SheafSystem™ Programmer's Guide

A Guided Tour of the SheafSystem™ Libraries

David M. Butler

Limit Point Systems, Inc.

1 Introduction

This document shows how to use many of the features of the SheafSystem. Part I describes the most basic features using examples that are as simple as possible. It is intended as an introduction for beginners, providing a guided tour of the essential capabilities of the SheafSystem libraries. Part II describes additional features using more complex examples. It is intended to extend the topics of Part I to describe usage for common cases that occur in practice. Both parts taken together still do not show how to use every feature of the system; readers are expected to use this tutorial as a starting place for further exploration of the reference documentation and class libraries.

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.

2 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, for viewing the reference documentation, and
- an installed copy of the SheafSystem.

3 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.

4 Part I: Basic features

4.1 The sheaf interface

4.1.1 Getting started

4.1.1.1 PartSpace metaphor

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

4.1.1.2 Sheaf tables

As described in the Part Space 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 primitives_poset_schema table, the primitives table, and the namespace table. The primitives_poset_schema table terminates the schema recursion, it is its own schema table. The primitives table describes each primitive type supported by the system.

Historical note*: The primitives_poset_schema table is called the "schema_part" table in the Part Space document. The name "primitives_poset_schema" is a historical artifact, originating in early attempts to understand termination of the schema recursion. The name "schema_part" is more accurate.*

4.1.1.3 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.

4.1.1.3.1 Example 1: 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, the first part of which reads:

```
#
# Path to your C++ compiler
#
CXX =
#
# Path the SheafSystem installation
#
```

```
SHEAF_INSTALL_DIR =
SHEAF_LIB_DIR = $(SHEAF_INSTALL_DIR)/Debug-contracts/lib
SHEAF_INC_DIR = $(SHEAF_INSTALL_DIR)/include

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 2 variables CXX and SHEAF_INSTALL_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
```

To run example32, the LD_LIBRARY_PATH also has to contain the path to the vtk libraries:

```
>setenv LD_LIBRARY_PATH  <sheaf_dir>/Debug-contracts/lib:<sheaf_dir>/lib/vtk
```

Now we can execute the example:

```
>./example1
Hello-sheaf
```

The "handcrafted" Makefile is provided to show the basic mechanics of creating an application with the SheafSystem. The examples module also comes with a CMake build system which doesn't show the details, but may be easier to use, especially on Windows. See the installation how-to document for instructions on using the CMake system.

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.

4.1.2 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.

4.1.2.1 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()
- name() == xname
- !in_jim_edit_mode()
- host() == 0
- !index().is_valid()

- index().same_scope(member_hub_id_space(false))
- has_standard_subposet_ct()
- current_namespace() == this
- state_is_not_read_accessible()

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() && (xname.find_first_not_of(name_legal_characters()) == 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()

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 x 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() == 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.

4.1.2.1.1 Example 2: 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.

4.1.2.2 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 Appendix A.

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.

4.1.2.3 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 explicit 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 implicit 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!

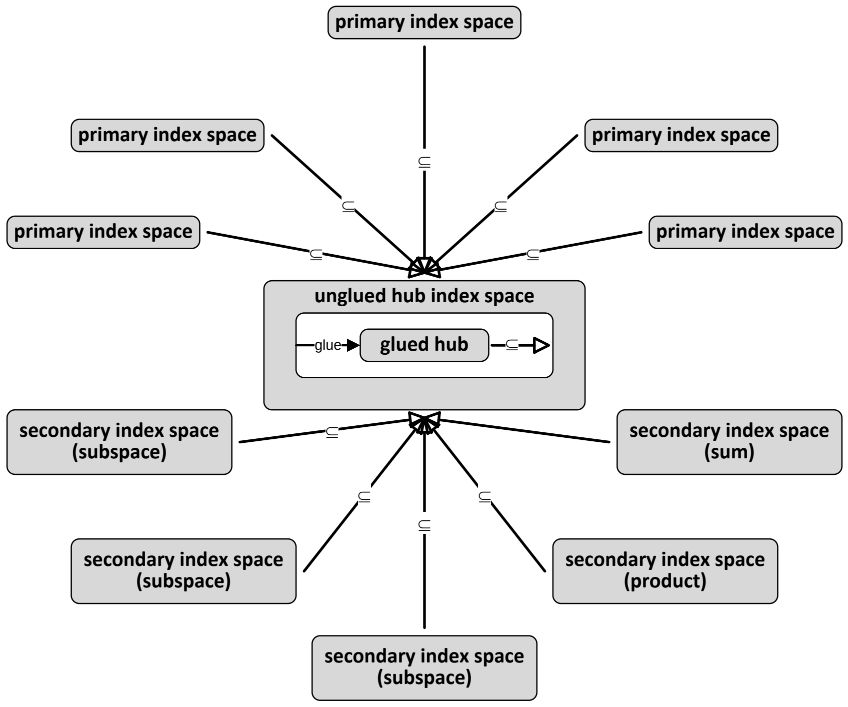
primary index space
primary index space
primary index space
primary index space
primary index space
unglued hub index space
glue
glued hub
⊆
secondary index space (subspace)
secondary index space (sum)
secondary index space (subspace)
secondary index space (product)
secondary index space (subspace)

Figure 1: Hub and spoke architecture of an index space family.

4.1.3 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. In this section, we will describe the basics of using id spaces automatically created by the system. In Part II: Intermediate features, we'll discuss creating client-defined id spaces.

4.1.3.1 Index spaces and iterators.

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 Figure 1. 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. For a detailed discussion of this structure see the document "Index Spaces". 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.

4.1.3.2 Example 3: 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 containers.
  // If the space is "gathered", begin() == 0 and end() = ct().
  // If the space is not gathered, it's "scattered".

  cout << "beginning 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.
beginning 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.
```

4.1.3.3 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 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.

4.1.3.4 Example 4: 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 << "beginning 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());

    // Assignment 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.
beginning 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.

4.1.4 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.

4.1.4.1 Example 5: 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 constructor, "example5.hdf".

```
>./example5
>ls *.hdf
example5.hdf
```

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

4.1.5 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.

4.1.5.1 Stream insertion operator

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

4.1.5.1.1 Example 6: 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 lot 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.

4.1.5.2 The dumpsheaf utility

The SheafSystem provides the dumpsheaf 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.

4.1.5.2.1 Example 7: View namespace with dumpsheaf

First, make sure you have set your environment using the set_env_vars script in the build directory of your Programmer's Guide installation:

```
>source set_env_vars.csh
```

Then run the dumpsheaf utility in the bin directory of the SheafSystem installation:

```
<sheaf_dir>/Debug_contracts/bin/dumpsheaf example5.hdf
```

4.1.5.3 The SheafScope interactive file browser

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

4.1.5.3.1 Example 8: View namespace with SheafScope

Make sure you've set your environment with the set_env_vars script as shown above, then run the SheafScope.

```
>java  -jar SheafScope.jar example5
```

4.1.6 Posets

4.1.6.1 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.

***Historical note**: Poset_state_handle, the abstract base class for all poset types, is as its name says, a handle. But this is another 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. "Abstract_poset" would be a more accurate name for this class.*

4.1.6.2 Creating posets

Two class hierarchies form the backbone of the SheafSystem: the poset hierarchy and the poset member hierarchy. The root of the poset hierarchy, class poset_state_handle, represents the general poset and its descendants represent various specialized posets, in particular the various "spaces" of the fiber bundle data model. The root of the poset member hierarchy, class abstract_poset_member, represents the general poset member and its descendants represent various particular object types, it particular the mesh, property, and section types of the fiber bundle model.

Each member type lives in some specific type of poset, called the host_type for the member. For instance:

typedef poset sheaf::abstract_poset_member::host_type

The type of host poset for this type of member.

Several member types may have the same host_type so an instance of the host type must be configured for the specific type of member that it will contain. In particular, the host poset must be created with the proper schema for the member type. For this reason, posets are created using factory methods provided by the member types.

Classes of the poset member hierarchy each provide a new_host factory method and, in some cases, a higher level standard_host factory method that automatically assigns a standard schema and other features. We'll start by creating a simple poset using the most basic new_host methhod:

static
sheaf::abstract_poset_member::host_type
sheaf::abstract_poset_member::
new_host(namespace_type& xns,
const poset_path& xhost_path,
const poset_path& xschema_path,
bool xauto_access)

Creates a new host table for members of this type. The poset is created in namespace xns with path xhost_path and schema specified by xschema_path.

As the signature suggests, 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. This is a difficult issue for a tutorial, since it means we have to already have a poset before we can explain how to create a poset! We'll see how to create a schema poset shortly, but to get started, we're going to use some unexplained magic. We can avoid creating a schema poset if the poset we want to create has only a single attribute because we can use a member of the primitives poset that every namespace already has as a schema. We'll do that to get started.

4.1.6.2.1 Poset_path

The signature for new_host says we need poset_path objects for the poset and its schema. A poset path is similar to a file path, but has only two elements, a poset name and a member name. If the path is "full", that is both the poset name and the member name are non-empty, then it specifies a member of a poset, while if the member name is empty it specifies only a poset. We can create a path by specifying the poset name and member name separately:

sheaf::poset_path::
poset_path(const string& xposet_name, const string& xmember_name)

Creates an instance with poset_name() == xposet_name and member_name() == xmember_name.

or by specifying the path as a string, with a "/" separating the poset and member:

sheaf::poset_path::
poset_path(const string& xpath)

Conversion from string.

The path for our poset requires only the poset name, we'll use "simple_poset", with an empty member name in this case:

```
poset_path lposet_path("simple_poset", "");
```

But if we provide a member name, for instance:

```
poset_path lposet_path("simple_poset/top");
```

new_host will ignore it.

We need a path for the schema member as well. 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, as we said above, 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 lschema_path("primitives", "INT");
```

So now we can construct our first poset:

```
poset& lposet = abstract_poset_member::new_host(lns, lposet_path,
      lschema_path, true);
```

4.1.6.3 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 meantime, 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 6, we can get a reference to it with:

```
poset_state_handle& lpsh1 = lns.member_poset(6, 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);
```

4.1.6.4 Deleting 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().

Example 9 ties the namespace functions we've been discussing into a single example.

4.1.6.5 Example 9: Creating, accessing, and deleting posets.

```
#include "sheaves_namespace.h"
#include "std_iostream.h"
#include "storage_agent.h"

using namespace sheaf;
```

```
int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example9:" << endl;

  // Create a namespace.

  sheaves_namespace lns("Example9");

  // Create a path for the poset.

  poset_path lposet_path("simple_poset", "");

  // Use the INT member of the primtives poset
  // as the schema for the simple poset.

  poset_path lschema_path("primitives", "INT");

  // Create the poset, will be id 6.

  poset& lposet =
    poset::new_table(lns, lposet_path, lschema_path, true);

  cout << "poset id: " << lposet.index().hub_pod() << endl;

  // Print the poset to cout.

  cout << lposet << endl;

  // Get another reference to the poset by id:

  poset_state_handle& lpsh1 = lns.member_poset(6, true);

  // and by path (string literal invokes conversion to poset_path):

  poset_state_handle& lpsh2 = lns.member_poset("simple_poset", true);

  // Get a reference to type poset:

  poset& lposet2 = lns.member_poset<poset>("simple_poset", true);

  // Write the namespace to a sheaf file.

  storage_agent lsa("example9.hdf");
  lsa.write_entire(lns);

  // 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.

4.1.7 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.

4.1.7.1 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 algebraically, 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.

***Historical note**: "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.*

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 Figure 2. Well, almost reproduce, Figure 2 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);
```

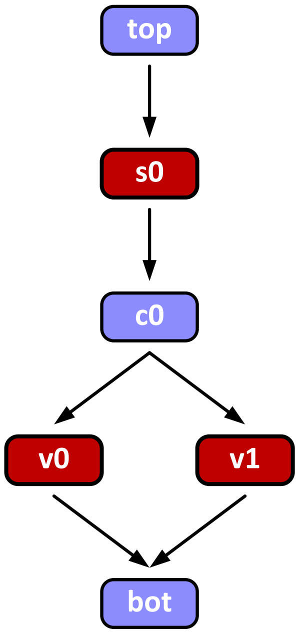

Figure 2: Line segment example from Part Space, Figure 10.

4.1.7.2 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(lv1_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 p, the set of lesser members p is linked to by outgoing links, that is the set of members p covers, is called the <u>lower</u> <u>cover</u> of p. The set of larger members that are linked to p by the incoming links, the set of members that cover p, is called the <u>upper cover</u> of p. 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();
```

4.1.7.3 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.

4.1.7.3.1 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.

4.1.7.3.2 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 attribute 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);
```

4.1.7.3.3 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);
```

4.1.7.3.4 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.

4.1.7.3.5 Primitive_value

The attribute value returned 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;
}
```

4.1.7.3.6 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.

4.1.7.3.7 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");
```

4.1.7.4 Creating join reducible members

We can create jrms and link them up at any time, whether we're in jim_edit_mode or not. 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.

4.1.7.5 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 Figure 3, 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?

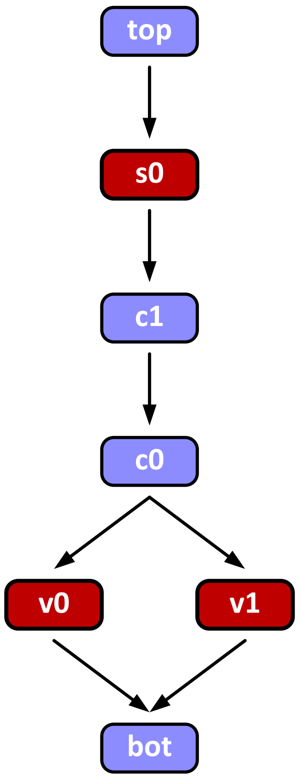

Figure 3: A join equivalent member.

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 Appendix B. 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 Example 11 we'll see some functions that make it easy to make copies.

4.1.7.6 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, use 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);
```

4.1.7.7 Example 10: 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.

```
#include "sheaves_namespace.h"
#include "poset.h"
#include "poset_dof_map.h"
#include "std_iostream.h"
#include "storage_agent.h"

using namespace sheaf;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example10:" << endl;

  // Create a namespace.

  sheaves_namespace lns("Example10");

  // Populate the namespace from the file we wrote in example9.
  // Retrieves the simple_poset example.

  storage_agent lsa("example9.hdf", sheaf_file::READ_ONLY);
  lsa.read_entire(lns);

  // Get a reference to the poset "simple_poset".

  poset_path lpath("simple_poset");
  poset& lposet = lns.member_poset<poset>(lpath, true);
  // Allow creation of jims.

  lposet.begin_jim_edit_mode(true);

  // Create jims for the two vertices and the segment.

  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);

  // Make the segment cover the vertices.

  lposet.new_link(ls0_pod, lv0_pod);
  lposet.new_link(ls0_pod, lv1_pod);

  // Top covers the segment.

  lposet.new_link(TOP_INDEX, ls0_pod);

  // The vertices cover bottom.

  lposet.new_link(lv0_pod, BOTTOM_INDEX);
  lposet.new_link(lv1_pod, lposet.bottom().index().pod());

  // We're finished creating and linking jims.

  lposet.end_jim_edit_mode();

  // Give each jim a name..

  lposet.put_member_name(lv0_pod, "v0", true);
  lposet.put_member_name(lv1_pod, "v1", true);
  lposet.put_member_name(ls0_pod, "s0", true);

  // Print the names to cout.

  cout << lposet.member_name(lv0_pod) << endl;
  cout << lposet.member_name(lv1_pod) << endl;
  cout << lposet.member_name(ls0_pod) << endl;

  // Get the row attribute id space and pod and
  // scoped ids for the only attribute.

  const index_space_handle& latt_id_space =
      lposet.schema().dof_id_space(false);
  pod_index_type latt_pod =
      lposet.schema().dof_id_space(false).begin();
  scoped_index latt_id(lposet.schema().dof_id_space(false), latt_pod);

  // Get the attribute tuple for vertex 0.

  poset_dof_map& ltuple = lposet.member_dof_map(lv0_pod, true);

  // Set the only attribute of v0 to its dimension, 0.
  // Do the first one explicitly, without any automatic conversion.
  primitive_value lpv(int(0));
  ltuple.put_dof(latt_pod, lpv);

  // Set attributes for v1 and s0 relying on conversions.

  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));

  // Get attributes back and write them to cout.

  int lv0_dim = lposet.member_dof_map(lv0_pod, false).dof(latt_pod);
  int lv1_dim = lposet.member_dof_map(lv1_pod, false).dof(latt_pod);
  int ls0_dim = lposet.member_dof_map(ls0_pod, false).dof(latt_pod);

  cout << "v0 dim= " << lv0_dim;
  cout << " v1 dim= " << lv1_dim;
  cout << " s0 dim= " << ls0_dim;
  cout << endl;

  // Create a jrm named c0.

  pod_index_type lc0_pod = lposet.new_member(false);
  lposet.put_member_name(lc0_pod, "c0", true);

  //  Link it up:

  lposet.new_link(ls0_pod, lc0_pod);
  lposet.new_link(lc0_pod, lv0_pod);
  lposet.new_link(lc0_pod, lv1_pod);

  // Delete the now obsolete links from s0 to the vertices.

  lposet.delete_link(ls0_pod, lv0_pod);
  lposet.delete_link(ls0_pod, lv1_pod);

  // Create a jem; a copy of c0, call c1.

  pod_index_type lc1_pod = lposet.new_member(false);
  lposet.put_member_name(lc1_pod, "c1", true);
  lposet.delete_link(ls0_pod, lc0_pod);
  lposet.new_link(ls0_pod, lc1_pod);
  lposet.new_link(lc1_pod, lc0_pod);

  // Output the finished poset to cout:

  cout << lposet << endl;

  // Delete c1.

  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);

  // Exit:

  return 0;
```

}

Make sure to run example9 before example10 and in the same directory so that example10 can find the file example9.hdf. The output from example10 is in the file example10.cout

4.1.8 Poset member handles

The poset member class hierarchy provides an alternate interface for manipulating poset members. Abstract_poset_member is the abstract base class for the hierarchy, with immediate descendants partial_poset_member and total_poset_member. The partial/total adjective refers to whether the interface supports restriction of a member to only part of its schema; partial_poset_member does and total_poset_member doesn't. Partial_poset_member is the base class for the various types of sections in the section_spaces cluster of the fiber_bundles component, for which restriction is an important operation. Total_poset_member is the type of member for ordinary posets and is the base class for the various algebraic types in the fiber_spaces cluster in fiber_bundles, for which restriction doesn't really make much sense.

4.1.8.1 Host factory methods

We've already seen that each class in the poset member hierarchy has a typedef host_type which specifies the type of host poset appropriate for the member class and exports a static new_host factory method for creating an instance of the host_type. The precise signature of the new_host method varies according to the specific needs of the class.

When appropriate, each class may also export a higher level standard_host factory method that automatically assigns a standard schema and other features. We'll see an example of the standard_host method in section 4.1.12 Schema_poset_member handles.

4.1.8.2 Member features

The most prominent feature of abstract_poset_member and its descendants it that they are handles. We've already seen index space handles. Poset member handles are similar in concept but are used somewhat differently. First, poset member handles are not stored in pools managed by the system; the client directly creates and destroys them. The default constructor creates an unattached handle (all the following functions are direct or inherited members of total_poset_member):

total_poset_member()

Default constructor; creates a new, unattached total_poset_member handle.

Once created, you can "attach" a handle to a state:

void attach_to_state(const poset_state_handle * xhost, pod_index_type xindex)

Attach this handle to the state with index xindex in the current version of host xhost.

The attach_to_state function is available in several signatures for specifying the member to attach to by pod id, scoped id, name and some other variations, see the reference documentation.

You can combine the construction and attachment into a single step, also available in several signatures, for instance:

total_poset_member(const poset_state_handle * xhost, pod_index_type xindex)

Creates a new total_poset_member handle attached to the member state with index xindex in xhost.

You detach a handle, so it is unattached again.

void poset_component::detach_from_state()

Detach this handle from its state, if any.

You can create a handle and a new jim state in a single step:

total_poset_member(poset_state_handle* xhost, poset_dof_map* xdof_map=0, bool xcopy_dof_map=false, bool xauto_access=true)

Creates a new jim (join-irreducible member) attached to a new member state in xhost.

You can use an existing handle to create a new state, as usual with several signatures, for instance:

void new_jim_state(poset_state_handle* xhost, poset_dof_map* xdof_map = 0, bool xcopy_dof_map = false, bool xauto_access = true)

Creates a new jim (join-irreducible member) state in xhost and attaches this to it.

The poset member classes provides most of the operations we've already seen in the poset interface. For instance:

void create_cover_link(abstract_poset_member* xlesser)

Insert a link from this to lesser; make lesser <= this.

and

void delete_cover_link(abstract_poset_member* lesser)

Delete the link from this to lesser; make lesser incomparable to this.

In addition, the poset member classes provide several operations not available (yet) in the poset interface. In particular, it provides the lattice algebra operations, join and meet:

total_poset_member* total_poset_member::l_join(abstract_poset_member* other, bool xnew_jem = true)

Lattice join of this with other, auto-allocated version. The lattice join is the least upper bound in the lattice generated by the jims in the poset.

total_poset_member* sheaf::total_poset_member::l_meet(abstract_poset_member* other, bool xnew_jem = true)

Lattice meet of this with other, auto-allocated version. The lattice meet is the greatest lower bound in the lattice generated by the jims in the poset.

4.1.8.3 Example 11: Manipulating poset member with the poset_member interface.

We provide examples of the poset member handle interface by redoing Example 10 using the poset member interface instead of the poset interface.

```
#include "sheaves_namespace.h"
#include "poset.h"
#include "poset_dof_map.h"
#include "std_iostream.h"
#include "storage_agent.h"
#include "total_poset_member.h"

using namespace sheaf;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example11:" << endl;

  // Create a namespace.

  sheaves_namespace lns("Example11");

  // Populate the namespace from the file we wrote in example9.
  // Retrieves the simple_poset example.

  storage_agent lsa("example9.hdf", sheaf_file::READ_ONLY);
  lsa.read_entire(lns);

  // Get a reference to the poset "simple_poset".

  poset_path lpath("simple_poset");
  poset& lposet = lns.member_poset<poset>(lpath, true);

  // Create an unattached handle.

  total_poset_member lmbr;
  cout << "lmbr is_attached() = " << boolalpha << lmbr.is_attached();
  cout << endl;

  // Attach it to the top member of our poset.

  lmbr.attach_to_state(&lposet, TOP_INDEX);
  cout << "lmbr attached to " << lmbr.name() << endl;

  // Reattach it to the bottom member.

  lmbr.attach_to_state(&lposet, BOTTOM_INDEX);
  cout << "lmbr attached to " << lmbr.name() << endl;

  // Unattach it.

  lmbr.detach_from_state();
  cout << "lmbr is_attached() = " << lmbr.is_attached() << endl;

  // Allow creation of jims.

  lposet.begin_jim_edit_mode(true);

  // Create jims for the two vertices and the segment.

  total_poset_member lv0(&lposet);
  total_poset_member lv1(&lposet);
  total_poset_member ls0(&lposet);

  // Make the segment cover the vertices.

  ls0.create_cover_link(&lv0);
  ls0.create_cover_link(&lv1);

  // Top covers the segment.

  lposet.top().create_cover_link(&ls0);

  // The vertices cover bottom.

  lv0.create_cover_link(&lposet.bottom());
  lv1.create_cover_link(&lposet.bottom());

  // We're finished creating and linking jims.

  lposet.end_jim_edit_mode();

  // Give each jim a name..

  lv0.put_name("v0", true, true);
  lv1.put_name("v1", true, true);
  ls0.put_name("s0", true, true);

  // Print the names to cout.

  cout << lv0.name() << endl;
  cout << lv1.name() << endl;
  cout << ls0.name() << endl;

  // Get the row attribute id space and pod and scoped ids
  // for the only attribute.

  const          index_space_handle&          latt_id_space           =
               lposet.schema().dof_id_space(false);
  pod_index_type                        latt_pod                    =
                lposet.schema().dof_id_space(false).begin();
  scoped_index latt_id(lposet.schema().dof_id_space(false), latt_pod);

  // Set attributes for v0, v1, and s0 relying on conversions.

  lv0.dof_map(true).put_dof(latt_pod, int(0));
  lv1.dof_map(true).put_dof(latt_pod, int(0));
  ls0.dof_map(true).put_dof(latt_pod, int(1));

  // Get attributes back and write them to cout.

  int lv0_dim = lv0.dof_map(false).dof(latt_pod);
  int lv1_dim = lv1.dof_map(false).dof(latt_pod);
  int ls0_dim = ls0.dof_map(false).dof(latt_pod);

  cout << "v0 dim= " << lv0_dim;
  cout << " v1 dim= " << lv1_dim;
  cout << " s0 dim= " << ls0_dim;
  cout << endl;

  // Create a jrm named c0.
  // C0 is the join of v0 and v1, so we can create it
  // and link it up in a single step using the join operator.

  total_poset_member* lc0 = lv0.l_join(&lv1, false);
  lc0->put_name("c0", true, true);

  // Create a jem; a copy of c0, Call it c1.

  total_poset_member lc1(*lc0, true);
  lc1.put_name("c1", true, true);

  // Output the finished poset to cout:

  cout << lposet << endl;

  // Delete c1.

  lc1.delete_state(true);

  cout << "c1 is_attached() = " << boolalpha << lc1.is_attached();
  cout << endl;

  // Exit:

  return 0;
}
```

4.1.9 Subposets

A subposet is, as the name suggests, a subset of a poset. We've already seen that subsets can be represented by id spaces, so why subposets? The answer is mostly historical, the id space concept emerged as a generalization of subposets and id maps. The id maps no longer exist as independent entities, they have been subsumed by the id spaces, but subposet lives on, at least for a while.

A subposet is essentially a bit vector with a bit for each member of the poset. The bit corresponding to a member is set (true) if the member of the poset is a member of the subposet. The bit vector representation is efficient in both time and memory for subsets that contain more than a few percent of the total poset, which is one reason the subposet class is still used.

The other reasons subposet is still around is that it is used for three critical roles in the system. The first critical role is the "whole" subposet. The whole subposet defines which ids are actually members of the poset; when a member is deleted the corresponding id continues to exist, but the member is no longer in the whole subposet. The functionality of whole subposet has been largely replaced by the member_hub_id_space.

The second critical role for subposet is the jims subposet. A member is a jim if and only if it is in the jims subposet. Finally, as we will learn shortly, subposets are used as filters in depth first searches of the graph.

Whenever possible, clients should use id spaces instead of subposets.

The following example shows the basic features of subposets.

4.1.9.1 Example 12: Subposets

```
#include "sheaves_namespace.h"
#include "poset.h"
#include "poset_dof_map.h"
#include "std_iostream.h"
#include "storage_agent.h"
#include "subposet.h"

using namespace sheaf;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example12:" << endl;

  // Create a namespace.

  sheaves_namespace lns("Example12");

  // Populate the namespace from the file we wrote in example9.
  // Retrieves the simple_poset example.

  storage_agent lsa_read("example10.hdf", sheaf_file::READ_ONLY);
  lsa_read.read_entire(lns);

  // Get a reference to the poset "simple_poset".

  poset_path lpath("simple_poset");
  poset& lposet = lns.member_poset<poset>(lpath, true);
  // Create a subposet called "jrms".

  subposet ljrms(&lposet);
  ljrms.put_name("jrms", true, false);

  // Test to see if it is empty (constructor ensures that it is).

  cout << "subposet " << ljrms.name();
  cout << " is empty()= " << boolalpha << ljrms.is_empty();
  cout << endl;

  // Put top, c1, c0, and bottom into the subposet.
  // Use all the different signatures for insert_member.

  scoped_index ltop_id(lposet.top().index());
  ljrms.insert_member(ltop_id);

  pod_index_type lc1_pod = lposet.member_id("c1", false);
  ljrms.insert_member(lc1_pod);

  pod_index_type lc0_pod = lposet.member_id("c0", false);
  ljrms.insert_member(lc0_pod);

  abstract_poset_member* lbot = &lposet.bottom();
  ljrms.insert_member(lbot);

  // Check if it contains the member we just put in.

  cout << "contains top: " << ljrms.contains_member(ltop_id) << endl;
  cout << "contains c1: " << ljrms.contains_member(lc1_pod) << endl;
  cout << "contains c0: " << ljrms.contains_member(lc0_pod) << endl;
  cout << "contains bottom: " << ljrms.contains_member(lbot) << endl;

  // Get an iterator for the members and print out their names.

  cout << "Subposet jrms contains:";
  index_iterator litr = ljrms.indexed_member_iterator();
  while(!litr.is_done())
  {
    // Print out the member name.

    cout << "  " << lposet.member_name(litr.index(), false);
    litr.next();
  }
  cout << endl;

  // Remove top and bottom.

  ljrms.remove_member(lposet.top().index());
  ljrms.remove_member(&lposet.bottom());

  // Print out the member names again.

  cout << "Subposet jrms contains:";
  litr.reset();
  while(!litr.is_done())
  {
    // Print out the member name.

    cout << "  " << lposet.member_name(litr.index(), false);
    litr.next();
  }
  cout << endl;

  // Exit:

  return 0;
}
```

4.1.10 Traversing the graph

Once you've constructed a poset or retrieved one from storage, you typically want to move around in it, explore it. The requires traversing (searching) the covering relation graph and the SheafSystem provides iterators for that purpose. There are two kinds cover iterators and depth-first iterators.

4.1.10.1 Cover id spaces and iterators

Local searches, in the immediate neighborhood of a given poset member are accomplished with cover iterators. Remember that every member except bottom has a set of outgoing links to the lower cover of the member and every member except top has a set of incoming links to its upper cover. The lower and upper cover of each member are represented as id spaces:

index_space_handle& poset_state_handle::
get_cover_id_space(bool xlower, pod_index_type xmbr_hub_id) const

Allocates a handle for the lower (xlower true) or upper (xlower false) cover id space of the member with hub id xmbr_hub_id from the pool of id spaces.

As always, if you get_ it you have to release_ it when you're finished with it.

void release_cover_id_space (index_space_handle &xcover_id_space) const

Returns xcover_id_space to the pool of id spaces.

We can get the id space from the poset and get/release an iterator from the id space or get/release the iterator directly from the poset:

index_space_iterator& poset_state_handle::
get_cover_id_space_iterator(bool xlower, pod_index_type xmbr_hub_id) const

Allocates an iterator for the lower (xlower true) or upper (xlower false) cover id space of the member with hub id xmbr_hub_id from the pool of id space iterators.

void poset_state_handle::
release_cover_id_space_iterator (index_space_iterator &xcover_itr) const

Returns xcover_itr to the pool of id space iterators.

The cover_id_space and cover_id_space_iterator functions are also available from poset member handles.

4.1.10.1.1 Example 13: Cover id spaces and iterators

```
#include "index_space_handle.h"
#include "index_space_iterator.h"
#include "poset.h"
#include "sheaves_namespace.h"
#include "std_iostream.h"
#include "storage_agent.h"
#include "total_poset_member.h"

using namespace sheaf;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example12:" << endl;

  // Create a namespace.

  sheaves_namespace lns("Example12");

  // Populate the namespace from the file we wrote in example10.
  // Retrieves the simple_poset example.

  storage_agent lsa_read("example10.hdf", sheaf_file::READ_ONLY);
  lsa_read.read_entire(lns);

  // Get a reference to the poset "simple_poset".

  poset_path lpath("simple_poset");
  poset& lposet = lns.member_poset<poset>(lpath, true);

  // Get the hub id for member "c0".

  pod_index_type lc0_pod = lposet.member_id("c0", false);

  // Get an iterator for the lower cover to member "c0".

  index_space_handle&    lc0_lc1    =    lposet.get_cover_id_space(true,
lc0_pod);
  index_space_iterator& lc0_lc_itr1 = lc0_lc1.get_iterator();

  // Iterate over the members of the lower cover.

  cout << "Lower cover of c0 is:";
  while(!lc0_lc_itr1.is_done())
  {
    cout << "  " << lposet.member_name(lc0_lc_itr1.hub_pod());
    lc0_lc_itr1.next();
  }
  cout << endl;

  // We're finished with the iterator, release it.
  lc0_lc1.release_iterator(lc0_lc_itr1);

  // Get an iterator for the upper cover of c0.
  // The enums LOWER and UPPER are defined in the sheaf namespace
  // true and false, respectively.

  index_space_iterator&                  lc0_uc_itr1                  =
              lposet.get_cover_id_space_iterator(UPPER, lc0_pod);

  // Iterate over the members of the upper cover.

  cout << "Upper cover of c0 is:";
  while(!lc0_uc_itr1.is_done())
  {
    cout << "  " << lposet.member_name(lc0_uc_itr1.hub_pod());
    lc0_uc_itr1.next();
  }
  cout << endl;

  // We're finished with the iterator, release it.

  lposet.release_cover_id_space_iterator(lc0_uc_itr1);

  // Repeat all the above using the poset member handle interface.

  // Get a handle for member c0.

  total_poset_member lc0_mbr(&lposet, "c0");

  // Get an iterator for the lower cover to member "c0".

  index_space_handle& lc0_lc2 = lc0_mbr.get_cover_id_space(true);
  index_space_iterator& lc0_lc_itr2 = lc0_lc2.get_iterator();

  // Iterate over the members of the lower cover.

  cout << "Lower cover of c0 is:";
  while(!lc0_lc_itr2.is_done())
  {
    cout << "  " << lposet.member_name(lc0_lc_itr2.hub_pod());
    lc0_lc_itr2.next();
  }
  cout << endl;

  // We're finished with the iterator, release it.

  lc0_lc2.release_iterator(lc0_lc_itr2);

  // Get an iterator for the upper cover of c0.
  // The enums LOWER and UPPER are defined in the sheaf namespace
  // true and false, respectively.

  index_space_iterator&                  lc0_uc_itr2                  =
              lc0_mbr.get_cover_id_space_iterator(UPPER);

  // Iterate over the members of the upper cover.
  total_poset_member luc_mbr;
  cout << "Upper cover of c0 is:";
  while(!lc0_uc_itr2.is_done())
  {
    luc_mbr.attach_to_state(&lposet, lc0_uc_itr2.hub_pod());
    cout << "  " << luc_mbr.name();
    lc0_uc_itr2.next();
  }
  cout << endl;

  // We're finished with the iterator, release it.

  luc_mbr.release_cover_id_space_iterator(lc0_uc_itr2);

  // Exit:

  return 0;
}
```

4.1.10.2 Depth first traversal

Global searches, over the larger portions of a graph than the immediate neighborhood of a single member are accomplished using the depth_first_itr family of iterators. As implied by the name, these iterators execute a depth first search of the graph. Readers unfamiliar with depth first search should consult a basic computer science text such as Aho and Ulman "Foundations of Computer Science".

A depth first iterator starts from a given member called the anchor for the iteration and transitively follows all the links in a given direction, either down or up. A depth first search always visits all the children of a node before it visits any of the siblings of the node. There are three locations relative to a given member p where some action may be performed in a depth first search, as shown in Figure 4. The previsit action occurs immediately before visiting the first child, so it is the first opportunity for acting on p. The postvisit action occurs after all the children have been visited, so it is the last action opportunity for p with p as the current member of the iteration. The link action for a given child occurs "on the link", immediately after visiting the child. The link action for a child, with the parent as current member, occurs immediately after the postvisit action for the child, with the child as current member. So a link action can still affect a node after the postvisit action on that node has occurred.

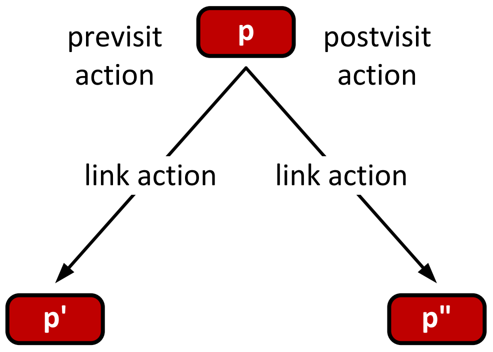

Figure 4: Actions in depth first search.

The members of the depth_first_itr hierarchy differentiate on which of the action opportunities are exposed to the client. Class preorder_itr returns control to the user only in the previsit position, immediately before iterating over the links. Class postorder_itr returns control to the client only in the postvisit position, after visiting all the children. Class biorder_itr returns control in both the previsit and postvisit positions. Class linkorder_itr exposes the link action opportunities. Class triorder_itr exposes all the action opportunities.

The postorder_itr is the most frequently used iterator. Postorder has the extremely useful property that by the time a given node is visited, all the nodes less than it in the order relation have been visited.

The preorder_itr is the next most frequently used. It is useful because the depth first search can be truncated from the preorder position. Instead of visiting all the children, transitively, iteration can forced to jump over the children to the next sibling. This is useful for finding maximal and minimal members of some subset, as we'll see in the example.

Biorder_itr, linkorder_itr, and triorder_itr are useful for advanced algorithms, especially iterations that change the graph.

All five of these classes are actually class templates, for instance:

template<typename T> class sheaf::postorder_itr< T >

Specialization of the filtered depth-first iterator which exposes the POSTVISIT_ACTION to the client.

The parameter specifies the type of data structure used to track which nodes in the graph have already been visited. Three specializations are provided: zn_to_bool (a bit vector type), set, and hash set. The specializations are provided because they have different performance characteristics. Let N be the number of nodes in the graph and M be the number visited in a given traversal. Zn_to_bool uses O(1) time to check if a node has been visited already, but uses one bit per node and hence uses O(N) memory, independent of how much of the graph is begin searched. Zn_to_bool is most efficient for traversal of the entire graph, but the O(N) memory means it takes O(N) time to initialize the marker structure, a serious problem for repeated small searches. Set uses O(log(M)) time to check if a node has been visited and O(M) memory, so it is better for small searches. Hash_set uses O(1) time and O(M) memory, so it is very good for small searches, but uses more memory than zn_to_bool when applied to the whole graph. Typedefs are provided for the three specializations, for instance:

typedef postorder_itr<zn_to_bool> sheaf::zn_to_bool_postorder_itr

Postorder_itr<T> using zn_to_bool for _has_visited markers.

typedef postorder_itr< set<pod_index_type> > sheaf::set_postorder_itr

Postorder_itr<T> using set for _has_visited markers.

typedef postorder_itr< hash_set<pod_index_type> > sheaf::hash_set_postorder_itr

Postorder_itr<T> using hash_set for _has_visited markers.

When we create an iterator, we specify the anchor, a filter, the search direction (down or up), and strictness (whether the anchor itself is visited):

postorder_itr(const abstract_poset_member & xanchor, pod_index_type xfilter_index, bool xdown, bool xstrict)

Creates an iterator anchored at xanchor, filtered by xfilter_index. If xdown, iterate in the down direction, otherwise iterate up. If xstrict, iterate over strict up/down set only.

The filter is a subposet; the iterator only returns control to the client if the node is a member of the subposet.

4.1.10.2.1 Example 14: Depth first iterators

```
#include "poset.h"
#include "postorder_itr.h"
#include "preorder_itr.h"
#include "sheaves_namespace.h"
#include "std_iostream.h"
#include "storage_agent.h"
#include "total_poset_member.h"

using namespace sheaf;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example14:" << endl;

  // Create a namespace.

  sheaves_namespace lns("Example14");

  // Populate the namespace from the file we wrote in example10.
  // Retrieves the simple_poset example.

  storage_agent lsa_read("example10.hdf", sheaf_file::READ_ONLY);
  lsa_read.read_entire(lns);

  // Get a reference to the poset "simple_poset".

  poset_path lpath("simple_poset");
  poset& lposet = lns.member_poset<poset>(lpath, true);

  // Create a postorder iterator for the down set of top.
  // Filter with the jims subposet.
  // DOWN and UP are enums for true and false, respectively.
  // Include the anchor itself in the search.

  zn_to_bool_postorder_itr
  lpost_itr(lposet.top(), JIMS_INDEX, DOWN, NOT_STRICT);

  // Find all the jims.

  while(!lpost_itr.is_done())
  {
    cout << lposet.member_name(lpost_itr.index(), false);
    cout << " is a jim";
    cout << endl;
    lpost_itr.next();
  }

  // Find just the minimal jims (atoms).
  // Iterate up from the bottom and truncate when we find something.

  zn_to_bool_preorder_itr
    lpre_itr(lposet.bottom(), JIMS_INDEX, UP, NOT_STRICT);
  while(!lpre_itr.is_done())
  {
    cout << lposet.member_name(lpre_itr.index(), false);
    cout << " is an atom";
    cout << endl;
    lpre_itr.truncate();
  }

  // Exit:

  return 0;
}
```

4.1.11 Schema posets

We took a short cut in section 4.1.6.2 "Creating posets" to avoid having to create a schema poset before we knew how to create any poset. Now that we've seen the basics of creating posets and poset members, we can return to the task of creating a schema poset.

As an example, we'll create the cell schema that was used for the line segment example in the Part Space tutorial. The cell schema is shown in Figure 5, which reproduces Figure 14 from the Part Space tutorial.

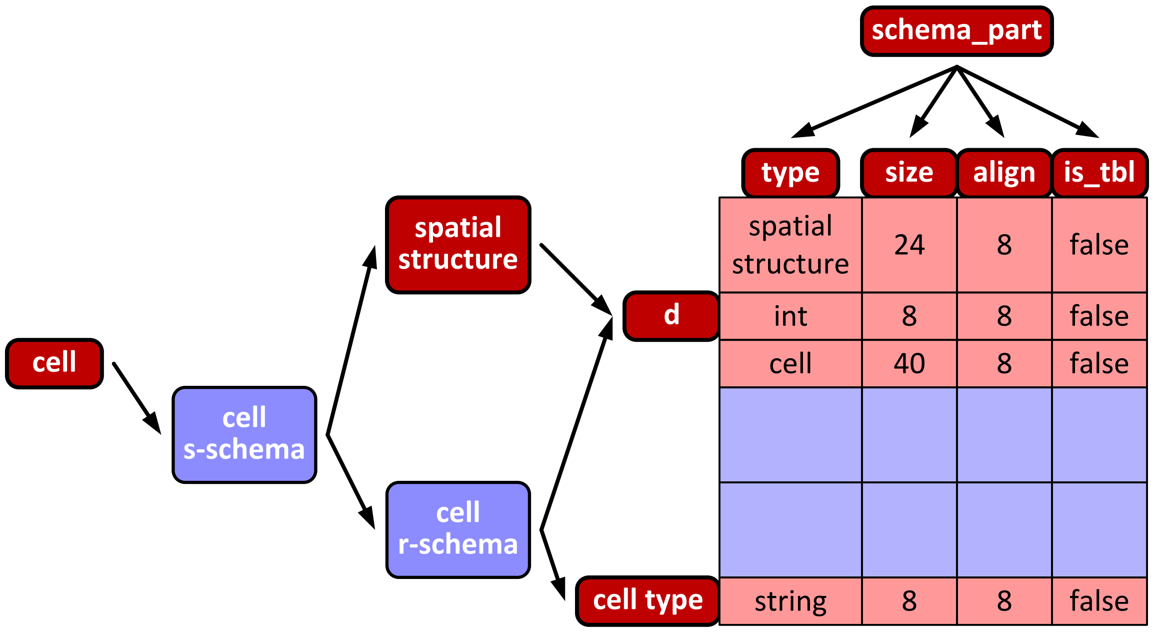

Figure 5: The cell schema table copied from the Part Space tutorial, Figure 14.

As we mentioned before, the poset that was called the "schema_part" poset in the Part Space tutorial is called the "primitives poset schema" poset in the SheafSystem library.

4.1.12 Schema_poset_member handles

The class schema_poset_member is a specialized poset member handle for creating and manipulating schema posets and members.

4.1.12.1 Host factory

Schema_poset_member is one of the classes in the poset member hierarchy that has a standard schema, identified by:

static
const sheaf::poset_path&
sheaf::schema_poset_member::standard_schema_path()

The path to the standard schema for this class.

This function returns the path to the member of the primitives_poset_schema poset that must be used for creating schema. We can use this path to create a new schema poset with schema_poset_member::new_host:

```
poset_path lcell_schema = schema_poset_member::standard_schema_path();

poset& lschema_poset =
  schema_poset_member::new_host(lns, "cell", lcell_schema, true);
```

But there's an even easier way to create a schema poset. Classes with a standard_schema_path() have a second factory method, called standard_host, that provides the standard schema automatically:

static
sheaf::schema_poset_member::host_type&
sheaf::schema_poset_member::standard_host(namespace_type& xns,
const poset_path& xhost_path,
bool xauto_access)

> The host with path xhost_path. Just returns the host if it already exists, otherwise, creates it in namespace xns with schema specified by standard_schema_path().

So we can also create the cell poset by:

```
poset& lschema_poset =
  schema_poset_member::standard_host(lns, "cell", true);
```

The standard_host method isn't a great deal more convenient than the new_host_method for class schema_poset_member. However in many poset member classes, especially the classes in the fiber_bundles component, the standard_host methods provide a dramatically simpler method than new_host.

4.1.12.2 Member features

Now that we have a schema poset, we can create some schema members. Typically the most convenient way to construct a schema member is with:

sheaf::schema_poset_member::
schema_poset_member(const namespace_poset & xns,
const string & xname,
const poset_path & xparent_path,
const wsv_block< schema_descriptor > & xdof_specs,
bool xauto_access)

> Creates a new jim which conforms_to the schema with path xparent_path and has additional dofs with names, types, and roles specified by xdof_specs.

This constructor supports single inheritance. The xparent_path argument specifies the path to the base class for the type being defined. The xdof_space argument specifies any additional data members. Class schema_descriptor is essentially a struct with members for the name, type, and is_tbl properties of an attribute. The "wsv" in wsv_block stands for "whitespace separated value" and the wsv_block template is essentially an array with features for easily entering elements as strings. Both define appropriate constructors and implicit conversions to make it easy to define the schema for individual attributes. The easiest way to explain how they work is with an example. Recall the cell class hierarchy from the Part Space tutorial:

```
class cell: public spatial_structure
{
   string cell_type;    // The type of spatial cell
}

class spatial_structure
{
   int d;               // The spatial dimension of the structure.
}
```

Class cell inherits class spatial_structure, so first we create the spatial_structure schema. It doesn't inherit anything, so we specify bottom as the parent. Poset_path has a constructor that takes a string literal, so we can specify the path using a string literal and rely on implicit conversion to poset_path. Spatial_structure has one data member, name "d", type int, not a table attribute. The wsv_block<schema_descriptor> combination lets you just string the attribute specifications together, separated by white space. The set of possible primitive types is specified in the enum primitive_type in file primitive_types.h in the include directory of the SheafSystem installation. The enumerator for type int is "INT". So the spec for the only data member of spatial structure is "d INT false". For more attributes, just string them together, separated by whitespace. So the call to create spatial_structure is:

```
schema_poset_member lspatial(lns, "spatial_structure",
      "cell_schema_poset/bottom", "d INT false", false, true);
```

Cell inherits spatial_structure and adds one data member, name cell_type, type string, not a table attribute. The primitive type corresponding to a C++ string object is a C string, with enumerator C_STRING, so we create the cell schema with:

```
schema_poset_member lcell(lns, "cell", lspatial.path(),
      "cell_type C_STRING false", false, true);
```

So now we've created the schema for the spatial_structure and cell types, with cell inheriting spatial_structure. Let's verify that cell inherits spatial_structure. We do that with:

bool sheaf::schema_poset_member::
conforms_to(const schema_poset_member & xother) const

True if the dofs defined by this agree in type and in order with the dofs defined by xother. (This schema may contain additional dofs as well.)

So we can test whether lcell inherits lspatial with:

```
cout << "cell conforms to spatial_structure= ";
cout << boolalpha << lcell.conforms_to(lspatial);
cout << endl;
```

Each schema member defines an id space for its row attributes and an id space for its table attributes. We can use ids from the attribute id space to access the properties of the attributes using, for instance:

sheaf::size_type sheaf::schema_poset_member::

size(pod_index_type xdof_id, bool xis_table_dof) const

The number of bytes in the table dof (xis_table_dof true) or row dof referred to by xdof_id in the schema defined by this.

There are similar accessors for name, type, alignment, and offset.

We demonstrate the use of all of these features and more in Example 15.

4.1.12.3 Example 15: Schema poset

```
#include "index_space_iterator.h"
#include "poset_path.h"
#include "schema_descriptor.h"
#include "schema_poset_member.h"
#include "sheaves_namespace.h"
#include "std_iostream.h"
#include "storage_agent.h"
#include "wsv_block.h"

using namespace sheaf;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example15:" << endl;

  // Create a namespace.

  sheaves_namespace lns("Example15");

  // Create the cell schema poset.

  poset& lposet =
    schema_poset_member::standard_host(lns,"cell_schema_poset", true);

  // Create the schema for spatial_structure.
  // It doesn't inherit anything,
  // so specify bottom as the parent.
  // It has one data member, name "d", type "INT",
  // not a table attribute.

  schema_poset_member lspatial(lns, "spatial_structure",
      "cell_schema_poset/bottom", "d INT false", true);

  // Cell inherits spatial_structure and adds one data member,
  // name cell_type, type C_STRING, not a table attribute.

  schema_poset_member lcell(lns, "cell", lspatial.path(),
      "cell_type C_STRING false", true);

  // Test cell for conformance to spatial_structure.

  cout << endl;
  cout << "cell conforms to spatial_structure= ";
  cout << boolalpha << lcell.conforms_to(lspatial);
  cout << endl;
  cout << endl;

  // Get an iterator for the row attribute id space of the cell schema.

  index_space_iterator& litr =
      lcell.dof_id_space(false).get_iterator();
  while(!litr.is_done())
  {
    // Print attribute info.

    pod_index_type latt_pod = litr.pod();

    cout << "name: " << lcell.name(latt_pod, false);
    cout << " size: " << lcell.size(latt_pod, false);
    cout << " alignment: " << lcell.alignment(latt_pod, false);
    cout << " type: " << lcell.type(latt_pod, false);
    cout << " offset: " << lcell.offset(latt_pod, false);
    cout << endl;

    litr.next();
  }
  lcell.dof_id_space(false).release_iterator(litr);

  // Print out the schema.

  cout << lposet << endl;

  // Test the schema by creating a poset using it.

  poset& ltest = poset::new_table(lns, "test", lcell.path(), true);

  ltest.begin_jim_edit_mode(true);
  pod_index_type lmbr0 = ltest.new_member(true);
  pod_index_type lmbr1 = ltest.new_member(true);
  pod_index_type lmbr2 = ltest.new_member(true);
  ltest.end_jim_edit_mode(true, true);

  ltest.put_member_name(lmbr0, "v0", true, false);
  ltest.member_dof_map(lmbr0, true).put_dof("d", int(0));
  ltest.member_dof_map(lmbr0, true).put_dof("cell_type", "vertex");

  ltest.put_member_name(lmbr1, "v1", true, false);
  ltest.member_dof_map(lmbr1, true).put_dof("d", int(0));
  ltest.member_dof_map(lmbr1, true).put_dof("cell_type", "vertex");

  ltest.put_member_name(lmbr2, "s0", true, false);
  ltest.member_dof_map(lmbr2, true).put_dof("d", int(1));
  ltest.member_dof_map(lmbr2, true).put_dof("cell_type", "segment");

  cout << ltest << endl;

  // Exit:

  return 0;
}
```

4.2 The fiber bundle interface

The sheaf component provides the fundamental mechanism for defining arbitrary persistent data types. The fiber bundle component uses this capability to define the types of the fiber bundle data model: base spaces, fiber spaces, and section spaces. We'll describe the basic functionality provided by the SheafSystem for each of these roles.

4.2.1 Fiber_bundles_namespace

The fiber_bundles_namespace class provides the sheaf schema for the specialized types defined by the fiber bundle component. As with the sheaf component, the first step in using the fiber bundle component is to create a namespace object, but for the fiber bundle component, it must be a fiber_bundles_namespace. You create a fiber_bundles_namespace object in the same way you create a sheaves_namespace, just give the constructor a name:

fiber_bundles_namespace(const string &xname)

Creates a fiber bundles namespace with name xname.

For example:

```
fiber_bundles_namespace lns("example21");
```

For the remainder of Part II, any namespace we mention is a fiber_bundles_namespace.

4.2.2 Base spaces

As described in the Part Spaces For Scientific Computing tutorial and the Sheaf System Analysis and Design Tutorial, the base space role in a property association is typically represented by a mesh. A mesh is a decomposition of some domain into a collection of smaller parts for the purpose of representing the dependence of one or more properties on position within the domain. In the sheaf system, a mesh is represented as a poset, with a jim for each of the basic parts in the decomposition. As with all tables in the SheafSystem, a base space table needs a schema.

4.2.2.1 Base space posets

Meshes are represented using the class base_space_poset. A base_space_poset inherits poset, so it is_a poset, and adds a number of member functions useful for manipulating meshes. As with all classes in the poset hierarchy, we create a base_space_poset using the new_host or standard_host function for the type of member we want the poset to contain. The standard_host function for class base_space_member creates a host that can contain any of the member types in the SheafSystem.

```
static
fiber_bundle::base_space_member::host_type&
fiber_bundle::base_space_member::standard_host(namespace_type& xns,
                    const poset_path& xhost_path,
```

int xmax_db,
bool xauto_access)

Finds or creates a host poset and any prerequisite posets for members of this type. The poset is created in namespace xns with path xhost_path and schema specified by standard_schema_path().

The input parameter xmax_db is the maximum dimension of any member the poset will contain. Currently all the member types supported by the system are dimension 3 or less,

Let's return to the line segment example we used in the discussion of the sheaf component and re-do it as a base space poset. For the line segment example the maximum dimension is 1. The complete code is given in Example 21.

4.2.2.2 Example 21: Creating a base_space_poset.

```
#include "base_space_member.h"
#include "base_space_poset.h"
#include "fiber_bundles_namespace.h"
#include "std_iostream.h"
#include "storage_agent.h"

using namespace sheaf;
using namespace fiber_bundle;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example21:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example21");

  // Create the poset.

  base_space_poset& lposet =
     base_space_member::standard_host(lns, "mesh", 1, true);

  // Print the poset to cout.

  cout << lposet << endl;

  // Write the namespace to a sheaf file.

  storage_agent lsa("example21.hdf");
  lsa.write_entire(lns);

  // Exit:

  return 0;
}
```

4.2.2.3 Base space members

As with ordinary poset members, there are two interfaces for creating base space members: the interface provided by base_space_poset or the interface provided by the handle classes of the base_space_member hierarchy

A base_space_poset is_a poset so we can create individual members just like we did previously for the "simple_poset" example. The only difference is that base_space_poset provides special support for creating and initializing the attribute tuples for standard cell types. To create a cell which is a copy of a standard prototype use:

sheaf::pod_index_type
fiber_bundle::base_space_poset::
new_member(const string& xprototype_name, bool xcopy_dof_map)

> Creates a disconnected jim using the prototype with name xprototype_name. If xcopy_dof_map or if xhost does not already contain a copy of the prototype dof map, create a copy of the dof map, otherwise just refer to an existing copy.

The names of the prototypes for individual members defined by the base_space_member_prototypes poset in the fiber_bundles_namespace are shown in Table 1.

Table 1: Standard prototypes for individual cells

0-cells	1-cells	2-cells	3-cells
point	segment	triangle	tetra
		quad	hex
		general_polygon	general_polyhedron

Any of these can be used with new_member to create a member with its attribute tuple initialized for the named type of cell.

If the xcopy_dof_map argument to new_member is true, the new cell will get its own copy of the prototype attribute tuple. If xcopy_dof_map is false, the new cell will use the same copy already being used by other cells of the same type. This can be an important memory optimization, so one typically sets xcopy_dof_map false.

As an example, we recreate the line segment mesh using base_space_poset.

4.2.2.3.1 Example 22: Base space members using the base_space_poset interface.

```
#include "base_space_poset.h"
#include "fiber_bundles_namespace.h"
#include "poset_dof_map.h"
#include "std_iostream.h"
#include "storage_agent.h"

using namespace sheaf;
using namespace fiber_bundle;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example22:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example22");

  // Populate the namespace from the file we wrote in example21.
  // Retrieves the simple_poset example.

  storage_agent lsa_read("example21.hdf", sheaf_file::READ_ONLY);
  lsa_read.read_entire(lns);

  // Get a reference to the poset "mesh".

  poset_path lpath("mesh");
  base_space_poset& lposet =
    lns.member_poset<base_space_poset>(lpath, true);

  // Create the vertices as copies of the point prototype.
  // Initializes all member attributes to be same as point prototype.

  pod_index_type lv0_pod = lposet.new_member("point", false);
  lposet.put_member_name(lv0_pod, "v0", true);

  pod_index_type lv1_pod = lposet.new_member("point", false);
  lposet.put_member_name(lv1_pod, "v1", true);

  // Create the segment as a copy of the segment prototype.
  // Initializes all member attributes to be same as segment prototype.

  pod_index_type ls0_pod = lposet.new_member("segment", false);
  lposet.put_member_name(ls0_pod, "s0", true);

  // Print the attribute tuples for the members.

  cout << "v0 tuple: " << lposet.member_dof_map(lv0_pod, false);
  cout << endl;
  cout << "v1 tuple: " << lposet.member_dof_map(lv1_pod, false);
  cout  << endl;
  cout << "s0 tuple: " << lposet.member_dof_map(ls0_pod, false);
  cout << endl;

  // Make the segment cover the vertices.

  lposet.new_link(ls0_pod, lv0_pod);
  lposet.new_link(ls0_pod, lv1_pod);

  // Top covers the segment.

  lposet.new_link(TOP_INDEX, ls0_pod);

  // The vertices cover bottom.
  lposet.new_link(lv0_pod, BOTTOM_INDEX);
  lposet.new_link(lv1_pod, lposet.bottom().index().pod());

  // We're finished creating and linking jims.

  lposet.end_jim_edit_mode();

  // Print the finished poset.

  cout << lposet << endl;

  // Write it to a file for later use.

  storage_agent lsa_write("example22.hdf", sheaf_file::READ_WRITE);
  lsa_write.write_entire(lns);

  // Exit:

  return 0;
}
```

4.2.2.4 Base space member handles

There are a number of handle classes for members of base space posets: base_space_member and the several block handle classes.

4.2.2.4.1 Base_space_member

Base_space_member is the general interface to members of base_space_posets. It is_a total_poset_member that knows about the attributes that all base space members have in common:

int db () const

The base space dimension.

const pod_index_type& type_id () const

The cell type id of this. The id of the prototype of this in the "cell_types" id space of the prototypes poset.

const char* type_name() const

The cell type name.

int refinement_depth() const

The refinement depth.

Base_space_member provides a convenient interface for constructing any of the cell types in Table 1:

fiber_bundle::base_space_member::
base_space_member(base_space_poset * xhost,

const string& xprototype_name,
bool xcopy_dof_map,
bool xauto_access)

Creates a new handle attached to a new jim state in xhost using the prototype with name xname. If xcopy_dof_map or if xhost does not already contain a copy of the prototype dof map, create a copy of the dof map, otherwise just refer to an existing copy.

For instance, we can create the two vertices and segment in the last example with:

```
base_space_member lv0(&lposet, "point", false, true);
base_space_member lv1(&lposet, "point", false, true);
base_space_member ls0(&lposet, "segment", false, true);
```

4.2.2.4.2 Blocks

As we said before, a block is a collection of cells of the same type. In other words, a block is pretty much what we think of as a mesh, or at least a simple mesh. Some meshes require multiple blocks, perhaps of different kinds of cells.

More precisely, a collection of cells in which the maximal dimension cells, which we refer to as "zones" or "elements", are all the same. Each maximal dimension cell may contain other kinds of cells in its downset. For instance, zones of any dimension typically contain vertices. The SheafSystem supports several different types of blocks, differentiated by the type of zones and how they are connected to each other:

Point blocks have zones are that isolated, disconnected 0-dimensional points. For class point_block_1d, the points can be indexed by a single index., class point_block_2d by a pair of indices (i, j) and class point_block_3d by a triple of indices (i, j, k)

Structured blocks are traditional i, (i,j) or (i,j,k) grids. The zones are 1, 2, or 3 dimensional boxes (line segments, quadrangles, or hexahedra, respectively) arranged in a regular 1, 2, or 3 index array and connected by their boundaries. Classes structured_block_1d, structured_block_2d, and structured_block_3d, respectively.

Zone-nodes blocks are finite element style meshes. The zones are line segments, triangles, quadrangles, tetrahedra, or hexahedra with explicit client-specified nodal connectivity. Class zone_nodes_block.

Unstructured blocks are more general finite element style meshes. The zones are copies of a client-specified zone template, connected with explicit client-specified nodal connectivity. The zone templates can, for instance, contain faces and or edges. The zone templates defined in the base space member prototypes poset are shown in Table 2.

Table 2: Standard prototypes for zone templates

0-cells	1-cells	2-cells	3-cells
	segement_complex	triangle_nodes	tetra_nodes
		triangle_complex	tetra_complex

		quad_nodes	hex_nodes
		quad_complex	hex_faces_nodes
			hex_complex

More general meshes, in fact any decomposition of a domain, can be represented using the sheaf system, but we will not discuss such meshes in this document.

The implementation of each block type is optimized for the type. Unstructured blocks can represent any of the other block types, but will not be as efficient.

4.2.2.4.2.1 Creating blocks

Blocks can only be created using the appropriate handle class. For instance, to create our line segment mesh as a structured_block_1d use:

fiber_bundle::structured_block_1d::
structured_block_1d(poset * xhost, const size_type & xi_size, bool xauto_access)

Creates a new handle attached to a new state in xhost with i_size() == xi_size.

The attribute i_size() is the size of the mesh, that is, the number of zones:

sheaf::size_type fiber_bundle::structured_block_1d::i_size() const

The number of local cells (zones) in the i direction. The number of vertices in the i direction is i_size() + 1.

So we can create the entire line segment mesh with just:

```
structured_block_1d lblock(&lposet, 1, true);
```

4.2.2.4.2.2 Block id spaces

Every block provides 3 block specific id spaces:

- the local id space for all the members of the block,
- the zone id space for the zones in the block, and
- the vertex id space for the vertices in the block.

The local id space is a data member of the block handle (homgeneous_block is a base class for all blocks):

const sheaf::index_space_handle&
fiber_bundle::homogeneous_block::
local_id_space() const

The id space for the members of this block, including the block itself.

The zone id space and the vertex id space are accessed with the usual get/release pattern. for instance:

sheaf::index_space_handle&
fiber_bundle::homogeneous_block::
get_zone_id_space(bool xauto_access) const

Allocates a handle from the pool of handles for the id space of zones in this block.

and

void
fiber_bundle::homogeneous_block::
release_zone_id_space(index_space_handle& xid_space, bool xauto_access) const

Returns the zone id space xid_space to the pool of handles.

Iterators for these id spaces are also available directly from the block handle, for instance:

sheaf::index_space_iterator&
fiber_bundle::homogeneous_block::
get_zone_id_space_iterator(bool xauto_access) const

Allocates a zone id space iterator from the pool of iterators.

and

void
fiber_bundle::homogeneous_block::
release_zone_id_space_iterator(index_space_iterator& xitr, bool xauto_access) const

Returns the zone id space iterator xitr to the pool of iterators.

Getting the iterators directly is more efficient than first getting the id space and then getting the iterator from the id space. This can be important in deeply nested loops.

As an example of the use of blocks, we recreate the line segment mesh yet again, this time using structured_block_1d.

4.2.2.4.2.3 Example 23: Blocks

```
#include "base_space_poset.h"
#include "index_space_iterator.h"
#include "fiber_bundles_namespace.h"
#include "poset_dof_map.h"
#include "std_iostream.h"
#include "std_sstream.h"
#include "storage_agent.h"
#include "structured_block_1d.h"

using namespace sheaf;
using namespace fiber_bundle;
```

```
int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example23:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example23");

  // Populate the namespace from the file we wrote in example21.
  // Retrieves the simple_poset example.

  storage_agent lsa_read("example21.hdf", sheaf_file::READ_ONLY);
  lsa_read.read_entire(lns);

  // Get a reference to the poset "mesh".

  poset_path lpath("mesh");
  base_space_poset& lposet =
    lns.member_poset<base_space_poset>(lpath, true);

  // Create the line segment mesh as a structured block.
  // Set xauto_access to true so it will invoke begin_jim_edit_mode
  // and end_jim_edit_mode as needed, so we don't have to do it
  // ourselves. See the precondition for details.

  structured_block_1d lblock(&lposet, 1, true);
  lblock.put_name("block", true, true);

  // Iterate over the zones and give them names. (There's only 1).

  index_space_iterator& lzitr =
    lblock.get_zone_id_space_iterator(true);
  while(!lzitr.is_done())
  {
    stringstream lstr;
    lstr << "s" << lzitr.pod();
    lposet.put_member_name(lzitr.hub_pod(), lstr.str(), true, false);
    lzitr.next();
  }
  lblock.release_zone_id_space_iterator(lzitr, true);

  // Iterate over the vertices and give them names.

  index_space_iterator& lvitr =
    lblock.get_vertex_id_space_iterator(true);
  while(!lvitr.is_done())
  {
    stringstream lstr;
    lstr << "v" << lvitr.pod();
    lposet.put_member_name(lvitr.hub_pod(), lstr.str(), true, false);
    lvitr.next();
  }
  lblock.release_vertex_id_space_iterator(lvitr, true);

  // Print the finished poset.

  cout << lposet << endl;
  // Write it to a file for later use.

  storage_agent lsa_write("example23.hdf", sheaf_file::READ_WRITE);
  lsa_write.write_entire(lns);

  // Exit:

  return 0;
}
```

4.2.2.5 Subposets

Class base_space_poset maintains a subposet for the cells in the poset:

const sheaf::subposet&
fiber_bundle::base_space_poset::
cells() const

The subposet containing all the cells, of any dimension.

It also maintains a separate subposet for cells of each dimension:

const sheaf::subposet&
fiber_bundle::base_space_poset::
d_cells(int xd) const

The subposet containing the cells of dimension xd (const version).

The accessor function elements() is an alias for d_cells(maximal dimension):

sheaf::subposet&
fiber_bundle::base_space_poset::
elements()

The subposet containing the elements or zones, that is, the cells of maximal dimension.

while the accessor function vertices() is an alias for d_cells(0):

sheaf::subposet&
fiber_bundle::base_space_poset::
vertices()

The subposet containing all the vertices, that is, the cells of dimension.

The d-cells subposets are also available as id spaces:

const sheaf::mutable_index_space_handle&
fiber_bundle::base_space_poset::
d_cells_id_space(int xd) const

The id space for the subposet containing the cells of dimension xd (const version).

It's better to use the id space instead of the subposet because, as we mentioned in section 4.1.9, id spaces will eventually completely replace subposets.

4.2.2.6 Traversing the graph

The cover id space iterators and depth first iterators can be used to traverse the row graph of a base_space_poset, just like any ordinary poset. In addition, base_space_poset provides two additional id spaces and iterators, connectivity and adjacency.

4.2.2.6.1 Connectivity

The connectivity id space indexes the vertices in the downset of a zone. The connectivity id space is available either from base_space_poset or from a block handle. For instance:

sheaf::index_space_handle&
fiber_bundle::base_space_poset::
get_connectivity_id_space(pod_index_type xzone_id, bool xauto_access) const

Allocates an id space handle from the connectivity handle pool attached to the connectivity id space state for zone with id xzone_id.

and

void
fiber_bundle::base_space_poset::
release_connectivity_id_space_iterator(index_space_iterator& xid_space, bool xauto_access)

Returns the id space iterator xid_space to the connectivity iterator pool.

As usual, there is a similar interface for get/release of connectivity iterators.

4.2.2.6.2 Adjacency

The adjacency id space indexes the zones in the upset of a vertex. The adjacency id space is available either from base_space_poset or from a block handle. For instance:

sheaf::index_space_handle&
fiber_bundle::base_space_poset::
get_adjacency_id_space(pod_index_type xvertex_id, bool xauto_access) const

Allocates an id space handle from the adjacency handle pool attached to the adjacency id space state for the vertex with id xvertex_id.

void
fiber_bundle::base_space_poset::
release_adjacency_id_space(index_space_handle& xid_space, bool xauto_access) const

Returns the id space handle xid_space to the adjacency handle pool.

There is a similar interface for get/release of adjacency iterators.

4.2.3 Fiber spaces

The classes in the fiber spaces cluster provide the various algebraic types used in theoretical physics to describe the properties of particles and systems. There are currently more than 40 such types, organized into an inheritance hierarchy based on their mathematical definitions. The core of the hierarchy is shown in Figure 6.

The central types of this hierarchy are:

- Vd: abstract vector of dimension d
- Tp: general tensor of degree p
- ATp: antisymmetric tensor of degree p
- Stp: symmetric tensor of degree p

As shown in the figure, the hierarchy differentiates on symmetry and degree. In addition, since a tensor is fundamentally a map on vectors and a tensor space is defined with respect to the specific kind of vector it maps, the hierarchy differentiates on vector type as well. The figure shows only a portion of the complete hierarchy supported by the Sheaf System, the portion corresponding to degrees 0 through 2 and Euclidean vector spaces E2 and E3.

Although the mathematical structure of this hierarchy is critical to the design and implementation of the Sheaf System class libraries, applications typically only use the leaf classes in the core of the hierarchy, as shown in the figure.

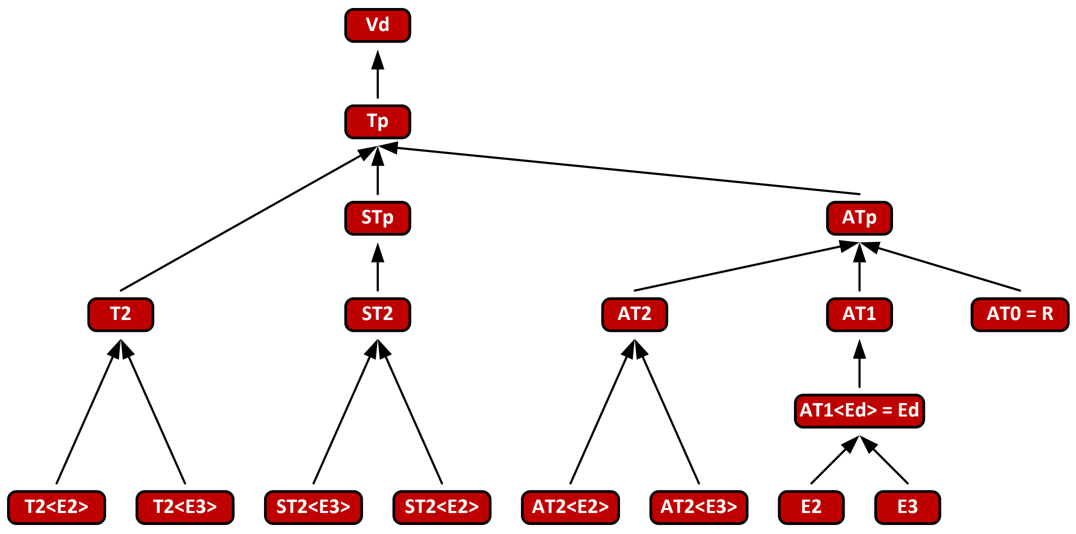

Figure 6: The core of the physical property inheritance hierarchy.

4.2.3.1 Fiber space schema

The sheaf schema for these types is defined in the "fiber_space_schema" table in the fiber_bundle_namespace. Example 24 prints a useful listing of all the types defined in the fiber_space_schema poset, and provides an interesting example of how to use the biorder_itr. The output is in file example24.cout

4.2.3.1.1 Example 24: Fiber schema subobject hierarchy.

```
#include "biorder_itr.h"
#include "fiber_bundles_namespace.h"
#include "poset.h"
#include "std_iostream.h"

using namespace sheaf;
using namespace fiber_bundle;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example24:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example24");

  // Get a handle to the fiber space schema poset.

  poset& lfiber_schema =
  lns.member_poset<poset>(lns.standard_fiber_space_schema_poset_name(),
                          true);

  // Use a biorder_itr to print out the subobject hierarchy.

  cout << endl;
  cout << "Subobject hierarchy in " << lfiber_schema.name(true);
  cout << endl << endl;

  schema_poset_member lschema(lfiber_schema.top());

  int ltab_ct = 0;
  set_biorder_itr litr(lfiber_schema.top(), true, true);
  while(!litr.is_done())
  {
    pod_index_type lhub_pod = litr.index().hub_pod();
    switch(litr.action())
    {
      case set_biorder_itr::PREVISIT_ACTION:
        if(lhub_pod != BOTTOM_INDEX)
        {
          if(lfiber_schema.is_jim(lhub_pod))
          {
            for(int i=0; i<ltab_ct; i++) cout << "   ";
            cout << lfiber_schema.member_name(lhub_pod, true);
            if(lfiber_schema.is_atom(lhub_pod))
            {
              lschema.attach_to_state(lhub_pod);
              cout << ": " << lschema.type();
              cout << (lschema.is_table_dof() ? "  table" : "  row");
            }
            cout << endl;
            ltab_ct++;
          }
        }
        break;
      case set_biorder_itr::POSTVISIT_ACTION:
        if(lhub_pod != BOTTOM_INDEX)
        {
          if(lfiber_schema.is_jim(lhub_pod))
          {
            ltab_ct--;
            if(ltab_ct == 0)
            {
              cout << endl;
            }
            litr.put_has_visited(lhub_pod, false);
          }
        }
        break;
      default:
        break;
    }
    litr.next();
  }

  // Exit:

  return 0;
}
```

4.2.3.2 Fiber space posets

There are a number of specialized poset types for fiber space, corresponding the central types of the hierarchy. Specifically:

- tuple_space: poset for abstract tuples
- vd_space: poset for abstract vectors of dimension d
- tp_space: poset for general tensors of degree p
- stp_space: poset for symmetric tensors of degree p
- atp_space: poset for antisymmetric tensors of deggree p
- at1_space: poset for antisymmetric tensors of degree 1 (ordinary vectors)
- at0_space: poset for antisymmetric tensors of degree 0 (scalars)
- gln_space: poset for general linear transformations of a vector space of dimension n
- jcb_space: poset for Jacobians

The Jacobian and linear transformation types are not shown in Figure 6.

As with other posets, we create these spaces using the new_host or standard_host factory methods for the type of member we want the space to contain. The new_host methods are more general and correspondingly more complex, so we'll show how to use the standard_host methods in this section and defer discussion of the new_host methods until Part II.

To show how this works, let's consider making a poset for ordinary 3D Euclidean vectors, represented by class e3. The standard_host method is:

static
fiber_bundle::e3::host_type&
fiber_bundle::e3::
standard_host(namespace_type& xns,
const string& xsuffix,
bool xauto_access)

> The host with path standard_host_path(static_class_name(), xsuffix). Returns the host if it already exists, otherwise, creates it in namespace xns with schema specified by standard_schema_path() and standard paths for prerequisites, which are also created if needed.

So we can create a space for e3 objects with:

```
e3::host_type& le3_host = e3::standard_host(lns, "", true);
```

The name for the host will be e3::standard_host_path<e3>(""), which is "e3". We can create a host with a (slightly) different name by providing a non-empty suffix. For instance:

```
e3::host_type& le3_other_host = e3::standard_host(lns, "_other", true);
```

will create a poset with with name "e3_other".

4.2.3.2.1 Example 25: Creating a fiber space.

```
#include "at1_space.h"
#include "e3.h"
#include "fiber_bundles_namespace.h"
#include "std_iostream.h"

using namespace sheaf;
using namespace fiber_bundle;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example25:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example25");

  // Create a space for e3 objects; empty suffix
  // e3::host_type is at1_space

  at1_space& le3_space = e3::standard_host(lns, "", true);

  cout << "e3 space name: " << le3_space.name() << endl;
  cout << "e3 space schema name: "
       << le3_space.schema().name() << endl;
  cout << "e3 space scalar space path: "
       << le3_space.scalar_space_path() << endl;

  // Create another space for e3 objects, suffix "_other".
  at1_space& le3_other_space = e3::standard_host(lns, "_other", true);

  cout << endl;
  cout << "another e3 space name: " << le3_other_space.name() << endl;
  cout << "another e3 space schema name: "
       << le3_other_space.schema().name() << endl;
  cout << "another e3 space scalar space path: "
       << le3_other_space.scalar_space_path() << endl;

  // Exit:

  return 0;
}
```

4.2.3.3 Fiber space members

The specialized fiber space posets all inherit poset and the fiber space members can be manipulated with the member interface on the poset classes the same way we've seen in previous examples. The fiber space poset classes do add a few member functions associated with the algebraic types they represent, especially the tensor classes, but those are advanced features beyond the scope of this introduction.

4.2.3.4 Fiber space member handles

The fiber space member handles are the heart of the fiber space subcomponent. As described above, there are a number of them. But for practical purposes, the leaf classes in Figure 6 are the important ones: the Euclidean vector spaces in dimension 1, 2 and 3 and the several tensor types defined over these vector spaces.

We've already seen how to create a poset for e3, now let's create some members in that space. We've already seen in section 4.1.6 how to use total_poset_member handles to create new members and fiber space members are no different. Every e3 vector we want to create is a jim, so we can use the constructor:

fiber_bundle::e3::
e3(poset_state_handle* xhost, bool xauto_access = true)

Creates a new e3 handle attached to a new jim state in xhost.

Class e3, like most of the fiber types, is a descendant of the abstract vector space class vd and hence has components. We can set and get component values is a variety of ways. The simplest are:

void
fiber_bundle::vd::
put_component(int xindex, value_type xvalue)

Sets the value of the xindex-th component to xvalue.

and

vd::value_type
fiber_bundle::vd::
component(int xindex) const

The value of the xi-th component.

4.2.3.4.1 Algebraic operations and namespaces

Each fiber type is associated with an algebra. For instance, the algebra associated with class e3 is Euclidean vector algebra. We refer to all these specific algebra operations collectively as the fiber algebra. The algebraic operations are defined as namespace scope functions rather than member functions and there is a subnamespace of fiber_bundle for each algebra. The subnamespace is named after the associated class, for instance the general vector algebra is in namespace fiber_bundles::vd_algebra and the 3D Euclidean vector algebra is in namespace fiber_bundle::e3_algebra. Appendix C summarizes the operations available in each algebra, see the namespace reference documents under Namespaces/Namespace list/fiber_bundle for a complete listing.

The fiber algebra is still under development. Currently, there is only partial support for the fiber algebra on the persistent fiber types, but more complete support for volatile fiber types.

4.2.3.4.2 Persistent, volatile, and POD types

The fiber space types are most often used for short calculations, typically inside a loop where the fiber variable is just a temporary. For such purposes, the schema, access control, and persistence properties of the sheaf tables are unnecessary, inconvenient, and frequently inefficient. For this reason, every persistent fiber type F has an associated volatile type, F_lite. For instance e3 has e3_lite. The lite types have all the same data members and algebraic operations as the full fiber types, but don't "live" in a table and hence aren't persistent. The volatile types don't have an explicit schema and aren't access-controlled.

Lite types are intended to be easy and efficient to use as temporary variables. In particular, they can be created without a table. For instance:

fiber_bundle::e3_lite::
e3_lite(const value_type& x, const value_type& y, const value_type& z)

Creates a new instance with components x, y, z.

Both the persistent fiber types and the lite fiber types are C++ classes with virtual functions. As a result, the byte by byte layout of instances of these types is compiler defined and usually contains data in addition to the data members defined by the SheafSystem. This makes it difficult to pass to any untyped, generic context, which includes the sheaf i/o subsystem as well as external C or FORTRAN functions. For this reason, each fiber type has a third associated type, the "plain old data" or "POD" type. (This nomenclature may seem informal to the point of being flippant, but it is in fact the official ISO C++ standard terminology!). For instance e3::row_dofs_type and e3_lite::row_dofs_type both are typedefed to the pod type e3_row_dofs_type.

The pod types satisfy the requirements of C++ POD types. They have no user-defined constructors or assignment operators and only public data members and member functions. They support data member access and conversion to and from full or lite types, but not algebraic operations.

The pod type provides the mechanism for easy conversion between persistent and volatile types. Both support construction from, conversion to, and assignment from the pod type. For instance:

fiber_bundle::e3_lite::
e3_lite(const row_dofs_type & xrow_dofs)

Creates a new instance with row dofs xrow_dofs.

fiber_bundle::e3_lite::
operator e3_lite::row_dofs_type& ()

Conversion (cast) operator to convert to the associated row dofs type (non const version).

fiber_bundle::e3_lite&
fiber_bundle::e3_lite::
operator= (const row_dofs_type & xrow_dofs)

Row_dofs_type assignment operator.

Implicit conversion to the pod type and construction or assignment from the pod type allows seemingly direct conversion and assignment between the persistent and volatile types. Example 26 shows how persistent and types work, and how they work together.

4.2.3.4.3 Example 26: Persistent and volatile types.

```
#include "at1_space.h"
#include "e3.h"
#include "fiber_bundles_namespace.h"
#include "poset.h"
#include "std_iostream.h"

using namespace sheaf;
using namespace fiber_bundle;
using namespace fiber_bundle::vd_algebra;
using namespace fiber_bundle::ed_algebra;
using namespace fiber_bundle::e3_algebra;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example26:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example26");
```

```
// Create a standard space for e3 objects; empty suffix
// e3::host_type is at1_space

at1_space& le3_space = e3::standard_host(lns, "", true);

// Create some persistent vectors.

e3 i_hat(&le3_space, true);
e3 j_hat(&le3_space, true);

// Set the components of i_hat individually.

i_hat.put_component(0, 1.0, false);
i_hat.put_component(1, 0.0, false);
i_hat.put_component(2, 0.0, false);

// Set the components of j_hat all at once.

j_hat.put_components(0.0, 1.0, 0.0);

// Create and initialize some volatile vectors

e3_lite i_hat_v(i_hat);
e3_lite j_hat_v;
j_hat_v = j_hat;

cout << "i_hat_v:" << i_hat_v << endl;
cout << "j_hat_v:" << j_hat_v << endl;

// "Auto-allocated" add creates result on heap.

e3_lite* v1_v = add(i_hat_v, j_hat_v);

cout << "v1_v:\t" << *v1_v << endl;

// Pre-allocated add requires result as argument

e3_lite v2_v;
add(i_hat_v, j_hat_v, v2_v);

cout << "v2_v:\t" << v2_v << endl;

// "Self-allocated" add uses first arg as result.

e3_lite v3_v(i_hat_v);
add_equal(v3_v, j_hat_v);

cout << "v3_v:\t" << v3_v << endl;

// Operations are also available as overloaded operators,
// whenever it makes sens.

e3_lite* v4_v = i_hat_v + j_hat_v;

cout << "v4_v:\t" << *v4_v << endl;
```

```
  e3_lite v5_v(i_hat_v);
  v5_v += j_hat_v;

  cout << "v5_v:\t" << v5_v << endl;

  // All the usual operations of Euclidean vector algebra are
  // available as functions, and also as operators, when it
  // makes sense. For instance:

  // Multiplication by scalar

  e3_lite v6_v;
  multiply(i_hat_v, 2.0, v6_v);

  cout << "v6_v:\t" << v6_v << endl;

  e3_lite v7_v(i_hat_v);
  v7_v *= 2.0;

  cout << "v7_v:\t" << v7_v << endl;

  // Dot product.

  e3::value_type ls0 = dot(i_hat_v, v5_v);
  e3::value_type ls1 = i_hat_v*v5_v;

  size_type lprec = cout.precision(18);
  cout << "ls0:\t" << scientific << setw(27) << ls0 << endl;
  cout << "ls1:\t" << scientific << setw(27) << ls1 << endl;
  cout.precision(lprec);

  // Cross product

  e3_lite k_hat_v;
  cross(i_hat_v, j_hat_v, k_hat_v);

  cout << "k_hat_v:" << k_hat_v << endl;

  k_hat_v = v6_v;
  k_hat_v ^= v5_v;

  cout << "k_hat_v:" << k_hat_v << endl;

  // Normalization.

  normalize(k_hat_v);

  cout << "k_hat_v:" << k_hat_v << endl;

  // Convert volatile to persistent;
  // uses implicit coversion to row_dofs_type.

  e3 k_hat(le3_space, k_hat_v, true);

  // Exit:

  return 0;
```

}

4.2.3.5 Subposets

There are no additional subposet features associated with fiber spaces.

4.2.3.6 Traversing the graph

There are no additional graph traversal methods associated with the fiber spaces.

4.2.4 Section spaces

We saw in Part Spaces For Scientific Computing and The Sheaf System Analysis and Design Tutorial that a section is a member of a section space poset and in practical cases the schema for a section space poset is the tensor product of the base space lattice and the fiber schema lattice. The set of jims of the tensor product is the Cartesian product of the jims of the base space and the jims of the fiber space schema. The members of the tensor product are pairs (base space member, fiber schema member) and joins of such members with no components in common. A pair (base space member, fiber schema member) corresponds to a "homogenous" representation, the fiber schema is the same everywhere on the given base space member. A join of pairs corresponds to an" inhomogeneous" representation, a collection of homogeneous schema for different parts of the base space. What the fiber schema is depends on which base space member you are talking about.

A direct representation of this poset, using the same method as the other posets we've described, would require instantiating the set of jims and any jrms we are interested in. Such a naive implementation is however infeasible - in practical cases the Cartesian product of the base space jims and fiber schema jims is just too large to store.

So some method other than just storing the product needs to be used. The approach currently implemented in the SheafSystem provides only a subspace of the entire tensor space, specifically the Cartesian subspace consisting of only the pairs (base space member, fiber schema member), and hence the current implementation can support only homogeneous representations.

This approximation, which we will refer to as the product subspace approximation, synthesizes the product on the fly, using special iterators, whenever it is needed. As a result, a section space schema poset is actually *empty*, it contains only the two standard members top and bottom. We can attach section_space_schema_member handles to members as if they actually existed, but there are no member objects stored in the poset. All this is transparent to the client, as long as one accesses the poset through the special member handles, iterators, and id spaces provided for the purpose.

4.2.4.1 Section space posets

There are a number of specialized poset types for section spaces, one for each specialized type of fiber space. Specifically:

- sec_tuple_space: poset for sections with fiber abstract tuple

- sec_vd_space.h: poset for sections with fiber abstract vectors of dimension d
- sec_tp_space: poset for sections with fiber general tensors of degree p
- sec_stp_space: poset for sections with fiber symmetric tensors of degree p
- sec_atp_space: poset for sections with fiber antisymmetric tensors of deggree p
- sec_at1_space: poset for sections with fiber antisymmetric tensors of degree 1 (ordinary vectors)
- sec_at0_space: poset for sections with fiber antisymmetric tensors of degree 0 (scalars)
- sec_jcb_space: poset for sections with fiber Jacobians

We create these spaces using the new_host or standard_host factory methods for the type of member we want the space to contain. The section space member hierarchy repeats the fiber space member hierarchy; there is a section space member class sec_<fiber type> for each fiber type in the fiber space member hierarchy. As with the fiber space factory methods, the new_host methods are more general and correspondingly more complex. We'll show how to use the standard_host methods in this section and defer discussion of the new_host methods until Part II.

Let's make a section space for sections of type sec_e2, that is, sections with ordinary 2D Euclidean vectors as the fiber type, over our line segment mesh. The standard_host method is:

```
static
fiber_bundle::sec_e2::host_type&
fiber_bundle::sec_e2::
standard_host(namespace_type& xns,
              const poset_path& xbase_path,
              const poset_path& xrep_path,
              const string& xsection_suffix,
              const string& xfiber_suffix,
              bool xauto_access)
```

The standard host for sections of this type with base space xbase_path, representation xrep_path, section suffix xsection_suffix, and fiber suffix xfiber_suffix. Creates the host and its prerequisites if necessary.

To explain the arguments to this function, we have to take a brief detour into section space schema.

4.2.4.2 Section space schema

Section spaces differ from the base spaces, fiber spaces, and other posets we've seen so far in an important way. Both the base spaces and the fiber spaces could use a predefined schema, but the schema of a section space depends on the base space, so the schema cannot be predefined. It has to be created once we know what the base space should be.

The section standard_host factory method does not require the client to specify the schema path explicitly, it will find the appropriate section space schema if it already exists or create it if it doesn't. But in order to do this, the factory method needs to know three pieces of information:

1. the base space, the mesh the section space is based on;
2. the fiber space schema, the type of the dependent variable of the section; and
3. the representation type ("rep type"), the method used to discretize a section.

The base space must always be specified by the client, so that is an argument to the standard_host function. Each section type knows what its fiber type is, so there is no need for fiber type to be an argument for the standard_host method. That leaves the rep type.

4.2.4.2.1 Representation type

A representation type is identified by three further pieces of information:

4. the discretization subposet, the subset of cells in the mesh with which the section data (attributes) are associated;
5. the evaluation subposet, the subset of cells with which the interpolation functions are associated; and
6. the evaluation method, the type of interpolation function.

A typical rep type would specify the vertices as the discretization subposet, the zones as the evaluation subposet, and linear interpolation as the evaluation method. We can visualize a representation type by tagging the members of the discretization and evaluation subposets in the graph of the base space, as shown for our line segment example in Figure 7.

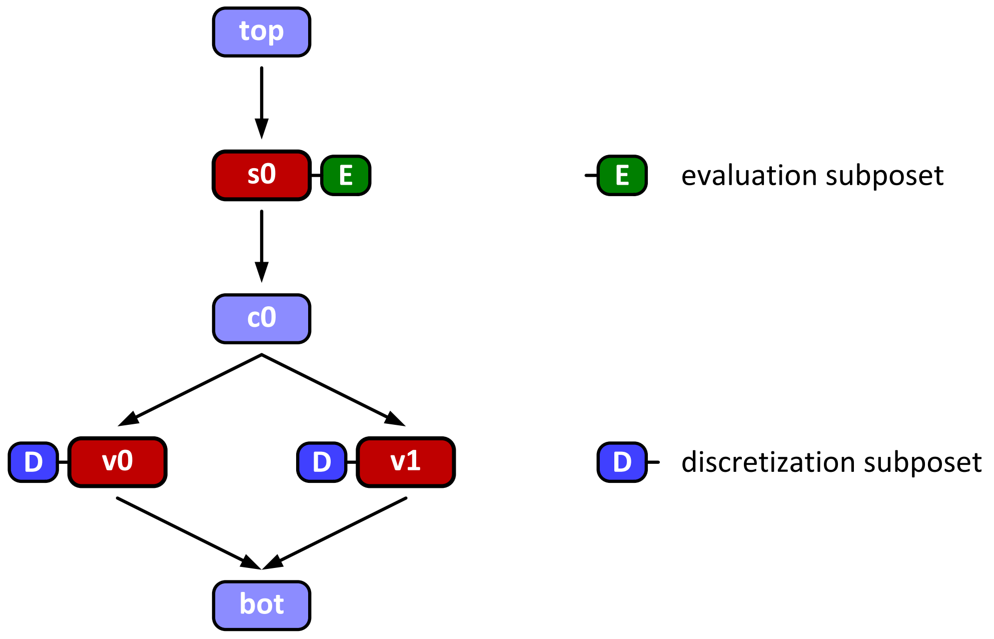

Figure 7: Rep type for the line segment mesh.

4.2.4.2.2 Sec_rep_descriptors

The poset "sec_rep_descriptors" in the fiber bundles namespace provides a number of predefined rep types, with names derived from these three quantities. For instance, "vertex_element_dlinear" is a representation with the data at the vertices, interpolated over the elements (zones) using linear, bilinear, or trilinear interpolation, depending on the dimension of the mesh and the type of zone. Another example is "element_element_constant", for which the data is attached to the zones and is "interpolated" over the zones as a constant value. There are a number of other predefined rep types that address common cases, or the client can define specialized rep types. The standard types predefined in the sec_rep_descriptors poset of the fiber_bundles_namespace are shown in Table 3.

Table 3: Standard sec_rep_descriptors

representation type	discretization subposet	evaluation subposet	evaluation method
vertex_element_dlinear	__vertices	__elements	dlinear
vertex_cells_dlinear	__vertices	__cells	dlinear
element_element_constant	__elements	__elements	constant
vertex_block_dlinear	__block_vertices	__blocks	dlinear
vertex_block_uniform	__block_vertices	__blocks	uniform
vertex_vertex_constant	__vertices	__vertices	constant

4.2.4.2.3 Standard representation path

Each section type defines a standard representation. For instance, for sec_e2 we have:

static
const sheaf::poset_path&
fiber_bundle::sec_tuple::
standard_rep_path()

The path to the standard rep for sections of this type.

Sec_e2, like many of the section classes, inherits its standard rep path from sec_tuple, the base of the section hierarchy. Currently, sec_tuple::standard_rep_path() defines its result to be vertex_element_dlinear, so the standard representation for most section classes is data on the vertices with linear, bilinear, or trilinear interpolation over the elements (zones).

4.2.4.3 Creating section spaces.

Returning to creating a section space for sec_e2 over our line segment mesh, the base space is the structured_block_1d mesh we created in section 4.2.2.4.2.1. If we leave the rep path argument empty, standard_host will use the standard rep path. So we can create the section space with:

```
  poset_path lbase_path("mesh/block");

  sec_e2::host_type& lsec_e2_host =
    sec_e2::standard_host(lns, lbase_path, "", "", "", true);
```

Example 28 creates an e2 section space on our line segment mesh.

Example 28: Creating a section space.

```
#include "fiber_bundles_namespace.h"
#include "sec_at1_space.h"
#include "sec_e2.h"
#include "std_iostream.h"
#include "storage_agent.h"

using namespace sheaf;
using namespace fiber_bundle;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example28:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example28");

  // Populate the namespace from the file we wrote in example25.
  // Retrieves the base space.
  storage_agent lsa_read("example23.hdf", sheaf_file::READ_ONLY);
  lsa_read.read_entire(lns);

  // Create a section space for sec_e2 sections
  // on the line segment mesh
  // using standard rep and no suffixes.

  poset_path lbase_path("mesh/block");

  sec_e2::host_type& lsec_e2_host =
    sec_e2::standard_host(lns, lbase_path, "", "", "", true);

  cout << endl;
  cout << "sec_e2 space path: " << lsec_e2_host.path() << endl;
  cout << "sec_e2 space schema path: ";
  cout << lsec_e2_host.schema().path() << endl;
  cout << "sec_e2 space scalar space path: ";
  cout << lsec_e2_host.scalar_space_path() << endl;

  // Print the finished poset.

  cout << lsec_e2_host << endl;

  // Create a section space for sec_e2 sections
  // on the line segment mesh using rep "element_element_constant"
  // and both section and fiber suffixes.

  poset_path
    lrep_path("sec_rep_descriptors", "element_element_constant");

  sec_e2::host_type& lsec_e2_other_host =
    sec_e2::standard_host(lns, lbase_path, lrep_path,
                          "_other", "_new", true);

  cout << endl;
  cout << "other sec_e2 space path: ";
  cout << lsec_e2_other_host.path() << endl;
  cout << "other sec_e2 space schema path: ";
  cout << lsec_e2_other_host.schema().path() << endl;
  cout << "other sec_e2 space scalar space path: ";
  cout << lsec_e2_other_host.scalar_space_path() << endl;

  // Write the namespace to a file for later use.

  storage_agent lsa_write("example28.hdf", sheaf_file::READ_WRITE);
  lsa_write.write_entire(lns);

  // Exit:

  return 0;
}
```

4.2.4.5 Section space members

The specialized section space posets all inherit poset and the section space members can be manipulated with the member interface on the poset classes the same way we've seen in previous examples. As with the fiber space poset classes, the section space class add a few member functions associated with the algebraic types they represent, especially the tensor classes, but those are advanced features beyond the scope of this introduction.

4.2.4.6 Section space member handles

There is a section space member handle type for each fiber type and, as with the fiber types, it is the leaf types in Figure 6 that are important in practice.

4.2.4.6.1 Creating and accessing sections.

We create sections in much the same way we create fiber objects. We'll use sec_e2 as an example:

```
fiber_bundle::sec_e2::
sec_e2(sec_rep_space* xhost,
        section_dof_map* xdof_map = 0,
        bool xauto_access = true)
```

Creates a new sec_e2 handle attached to a new jim state in xhost.

We usually create let the constructor create the attribute tuple by accepting the default arguments.

4.2.4.6.1.1 Section id spaces

In the product subspace approximation, an instance of the fiber is stored for each member of the discretization subposet in the base space and the fiber_space_schema defines the data that represents each instance of the fiber.

This means that there are 3 id spaces associated with the section attributes: the fiber attribute id space, the discretization id space, and the section attribute id space. The fiber attribute id space is the row attribute id space of the fiber schema. The discretization id space is the id space of the discretization subposet. The section attributes id space is the Cartesian product of the discretization id space and the fiber attribute id space.

The interface for accessing the section values is based on these 3 id spaces, with an accessor function, a mutator function and an iterator for each id space.

4.2.4.6.1.2 Discretization id access

Typically the most convenient access is via discretization id. One can iterate over all the discretization "points" of a section and, since an instance of the fiber type is associated with each discretization id, one can get or put a fiber value:

The discretization id space is available from the schema:

const index_space_handle&
fiber_bundle::section_space_schema_member::
discretization_id_space() const

The id space for the discretization members in the down set of the base space of this (const version).

The value at each discretization id is a copy of the fiber. The accessor, get_fiber, and the mutator, put_fiber, are defined in the base class sec_vd:

void
fiber_bundle::sec_vd::
get_fiber(pod_index_type xdisc_id, vd_lite& xfiber) const

Sets xfiber to the fiber referred to by discretization id xdisc_id.

void
fiber_bundle::sec_vd::
put_fiber(pod_index_type xdisc_id, const vd_lite& xfiber)

Sets the fiber referred to by discretization id xdisc_id to xfiber.

4.2.4.6.1.3 Section attribute id access

Occasionally it is more convenient to access the individual attributes. In that case, one can iterate directly over the attribute ids of a section and get or put an individual attribute value. The row attribute id space is available from a section schema just like it is from an ordinary schema member, in fact the section schema inherits the dof_id_space accessor from schema_poset_member:

const sheaf::index_space_handle&
sheaf::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 convenience and clarity, the accessor row_dof_id_space() is defined as a synonym for dof_id_space(false).

The individual attributes can be accessed in the same way as those of ordinary poset members, as described in section 4.1.7.3.4. In addition, since every attribute corresponds to a pair (discretization id, fiber attribute id), there are accessor and mutator signatures for this combination of ids, inherited from base class sec_rep_space_member:

```
void
fiber_bundle::sec_rep_space_member::
get_dof(pod_index_type xdisc_id,
        pod_index_type xfiber_dof_id,
        void* xdof,
```

size_type xdof_size) const

Copies the dof referred to by xdisc_id, xfiber_dof_id into xdof.

void
fiber_bundle::sec_rep_space_member::
put_dof(pod_index_type xdisc_id,
pod_index_type xfiber_dof_id,
const void* xdof,
size_type xdof_size)

Sets the dof referred to by xdof_id to the value at xdof.

4.2.4.6.1.4 Fiber attribute id access

One can also access the section values by fiber attribute id. This access method is included mostly for mathematical completeness. It is rarely used in modern applications but can be useful in the context of legacy code oriented towards vector processor architectures.

Since the fiber attributes are usually the components of the fiber, the collection of section attributes with a given fiber attribute id usually constitute a component of the section. Hence, this access method is called component access.

void
fiber_bundle::sec_rep_space_member::
get_component(pod_index_type xfiber_dof_id,
void* xcomponent,
size_type xcomponent_size) const

Sets xcomponent to the component referred to by fiber id xfiber_dof_id.

void
fiber_bundle::sec_rep_space_member::
put_component(pod_index_type xfiber_dof_id,
const void* component,
size_type xcomponent_size)

Sets the xcomponent referred to by fiber id xfiber_dof_id to xcomponent.

4.2.4.6.1.5 Example 29: Creating and accessing sections.

```
#include "e2.h"
#include "fiber_bundles_namespace.h"
#include "index_space_handle.h"
#include "index_space_iterator.h"
#include "sec_at1_space.h"
#include "sec_e2.h"
#include "std_iostream.h"
#include "storage_agent.h"

using namespace sheaf;
using namespace fiber_bundle;
using namespace fiber_bundle::vd_algebra;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example29:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example29");

  // Populate the namespace from the file we wrote in example28.
  // Retrieves the section space.

  storage_agent lsa_read("example28.hdf", sheaf_file::READ_ONLY);
  lsa_read.read_entire(lns);

  // Get a handle for the section space.

  poset_path lssp_path("e2_on_block");

  sec_e2::host_type& lhost =
    lns.member_poset<sec_e2::host_type>(lssp_path, true);

  // Create a section.

  sec_e2 lsec1(&lhost);

  // Initialize the first section using the disc id interface.

  // First, get an instance of the lite fiber type
  // (e2_lite in this case).

  sec_e2::fiber_type::volatile_type lfiber(0.0, 1.0);

  // Iterate over the discretization setting the fibers.

  index_space_handle& ldisc_id_space =
    lsec1.schema().discretization_id_space();

  index_space_iterator& ldisc_itr = ldisc_id_space.get_iterator();
  while(!ldisc_itr.is_done())
  {
    // Note that we use the pod id, not the hub pod id,
    // the id has to be in the disc id space.

    lsec1.put_fiber(ldisc_itr.pod(), lfiber);
    ldisc_itr.next();
  }

  // Create a second section and initialize it
  // from the first section using disc id interface.

  sec_e2 lsec2(&lhost);

  sec_e2::value_type lval = 1.0;
  ldisc_itr.reset();
  while(!ldisc_itr.is_done())
  {
    lsec1.get_fiber(ldisc_itr.pod(), lfiber);

    lfiber *= lval;
    lval *= 2;

    lsec2.put_fiber(ldisc_itr.pod(), lfiber);
    ldisc_itr.next();
  }

  // Create a third section and initialize it
  // from the first section using section attribute id interface.

  sec_e2 lsec3(&lhost);

  const index_space_handle& lsec_id_space =
    lsec1.schema().row_dof_id_space();
  index_space_iterator& lsec_id_itr = lsec_id_space.get_iterator();
  while(!lsec_id_itr.is_done())
  {
    lval = lsec1.dof(lsec_id_itr.pod());
    lval *= 2.0;
    lsec3.put_dof(lsec_id_itr.pod(), lval);
    lsec_id_itr.next();
  }
  lsec_id_space.release_iterator(lsec_id_itr);

  // Create a fourth section and initialize it
  // from the first section using disc id, fiber id interface.

  sec_e2 lsec4(&lhost);

  const index_space_handle& lfiber_id_space =
    lsec1.schema().fiber_schema().row_dof_id_space();
  index_space_iterator& lfiber_id_itr = lfiber_id_space.get_iterator();

  ldisc_itr.reset();
  while(!ldisc_itr.is_done())
  {
    while(!lfiber_id_itr.is_done())
    {
      lsec1.get_dof(ldisc_itr.pod(), lfiber_id_itr.pod(),
                    &lval, sizeof(lval));

      lval *= 3;

      lsec4.put_dof(ldisc_itr.pod(), lfiber_id_itr.pod(),
                    &lval, sizeof(lval));

      lfiber_id_itr.next();
    }
    lfiber_id_itr.reset();
    ldisc_itr.next();
  }

  // Create a fifth section
  sec_e2 lsec5(&lhost);

  // Allocate a buffer to store a section component
  // All the components are the same size for a sec_e2.

  size_type lcomp_size = lsec1.schema().component_size(0);

  sec_e2::dof_type* lcomp = new sec_e2::dof_type[lcomp_size];

  // Initialize the 5th section using component access.

  lfiber_id_itr.reset();
  while(!lfiber_id_itr.is_done())
  {
    lsec1.get_component(lfiber_id_itr.pod(), lcomp, lcomp_size);
    lsec5.put_component(lfiber_id_itr.pod(), lcomp, lcomp_size);
    lfiber_id_itr.next();
  }

  lfiber_id_space.release_iterator(lfiber_id_itr);
  ldisc_id_space.release_iterator(ldisc_itr);

  // Print the finished poset.

  cout << lhost << endl;

  // Write it to a file for later use.

  storage_agent lsa_write("example29.hdf", sheaf_file::READ_WRITE);
  lsa_write.write_entire(lns);

  // Exit:

  return 0;
}
```

4.2.4.6.2 Algebraic operations and namespaces

The section type hierarchy repeats the fiber type hierarchy: for every type F in the fiber hierarchy there is a section type S with fiber of type F. Every fiber operation has a corresponding section operation. See Appendix D for additional discussion.

4.2.4.6.3 Coordinate sections

The positions of the cells in a mesh are not an intrinsic feature of the mesh in the sheaf system. Positions are assigned to the cells by defining a coordinates section on the mesh. A mesh can have a single coordinates section or many coordinates sections. For instance, if the positions of the vertices change with time, then the coordinates at each time step can be stored as a separate section.

Any section which defines a one-to-one mapping of points in the base space to position can be used as a coordinates section, but the sheaf system provides two types of sections specifically intended for coordinates: uniform coordinates and general coordinates.

4.2.4.6.3.1 Creating uniform coordinates on structured blocks

Uniform coordinates can be used only on structured blocks. Uniform coordinates are defined by positions at the corners of the mesh and are interpolated uniformly (actually, d-linearly, where d is the dimension of the mesh) on the interior of the mesh. Assuming the positions are chosen to be a d-dimensional box, uniform coordinates make the mesh a "regular" grid - the vertices are evenly spaced and the zones are all the same size.

4.2.4.6.3.2 Creating general coordinates

General coordinates can be created on any type of block using the sec_e1, sec_e2, or sec_e3 classes. The client has to explicitly set the attributes of the coordinates after creating the section. In the case of complex mesh shapes, as is usually the case with zone-node and unstructured blocks, the block vertex id space is just a simple id space. The positions of the vertices were presumably determined by the mesher, at the same time the nodal connectivity was defined, and this position information can be used.

4.2.4.6.3.3 Example 30: Creating coordinate sections

```
#include "e2.h"
#include "fiber_bundles_namespace.h"
#include "index_space_handle.h"
#include "index_space_iterator.h"
#include "sec_at1_space.h"
#include "sec_e1.h"
#include "sec_e1_uniform.h"
#include "std_iostream.h"
#include "storage_agent.h"
#include "structured_block_1d.h"

using namespace sheaf;
using namespace fiber_bundle;
using namespace fiber_bundle::vd_algebra;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example30:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example30");

  // Create a new base space with structured_block_1d with 2 segments.

  base_space_poset& lbase_host =
    structured_block_1d::standard_host(lns, "mesh2", true);

  structured_block_1d lblock(&lbase_host, 2, true);
  lblock.put_name("block", true, true);

  // Create a section space for uniform coordinates

  poset_path lbase_path("mesh2/block");
  sec_e1_uniform::host_type& le1u_host =
    sec_e1_uniform::standard_host(lns, lbase_path, "", "", "", true);

  // Create the coordinates section.
  // Uniform coordinates are initialized by the constructor.

  sec_e1_uniform le1u_coords(&le1u_host, -1.0, 1.0, true);
  le1u_coords.put_name("uniform_coordinates", true, true);

  // Create a section space for general coordinates;
  // use the default vertex_element_dlinear rep.

  sec_e1::host_type& le1_host =
    sec_e1::standard_host(lns, lbase_path, "", "", "", true);

  // Create a general coordinates section.

  sec_e1 le1_coords(&le1_host);
  le1_coords.put_name("general_coordinates", true, true);

  // Have to explicitly initialize general coordinates.
  // In a more realitistic case, the coordinate values
  // would come from some external source such as a mesher.
  // But this mesh only has 3 vertices, so we'll just make
  // something up that's definitely not uniform..

  sec_e1::dof_type lcoord_values[3] = {0.0, 1.0, 10.0};

  sec_e1::fiber_type::volatile_type lfiber;
  index_space_handle& ldisc_id_space =
    le1_coords.schema().discretization_id_space();
  index_space_iterator&  ldisc_itr = ldisc_id_space.get_iterator();
  while(!ldisc_itr.is_done())
  {
    lfiber = lcoord_values[ldisc_itr.pod()];
    le1_coords.put_fiber(ldisc_itr.pod(), lfiber);
    ldisc_itr.next();
  }
  ldisc_id_space.release_iterator(ldisc_itr);

  // Print the finished posets.

  cout << le1u_host << endl;
  cout << le1_host << endl;

  // Write it to a file for later use.

  storage_agent lsa_write("example30.hdf", sheaf_file::READ_WRITE);
  lsa_write.write_entire(lns);

  // Exit:

  return 0;
}
```

4.2.4.6.4 Multi-valued sections

Sometimes we need to represent a property that is multivalued. Consider a domain which consists of two parts ("patches") made of different materials. At the boundary between the two patches, density may have some value in the one material and a different value in the other material, making it effectively double-valued on the boundary. As another example, some objects, such as the surface of the Earth, cannot be completely covered by a single coordinate system. In such cases the standard practice is to decompose the object into overlapping patches with a single-valued coordinate system on each patch.

In both these cases, the result is a set of ordinary single-valued sections, one on each patch. It is very useful to treat such a collection as a single section which is multi-valued where the patches overlap. We call such a multi-valued section a multisection and each single-valued piece is referred to as a branch of the multisection. Since a multisection is a collection of single-valued sections, each of which is a jim in section space, a multisection is represented by a jrm in section space.

4.2.4.6.4.1 Creating a multisection

Creating a multisection just requires specifying the patches on which the section is single-valued, for instance:

fiber_bundle::sec_e2::
sec_e2(sec_rep_space* xhost, const subposet& xbase_parts, bool xauto_access)

> Creates a new handle attached to a new jrm state which is a multi-section with a branch for each member of the base space contained in xbase_parts.

4.2.4.6.4.2 Accessing the attributes of a multisection

A multisection is a jrm and hence does not itself have attributes, only the branches have attributes. Each multisection defines an id space for its branches. The base class sec_rep_space_member supplies the usual get/release protocol, inherited by all section types:

sheaf::mutable_index_space_handle&
fiber_bundle::sec_rep_space_member::
get_branch_id_space(bool xauto_access)

> Allocate an id space handle from the handle pool attached to the branch id space for this member.

void
fiber_bundle::sec_rep_space_member::
release_branch_id_space(index_space_handle& xid_space, bool xauto_access) const

> Returns the id space handle xid_space to the handle pool.

Get_branch_id_space_iterator and release_branch_id_space_iterator are also provided for direct access to the iterators.

The attributes of each branch may be accessed as described above for an ordinary section.

Example 31 shows how to create and access multisections.

4.2.4.6.4.3 Example 31: Multisections

```
#include "index_space_handle.h"
#include "index_space_iterator.h"
#include "sec_at1_space.h"
#include "sec_e2.h"
#include "std_iostream.h"
#include "std_sstream.h"
#include "storage_agent.h"
#include "subposet.h"
#include "tern.h"

using namespace sheaf;
using namespace fiber_bundle;
using namespace fiber_bundle::vd_algebra;

int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example31:" << endl;

  // Create a namespace.

  fiber_bundles_namespace lns("Example31");

  // Populate the namespace from the file we wrote in example30.

  storage_agent lsa_read("example30.hdf", sheaf_file::READ_ONLY);
  lsa_read.read_entire(lns);

  // Get the base space.

  base_space_poset& lbase_host =
    lns.member_poset<base_space_poset>("mesh2", true);

  // Create some patches in the base space:
  // First get elements id space.

  index_space_handle& lseg_id_space =
    lbase_host.elements().id_space();

  // Create the first patch - a jrm that
  // contains the first two segments.

  scoped_index lpatch_segs[2];

  lpatch_segs[0].put(lseg_id_space, 0);
  lpatch_segs[1].put(lseg_id_space, 1);
  base_space_member
    lpatch0(&lbase_host, lpatch_segs, 2, tern::TRUE, true);
  lpatch0.put_name("patch0", true, true);
```

```
// Create the second patch.

lpatch_segs[0].put(lseg_id_space, 2);
lpatch_segs[1].put(lseg_id_space, 3);
base_space_member
  lpatch1(&lbase_host, lpatch_segs, 2, tern::TRUE, true);
lpatch1.put_name("patch1", true, true);

// Create a subposet for the patches.

subposet lpatches(&lbase_host);
lpatches.put_name("patches", true, true);
lpatches.insert_member(lpatch0.index());
lpatches.insert_member(lpatch1.index());

// Print the base space with the patches.

cout << lbase_host << endl;

// Create a section space for sec_e2.

sec_e2::host_type& le2_host =
  sec_e2::standard_host(lns, "mesh2/block", "", "", "", true);

// Create a multisection on the patches.

sec_e2 lmulti(&le2_host, lpatches, true);

// Set the attributes of the branches.
// Because lfiber is incremented every time
// and middle vertex is in both branches,
// multisection will be discontinuous at middle vertex.

e2_lite lfiber(0.0, 0.0);
e2_lite lfiber_inc(1.0, 1.0);

// Get a branch iterator for the multisection.

index_space_iterator& lbranch_itr =
  lmulti.get_branch_id_space_iterator(false);
while(!lbranch_itr.is_done())
{
  // Attach a handle to the current branch.
  // Note that arg has to be in the hub id space.

  sec_e2 lbranch(&le2_host, lbranch_itr.hub_pod());

  // Give the branch a name.

  stringstream lstr;
  lstr << "branch" << lbranch_itr.pod();
  lbranch.put_name(lstr.str(), true, false);

  // Get discretization iterator for the current branch
  // and iterate over fibers, setting the section.

  index_space_handle& ldisc_id_space =
      lbranch.schema().discretization_id_space();
    index_space_iterator& ldisc_itr = ldisc_id_space.get_iterator();
    while(!ldisc_itr.is_done())
    {
      lbranch.put_fiber(ldisc_itr.pod(), lfiber);
      lfiber += lfiber_inc;
      ldisc_itr.next();
    }
    ldisc_id_space.release_iterator(ldisc_itr);
    lbranch_itr.next();
    lbranch.detach_from_state();
  }
  lmulti.release_branch_id_space_iterator(lbranch_itr, false);

  // Print the finished section space.

  cout << le2_host << endl;

  // Exit:

  return 0;
}
```

4.2.4.7 Subposets

we've already discussed the discretization and evaluation subposets, and the use of subposets for definition of branches in a multisection. All of these subposets are actually in the base space. Section spaces and section space schema define no additional client-visible subposets/

4.2.4.8 Traversing the graph

All the section space posets are ordinary posets as far as graph traversal is concerned, so the cover iterators and depth_first_itrs we discussed above can be used for section spaces. The only additional feature introduced in section spaces is the branch id space and iterators for multisections, which we discussed in the preceding section.

Section space schema posets are a different story. As we've already mentioned, section space schema posets are virtual. The membership of these is not actually instantiated as ordinary members, so ordinary traversal via either cover iterators or depth_first_itrs will not produce the expected results. Section space schema posets are traversed with special iterators that know about the product subspace approximation used in these posets. However, although not protected or private in any way, these iterators are essentially internal to the system. Clients need rely only on the attribute id spaces and iterators we've discussed above, at least for the introductory level use which is the subject of this tutorial.

4.3 The fields interface

The classes in the fields component represent the notion of field as found in typical undergraduate physics text books, that is, some property as a function of global coordinates. The field abstraction is the natural context for a number of useful functions, most notably calculus. The fields component follows the pattern already established by the section space cluster: there is a field type field_F for each fiber type F in the fiber spaces cluster. The fields component is still under construction at this time, so the functionality discussed here will only discuss a few of the features already implemented.

4.3.1 Creating fields

A fields is just a pair of sections, a coordinates section and a property section. It is C++ object, but not a sheaf object. It does not have a schema, does not live in a table, and is not persistent. We create a field from the two sections, a coordinates section and a property section. A scalar field, for instance:

fields::field_at0::
field_at0 (const sec_ed & xcoordinates, const sec_at0& xproperty, bool xauto_access)

Create an instance with coordinates xcoordinates and property xproperty.

4.3.2 Creating a property as a function of the coordinates

It's quite common to want to set the property to some given function of the coordinates. If the coordinates and the property sections both use the same discretization, the task is simple - we just iterate over the discretization id space, get the coordinate fiber, compute the given function and put the property fiber. But if the property and coordinates are not on the same discretization, the task is substantially more complicated. For each property discretization point, we have to find the coordinate evaluation member that contains the point, gather the coordinate attributes for that evaluation member, evaluate the coordinate section at the property discretization point, then compute the property attributes from the coordinate attributes. The put_property_dofs function takes care of all this for us. All the client has to provide is a simple function that does the last step - calculates the property given the coordinates:

void
fields::field_vd::
put_property_dofs(property_dof_function_type xproperty_dofs_fcn, bool xauto_access)

Sets the dofs of property() to values computed by xproperty_dofs_fcn.

The typedef for the function signature is:

typedef void(*fields::field_vd::property_dof_function_type)(block<sec_vd_value_type>& xglobal_coords, block<sec_vd_dof_type>& xproperty_dofs)

The type of the function for computing the property dofs at given global coordinates.

This is one of those C/C++ typedefs that is virtually impossible to decipher, so the library provides an example, largely so you can understand the signature:

void
fields::field_vd::
property_dof_function_example(block< sec_vd_value_type >& xglobal_coords,

```
                block< sec_vd_dof_type >& xproperty_dofs) static
```

Example property dof function; property value is x*1000000 + y*1000 + z, intended to be easy to check field dofs for correctness.

4.3.3 Evaluating the property at given coordinates

The defining function of the field object is to evaluate the property at given global coordinates. This is more complicated than it seems at first. The property is a section, a function of patch and local coordinates. To evaluate it, we need to know what patch and local coordinates the global coordinates value corresponds to, in other words, we have to invert the global coordinate section. The task is even more complicated if the coordinates and the property aren't defined on the same discretization and evaluation subposets, for example if the coordinate representation is uniform and the property representation is vextex_element_dlinear. The property_at_coordinates method takes care of all this complexity:

```
void
fields::field_vd::
property_at_coordinates(sec_vd_value_type  xcoord_base[],
                        int xcoord_ct,
                        sec_vd_value_type xprop_base[],
                        int xprop_ub) const
```

The value of the property at the given global coordinates.

4.3.4 Pushing a property from one mesh to another

One of the most useful features of the field abstraction is that it provides enough context to support moving a property from one mesh to another. This process, often called "remapping" in the scientific computing literature, is called "pushing" in the sheaf system because that is what it is called in the fiber_bundle literature. As with put_property_dofs and property_at_coordinates, this task is considerably more complicated than it appears at first. The push function takes care of the complexity and makes it easy:

```
void
fields::
push(const field_vd& xsrc, field_vd& xdst, bool xauto_access)
```

Pushes xsrc.property() to xdst.property().

4.3.5 Algebraic operations and namespaces

The field type hierarchy repeats the fiber type hierarchy: for every type F in the fiber hierarchy there is a field type field_F with property section of type sec_F. Every fiber algebra has a corresponding field algebra in namespace fields::field_F_algebra. See the reference documentation for a complete listing of the namespaces and operations.

Example 32 illustrates the features of fields.

4.3.6 Example 32: Fields

```
#include "at0.h"
#include "e1.h"
#include "fiber_bundles_namespace.h"
#include "field_at0.h"
#include "index_space_handle.h"
#include "index_space_iterator.h"
#include "sec_at0.h"
#include "sec_at0_space.h"
#include "sec_at1_space.h"
#include "sec_e1.h"
#include "sec_e1_uniform.h"
#include "std_iostream.h"
#include "std_sstream.h"
#include "storage_agent.h"
#include "structured_block_1d.h"

using namespace sheaf;
using namespace fiber_bundle;
using namespace fields;


namespace
{

// Function to compute the property dofs as a function of the
// coordinates. As an example, assume the property is scalar and
// set it to the distance from the coordinate origin.
// For the 1d case we will use, this will just give us the
// coordinate back again, so it will be easy to see if we get
// the right answer.

void
property_dof_function(block<sec_vd_value_type>& xglobal_coords,
                      block<sec_vd_dof_type>& xproperty_dofs)
{
  sec_vd_value_type dist = 0.0;
  for(int i= 0; i<xglobal_coords.ct(); ++i)
  {
    dist += xglobal_coords[i]* xglobal_coords[i];
  }

  dist = sqrt(dist);
  xproperty_dofs[0] = dist;
  return;
}

} // end unnamed namespace.


int main( int argc, char* argv[])
{
  cout << "SheafSystemProgrammersGuide Example32:" << endl;

  // Create a namespace.
```

```
fiber_bundles_namespace lns("Example32");

// Populate the namespace from the file we wrote in example30.

storage_agent lsa_read("example30.hdf", sheaf_file::READ_ONLY);
lsa_read.read_entire(lns);

// Get the coordinates space.

poset_path le1_path("e1_on_block");
sec_e1::host_type& le1_host =
  lns.member_poset<sec_e1::host_type>(le1_path, true);

// Get a handle for the coordinates section

sec_e1 le1_coords(&le1_host, "general_coordinates");

// Create a scalar section space.

poset_path lat0_path1("at0_on_block");
poset_path lbase_path1("mesh2/block");

sec_at0::host_type& lat0_space1 =
  lns.new_section_space<sec_at0>(lat0_path1, lbase_path1);

// Create a scalar section for the property

sec_at0 lprop1(&lat0_space1);

// Create a scalar field

field_at0 lscalar_field1(le1_coords, lprop1, true);

// Set the property to the value defined by
// the put_property_dofs_example function above.

lscalar_field1.put_property_dofs(property_dof_function, true);

// Get the property value at coordinate = 5.0.

sec_e1::value_type lx = 5.0;
sec_at0::value_type lp;

lscalar_field1.property_at_coordinates(&lx, 1, &lp, 1);

cout << endl << "property at x= 5.0 is " << lp << endl;

// Print the finished section space.

cout << lat0_space1 << endl;

// Create a different base space with 3 segments

arg_list largs = base_space_poset::make_args(1);
base_space_poset& lbase_host =
  lns.new_base_space<structured_block_1d>("mesh3", largs);
  structured_block_1d lblock3(&lbase_host, 3, true);
  lblock3.put_name("block3", true, true);

  // Create a section space for uniform coordinates
  // on the new base space.

  poset_path le1u_path("e1_uniform_on_block3");
  poset_path lbase_path2("mesh3/block3");
  poset_path le1u_rep_path("sec_rep_descriptors/vertex_block_uniform");

  sec_e1_uniform::host_type& le1u_host =
    lns.new_section_space<sec_e1_uniform>(le1u_path, lbase_path2,
                                          le1u_rep_path, true);

  // Create the coordinates section for the new base space.
  // Uniform coordinates are initialized by the constructor.

  sec_e1_uniform le1u_coords(&le1u_host, 0.0, 3.0, true);
  le1u_coords.put_name("uniform_coordinates", true, true);

  // Create a scalar section space on the new base space.

  poset_path lat0_path2("at0_on_block3");

  sec_at0::host_type& lat0_space2 =
    lns.new_section_space<sec_at0>(lat0_path2, lbase_path2);

  // Create a scalar section for the property

  sec_at0 lprop2(&lat0_space2);

  // Create a scalar field on the new base space.

  field_at0 lscalar_field2(le1u_coords, lprop2, true);

  // Push the property from the first base space to the second.

  push(lscalar_field1, lscalar_field2, true);

  // Print out the property section space on the new base space.

  cout << lat0_space2 << endl;

  // Exit:

  return 0;
}
```

5 Part II: Intermediate features

Note: Part II is still under construction

Part I described the basic capabilities of the SheafSystem using the simplest possible examples. In Part II we will revisit many of the same topics, but describing more advanced capabilities using more complex examples. The topics in Part II are intended to address many of the issues that arise when using the SheafSystem in typical practical applications.

5.1 The sheaf interface

5.1.1 Index spaces and scoped indices, part 2

We introduced the basic notions of id spaces in section 4.1.3 and showed how to use them in several of the examples that followed. In this section we will show how to create and modify id spaces.

5.1.1.1 Primary and secondary index spaces.

As suggested by the index space family diagram in Figure 1, there are two main catergories of index spaces: primary and secondary.

Primary id spaces are created internally by the SheafSystem to represent the members of the row graph. For instance, when we create a structured_block_1d as we did for the line segment mesh in Example 23, the system allocates a primary id space for the members of the block. Each member of the poset is represented by a unique member of a primary id space. The hub id space is defined to be the sum (disjoint union) of the primary id spaces. The primary id spaces are maintained as needed by the system and hence are not typically of direct interest to clients.

Secondary id spaces are derived from other id spaces using one of the following methods: subspace, sum, and product.

5.1.1.2 Subspaces

A subspace id space is defined as a subset of the hub id space. In other words, a subspace is an indexing scheme for some subset of the members of a poset. The SheafSystem provides a number of id space classes, each specialized for some particular usage. Many of these are intended only for internal use in the SheafSystem itself. The most useful for clients are the mutable index spaces.

5.1.1.2.1 Mutable index space representations

The most commonly used index space types are:

sheaf::array_index_space_state

An array implementation of class mutable_index_space_state. This representation is intended to efficiently represent id spaces that are positive and dense, that is, the domain ids are in the domain (~0, ~end()) and hence the id to hub id map can be efficiently stored in an array. In particular it is intended for id spaces used to index arrays, in which case the ids are in the domain [0, end()).

sheaf::hash_index_space_state

An hash map implementation of class mutable_index_space_state. This representation is intended to efficiently represent id spaces that are possibly negative and are sparse, that is, end() is much greater than ct() and hence using an array to store the id to hub id map would make inefficient use of memory. Instead, this class uses hash maps to represent both directions of the map.

5.1.1.2.2 Creating mutable index spaces

5.1.1.2.3 Populating mutable index spaces

5.1.1.2.4 Other useful index space types

sheaf::offset_index_space_state

A computed implementation of abstract class explicit_index_space_state. This representation assumes the ids are an open gathered set of ids. The equivalence between the ids in this space and the hub id space is computed using an offset.

5.1.1.3 Sums

5.1.1.4 Products and product structures

5.2 The fiber bundle interface

5.2.1 Base spaces

5.2.1.1 Point blocks

5.2.1.2 Structured_block_2d and structured_block_3d

5.2.1.3 Zone_nodes_block

5.2.1.4 Unstructured_block

5.2.2 Fiber spaces

5.2.2.1 Fiber space dependency graph

5.2.3 Section spaces

5.2.3.1 Creating section space schema

5.2.3.2 Example 33: Creating a section space schema.

Appendix A 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.

A.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
  // cannot 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
```

A.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
```

Appendix B Join equivalent members and lexicographic ordering

Is it 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 relation 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 $p \le p'$. 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: $p \leq p'$ 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 lexicographic order. A lexicographic 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 lexicographic 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.

Appendix C Fiber Algebra

As described above, the classes in the fiber spaces cluster provide the various algebraic types used in theoretical physics to describe the properties of particles and systems. Each of these algebraic types has an associated set of operations, an algebra. In this appendix we summarize the operations associated with each type. See the reference documentation for a complete description of the operations and the numerous signatures supported for each operation.

C.1 Abstract vectors (vector algebra)

Class vd and its descendants support the operations of vector algebra (linear algebra). The main operations are:

add	add one vector to another.
subtract	subtract one vector from another.
multiply	multiply a vector by a scalar.
divide	divide a vector by a scalar.
min	find the minimum component of a vector.
max	find the maximum component of a vector.

contract	contract a vector with a covector.

C.2 Euclidean vectors (Euclidean vector algebra)

dot	the Euclidean scalar product of two vectors.
length	the length of a vector.
put_length	sets the length of a vector.
normalize	sets the length of a vector to 1.
cross	the vector ("cross") product of an e3 vector with another.

C.3 General tensors

Class tp and its descendants support general tensor algebra. The main operations are:

tensor	the tensor product of one tensor with another.
alt	the antisymmetric ("alternating") part of a tensor.
sym	the symmetric part of a tensor.
contract	contract a tensor on one contravariant and one covariant index.

C.4 Antisymmetric tensors (exterior algebra)

Class atp and its descendants support exterior algebra. the main operations are:

wedge	the exterior ("wedge") product of two antisymmetric tensors.
hook	the interior ("hook") product of an antisymmetric tensor and a vector.
star	the Hodge star ("dual") of an antisymmetric tensor.

C.5 Symmetric tensors (symmetric algebra)

symmetric_product	the symmetric product of one symmetric tensor with another.
trace	the trace of a degree 2 symmetric tensor.
determinant	the determinant of a degree 2 symmetric tensor.
to_principal_axes	diagonalizes a degree 2 symmetric tensor

C.6 Metric tensors

raise	make a given index of a tensor contravariant.
lower	make a given index of a tensor covariant

C.7 Jacobians

push	push a vector from the domain to the range of a Jacobian
pull	pull a covector from the range to the domain of a Jacobian

C.8 Transformation groups

inverse	the inverse of a linear transformation
transform_basis_by	transform the basis and components of a tensor

C.9 Multiple signatures

Each of the above operations is represented by a family of functions implementing multiple signatures for the operation. Typically each operation is provided for both regular fiber types and the associated lite fiber types. In addition, auto-allocated, pre-allocated and self-allocated variants are provided. Finally, whenever it makes sense, operator variants are provided.

The full fiber algebra consists of all the various signatures for all the various operations and thus contains approximately100 functions.

Appendix D Section algebra

The section type hierarchy repeats the fiber type hierarchy: for every type F in the fiber hierarchy there is a section type S with fiber of type F. Every fiber operation has a corresponding section operation. Sections also support additional operations, not defined for fibers. The most important of these are the real-valued functions of scalar sections. A scalar section is essentially a map from the base space to reals and we can compose this with any map from reals to reals to form another map from the base space to reals, i.e. another scalar section. Namespace fiber_bundle::sec_at0_algebra defines such an operation for each math library function defined by the C++ standard. We summarize these in the following:

fabs	absolute value of a scalar section.
ceil	ceiling of a scalar section.
floor	floor of a scalar section.
sqrt	square root of a scalar section.
pow	power of a scalar section.
cos	cosine of a scalar section.
sin	sine of a scalar section.
tan	tangent of a scalar section.
acos	arc cosine of a scalar section.
asin	arc sine of a scalar section.
atan	arc tangent of a scalar section.
atan2	arc tangent of a scalar section.
cosh	hyperbolic cosine of a scalar section.
sinh	hyperbolic sine of a scalar section.

tanh	hyperbolic tangent of a scalar section.
exp	exponential of a scalar section.
log	log base e of a scalar section.
log10	log base 10 of a scalar section.
modf	integer and fractional parts of a scalar section.
frexp	fractional and exponent parts of a scalar section.
fmod	remainder of scalar section.
ldexp	scalar section times a power of 2.