Limit Point Systems, Inc.

# Introduction

This document shows how to use the basic features of the SheafSystem; it is intended as an introduction for beginners. The C++ examples in the document are available as source code in the examples subdirectory of the SheafSystemProgrammersGuide module and the reader is encouraged to build and execute the examples along with reading the text. The examples are numbered and the source for example N is in exampleN.cc. There are a few compilation and execution examples in the text, these are given in Linux using the csh shell, Gnu C++, and Gnu make.

# What you'll need

To take full advantage of this document, you'll need a few things in addition to the document itself, namely:

* an installed copy of the SheafSystemProgrammersGuide module, which includes the examples,
* a C++ compiler,
* a web browser, and
* an installed copy of the SheafSystem.

# The SheafSystem installation

The SheafSystem installer installs all the files of the SheafSystem in a directory tree. We will have to refer to the root of this directory tree repeatedly, so to simplify the notation, we'll let <sheaf\_dir> refer to the full path to the root directory of the installation, for instance:

<sheaf\_dir> = /usr/local/SheafSystem

Wherever you see <sheaf\_dir> in this document, mentally replace it with the full path to your SheafSystem installation.

The installation includes 4 configurations of the libraries: Debug-contracts, Debug-no-contracts, Release-contracts, and Release-no-contracts. The "Debug" configurations are unoptimized and contain symbol information for use by interactive debuggers such as gdb. The "Release" configurations are optimized and contain no debugging information. We'll describe "Contracts" below. Generally speaking, the Release configurations are higher performance that the Debug configurations and the no-contracts configurations are much faster than the contract configurations.

The examples will compile and execute with any configuration, but we will always use the Debug-contracts configuration in the text below.

# Getting started

## PartSpace metaphor

The PartSpace document describes the fundamental concepts of the SheafSystem in non-mathematical terms using the common notion of basic and composite parts, tables, and table schema. This document assumes the reader is familiar with the PartSpace metaphor.

## Sheaf tables

As described in the PartSpace document, a SheafSystem database is a collection of tables. Each table is equipped with a covering relation graph describing the lattice order of its rows and another graph describing the lattice order of its columns. Each such object table has an associated table called its schema table and the row graph of the schema table defines the column graph of the object table. A member of the row lattice is represented by a node in the row graph. A member also has a corresponding row in the table if and only if it is a basic part, a join irreducible member (JIM) in the row lattice.

There are 3 special tables. the primitive schema table, the primitives table, and the namespace table. The primitives schema table terminates the schema recursion, it is its own schema table. The primitives table describes each primitive type supported by the system.

## Namespaces

A namespace table is a special table in each database that serves as a container and table of contents for all the other tables. The SheafSystem includes 3 predefined namespace types: the sheaves\_namespace, the fiber\_bundles\_namespace, and the geometry\_namespace. Each of these predefines the sheaf schema for the C++ types defined in the sheaf, fiber\_bundle, and geometry components, respectively. (The fields component doesn't have its own schema).

Creating an instance of a namespace is typically the first thing a client must do to use the SheafSystem, so we start with an example of how to do that using the most basic namespace, sheaves\_namespace. This example will also cover the basic mechanics of compiling and linking with the SheafSystem.

### Example : Hello, Sheaf

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

// Create a standard sheaves namespace.

sheaves\_namespace lns("Hello-sheaf");

// Write its name to cout.

cout << lns.name();

return 0;

}

This code is in the SheafSystemProgrammersGuide module in examples/sheaf/example1.cc along with a Makefile:

#

# Full path to your C++ compiler, for instance /usr/bin/g++

#

CXX = /usr/bin/g++

#

# Full path the SheafSystem installation include and library directories

#

SHEAF\_INC\_DIR =<sheaf\_dir>/include

SHEAF\_LIB\_DIR = <sheaf\_dir>/Debug-contracts/lib

example1: example1.cc

$(CXX) -o example1 -I$(SHEAF\_INC\_DIR) -L$(SHEAF\_LIB\_DIR) example1.cc -lsheaves

To compile and link the example, you first have to configure the Makefile to your installation by setting the 3 variables CXX, SHEAF\_INC\_DIR, and SHEAF\_LIB\_DIR to the actual values for your installation. Then we can compile and link by:

>make example1

This command will compile example1.cc and link it with the shared library libsheaves.so to create an executable example1 in the current directory. We have to tell the dynamic loader where to find the shared library by setting the environment variable LD\_LIBRARY\_PATH to contain the path to the SheafSystem library directory, that is, the same value we set SHEAF\_LIB\_DIR to in the Makefile, for instance:

>setenv LD\_LIBRARY\_PATH <sheaf\_dir>/Debug-contracts/lib

Now we can execute the example:

>./example1

Hello-sheaf

That's the basic mechanics of creating an application with the SheafSystem. We've created a sheaves\_namespace in this example, but before we can do much with it, we need to learn a few programming patterns that the SheafSystem uses repeatedly.

# Programming patterns

There are a few design features shared by all the classes in the SheafSystem. In this section we will give a quick introduction to the most ubiquitous of these patterns. We'll introduce some more patterns later, as we need them, and also go into some of these initial patterns in more detail.

## Design by contract

The sheaf system is implemented using the "design by contract" programming paradigm. We'll cover the essentials of the method and how they are used in the SheafSystem. For a more detailed introduction, see the excellent book Design By Contract, by Example by Richard Mitchell and Jim McKim.

When using design by contract, each class is equipped with an invariant, a set of assertions that must be true at any time control returns the client. (The invariant is not defined when control is within a member function of the class.) Every member function is equipped with preconditions and postconditions. The preconditions are assertions that must be true when control enters the member function; the postconditions must be true when control leaves the member function. The "contract" in "design by contract" is between the client and the member function: if the client guarantees the preconditions are true, the member function ensures the invariant and the postconditions are true.

The invariant, precondition, and postcondition assertions are specified using "invariance", "require", and "ensure" macros, respectively, in the source code. If contracts are enabled when the library is compiled, these clauses will be evaluated as part of the execution of the member functions. If the conditions specified in the clauses are not true, execution throws an exception with an error message, which usually terminates the program.

The contracts are extremely useful for detecting improper use of the classes and member functions and are thus an important debugging tool. Once client code is correct, the contracts can be disabled to improve efficiency.

The SheafSystem Debug-contracts and release-contracts configurations are compiled with contracts enabled. The Debug-no-contracts and Release-no-contracts are compiled with contracts disabled.

The contracts are also published as an essential part of the reference documentation and are critical to using the sheaf system correctly. Let's look at the reference documentation for the sheaves\_namespace constructor we used in example1. The reference documentation is generated in html, so you can open it with your browser. The main page is <sheaf\_dir>/documentation/C++/index.html. If you browse to the documentation for class sheaves\_namespace and click on the constructor sheaves\_namespace(const string& xname), you'll find:

sheaf::sheaves\_namespace::sheaves\_namespace ( const string & *xname* )

Creates a sheaves namespace with name xname.

Precondition

* poset\_path::is\_valid\_name(xname)

Postcondition

* [invariant()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#a952742bdad45c56c22fd9509a00e9c07)
* [name()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#aec09bcd260a52a459c8a35ae5bc1bef5) == xname
* !in\_jim\_edit\_mode()
* [host()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#adc8f6d6d2b952a6842a1d09de75bff9a) == 0
* !index().[is\_valid()](http://192.168.4.199/comp-tutorial-dev-4/df/d4b/namespacesheaf.html#a3dd8f96a360e1b63c6caa744e5ccf7b3)
* [index()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#a9a283b1819bc8e75b212bff26fc645b0).same\_scope(member\_hub\_id\_space(false))
* [has\_standard\_subposet\_ct()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#af5786ce90013ec6e72dbacd9b67e1c13)
* [current\_namespace()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#ae8ca3a11bc745cf0b275a70ab71b2d70) == this
* [state\_is\_not\_read\_accessible()](http://192.168.4.199/comp-tutorial-dev-4/d1/d3c/classsheaf_1_1read__write__monitor__handle.html#adc32a6090b2df1e5673444d5170539f3)

So what does this tell us? The precondition:

* poset\_path::is\_valid\_name(xname)

tells us exactly what conditions the argument xname has to satisfy if we want this call to the constructor to work correctly, namely is\_valid\_name(xname) has to be true. Well, what does that take? If we look up poset\_path∷is\_valid\_name we find:

static bool sheaf::poset\_path::is\_valid\_name( const string &  xname )

True if xname is not empty and contains only name legal characters.

Postcondition

* result == (!xname.[empty()](http://192.168.4.199/comp-tutorial-dev-4/d0/d38/classsheaf_1_1poset__path.html#a05ee8f14bcc22701b551059341f16749) && (xname.find\_first\_not\_of([name\_legal\_characters()](http://192.168.4.199/comp-tutorial-dev-4/d0/d38/classsheaf_1_1poset__path.html#a34019af3a5bee6f34d3ec2c2657a8671)) == string::npos))

So xname can't be empty and can't contain any characters not in name\_legal\_characters(). If we click on name\_legal\_characters we find:

static const string & sheaf::poset\_path::name\_legal\_characters( )

The characters a name is allowed to contain.

Postcondition

* result == "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789\_ -,.=+()\*:?"

So xname has to be non-empty and contain only the above characters.

If xname satisfies these conditions, which it does in example1, then the postcondition gives a great deal of information about what state the sheaves\_namespace object is in after construction.

The first postcondition is:

* [invariant()](http://192.168.4.199/comp-tutorial-dev-4/d4/d91/classsheaf_1_1namespace__poset.html#a952742bdad45c56c22fd9509a00e9c07)

that is, the invariant has to be satisfied. As we said above, this is an implicit postcondition of every member function, even if we don't explicitly provide it as part of the contract. So what does this mean for sheaves\_namespace? Well, click on invariant() to find:

virtual bool sheaf::namespace\_poset::invariant ( ) const

Class invariant.

Invariant

* poset\_state\_handle::invariant()
* host() == 0
* !index().is\_valid()
* !is\_external()
* is\_attached() ? primitives().is\_attached() : true
* is\_attached() ? (primitives().index() == PRIMITIVES\_INDEX) : true
* state\_is\_read\_accessible() ? primitives().state\_is\_read\_accessible() : true
* is\_attached() ? primitives\_schema().is\_attached() : true
* is\_attached() ? (primitives\_schema().index() == PRIMITIVES\_SCHEMA\_INDEX) : true
* state\_is\_read\_accessible() ? primitives\_schema().state\_is\_read\_accessible() : true

Sheaves\_namespace inherits namespace\_poset and doesn't override the invariant, which is a virtual function, so the invariant of sheaves\_namespace is the invariant of namespace\_poset. The invariant in a derived class must be at least as strong as the invariant in the base space, so the invariant of namespace\_poset calls the invariant of its base class, poset\_state\_handle. Beyond whatever poset\_state\_handle∷invariant() ensures, the namespace\_poset invariant ensures several properties of the data members, primitives() and primitives\_schema() in particular.

As this invariant shows, the conditional expression

* x ? y : true

appears frequently in the contracts, so it is worth describing in more detail. As an assertion, this expression can be read "x implies y", that is, x can be either true or false, but if is true, then y must be true as well. If x is false, there is no condition on y.

The reader is encouraged to examine the poset\_state\_handle invariant to learn what additional invariances sheaves\_namespace has inherited, but we'll move on to the rest of the postcondition of the constructor. The next postcondition is one you'd expect:

* [name()](http://192.168.4.199/comp-tutorial-dev-4/d0/d99/classsheaf_1_1poset__state__handle.html#aec09bcd260a52a459c8a35ae5bc1bef5) == xname

that is, the name of the namespace is the name we gave it.

The remainder of the postconditions ensure various arcane properties of the namespace that we're not very interested in right now. But when your tackling a tough debugging problem, any of these may be just the piece of information you need!

The power of the design by contract method comes from the great amount of detailed information contained in the assertions and two further properties. First, if contracts are turned on, that is if you are using either the Debug-contracts or Release-contracts configuration of the library, the pre- and post-conditions of a function are executed whenever the function is called. Second the contracts exhibited in the documentation are extracted directly from the source code. The combination of the two allows you to reason about the behavior of the code with great confidence while designing, programming, and especially while debugging.

So what happens if the contract for a member function is not satisfied? Let's find out by trying to create a sheaves\_namespace without a name.

### Example : contract for sheaves\_namespace constructor.

#include "sheaves\_namespace.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example2:" << endl;

// Attempt to create a standard sheaves namespace

// with an empty name. This violates the preconditions

// of the constructor and will throw an exception and abort.

sheaves\_namespace lns("");

// Done.

return 0;

}

If we compile and run this, assuming we still have LD\_LIBRARY\_PATH set from running example1, we get:

>make example2

>./example2

terminate called after throwing an instance of 'std::logic\_error'

what(): 'poset\_path::is\_valid\_name(xname)' in file namespace\_poset.cc at line 1941

Abort

The error message tells you exactly what assertion failed. If you're debugging, you can walk back up the stack from where the exception was actually thrown to the assertion that failed and inspect local variables, for instance xname, to determine what went wrong.

## Concurrency control

One of the attractive features of the sheaf data model is that its mathematical formalism provides a natural language for describing concurrency and parallelism. The sheaf system libraries were designed for concurrent programming using an access control mechanism based on the monitor design pattern. Currently, this mechanism is only partially implemented and the SheafSystem libraries are delivered with the access control mechanism disabled. Programmers nevertheless must be aware of certain aspects of the access control mechanism, which we describe in this section. More complete examples are included in .

Access to every table is controlled. A client thread can have no access, read access, or read-write access. At any given time, either no client has access, exactly one client has read-write access, or one or more clients have read access. Before reading or writing a table or any of its members, a client must request read access or read-write access, respectively. After accessing the table, the client must release access. If a client requests read access and another client already has write access, or vice versa, the request blocks until the other client releases the conflicting access.

The concurrency control mechanism is "enforced" through precondition clauses in the table member functions. In order to make concurrency control apparent to the client and avoid dead lock, the library routines do not themselves request or release access without the client knowing it. Instead, they "publish" their access requirements as preconditions and let the client control the access.

For instance, in example 1 we invoked sheaves\_namespace∷name(). Consulting the reference documentation, we find for the name() member function:

virtual string sheaf::namespace\_poset::name() const

The name of this namespace.

Precondition

* state\_is\_read\_accessible()

So, if the access control mechanism is enabled, the client must request read access, invoke the name function, and release access:

lns.get\_read\_access();

cout << lns.name() << endl;

lns.release\_access();

Getting and releasing access can be a tedious programming chore. Furthermore, it is syntactically impossible in some cases, for instance within a pre- or post-condition clause. So many member functions offer an "auto-access" option. These routines will automatically get and release the access they need, if the client allows it by setting an auto-access argument to true. If invoked with the auto-access argument false, the client must get the required access before making the call. These routines also publish their access requirements as preconditions. For instance, the auto-access version of the name function is:

virtual string sheaf::namespace\_poset::name( bool  xauto\_access ) const

The name of this namespace.

Precondition

* state\_is\_auto\_read\_accessible(xauto\_access)

Using this version of the name function, the client need only invoke the function with argument "true":

cout << lns->name(true) << endl;

The function will request read access, get the name, release access, and return the name.

When the access control mechanism is disabled, the client always has read-write access and neither requesting nor releasing access is necessary. Functions with an auto-access argument can be called with either true or false, either will work. However, the access control mechanism doesn't quite disappear from the programmer's view. The auto-access signatures are still present and the access requirements still appear as preconditions in the contracts.

## Handles and states

The Sheaf System is object-oriented, so the client interacts with the library by manipulating the various objects presented by the library interface. Lattice members are a prime example. Many of the objects exported by the interface are not however stored as explict objects internally. Both memory and performance efficiency often require that such objects be implicit - stored as disjoint data items in bulk arrays. The problem of how to present an externally explicit object interface to an internally implict object is a common software design problem and several similar design patterns - flyweight, proxy, surrogate, etc, have been developed to address this problem. In the Sheaf System, we call such a surrogate object a handle and the internal data it accesses is called its state.

For the most part, the distinction between handles and states is an implementation detail that the client needs to be only vaguely aware of. The client uses the handle object as if it were stored internally without worrying about the internal details. But there is one aspect the client has to be aware of: the client has to somehow get a handle to the desired object and when finished with it the client may have to explicitly release it.

There are two basic patterns. In the first pattern, some object has a data member which is a handle and it provides an accessor to this data member. For instance, sheaves\_namespace, like every lattice, has a top member. This member is represented by a data member which is a handle and sheaves\_namespace exports an accessor:

sheaves\_namespace lns;

namespace\_poset\_member& ltop = lns.top()

The namespace object allocated and owns the handle. The client need not and should not worry about releasing or otherwise deallocating the handle.

The second pattern addresses the more general case in which the number of handles the client needs and what states they should be attached to is not known at compile time. In order to support efficient allocation and deallocation of handles, the system maintains pools of handles which the client can "borrow", use, and return. For instance, we'll see in the next section that index spaces are accessed via handles and the client can get a handle from the appropriate index space family:

index\_space\_handle& lids =

lns.member\_id\_spaces(true).get\_handle("member\_poset\_id\_space");

When accessed in this way, the handle must be released when the client is finished with it:

lns.member\_id\_spaces(true).release\_handle(lids, true);

How does a client know whether to release a handle or not? Simple, if you got the handle by calling get\_<whatever>, you need to release it by calling release\_<whatever>. Release if and only if get!



Figure : Hub and spoke architecture of an index space family.

# Index spaces and scoped indices, part 1

The members of the row lattice of a table (and hence the members in the column lattice as well) are identified by integer ids. Subsets of the members are very important in the SheafSystem and it is frequently useful to generate a special purpose index scheme for a given subset. Such an index scheme is referred to as an "index space", or "id space" for short. The SheafSystem provides extensive support for defining and using id spaces.

## Index spaces and iterators.

More specifically, an index space is a set of integer ids. The system supports the creation and use of a family of index spaces. The fundamental id space of the family is the member id space - the ids automatically generated for the nodes in the row graph. This index space is called the hub id space because the index space family has a hub and spoke architecture as shown in . As you can see from the diagram, there are several different kinds of id space and even two hub id spaces, the "unglued" and "glued" versions. We'll describe this structure and how to create and modify id spaces later. For the moment, hub id space means unglued hub id space and you should just think of each id space on the rim as a way of indexing some subset of the hub id space, with each spoke representing a map. We'll focus on the basics of how to use id spaces.

As one might expect, the principal use for a member id is to access the features of the member the id refers to. The principal use of a member id space is to iterate over all the members in the subset defined by the id space. Let's look at an example.

## Example : Iterates over the member hub id space.

#include "hub\_index\_space\_handle.h"

#include "index\_space\_iterator.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example3:" << endl;

// Create a standard sheaves namespace.

sheaves\_namespace lns("Example3");

// Get a handle for the member hub id space.

const index\_space\_handle& lmbr\_ids = lns.member\_hub\_id\_space(true);

// Find out how many ids are in the id space.

cout << lmbr\_ids.name();

cout << " has " << lmbr\_ids.ct() << " ids.";

cout << endl;

// Id spaces are defined as half open intervals, like STL iterators.

// If the space is "gathered", begin() == 0 and end() = ct().

// If the space is not gathered, it's "scattered".

cout << "begining at " << lmbr\_ids.begin();

cout << " and ending at " << lmbr\_ids.end();

cout << " " << (lmbr\_ids.is\_gathered() ? "gathered" : "scattered");

cout << endl;

// The main thing one does with id spaces is iterate over them.

// Get an iterator from the iterator pool.

index\_space\_iterator& lmbr\_itr = lmbr\_ids.get\_iterator();

cout << endl << "Iterate:" << endl;

while(!lmbr\_itr.is\_done())

{

// The current member of the iteration is "pod()".

// "POD" is an ISO C++ acronym for "plain old data".

// A pod is an ordinary integer id, in contrast with

// a "scoped\_index" id, to be discussed shortly.

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

// Use the id to get the member name.

// Member name requires a hub id, but since we're using

// the hub id space, pod and hub pod are the same thing.

cout << "id: " << lpod;

cout << " hub id: " << lmbr\_itr.hub\_pod();

cout << " name: " << lns.member\_name(lpod, true);

cout << (lns.is\_jim(lpod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// You can reuse an iterator by resetting it.

lmbr\_itr.reset();

cout << endl << "Reiterate:" << endl;

while(!lmbr\_itr.is\_done())

{

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

cout << "id: " << lpod;

cout << " hub id: " << lmbr\_itr.hub\_pod();

cout << " name: " << lns.member\_name(lpod, true);

cout << (lns.is\_jim(lpod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// If you got an id space or iterator from the pool with get\_

// you have to return it to the pool with release\_.

lmbr\_ids.release\_iterator(lmbr\_itr);

// The id space itself is a data member of the id space family,

// we didn't get it from the pool with get\_, so we don't have to

// release it.

// Exit:

return 0;

}

If we execute example3 we get:

>./example3

SheafSystemProgrammersGuide Example3:

\_\_hub has 6 ids.

begining at 0 and ending at 6 gathered

Iterate:

id: 0 hub id: 0 name: bottom is a jrm.

id: 1 hub id: 1 name: top is a jrm.

id: 2 hub id: 2 name: primitives\_schema is a jim.

id: 3 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 4 hub id: 4 name: primitives is a jim.

id: 5 hub id: 5 name: schema definitions is a jrm.

Reiterate:

id: 0 hub id: 0 name: bottom is a jrm.

id: 1 hub id: 1 name: top is a jrm.

id: 2 hub id: 2 name: primitives\_schema is a jim.

id: 3 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 4 hub id: 4 name: primitives is a jim.

id: 5 hub id: 5 name: schema definitions is a jrm.

## Id maps and scoped ids.

As we said above, id spaces are used for indexing subsets. For instance, in a namespace, the member poset id space indexes just the jims, which represent the member posets - the other posets contained in the namespace. There may be several or even many id spaces available in a practical setting. Various member functions may require an index to be in a particular id space, most commonly in the hub id space. The id maps associated with the spokes in the id space family provide the mechanism for translating between id spaces.

Every id space has a map to the (unglued) hub id space. The index\_space\_handle class provides member functions for mapping ids between the id space and the hub id space:

pod\_type hub\_pod (pod\_type xid) const

The pod index in the unglued hub id space equivalent to xid in this id space; synonym for unglued\_hub\_pod(pod\_type).

and

pod\_type pod (pod\_type xid) const

The pod index in this space equivalent to xid in the hub id space.

Using these functions we can map between id spaces. For instance, if id1 is an id in id\_space1 and id\_space2 is a different id space, then

pod\_type id2\_eqv\_1 = id\_space2.pod(id\_space1.hub\_pod(id1));

is the id in id\_space2 that identifies the same member identified by id1 in id\_space1, if such an equivalent member exists. The postcondition for the pod(pod\_type) function is:

* !is\_valid(result) || contains(result)

So if id\_space2 does not have an equivalent member, id2\_eqv\_1 is assigned an invalid value. The value

sheaf::pod\_index\_type sheaf::invalid\_pod\_index()

The invalid pod index value.

is reserved as a "null" value for index types. It is currently set to numeric\_limits<pod\_index\_type>::max(), but that may change and I only mention that so you will recognize it if you see it in a print out or in the debugger. It should be used as an opaque value. A pod type can be tested for validity using:

bool sheaf::is\_valid(sheaf::pod\_index\_type xpod\_index)

True if an only if xpod\_index is valid.

Using the pod and hub\_pod functions, the programmer can map ids between id spaces. But it can be tedious. Worse, it may be difficult or impossible for a programmer to track just what id space a given index is in. The scoped\_index class provides a convenient mechanism for both managing the connection between an id and the space it belongs to and for automatically mapping between id spaces. We call the id space an id belongs to the scope of the id. A scoped\_index is a a pair (id, scope). Most member functions that require an id as input are available in two signatures; one signature that takes a pod id and one that takes a scoped id. One can use a scoped id, once it has been initialized, without worrying what scope it is in; any function that accepts a scoped id will translate it to the scope it requires. We'll see more complex examples of mapping between id spaces later, for now let's redo example3 using the member poset id space, the hub id space, id maps and scoped ids.

## Example : Iterates over the member poset id space.

#include "index\_space\_handle.h"

#include "index\_space\_iterator.h"

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example4:" << endl;

sheaves\_namespace lns("Example4");

// Get a handle for the member poset id space;

// has one member for each poset in the namespace.

const index\_space\_handle& lmbr\_ids =

lns.get\_member\_poset\_id\_space(true);

// Print out the same info we did for the hub id space.

cout << lmbr\_ids.name();

cout << " has " << lmbr\_ids.ct() << " ids.";

cout << endl;

cout << "begining at " << lmbr\_ids.begin();

cout << " and ending at " << lmbr\_ids.end();

cout << " " << (lmbr\_ids.is\_gathered() ? "gathered" : "scattered");

cout << endl;

index\_space\_iterator& lmbr\_itr = lmbr\_ids.get\_iterator();

cout << endl << "Iterate:" << endl;

while(!lmbr\_itr.is\_done())

{

index\_space\_iterator::pod\_type lpod = lmbr\_itr.pod();

// Use the id to get the member name.

// Member name requires a hub id which we can get in two ways.

// The id space will use the map from the id space to the hub

// to translate any id in the id space to its equivalent in the

// hub:

index\_space\_iterator::pod\_type lhub\_pod = lmbr\_ids.hub\_pod(lpod);

// The iterator can provide the hub id equivalent for the current

// id, and it can be faster because for some id space types it can

// avoid the map lookup.

lhub\_pod = lmbr\_itr.hub\_pod();

cout << "id: " << lpod;

cout << " hub id: " << lhub\_pod;

cout << " name: " << lns.member\_name(lhub\_pod, true);

cout << (lns.is\_jim(lhub\_pod) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

// Most member functions are available with two signatures, one

// that takes a pod\_index\_type and one that takes a scoped\_index.

// If you don't want to think about what the scope for an argument

// should be, you can use the scoped\_index signature.

// Create a scoped id with scope = member poset id space.

scoped\_index lscoped\_id(lmbr\_ids);

// The value sheaf::invalid\_pod\_index() is reserved as a

// "null" value for index types. It is currently set to

// numeric\_limits<pod\_index\_type>::max(), but don't count on it.

cout << endl << "sheaf::invalid\_pod\_index()= ";

cout << sheaf::invalid\_pod\_index() << endl;

// When a scoped id is created without a specific pod value,

// it is invalid by default.

cout << "lscoped\_id= " << lscoped\_id;

cout << " is\_valid() ";

cout << boolalpha << lscoped\_id.is\_valid() << noboolalpha;

cout << endl;

// Reset the iterator and re-iterate using

// the scoped\_index signature for member\_name.

lmbr\_itr.reset();

cout << endl << "Reiterate:" << endl;

while(!lmbr\_itr.is\_done())

{

// Set the scoped id for the current member of the iteration.

lscoped\_id.put\_pod(lmbr\_itr.pod());

// Assigment is overloaded, so you can also say:

lscoped\_id = lmbr\_itr.pod();

// Use the scoped\_index signature to get the member name.

cout << "scoped\_id: " << lscoped\_id;

cout << " name: " << lns.member\_name(lscoped\_id, true);

cout << (lns.is\_jim(lscoped\_id) ? " is a jim." : " is a jrm.");

cout << endl;

// Move on.

lmbr\_itr.next();

}

lmbr\_ids.release\_iterator(lmbr\_itr);

// Exit:

return 0;

}

When we run example4 we get:

>./example4

SheafSystemProgrammersGuide Example4:

member\_poset\_id\_space has 3 ids.

begining at 0 and ending at 3 gathered

Iterate:

id: 0 hub id: 2 name: primitives\_schema is a jim.

id: 1 hub id: 3 name: namespace\_poset\_schema is a jim.

id: 2 hub id: 4 name: primitives is a jim.

sheaf::invalid\_pod\_index()= 2147483647

lscoped\_id= (2, 2147483647) is\_valid() false

Reiterate:

scoped\_id: (2, 0) name: primitives\_schema is a jim.

scoped\_id: (2, 1) name: namespace\_poset\_schema is a jim.

scoped\_id: (2, 2) name: primitives is a jim.

Note the value of the invalid id in the above output; we'll see it again shortly.

# Storage\_agent

We've talked about the notion of a SheafSystem database, so there must be some way to make a namespace persistent and indeed there is. Persistent storage is managed by the storage\_agent class. A storage\_agent makes it particularly easy to save an entire namespace to disk, as we show in the next example.

## Example : Write a namespace to a file

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

#include "storage\_agent.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example5:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example5");

// Write the namespace to a file.

storage\_agent lsa("example5.hdf");

lsa.write\_entire(lns);

// Exit:

return 0;

}

If we build and run example5, it writes the file named in the storage\_agent constructior, "example5.hdf".

>./example5

>ls \*.hdf

example5.hdf

We'll see shortly how we can view the contents of this file.

# Viewing Namespaces

Once we have a namespace, we'd like to know what it contains. We've already seen how to iterate over the members of the namespace and display their names. Now we'll look at 3 ways that are easier and provide a lot more information.

## Stream insertion operator

The base class for namespaces, namespace\_poset, has a stream instertion operator for writing the contents of the namespace to a stream. The insertion operator is most commonly used for dumping a namesapce to cout for debugging purposes.

### Example : Write namespace to cout

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example6:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example6");

// Write the namespace to cout.

cout << lns << endl;

// Exit:

return 0;

}

When we execute example6, it creates quite a lost of output, even for sheaves\_namespace, which is as close to being empty as a namespace can get. We won't include it here, but the reader should examine the file example6.cout in the same directory with example6.cc.

For each poset in the namespace, including the namespace itself, the stream insertion operator prints information about the row graph, the subposets of the row graph, and the table. We'll learn more about how to interpret all this information as we go along.

## The dump\_shf utility

The SheafSystem provides the dump\_shf utility for reading a sheaf file and dumping its contents to cout using the stream insertion operator. So if you've written a sheaf file, as we did in example 5, then we can view its contents easily.

### Example : View namespace with dump\_sheaf

>dump\_shf example5.hdf

## The SheafScope interactive file browser

The SheafScope is another SheafSystem utility. It provides an interactive, graphical browser for sheaf files.

### Example : View namespace with SheafScope

>java SheafScope example5

# Posets

## Table or part space or lattice or poset?

We've so far talked about a sheaf database being a collection of sheaf tables. In the Part Space tutorial we talked about sheaf tables as part spaces and in the Analysis and Design tutorial we revealed that a sheaf table could be thought as either a poset-ordered or lattice-ordered relation. So which is it? The answer of course, is (e) all of the above. But when it comes to naming classes, we had to pick one. The one we picked was poset. Most of the classes that implement sheaf tables are posets of some kind or another. The most commonly used type is class poset. The type that is used to represent meshes in the fiber bundles component is base\_space\_poset. The abstract base class for all poset types is poset\_state\_handle.

|  |
| --- |
| **Technology Roadmap**: Poset\_state\_handle, the abstract base class for all poset types, is as its name says, a handle. But this is a historical artifact. There is no longer any reason for it to be a handle, the state of a poset is an explicit object, and the various handle features are in fact protected so they can't be used. The poset\_state\_handle class will be renamed and refactored in a future release to eliminate the handle features. |

## Creating posets

A namespace is a factory for posets. Posets are created using the member function template new\_member\_poset<T>, where T is the type of poset to create:

T& new\_member\_poset (const string &xname,

const poset\_path &xschema\_path,

const arg\_list &xargs,

bool xauto\_access)

Creates a new poset with name xname, schema specified by xschema\_path, and table attributes initialzied by xargs.

But first, we have to have a schema if we want to create a poset. Typically, this means we have to create a schema poset before creating an object poset. We'll see how to create a schema poset shortly, but we can avoid creating a schema poset if the poset we want to create has only a single attribute. Instead, we can use a member of the primitives poset that every namespace has as a schema. We'll do that to get started.

### Poset\_path

The schema for a poset is a member of a schema poset and a member can be identified by a path. A poset path is similar to a file path, but has only two elements, a poset name and a member name. We can create a path by specifying the poset name and member name separately:

poset\_path lschema\_path("primitives", "INT");

or by specifying a complete path, like a file path:

poset\_path lschema\_path("primitives\_poset/INT");

### Arg\_list

Some types of poset can not be default constructed, so the factory method requires an argument list for the constructor. Each poset type provides a static function make\_args(...) that sets up the arg\_list. Every poset type has such a function, even if it can be default constructed. In that case, the make\_args function itself has no arguments and the arg\_list it creates is empty. This is in fact the case for the ordinary poset class we want to construct:

arg\_list largs = poset::make\_args();

The arg\_list class has a constructor that takes a string, so if you know the arg\_list is empty can also just pass an empty string for the arg\_list, but we'll do it the general way for now.

So now all we need is a name and we can construct the poset. We'll call it "simple\_poset":

poset& lposet = lns.new\_member\_poset<poset>("simple\_poset",

lschema\_path, largs, true);

## Accessing posets

Once you've created a poset and have a reference to it, you can access its features. For instance, you can find out what its id is:

cout << lposet.index().hub\_pod() << endl;

All poset types have a stream insertion operator, so once you have access to a poset, you can use the stream insertion operator to print it out:

cout << lposet << endl;

We'll do more with posets, like creating members, in the next section.

In the mean time, what if a poset already exists, how do you get a reference to it? You can get a reference to a poset by id, which is available in two variants, pod and scoped\_index:

poset\_state\_handle & member\_poset (pod\_index\_type xindex, bool xauto\_access) const

The poset\_state\_handle object referred to by the member with hub id xindex.

poset\_state\_handle & member\_poset(const scoped\_index &xindex, bool xauto\_access) const

The poset\_state\_handle object referred to by the member with index xindex.

For instance, assuming that the id of the poset we created above is 5, we can get a reference to it with:

poset\_state\_handle& lpsh1 = lns.member\_poset(5, true);

You can also access it by path. The member name part of the path can be empty or not, only the poset name will be used.

poset\_state\_handle & member\_poset (const poset\_path &xpath, bool xauto\_access) const

The poset\_state\_handle object referred to by the member with name xpath.poset\_name().

Furthermore, since poset\_path has a constructor that takes a string literal, you can specify the path as a string literal:

poset\_state\_handle& lpsh2 = lns.member\_poset("simple\_poset", true);

Poset\_state\_handle is the abstract base class for all poset types. If you know the specific type of a poset and want a reference to that type, all three of the above signatures are also available in a templated version:

P& member\_poset(pod\_index\_type xindex, bool xauto\_access) const;

The poset\_state\_handle object referred to by the member with hub id xindex dynamically cast to type P&.

invoked with bracket notation:

poset& lposet1 = lns.member\_poset<poset>("primitives", true);

## Deleteing posets

Posets live in the namespace. Once you've created one, it stays in the namespace whether you have a reference to it or not. If you want to delete a poset, use the namespace delete\_poset function, available in the same id and path signatures as member\_poset, for instance:

void delete\_poset (const poset\_path &xpath, bool xauto\_access)

Delete the poset with name xpath.poset\_name().

ties the namespace functions we've been discussing into a single example.

## Example : Creating, accessing, and deleting posets.

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide Example9:" << endl;

// Create a namespace.

sheaves\_namespace lns("Example9");

// We use a schema with a single integer attribute.

poset\_path lschema\_path("primitives", "INT");

// The contructor for the ordinary poset class

// doesn't need any arguments.

arg\_list largs = poset::make\_args();

// Create the poset.

poset& lposet = lns.new\_member\_poset<poset>("simple\_poset",

lschema\_path, largs, true);

// Print the poset to cout.

cout << lposet << endl;

// You can get another reference to the poset by id,

// if you know the id:

poset\_state\_handle\* lpsh1 = lns.member\_poset(6, true);

// and by path:

poset\_state\_handle\* lpsh2 = lns.member\_poset("simple\_poset", true);

// Delete the poset by path.

// Invalidates all the above references.

lns.delete\_poset(lposet.path(), true);

// Exit:

return 0;

}

When we run example9, the output is once again a bit lengthy to include here, take a look at example9.cout.

# Poset members

We created a poset in the last section, but it was empty. Well, almost empty. As we discussed in the Part Space tutorial, a sheaf table represents a part space. A part space always has a bottom member// corresponding to the empty assembly of basic parts (jims). It also has a top member, corresponding to the assembly of all the basic parts. We automatically create these two composite parts (jrms) when we create the poset. When the poset has no basic parts, the top is equivalent to the bottom as an assembly, but they are still distinct members.

## Creating join irreducible members

We can't create any interesting jrms until we have some jims, so we'll start by creating some jims. A poset has a special editing mode, called "jim\_edit\_mode", for creating jims. Jrms can be created at any time, but to create a jim you have to put the poset into jim\_edit\_mode. Jim edit mode allows you to directly edit the row graph, creating or deleting members and cover links. Once the jims poset has been defined, you can create composite parts either directly by creating the members and links or algebraicly, using the join and meet operations.

Jim edit mode is off by default, so we have enter jim edit mode to create a jim:

void begin\_jim\_edit\_mode (bool xauto\_access)

Allow editing of jims and jim covering relation.

(All the functions we will discuss in this section are member functions of class poset or its ancestors unless specifically scoped. Click the "List of all members" item in the extreme upper right of the poset class documentation web page to get a listing of all member functions, both direct and inherited.)

You create a jim with the new\_member function:

sheaf::pod\_index\_type new\_member (bool xis\_jim,

poset\_dof\_map \* xdof\_map = 0,

bool xcopy\_dof\_map = false )

Create a disconnected member with is\_jim == xis\_jim. If xdof\_map != 0, the new member uses it for dof storage, otherwise it creates an instance of array\_poset\_dof\_map. WARNING: this routine leaves a disconnected member in the poset and hence leaves the poset in an invalid state. The client must properly link the member created by this routine using new\_link in order to return the poset to a valid state.

|  |
| --- |
| **Technology Roadmap**:."Dof" is an acronym for "degree of freedom" but it means "attribute". A "dof\_map" is a tuple. Both are historical artifacts, originating in early attempts to interpret fields as relational tuples. We expect to replace the "dof" and "dof\_map" nomenclature with the more standard "attribute" and "tuple" nomenclature in some future release. |

To create a new jim, we call new\_member with xis\_jim true and let it create a tuple for the new member by accepting the default values for xdof\_map and xcopy\_dof\_map. Let's create three new jims, corresponding to the basic parts in the line segment example from the Part Space tutorial, Figure 10, which we reproduce in . Well, almost reproduce, has a top member because sheaf tables always have top and bottom members.

The code to create the jims is:

lposet.begin\_jim\_edit\_mode(true);

pod\_index\_type lv0\_pod = lposet.new\_member(true);

pod\_index\_type lv1\_pod = lposet.new\_member(true);

pod\_index\_type ls0\_pod = lposet.new\_member(true);



Figure : Line segment example from Part Space, Figure 10.

## Ordering poset members

You define the ordering relation for the poset by explicitly creating cover links between the jims, using new\_link:

void new\_link( pod\_index\_type xgreater, pod\_index\_type xlesser)

Insert a cover link from greater to lesser (that is, xgreater covers xlesser). WARNING: this routine does not ensure that the link is a cover link, that is, it does not remove redundant or conflicting links. Improper use of this routine can produce inconsistent poset states.

Continuing with the line segment example, the segment member should cover the two vertices:

lposet.new\_link(ls0\_pod, lv0\_pod);

lposet.new\_link(ls0\_pod, lv1\_pod);

Each vertex is an atom, there is no smaller basic part than a vertex, so each vertex should have a cover link to bottom. Similarly, there is no larger part than the segment, so top should cover the segment. We can put these links in explicitly. The id of the top and bottom member are defined in the enumeration sheaf::standard\_member\_index as TOP\_INDEX and BOTTOM\_INDEX, respectively. Or, we can get the id from the top() or bottom() accessors:

lposet.new\_link(TOP\_INDEX, ls0\_pod);

lposet.new\_link(lv0\_pod, BOTTOM\_INDEX);

lposet.new\_link(lv0\_pod, lposet.bottom().index().pod());

However, we don't have to explicitly put the links to top and bottom. Remember that the cover relation graph is a directed graph, with links pointing in the "covers" direction. For a given member, the set of all outgoing links to lesser members is called the lower cover of the member and the set of all incoming links from larger members is called the upper cover. It is an invariant of a lattice that bottom is the only member with an empty lower cover and top is the only member with an empty upper cover. The end\_jim\_edit\_mode function:

void end\_jim\_edit\_mode(bool xensure\_lattice\_invariant = true, bool xauto\_access = true)

Prevent editing of jims and jim order relation.

will enforce the invariant if we request it by setting the xensure\_lattice\_invariant argument to true. In that case, it automatically links anything with an empty lower cover to bottom and links top to anything with an empty upper cover. This requires a search of the graph for empty covers, so it is more efficient in large graphs to do the linking explicitly.

When we're finished creating and linking jims, we leave jim edit mode. We'll just take the default arguments, even though we've already linked everything, it won't make any difference for a tiny graph like this one:

lposet.end\_jim\_edit\_mode();

## Accessing poset members

We've created the basic parts and ordered them, but we haven't set any of their attributes, so let's do that now.

### Member names

We've already seen that every poset member has at least one implicit attribute, it's id, automatically assigned by the system. Every member also has another implicit attribute, supported by the system but not automatically assigned: a name. Any member can be given a name, in fact any member can be given multiple names, but naming is optional. Names are assigned with the put\_member\_name function:

void put\_member\_name(pod\_index\_type xindex, const string & xname,

bool xunique, bool xauto\_access = false )

Make xname a name for the member with hub id xindex; if xunique, make xname the only name.

We'll give all our basic parts the obvious names:

lposet.put\_member\_name(lv0\_pod, "v0", true);

lposet.put\_member\_name(lv1\_pod, "v1", true);

lposet.put\_member\_name(ls0\_pod, "s0", true);

We can retrieve a name with the member\_name function:

string member\_name(pod\_index\_type xindex, bool xauto\_access = false)

A name for the member with hub id xindex.

For instance:

cout << lposet.member\_name(lv0\_pod);

Members can have more the one name. The reader is encouraged to review the other member name functions in the reference documentation for poset\_state\_handle.

### Schema

The explicit attributes of a member are whatever is defined by the schema for the poset. The schema for a poset is a member of a schema poset. A handle for the schema member is available from the poset:

const schema\_poset\_member& schema () const

The schema for this poset (const version).

We'll learn more about member handles shortly. In the meantime, we'll just introduce a few features we need.

Mathematically, each attribute is a component of a tuple and components are traditionally accessed by component id. There is an id space defined for the attributes specified by each schema member. In fact, since the table is partitioned into a row part and a table part for storage efficiency, each schema member defines two id spaces, one for the row attributes and one for the table attributes. We can get either id space from the schema using the dof\_id\_space accessor function:

const index\_space\_handle&

schema\_poset\_member::dof\_id\_space(bool xis\_table\_dofs) const

The table dof (xis\_table\_dof true) or row dof id space for the schema defined by this.

For instance, the row atttribute id space for our example is:

const index\_space\_handle& latt\_id\_space = poset.schema().dof\_id\_space(false);

Now, because we took a short cut defining the schema, we don't have all the attributes of the original example in the Part Space tutorial. All we have is a single integer attribute called "INT". Since we know that the schema has only a single row attribute, it has to be the first id in the id space, so we can get its id with:

pod\_index\_type latt\_pod = lposet.schema().dof\_id\_space(false).begin();

We can also create a scoped id for an attribute:

scoped\_index latt\_id(lposet.schema().dof\_id\_space(false), latt\_pod);

### Member tuple

The attributes are components in the relation tuple of the member, so to access an attribute we need a reference to the tuple or "dof\_map", which we can get with:

poset\_dof\_map&

member\_dof\_map(pod\_index\_type xmbr\_index, bool xrequire\_write\_access)

The dof map associated with the member identified by xmbr\_index (mutable version).

We can get the attribute tuple for the first vertex for instance:

poset\_dof\_map& ltuple = lposet.member\_dof\_map(lv0\_pod, true);

### Member attributes

Once we have the tuple, we can get an attribute value with the dof accessor function:

primitive\_value poset\_dof\_map::dof(pod\_index\_type xdof\_id) const

The dof with name xname.

and set an attribute value with the put\_dof mutator function:

void poset\_dof\_map::put\_dof(pod\_index\_type xdof\_id, const primitive\_value& xdof)

Sets the dof with name xname to xdof.

The accessor and mutator functions are each available in 3 signatures corresponding to specifying the attribute by pod id, scoped id, or name.

### Primitive\_value

The attribute value return by the accessor functions and accepted by the mutator functions is of type primitive\_value, which is essentially a wrapper for any primitive type. It allows a single signature for each function to support any attribute type. Primitive\_value is essentially a union that can store any primitive plus some type information. It also has conversion operators to and from every primitive type that make it easy to put a primitive value in to a primitive\_value and get it back again.

You can put a value into a primitive\_value with either a constructor or an assignment:

primitive\_value lpval(int(0));

lpval = float(1);

and get it back again with an assignment:

float lf = lpval;

You can find out what type of value a primitive\_value is currently holding with the id function by testing it against the id() member of the primitive\_traits template for the appropriate type:

if(lpval.id() == primitive\_traits<float>.id())

{

float lf = lpval;

}

### Setting an attribute

Setting the attribute for the vertex v0 takes three steps: get the tuple, put the attribute in a primitive\_value wrapper, put the attribute in the tuple.

poset\_dof\_map ltuple = lposet.member\_dof\_map(lv0\_pod, true);

primitive\_value lpv(int(0));

ltuple.put\_dof(latt\_pod, lpv);

These three steps have to be done, but they don't have to be done explicitly. Implicit conversions actually make it much simpler:

lposet.member\_dof\_map(lv1\_pod, true).put\_dof(latt\_pod, int(0));

lposet.member\_dof\_map(ls0\_pod, true).put\_dof(latt\_pod, int(1));

We can also set an attribute value using either the scoped id signature:

lposet.member\_dof\_map(lv0\_pod, true).put\_dof(latt\_id, int(0));

or the name signature:

lposet.member\_dof\_map(lv0\_pod, true).put\_dof("INT", int(0));

In practice, you use whichever signature is most convenient.

### Getting an attribute

The implicit conversions work when getting an attribute value as well:

int ldim = lposet.member\_dof\_map(lv0\_pod, false).dof(latt\_pod);

As with the mutator, the accessor works with scoped id or name:

int ldim = lposet.member\_dof\_map(lv0\_pod, false).dof(latt\_id);

int ldim = lposet.member\_dof\_map(lv0\_pod, //false).dof("INT");

## Creating join reducible members

We can create jrms and link them up at any time, whether we're in jim\_edit\_mode or not. We'll finish the poset corresponding to To create a jrm, just set the xis\_jim argument to false in new\_member.

poset\_index\_type lc0\_pod = lposet.new\_member(false);

lposet.put\_member\_name(lc0\_pod, "c0", true);

Then link it up:

lposet.new\_link(ls0\_pod, lc0\_pod);

lposet.new\_link(lc0\_pod, lv0\_pod);

lposet\_new\_link(lc0\_pod, lv1\_pod);

But we can't stop here. We have to make sure the cover relation is indeed a cover relation. The segment no longer covers the two vertices, so we have to remove those links using delete\_link:

void delete\_link (pod\_index\_type xgreater, pod\_index\_type xlesser)

Delete the cover link between xgreater and xlesser.

which is pretty straight forward:

lposet.delete\_link(ls0\_pod, lv0\_pod);

lposet.delete\_link(ls0\_pod, lv1\_pod);

Remember, when you're editing the graph, it's your job to get it right! In particular, it's your job to make sure there are no transitive links (links equivalent to a path) like the two we just removed. An invalid graph can produce subtle errors that are difficult to track down. We'll see shortly that in many cases, you can create and link up a jrm in a single step using the join operation. In that case, the system does all the graph editing and makes sure the graph is valid.

## Creating join equivalent members

What if we create another jrm, let's call it c1, and link it between s0 and c0:

pod\_index\_type lc1\_pod = lposet.new\_member(false);

lposet.delete\_link(ls0\_pod, lc0\_pod);

lposet.new\_link(ls0\_pod, lc1\_pod);

lposet.new\_link(lc1\_pod, lc0\_pod);

The result is shown in , but what does it mean? Is it even legal, since c1 has only a single member in its lower cover, doesn't it have to be a jim?



Figure : A join equivalent member.

Well, to answer the first question, let's think about part space. A jrm is a composite part in the part space metaphor and a composite part is an assembly of basic parts. More specifically, it is the assembly of the basic parts in its down set, the set of all parts below it in the graph. Pretty clearly, the set of basic parts in the down set of our new jrm is exactly the same as the set of basic parts in the down set of c0. So our new jrm is a distinct member, but as an assembly it is identical to c0. In other words, the new jrm is a copy of c0. We say that c1 is join equivalent to c0 and call it a join equivalent member or "jem", pronounced like "gem".

Now for the second question: is it legal? The short answer is yes, but why this is true takes a little bit of explanation. The interested reader can find the full answer in . For those not interested in the mathematical details, feel free to copy members whenever you please, as many times as you please. In fact, in section we'll see some functions that make it easy to make copies.

## Deleting poset members

Deleting a member is the inverse of creating it. You have to unlink it, delete it, and relink the remaining members appropriately. We've already seen how to delete links. To delete a member uses, the delete\_member function:

virtual void delete\_member (pod\_index\_type xindex)

Delete the member with index xindex. Warning: this routine does not delete links; it will leave any links to this member dangling.

So let's delete the jem we just created:

lposet.delete\_link(lc1\_pod, lc0\_pod);

lposet.delete\_link(ls0\_pod, lc1\_pod);

lposet.new\_link(ls0\_pod, lc0\_pod);

lposet.delete\_member(lc1\_pod);

## Example : Reading a sheaf file; manipulating poset members with the poset interface

We've covered the basics of creating, linking, accessing, and deleting poset members using the poset interface. Let's collect everything we've covered together into a single example. We'll learn to read a poset from a file as well.

# Poset member handles

Repeat all the same examples with handles

# Traversing the graph

## Cover id spaces and iterators

## Depth first traversal

# Schema posets

1. Concurrency control examples

The access control mechanism is a work in progress. The control mechanism itself is complete and is implemented both for multiple threads using pthreads and for single threads. When the library is compiled with threads enabled and a client requests read access and another client already has write access, or vice versa, the request blocks until the other client releases the conflicting access. When the library is compiled with threads disabled, requests do not block, they return immediately. The library is currently delivered with threads disabled because the use of threads and concurrency in the library is only partially implemented and not tested. The access control mechanism is disabled by default but can be enabled by the programmer. These examples demonstrate use of the manual and auto-access mechanisms.

* 1. Example A1: manual access control

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide ExampleA1:" << endl;

// Enable concurrency control; must be called

// before any other library call.

read\_write\_monitor::enable\_access\_control();

// Create a standard sheaves namespace.

sheaves\_namespace\* lns = new sheaves\_namespace("ExampleA1");

// Write its name to cout.

// Requires read access to the namespace.

// Be polite, request access.

// If threads are enabled and another thread has

// read-write access, execution will block until it

// releases access. Otherwise, the request will succeed

// immediately.

// You can nest requests as deep as you want, or at least

// until the integer depth counter overflows.

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->get\_read\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->get\_read\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

// Invoke the operation.

cout << lns->name() << endl;

// Be proper, release access so this thread

// or another can get write access.

// Have to match every request with a release.

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->release\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

lns->release\_access();

cout << "request depth " << lns->access\_request\_depth() << endl;

// Delete the namespace, requires read-write access.

// Be polite, request access. If threads are enabled

// and another thread has either read or read-write

// access, execution will block until it releases access.

// Otherwise, the request will succeed immediately.

// This client must not already have read-only access,

// see precondition for details.

lns->get\_read\_write\_access(false);

// Invoke the operation.

delete lns;

// Deletion is the only case where the client

// can not be proper and release access.

// Create another namespace.

lns = new sheaves\_namespace("Example3B");

// Invoking a function that requires access

// without first getting access violates the

// precondition of the function.

// The following will throw an exception and abort.

cout << lns->name() << endl;

return 0;

}

If you compile and run example A1, the output is:

SheafSystemProgrammersGuide ExampleA1:

request depth 0

request depth 1

request depth 2

Example3A

request depth 2

request depth 1

request depth 0

terminate called after throwing an instance of 'std::logic\_error'

what(): 'is\_external() ? name\_space()->state\_is\_read\_accessible() : state\_is\_read\_accessible()' in file poset\_state\_handle.cc at line 1178

Abort

* 1. Example A2: auto-access control

#include "sheaves\_namespace.h"

#include "std\_iostream.h"

using namespace sheaf;

int main( int argc, char\* argv[])

{

cout << "SheafSystemProgrammersGuide ExampleA2:" << endl;

// Enable concurrency control; must be called

// before any other library call.

read\_write\_monitor::enable\_access\_control();

// Create a standard sheaves namespace.

sheaves\_namespace\* lns = new sheaves\_namespace("ExampleA2");

// Write its name to cout.

// Requires read access to the namespace.

// Invoke the auto-access version of the operation with

// auto-access set to true.

// Operation will request and release access as needed.

cout << lns->name(true) << endl;

return 0;

}

If you compile and run example A2, the output is:

SheafSystemProgrammersGuide ExampleA2:

ExampleA2

1. Join equivalent members and lexicographic ordering

Is is legal to have a jrm with a single member in its lower cover? The answer is no, in an ordinary finite distributive lattice as defined in the text books, but yes, in a SheafSystem lattice. In the mathematical view, the lattice is fully instantiated, the order relation fully enumerated, and a member is a jim if and only if it has a single member in its lower cover. So c1 couldn't be a jrm. In this view, the join operator is a query that finds the member which is the least upper bound of the joinands. The Birkhoff representation theorem is a consequence of the order releation and it, in turn, implies two additional facts. First, for lattice members p and p', the set of jims in the downset of p is included in the set of jims of p' if and only if p  p'. Second, the jims in the downset of a join is the union of the jims of the joinands.

But on the computer, we can't afford the memory to fully instantiate all the members of the lattice and enumerate the order relation. So the SheafSystem reverses the mathematical argument. We initially instantiate only the poset of join irreducible members and the covering relationships between them. That is, a member is a jim because we specify that it is when we create it, not because of some property of the cover or order relation. The join operator isn't a query, it's a constructor that creates a jrm and places it in the cover relation so that the set of jims in its down set is the union of the jims of the joinands. The Birkhoff representation theorem is satisfied by construction and the order relation is derived form it: p  p' if and only if the set of jims in the down set of p is included in the jims of p'.

But what if we join the same set of joinands twice? We can interpret the second operation in two ways. Either the join operator finds the existing join and returns it as the result, or it constructs a second member, with the same jims in its downset as the first join. The SheafSystem supports both approaches, but in the latter case, where should the second join be placed in the order relation? The set of jims of the second join is equal to the set of jims of the first, so the order relation says they are equal, but they are not the same object. The SheafSystem breaks the tie by extending its definition of the order relation to what is called a lexicographical order. A lexicographical order is a generalization of ordinary dictionary order. To place words in order, first we sort on the first letter. If two words have the same first letter, we sort on the second letter, and so on. The SheafSystem uses a lexicographical order in which the first letter is the set of jims in the down set and the second letter is the order of creation. So when we construct the second join, it has the same set of jims but it was constructed after the first join, so it is greater than the first join. The second join is thus linked immediately above the first join. A third copy would be linked above the second copy, and so on.