template<class UniqueType>
Handle<UniqueType> &Handle<UniqueType>::operator = (Handle<UniqueType> &u)
{
    if (class_data.dirty_obj) cleanup();
    if (obj != u.obj) {
        unref();
        if (obj = u.obj) {
            obj->ref();
        } else {
            ++class_data.num_null;
        }
    }
    return *this;
}

template<class UniqueType>
Handle<UniqueType>::operator == (const Handle<UniqueType> &u) const
{
    if (class_data.dirty_obj) cleanup();
    return obj == u.obj;
```

```
}

template<class UniqueType>
Handle<UniqueType>::~Handle(void)
{
    unref();
    if (class_data.dirty_obj) cleanup();
    assert(class_data.num_handles > 0);
    --class_data.num_handles;
}

template<class UniqueType>
void Handle<UniqueType>::unref(void)
{
    if (obj) {
        obj->unref();
        if (class_data.dirty_obj == &obj)
            class_data.dirty_obj = NULL;
        if (obj->refs() == 0) {
            if (obj->is_registered()) {
                int stat = class_data.objs.remove(*obj);
                assert(stat);
                assert(class_data.num_objs > 0);
                --class_data.num_objs;
            }
            delete obj;
        }
        obj = NULL;
    } else {
        assert(class_data.num_null > 0);
        --class_data.num_null;
    }
}

// setup obj for modification (copy-on-write)
template<class UniqueType>
void Handle<UniqueType>::modify(void)
{
    assert(class_data.dirty_obj == NULL);
    assert(obj->refs() > 0);
    if (obj->refs() == 1) {
        if (obj->is_registered()) {
            int stat = class_data.objs.remove(*obj);
            assert(stat);
            assert(class_data.num_objs > 0);
            --class_data.num_objs;
            obj->unset_registered();
        }
```

```
    } else {
        obj->unref();
        obj = new UniqueType(*obj);
    }
    class_data.dirty_obj = &obj;
}

// search for identical obj and share it
template<class UniqueType>
void Handle<UniqueType>::share(UniqueType *&obj)
{
    assert(obj->refs() == 1);
    assert(!obj->is_registered());
    int found;
    UniqueType &o = class_data.objs.findMatch(*obj, found);
    if (found) {
        assert(obj != &o);
        delete obj;
        obj = &o;
        obj->ref();
    } else {
        class_data.objs.add(*obj);
        obj->set_registered();
        ++class_data.num_objs;
    }
}

template<class UniqueType>
void Handle<UniqueType>::cleanup(void)
{
    if (class_data.dirty_obj) {
        assert(!(*class_data.dirty_obj)->is_registered());
        UniqueType **obj = class_data.dirty_obj;
        class_data.dirty_obj = NULL;
        share(*obj);
    }
}
```

Appendix D. Classes from START

D.1 SyncGraph

An *SyncGraph* is a Sync Graph. It contains nodes which represent synchronization events in a parallel program. Each Sync Node (SyncNode) in the SyncGraph has information particular to its type. For example, a dispatch node contains a task id and function id, and a post node contains an event id.

The SyncGraph constructor is used to build the SyncGraph. It takes a filename as its only argument. A print member function is also used for debugging. It outputs the Sync Graph in a human-readable format, calling the `print` virtual function for each SyncNode.

D.2 SyncNode

An *SyncNode* in an SyncGraph corresponds to a synchronization event in the program. There is a different class derived from the base SyncNode class for each type of synchronization event.

Each SyncNode has a virtual function getOp that uniquely identifies its type. Depending on the type of SyncNode, it can also have members functions that return information such as a task-id, function begin node, or event name.

D.3 Action

An action is a pointer to a node in the Sync Graph (SyncNode). For each type of SyncNode there is a corresponding derived Action class.

The key member function of the Action class is its `exec` virtual function. This function fires the action on a `TaskInCstate`, appending to a `CstateSet` any new *Cstates* it generates. Some action types can perform automatic optimizations, an example of which is "overwrite". An action such as non-clearing wait can detect that the new state can be merged with the previous state, since only an ActionSet was changed. In this case, the new Cstate will replace (overwrite) the current Cstate, resulting in a more general state because `addAction` will be performed instead of `changeAction`.

D.4 List<T>

This is a simple generic list class that allows you to easily build singly-linked lists of different types. It does not provide every possible list operation. Noticeably missing is a function to remove an element from the list. That function was left out because it was not needed by the application, but for the sake of completeness the source code contains a new class `RList<T>` derived from `List<T>` that provides a `remove` member function. There is also a `SortedList<T>` class that inherits from RList.

The type parameter `T` can be a class, a pointer to a class, or a reference to a class. If `T` is a pointer or a reference, the list will not store copies of each element. Both save space because a copy is not made, but using a reference has the advantage over a pointer that when the time comes to compare elements, the actual elements will be compared, not their addresses.

```
#include <iostream.h>
#include "list.h"

void func(void)
{
    List<int> list;
    list.prepend(5);
    list.prepend(3);

    ListElem<int> iter(list);
    while (iter.advance()) {
        cout << (int)iter << '\n';
    }
}
```

Figure D.1: Example using the ListElem list iterator

`List` has a default constructor for creating an empty list, and a copy constructor for copying a list. It has a member function `first` that returns the first element in the list. This function is not strictly necessary, since the ListElem iterator described next can be used to do the same thing. `List` also has member functions `hasData` to check if a certain element is in the list and `findMatch` to find a matching element in the list. `HasData` is also redundant as `findMatch` returns the same information but also returns the matched element. To add an element to the front of the list, there is a member function `prepend`. To remove the first element, there is `dequeue`. These have the same semantics as `push` and `pop` for a stack, but are not called those names because some other class might want to inherit from this class and add `append` to add an element to the end of the list. If `prepend` was called `push`, the new class would have `push` and `append` instead of `prepend` and `append`. Section A.1 describes the interface in some detail.

The `ListElem<T>` class is the iterator for the generic list class. It allows the elements of the list to be examined sequentially. The full description of the interface is in Section A.2. To iterate over the elements of a list, first create a `ListElem<T>` with a `List<T>` as the argument. Then while the `advance` member function returns `TRUE`, pull out each element using the conversion operator (Figure D.1).

Another possible interface would have functions first and next returning pointers to elements, with first returning `NULL` if the list is empty and next returning `NULL` after the last element has been returned. This would require disallowing references, as described above for the list class. The interface using `advance` has the advantage that you can step through the list, returning only those elements asked for explicitly. For example, you might want to examine only every other element.

D.5 SortedList<T>

SortedList is a parameterized-type (template) class that implements a sorted list of elements of type `T`. It is derived from the `List<T>` class and adds the member functions `add`, `remove`, and `operator ==`. `Add` will insert an element, keeping the list sorted, based

on the `<` operator of the element class. `Remove` will delete an element if it is in the list, using the `==` operator to find it.

D.6 LabelList

A `LabelList` is a sorted list of Labels. A Label is a class that contains a string.

`LabelList` inherits from SortedList<Label>, but also adds a function `hasLabel` as a more symbolic interface to `hasData`.

D.7 TaskMap

A `TaskMap` is a type of graph used to represent the state of the tasks. There is a source node for each task and a sink node for each source node. The number of tasks in the TaskMap determines its *width*. The number of levels (rows) in the graph is called the height. The graph is implemented as two objects, the `TaskVector` and the `LabelMap`. The TaskVector is implemented as a one-dimensional array of `ActionSet`s, and the `LabelMap` is implemented as a two-dimensional array of `NodeRep`s. The TaskVector forms the first row of the graph and the LabelMap forms the remaining rows, resulting in a rectangular graph of nodes. The TaskVector is conceptually connected to the top of the LabelMap, so that the node in column n of the TaskVector has an edge to the node in column n in the first row of the LabelMap. Each node in the TaskVector contains an `ActionSet` object, while each node in the LabelMap contains a `NodeRep` object. A node can only have edges to nodes that are on an adjacent level (row). Also, the connected components of the sub-graph consisting of the nodes of two adjacent levels and the edges between those two levels must be complete bipartite graphs.

Each NodeRep in the LabelMap contains a LabelList and information describing its connections to other nodes (edges).

The Unify member function is passed an Action, and modifies the TaskMap so that all tasks with that Action in their ActionSets will be clustered together in the same CBG (complete bipartite graph). This modification involves the equivalent of a SplitNode followed by a MakeCluster [Helmbold and McDowell, 1990].

GetActionSet returns a pointer to the ActionSet for the given task. GetCBG returns the CBG to which the task belongs (only the lower CBG is useful). Num˙tasks returns how many tasks are represented by the TaskMap (also known as the width). The boolean function identicalTask returns whether two tasks can be regarded as identical. In this case, identical means they are in the same CBG, have the same labels, and have the same ActionSet. Edge is a boolean function that says if there is an edge between two nodes. This is currently only used by the draw function. AddTask modifies the TaskMap by adding a new task with the given Action as its ActionSet. ChangeAction replaces the ActionSet of a task with an ActionSet containing just the given Action. HasAction returns whether the ActionSet of the given task contains the given Action. AddAction adds an Action to the ActionSet of a task. ChangeAction, addAction, and hasAction are forwarded to the corresponding ActionSet function. AddLabel adds a label to the LabelList for a task. The label will appear at the top level, along with the ActionSet. Likewise, removeLabel removes a label. Print and draw perform ASCII and X Windows dumps of the TaskMap, respectively.

Operations that specify a task do so with a TaskMarker, which is a handle for a task. Usually this fact is hidden by the use of a TaskInCstate, which implicitly specifies a specific task in a Cstate.

D.8 NodeRep

Besides a LabelList, a `NodeRep` keeps track of its edges with two CBG identifiers. One CBG identifier is for edges to the next higher level and the other is for edges to the next lower level. A CBG identifier is currently implemented as the minimum column number of any node in the CBG. The LabelList holds a list of Labels containing task-ids.

A NodeRep has functions `addLabel`, `removeLabel`, and `getLabel` for manipulating its LabelList. It also defines `operator ==` for comparisons with other nodes. Equivalence requires equivalent label lists and CBG numbers.

D.9 ActionSet

An `ActionSet` is a set of Actions, and is implemented as a sorted list. It should be based on SortedList, but is currently its own class. The list is recursive; an ActionSet contains an Action and a pointer to another ActionSet. This implementation implies functions `next` for advancing to the next ActionSet and `getAction` for returning the Action of the current ActionSet. In addition, `hasAction` and `append` mimic the behavior of `hasData` and `add` of the SortedList class. `ChangeAction` destroys the entire ActionSet and replaces it with a new ActionSet containing the given Action. Note: Actions above refer to Action pointers in the actual implementation.

D.10 Cstate

A `Cstate` contains a LabelList and a TaskMap. Currently, the Labels in the LabelList only contain information about posted events. I call these meta-labels to distinguish them from labels in the TaskMap.

Cstate hands off to TaskMap the member functions `removeTaskLabel` and `addTask`. The member functions `addMetaLabel` and `removeMetaLabel` are delegated to the `add` and `remove` functions of the LabelList. The constructor `Cstate(const Cstate &, Action *)` employs the TaskMap constructor followed by a call to `TaskMap::unify` with the given Action.

D.11 TaskMarker

A `TaskMarker` is a type that is guaranteed to identify a task in a Cstate (or TaskMap) even across copies. It is currently implemented as an integer. The TaskInCstate class uses a TaskMarker for specifying a task in a TaskMap.

D.12 TaskInCstate

This class is used for referring to a particular task in a Cstate. A `TaskInCstate` is a (Cstate, TaskMarker) pair, so all the member functions operator on the Cstate using the TaskMarker to specify the task. In the current implementation, however, all of the calls go directly through the Cstate to the TaskMap. These functions are `getActionSet`, `changeAction`, `addAction`, `hasAction`, and `addLabel`, and are simply forwarded to the corresponding function in the TaskMap.

D.13 HistNode

A `HistNode` is a node in the CHG whose job is to hold a Cstate. It also contains predecessor and successor list for storing the graph structure.

The first implementation of HistNode provided a function `getCstate` for accessing the Cstate which it contains, but the current implementation actually has HistNode as a derived class of Cstate, which turns out to be quite useful for some of the list and comparison operations. The function `addSuccessor` will add successors to the current node.

D.14 CHG

A `CHG` is a graph of HistNodes. All of the work done on a CHG is done in the constructor. The constructor is passed an SyncGraph, and builds a CHG based on that Sync Graph. To access the nodes, a function `firstNode` is provided for returning the "root" node.