iRel: An Interface For Inter-Interface Associations

– WORKING DRAFT –

The ITAPS Team

August 18, 2010

Contents

1 Introduction

The ITAPS project is developing technology to improve interoperability in mesh-based scientific computing. Three primary activities are: designing common interfaces for mesh, geometry, and other data; developing and porting services involving those data to those interfaces; and building applications or higher-level services based on those services and interfaces. Current interfaces include iMesh (interface to discretized definition of spatial domain), iGeom (continuous definition of spatial domain), iField (mathematical field and operators, on continuous or discretized domain). Interfaces have been developed or imagined for other kinds of data, including: material models.

1.1 Interface Design Philosophy

There are several characteristics interfaces should have, regardless of how the interface is designed: b

- *Scope*: The types of data and functionality to be included in an interface. An interface should include data and functionality that are intrinsically related and would be difficult or awkward

to separate in different interfaces, but no more. A more practical definition is that an interface include data and functions that most applications dealing with those types of data are likely to need. Separation of a collection of data and functions into multiple interfaces is appropriate when there are many applications which need one interface without needing other interfaces.

- *Level of abstraction*: The level of abstraction used to expose data and functionality to applications strongly affects both the usability and versatility of an interface. The level of abstraction falls on a spectrum between concrete and abstract. If an interface is too concrete, it is large and unwieldy, and can have limited versatility for new applications. Too abstract, and the interface is difficult to use by practitioners only casually familiar with the data and functionality exposed by the interface.

The actual definition of an interface consists of two parts: a data model, which is the language and types used to define data exposed by an interface; and an API, which defines the data types and functions used in the specification of the interface. The design of a data model is guided by the level of abstraction desired in an interface.

1.2 ITAPS Interface Design

The separation of ITAPS interfaces into iMesh, iGeom, and iField follows from the definition of scope above. Speculation of other interfaces which may arise in the context of scientific computing, e.g. a materials interface, also follow from that principle. The data model exposed by iMesh and iGeom includes four fundamental types: entity, set, tag, and interface or root set. Although it is desirable to have separate interfaces for various types of data, there is often a need to relate data in one interface to data in another. For example, mesh generation applications generate mesh, stored in iMesh, that discretizes geometric entities in iGeom. For this purpose, ITAPS also defines a relations interface, iRel, to accomplish this task.

1.3 Requirements

Defining requirements for a relations interface is straightforward, based on experience in mesh generation and mesh-based applications. Requirements are in two areas: representation, describing the type of relation that needs to be represented; and function, the functionality needed by various applications. These are indicated with (R) or (F) in the requirements below. While many of the requirements below are phrased in interface-specific terms (mesh, geometry, etc.), all of them have more abstract definitions and applications with other interface types.

- *Geometry entity to mesh entities (R)*: From the perspective of mesh generation and related fields, it is necessary to store the relation of a geometric entity (e.g. model face) with the mesh entities which discretize or are positioned on that model entity (e.g. mesh faces, along with non- boundary edges and vertices). Most commonly, these relations need to be symmetric; that is, they can be queried starting from either side of the relation.
- *Geometry entity to collection of mesh entities (R)*: There are several cases where it is desirable to relate geometry entities to collections of mesh entities, i.e. to a collection of mesh entities

represented as an object in the mesh state. The accepted mechanism for doing this is to use entity sets. First, it may be more efficient to handle sets of mesh than individual entities. This may be due to the implementation of iMesh, for example when storing a relation in this way is more memory efficient; or it may be due to the application, for example mesh generation on non-manifold models or manifold models with boundaries, where the container for the collection may exist before the actual mesh entities do. Second, relating a geometry entity to a mesh set may be desirable for the actual mesh generation process, where a mesh set is created for each geometry entity first, then the mesh generation application is coordinated using mesh sets. Relations of this type (geometry entity to mesh set) are also usually symmetric.

- *Mesh entity to geometry entity (R)*: In some cases it is more compute-efficient to access mesh to geometry relations individually, without first going through a mesh set. Two straightforward examples are adaptive mesh refinement and mesh smoothing, where mesh vertices must be projected to the owning geometry entity. These relations are usually not symmetric; that is, a geometry entity is usually not related to a single mesh entity.

- *Field data to tag on mesh (R)*: The iField interface is being designed to handle both generalized tensor fields and operators on those fields. The scope of this interface, as currently targeted, includes both discrete and continuous fields, with discrete fields including both those defined on a mesh as well as those from external sources, e.g. measured experimental data. In the case of mesh-based discrete fields, where the actual field data may be stored in iMesh, there will need to be a relation to the field meta-data, which describes the semantics of the field. iRel could be used to store that relation. Because iField is in an early stage of development, it is difficult to speculate how best to represent that meta-data in the data model. These types of relations could be symmetric or asymmetric, depending on whether multiple entities on one side of the relation would be related to single entities on the other side.

- *Material mixture to geometry entity or set (R)*: Consistent with the principle of scope, it is logical to separate the definition of material properties from that of the spatial domain. However, at the application level, the two are often specified at the same time. For example, CAD systems often have the ability to specify material properties as the geometric model is developed. A material interface is likely to have application-defined materials, for example to define material or isotopic mixtures. If the geometry and material interfaces are independent, then the relations interface must be able to relate things between those interfaces. Both entities and sets are needed on the geometry side, since material identifiers on CAD models are often specified on collections of these entities.

- *Assignment (F)*: Since iMesh and iGeom are usable independent of the other, the data in each is not automatically related to the data in the other. Relations must be assigned before they can be used by applications. Assignment of relations happens at a lower level; that is, individual entities or sets in one interface are related to those in another interface. The most obvious use of this functionality is in mesh generation, where mesh-geometry relations are assigned as the mesh is constructed. A relation should be available for query at any time after it has been assigned.

- *Infer (F)*: Because the interfaces related by iRel are independent, the data in each is often saved and restored independently. Applications using relations may be run separate from the application creating the relations in the first place, after a save and restore of the data in each interface. These querying applications may not know how to re-assign relations either; in this case, relations must be inferred. Specific approaches to inferring relations between

interfaces are discussed later in this document; here, we assume only that iRel provides the functionality to bootstrap relations data based on data in the interfaces being related. For example, in a physics application using an adaptive mesh refinement service, that service tells iRel to restore relations between mesh and geometry, on individual entities or for the entire model, before starting the refinement process.

- *Save/restore (F)*: There are many times where the application creating the relations is different from the application which later uses those relations. For example, mesh is created in a mesh generation application, and adaptively refined in a physics application. It may be desirable to specify that relations be saved and restored explicitly, rather than being inferred by later applications.

- *Change type (F)*: The actual representation of a relation between an entity in one interface and a collection of entities in another interface has characteristic memory and cpu time costs. In many cases, application requirements may vary, even during the same run. For example, an application may want the AMR service to minimize memory usage when it is not actively refining mesh while also allowing that service to use more memory during refinement. Another example of this is mesh generation on non-manifold models, where the mesh entity to geometry entity relation evaluation is needed only for the geometric region and boundary being meshed at a given time. To accommodate the varying needs of applications, the relations interface must allow the application to request different forms of representations and changes to those representations during a given run.

2 iRel Interface

Given the above requirements, we now describe the iRel interface.

2.1 Data Model

There are many ways to formulate a data model for storing relations. For example, each type instance of relation in an application (e.g. mesh-geometry) could be represented by a distinct instance of the iRel interface. Or, each direction of a given relation could be a distinct relation, whether represented by a distinct iRel instance or not. The data model describes these and other specifics of how relations are described through the iRel intereface. We define the data model in iRel using the following subsections:

2.1.1 Interface type enumeration

The interfaces being related, e.g. iMesh, iGeom, iField. In higher-level languages, interface instances would be passed to iRel using some parent data type, e.g. iBase, with member functions called by iRel passing through to the concrete interface type (iMesh, iGeom). This could also be done in a lower level language, using indirection (e.g. function pointers). In practice, though, we anticipate a relatively low number of distinct interfaces being related by iRel, and in most cases those interface types will seldom change and will likely be known by both iRel and the applications

using iRel. Therefore, we specify an interface type enumeration *IfaceType*, whose values currently include (*iRel_IGEOM_IFACE*, *iRel_IMESH_IFACE*, *iRel_IFIELD_IFACE*). Interface handles are specified to iRel as (*iBase_Instance*, *IfaceType*) tuples.

2.1.2 Relation type

The Relation Type denotes what kind of data in one interface are related to what kind of data in another. Thus, a Relation Type has two sides, one for each side of the relation. The following Relation Types are allowed:

- Entity - Entity
- Set - Entity
- Set - Set
- Entity - Both
- Set - Both
- Both - Entity
- Both - Set
- Both - Both

In the first three types of relations, entities or sets in iface1 are related directly to entities or sets in iface2; for those types of relations, it is assumed that, given an entity to be related, the relation has O(1) storage and retrieval costs. Relation Types with -Both are *asymmetric*; entities and sets on a Both side point to sets on the other side, and vice versa.

Suggestion

- *(RPI)* Need to describe *asymmetric* in detail for clarity.
- *(RPI)* Added Both-Entity and Both-Set.

2.1.3 Relation

An iRel Relation is an entity in an iRel intance; it stores a relation of a specific Relation Type between one interface (iface1) and another (iface2). A Relation is created and used analogous to a Tag in iGeom or iMesh, where it is first created, then given a value for various entities. A Relation stores the interface handles it relates (in the order they were specified when the Relation was created), and the Relation Type A Relation is passed through iRel in the form of a Relation Handle. Relations can be symmetric or asymmetric, depending on the type. They can be consistent or inconsistent, with the latter being used in coordination with lower-level functions in iRel, e.g. when a mesh is being adapted or generated.

2.1.4 Relation side

Used in query functions to denote the starting side of the query; side1 implies a query from iface1 to *iface2*, while *side2* implies a query from *iface2* to *iface1*.

2.2 API

The iRel API as currently envisioned can be found on the ITAPS project website (http://www.itaps.org). This section describes the API in general terms. Functions indicated with an asterisk are proposed and not yet implemented.

- *Create/infer*: Functions are available for creating a specific Relation, and inferring a Relation in the interfaces that Relation pertains to. Inference can be for individual entities or sets, or for the entire database stored in the interfaces.
- *Get*: Relations can be retrieved given a Relation handle, an entity, set, or arrays of them, and a side, which specifies the starting side of the query.
- *Set*: Functions are available for setting individual relations. These functions are somewhat low-level, and should be used with care, as they have the potential for invalidating the relations data with no way to recover. It is expected that at least one application will set relations at some point in the lifetime of entities being related, and it is these relations that are inferred by later applications.
- *Change type**: Functions must be added for changing type; this flexibility can result in significant memory savings, e.g. when changing from a -Both to a -Set type after smoothing or mesh adaptation is finished.
- *Higher-level functions*: The current iRel specification includes higher-level functions specific to mesh- geometry relations. These functions were put in iRel as a compromise between a completely abstract iRel interface and one pertaining only to mesh-geometry relations. These functions may be moved to a higher-level service, or may be specified in an interface-specific part of iRel.

3 Discussion

Specific issues have arisen in the development of iRel which are described further below.

3.1 Storage

In the only current implementation of *iRel* (Lasso), it is expected that relations are established when a mesh is generated, and inferred by later applications. That is, the implementation assumes that relations do not get stored with the data in a given interface when the save function is called

on that interface. This is the case even though Lasso uses tags in the interfaces being related, and tags are usually stored during a save. The rationale for this is that relation data stored on a given entity refers to data in a different interface, and that data cannot be saved and restored reliably. This is in contrast to storing Entity Handle-type tags in an interface, where these tags refer to entities in the same interface.

3.2 Inference Method

Since we assume *iRel* does not store its relations data directly with the interface data, we require that relations be inferred. The method used to infer relations depends on the data being related, which in turn can depend on the implementation of the interface that data is read from. It is an open issue how to specify criteria for how to relate entities between interfaces, in a way which is interoperable. One obvious requirement, or at least a recommended path forward, is that those criteria be specified in terms of the data model used in *iBase*. In that language, the criteria used by Lasso to relate entities between iGeom and iMesh can be described as: Relate e(iGeom), s(iMesh) such that:

```
1. getEntType(e) == val(GEOM_DIMENSION)_s &&
2. val(GLOBAL_ID)_e == val(GLOBAL_ID)_s
```

Here, *e* and *s* denote entity and entity set; *getEntType(e)* is the unary result of a function in iGeom; *val(xxx)* is the value of *tag xxx* on the indicated entity. Thus, the elements used to specify the relation criteria include entities, sets, tags and possibly values on those things, and unary function results for those things. We have delayed proposing an interface for that sort of specification until such time as another implementation of iRel emerges. Note, we believe there is a place for implementation-specific iRel instances, which will probably have embedded relation criteria specific to particular data formats. These versions of iRel will probably not be interoperable with different implementations of the interfaces being related, but will still be useful to applications needing to query the interfaces for which these iRel instances are developed. The iRel API has been modified to allow the iRel implementation to determine the relation type during the infer function.

3.3 Interface-Specific Functionality and API

Another way to approach an iRel specification would be to write it in terms of the specific data being related. For example, there have been many requests for specific functions for relating mesh and geometry. We assert that this should not be the basis for iRel because:

1. This would be shortsighted, as it would not address relations with other interfaces, planned (e.g. iField) and unplanned (e.g. materials). As a consequence, iRel would have to be modified before any new interface could be treated by iRel. This would go against the principle of versatility of interfaces. Also, given the difficulty of negotiating interfaces, it should not be assumed that this extension would be a trivial matter.
2. Narrowing the scope of iRel to only mesh and geometry is actually a relatively minor issue; the deeper issues have to do with the data model and communication between iRel and other

interfaces in terms of sets and tags. Choosing to narrow the scope would not address these other issues.

3. Focusing specifically on geometry and mesh, rather than considering it as an abstract problem, would likely leave out functionality important to other interfaces. For example, the current implementation of iRel, Lasso, does not treat relating tags between interfaces; yet, that is one of the obvious uses of iRel in the context of iField. No matter what the resolution of this issue, a recommended follow-up would be to review iRel for completeness while considering other interfaces.

It is recommended that if an API specific to geometry-mesh relations is developed, it be offered as a service on top of iRel, rather than part of the iRel specification.

3.4 Serving and Maintaining Relations under Mesh Modification

Services performing mesh modification, e.g. AMR, have very specific requirements for a relations interface. Mesh to geometry relations need to be accessed from a mesh entity level with O(1) complexity, and frequent addition and removal of mesh makes it difficult to maintain sets efficiently. This could make serving the geometry-mesh relations in terms of sets inefficient, due to the difficulty of maintaining sets under modification. Looking at this situation from the AMR service point of view, though, it seems clear that one of the following will be true:

1. The lists of mesh entities related to geometry entities become out of date as entities are created and destroyed during the adaptation process; these lists are updated after mesh adaptation is completed.

2. The lists of mesh entities related to geometry entities are cleared before adaptation, and repopulated after adaptation is completed.

3. The lists of mesh entities are updated as entities are created and destroyed during adaptation.

If 3 is true, then the implementation already has the basis of a set which performs efficient removal of entities. If 1 or 2 is true, then the implementation allows for the possibility of an inconsistent relation state between geometry and mesh; this could also be true of a relations interface phrased in terms of sets. In all cases, there is nothing which prevents the implementation from returning set objects which have constraints which are more strict than generic sets themselves. These constrained sets would return errors if unallowed operations were attempted on them by applications.

Suggestion

(RPI/Simmetrix) In case of the mesh implementation which maintains *mesh entity → model entity*[1] and/or *model entity → mesh entities*[2] relations at the implementation level (e.g. FMDB,

[1] classification
[2] reverse classification

Simmetrix), with the help of a relation created with *iRel_createRelation (iRel_IGEOM_IFACE, iRel_ENTITY, iRel_IMESH_IFACE, iRel_BOTH)*, where relations *model entity* → *mesh set* and *mesh entity* → *model entity* are generated, classification and reverse classification information can be inferred and obtained through iRel at the reasonable cost.

However, in case of FMDB and Simmetrix, proposed *ent-both* type relation between model and mesh is inappropriate for effective mesh modification, especially in terms of continuous on-the-fly update of classification and reverse classification information as a mesh is adapted. In other words, for a mesh impementation which doesn't use entity set for representing classification, *iRel_setEntEntRelation* with *ent-both* type relation is not an effective method to update the classification (*model entity* ↔ *mesh entities*) when a new mesh entity is created through *iMesh_create(Ent|Vtx)*.

Therefore, we suggest adding the following function in iMesh for supporting mesh entity creation and classification setting in the same interface without requiring entity set as a necessary part explicitly or implicitly.

```
/**\brief  Set geometric model classification of mesh entity
 * \param  mesh_entity_handle    mesh entity handle
 * \param  geom_entity_handle    geometry entity handle
 * \param  *err    pointer to error type returned from function
 */
void iMesh_setGeomClas (/* in */  const iBase_EntityHandle mesh_entity_handle,
                        /* in */  const iBase_EntityHandle geom_entity_handle,
                        /* out */ int *err)
```

Apparently, at the interface level, this function doesn't impose model/mesh dependency since iMesh is still stand-alone. And at the implementation level, this function doesn't impose model/mesh dependency either since, this function won't be used by a mesh implementation without model dependency (e.g. MOAB) and a mesh data loaded without classification (e.g. FMDB mesh without geometric model association).

Be noted that in case of FMDB and Simmetrix, when a mesh entity is removed, the classification is automatically taken care of (removed) at the implementation level. Therefore, as long as *iMesh_setGeomClas* follows *iMesh_create(Ent|Vtx)*, additional *iRel_inferAllRelations* or *iRel_inferEntRelations* is not required for maintaining model-mesh relation during mesh modification. In other word, model-mesh relation update is not needed for mesh entity removal.

3.5 Iterator-Based Communication of Relations

Certain relation types imply that entity sets are used to store relations. This may constrain those implementations which do not use sets to store those relations. To accommodate these implementations, iRel also contains functions which communicate collections of entities in the form of set iterators. All iRel implementations are expected to implement iterator-based functions, while only some implementations may implement the corresponding set-based functions. The *iBase_NOT_IMPLEMENTED* error code is used to convey this information to the application when a set-based function is called on such implementations.

3.6 Interface Function Improvement

See Appendix A for suggestions and questions on interface functions.

4 Revision History

4.1 Document Revision

4.1.1 8/12/10 by RPI

- Added suggestions and questions on design concept in this document with passages starting with *Question* or Suggestion.
- Added suggestion for effectively relating mesh/geom entity in §3.4.
- Added interface source *iRel.h* in Appendix A
- Added suggestions and questions on *iRel.h* in Appendix A

4.1.2 1/7/10 by Tautges, T.

- Added paragraph under iRel Interface-API describing *Change type* functionality, for consistency with requirements
- Changed discussion of time/storage complexity to clarify that any O(1) complexity assumed you already had an entity for one side of the relation
- Modified discussion of relation type symmetry for -Both type relations to remove the words forward and reverse, to avoid confusion with historical use of these terms describing model-mesh classification and preferred directionality implied by those terms

4.1.3 1/27/10 by Tautges, T.

- Added various wordsmithing changes
- Restricted interface type enumeration to concrete types only (i.e. removed *iRel_IBASE_IFACE* from enumeration)
- Added conclusions of the discussion on iRel

4.2 Interface Revision

4.2.1 07/08/10 by Porter, J.

- Removed *iRel_getRelatedInterfaces* in favor of *iRel_findRelations*, which takes an interface as input and returns an array of *iRel_RelationHandles*, which can then be queried with

iRel_getRelationInfo

- Added interface type parameters to *iRel_getRelationInfo*, so that users of this function can be sure they have, say, an iMesh interface. This also makes the function more consistent with *iRel_createRelation* (the inputs of one would be the outputs of the other, and vice versa).
- Re-added *iRel_getSetEntRelation* and friends, since otherwise there's no way to pass in a set and get a related entity
- Moved *iRel_newRel* to the top of the file to match iMesh and iGeom
- Another change I'll suggest but have no strong attachment to is that we change *iRel_RelationHandle* to *iRel_PairHandle* to make a distinction between the relation of two interfaces (a pair) and the relation of entities/sets (a relation). The changes would be as follows:
  - iRel_RelationHandle → iRel_PairHandle
  - iRel_createRelation → iRel_createPair
  - iRel_destroyRelation → iRel_destroyPair
  - iRel_getRelationInfo → iRel_getPairInfo
  - iRel_findRelations → iRel_findPairs

4.2.2 1/27/10 by Tautges, T.

- Aadded getRelationInfo function returning types and interfaces
- Removed *getSetEntRelation* and others like it; since the get functions have a *switch_order* argument, you don't need both *getEntSetRelation* and *getSetEntRelation*
- Removed *iRel*_moveTo, *iRel_createVertexandRelate*, *createEntAndRelate*, and array-based versions, after agreement that these belong in higher-level service
- Added RelationType enumeration
- Removed *iRel_IBASE_IFACE* from *iFaceType* enumeration
- Added *IREL_MAJOR_VERSION*=1 and *IREL_MINOR_VERSION*=0 to header.

In our original ITAPS proposal, we mention partitioning of geometry information. We have focused on parallel meshes so far. But what are your needs for distributed geometry information and mapping between distributed geometries and meshes? How do you handle this information now? What do you envision you and your applications people will need going forward?

A iRel.h

Last update: July 08, 2010

A.1 Suggestions

- *(Porter, J.)* Add *iRel_getDescription* to return some text describing the last error in iRel
- *(Porter, J.)* Instead of *i(Mesh|Geom)_EntityIterator*, define *iBase_EntityIterator* in iBase.h
- *(RPI)* Rename *__iRel_LASSO_HPP__*
- *(RPI)* Include iBase.h instead of iMesh.h and iGeom.h
- *(RPI)* Argument type (in, out, inout) is not specified
- *(RPI)* How to increment, reset, and delete *iBase_EntityIterator* obtained through *iRel_getEntSetIterRelation* and *iRel_getEntArrSetIterArrRelation* is not specified. Shall we use the following iMesh iterator functions as iMeshP does?

  . `iMesh_getNextEnt(Arr)Iter`
  . `iMesh_resetEnt(Arr)Iter`
  . `iMesh_endEnt(Arr)Iter`

  If iRel and iMeshP will share the above iMesh iterator functions, will it be better to have them in iBase.h?

- *(RPI)* For consistant naming, we suggest *iRel_create* and *iRel_destroy* (respecting the discussion of create/destroy name change proposal in http://lists.mcs.anl.gov/pipermail/tstt-interface/2010-May/000166.html)
- *(RPI)* Incompleteness: *iRel_getEntSetArrRelation* and *iRel_getSetSetArrRelation* are not defined.
- *(RPI)* The argument *switch_order* is missed in the following functions

  . `iRel_setEntEntRelation`
  . `iRel_setEntSetRelation`
  . `iRel_setSetEntRelation`
  . `iRel_setSetSetRelation`

  . `iRel_setEntArrEntArrRelation`
  . `iRel_setEntArrSetArrRelation`
  . `iRel_setSetArrEntArrRelation`
  . `iRel_setSetArrSetArrRelation`

- *(RPI) is_set* shall be removed from the comment of *iRel_inferEntRelations* and *iRel_inferEntArrRelations*

A.2 Questions

- *(RPI)* The purpose of *iRel_inferAllRelationsAndType* is unclear.

  iRel_createRelation creates a relation based on the relation type provided as an input. Given a relation handle, *iRel_getRelationInfo* gets the relation type information back. Given a relation handle, two functions are provided to infer all relations underneath; *iRel_inferAllRelations* for inferring relations between entities in specified pair of interfaces and *iRel_inferAllRelationsAndType* for inferring both relations between entities and relation type. Since the relation type is obtainable from *iRel_getRelationInfo* without inferring, the purpose of *iRel_inferAllRelationsAndType* seems to unclear.

- *(RPI)* Need to extend *iBase_ErrorType* to include *iRel* specific errors[3].
  E.g. *iBase_INVALID_RELATION_HANDLE*, *iBase_RELATION_NOT_FOUND*, etc.

  In doing this, is it desirable to make an individual error type for each interface? *iBase_ErrorType*, *iMesh_ErrorType*, *iMeshP_ErrorType*, *iGeom_ErrorType*, *iField_ErrorType*, *iRel_ErrorType*.

[3]The same for iMeshP

A.3 Source Code

```
#ifndef __iRel_LASSO_HPP__
#define __iRel_LASSO_HPP__

#define IREL_MAJOR_VERSION 1
#define IREL_MINOR_VERSION 0

  /** \mainpage The ITAPS Relations Interface iRel
   *
   * Each ITAPS interface encapsulates functionality that "belongs"
   * together, for example mesh or geometric model functionality.  In
   * some cases, however, data in several of these interfaces need to
   * be related together.  For example, a collection of mesh faces
   * should be related to the geometric model face which they
   * discretize.  The ITAPS Relations interface accomplishes this in a
   * way which allows the lower-level interfaces to remain
   * independent.
   *
   * iRel defines relations as pairwise relations between entities
   * or entity sets.  Related entities can be in the same or different
   * interfaces.  A given relation is created for a given pair of
   * interfaces and returned in the form of a \em Relation \em Handle.
   * After a specific relation pair has been created, concrete
   * relations for that pair can be assigned and retrieved for
   * specific entities using set and get functions on the iRel
   * interface.  A given interface instance can appear in one or many
   * relation pairs, each identified by the relation handle.
   *
   * \section Relation Types
   *
   * Relations are also distinguished by a pair of relation types.
   * For each interface in a relation pair, a corresponding type
   * indicates whether the relation applies to entities, entity sets,
   * or both entities and sets in the corresponding interface in the
   * pair.  If only one of the interfaces in a given pair has a
   * 'both'-type, entities and entity sets in that
   * interface are each related to either entities or sets in the other
   * interface in the pair.  If both of the sides of a relation are of
   * 'both'-type, entities and sets on one side of a relation point to
   * sets on the other side.
   *
   * \section Argument Order
   *
   * Many functions in the iRel interface take as input two entities,
   * or two lists of entities, along with a relation handle.  For
   * these functions, the entities or lists are assumed to be in the
   * same order as the interfaces used to create that relation pair.
   * For example, if a relation pair is created by calling:
   * \code
   * iRel_createRelation(instance, iface1, ent_or_set1, type1,
   *                     iface2, ent_or_set2, type2,
   *                     &relation_handle, &ierr)
```

```
   * \endcode
   * and relations set by calling
   * \code
   * iRel_setEntEntRelation(instance, relation_handle,
   *                        ent1, is_set1, ent2, is_set2, &ierr)
   * \endcode
   * it is assumed that ent1 is contained in iface1 and ent2 in
   * iface2.
   *
   * For functions taking only one entity or list as input, and
   * returning an entity or list, an additional argument indicates
   * whether the input entity or list belongs to the first or second
   * interface in that relation pair.
   *
   */

#include "iGeom.h"
#include "iMesh.h"
#include "iRel_protos.h"

#ifdef __cplusplus

extern "C"
{
#endif

    /**\brief  Type used to store iRel interface handle
     *
     * Type used to store iRel interface handle
     */
  typedef void* iRel_Instance;

    /**\brief  Type used to store references to relation pairs
     *
     * Type used to store references to relation pairs
     */
  typedef struct iRel_RelationHandle_Private* iRel_RelationHandle;

    /**\brief  \enum IfaceType Enumerator specifying interface types
     *
     * Enumerator specifying interface types.  This enumeration is
     * necessary because functions to get entities of a given dimension
     * are part of the higher-level interfaces (e.g. iGeom, iMesh) instead
     * of iBase.
     */
  enum IfaceType
  {iRel_IGEOM_IFACE = 0,
   iRel_IMESH_IFACE,
   iRel_IFIELD_IFACE,
   iRel_IREL_IFACE};

    /**\brief  \enum RelationType Enumerator specifying relation types
     *
     * Enumerator specifying relation types.  A relation has two types, one
```

```
   * for each side of the relation.
   */
enum RelationType
{iRel_ENTITY = 0,
 iRel_SET,
 iRel_BOTH};

extern struct iBase_Error iRel_LAST_ERROR;

  /**\brief  Create a new iRel instance
   *
   * Create a new iRel instance.  Currently no options are implemented.
      \param options Options for the implementation
      \param *instance Interface instance
      \param *ierr Pointer to error value, returned from function
      \param options_len Length of options string
  */
void iRel_newRel(const char *options,
                 iRel_Instance *instance,
                 int *ierr,
                 const int options_len);

  /**\brief  iRel_dtor Destroy the interface object
   *
   * Calls destructor on interface object
      \param instance Interface object handle to destroy
      \param *ierr Pointer to error value, returned from function
  */
void iRel_dtor(iRel_Instance instance, int *ierr);

  /**\brief  Create a relation pair between two interfaces
   *
   * Creates a relation pair between two interfaces, passing
   * back a handle to the pair.
      \param instance Interface instance
      \param iface1 1st interface object in the relation pair
      \param ent_or_set1 This relation relates entities, sets, or both from
             1st interface object
      \param iface_type1 Type of 1st interface
      \param iface2 2nd interface object in the relation pair
      \param ent_or_set2 This relation relates entities, sets, or both from
             2nd interface object
      \param iface_type2 Type of 2nd interface
      \param *rel Pointer to relation handle, returned from function
      \param *ierr Pointer to error value, returned from function
  */
void iRel_createRelation (
  iRel_Instance instance,
  iBase_Instance iface1,
  const int ent_or_set1,
  const int iface_type1,
  iBase_Instance iface2,
  const int ent_or_set2,
  const int iface_type2,
```

```
  iRel_RelationHandle *rel,
  int *ierr);

  /**\brief Get information for this relation handle
   *
   * Get information about the interfaces and relation type for this
   * relation.  Relation type for each side is passed back as integers,
   * but values will be from RelationType enumeration.
      \param instance Interface instance
      \param rel Handle of relation pair being queried
      \param *iface1 Side 1 instance for this relation
      \param *ent_or_set1 Relation type for side 1 of this relation
      \param *iface_type1 Inferface type for side 1 of this relation
      \param *iface2 Side 2 instance for this relation
      \param *ent_or_set2 Relation type for side 2 of this relation
      \param *iface_type2 Inferface type for side 2 of this relation
      \param *ierr Pointer to error value, returned from function
  */
void iRel_getRelationInfo (
    iRel_Instance instance,
    iRel_RelationHandle rel,
    iBase_Instance *iface1,
    int *ent_or_set1,
    int *iface_type1,
    iBase_Instance *iface2,
    int *ent_or_set2,
    int *iface_type2,
    int *ierr);

  /**\brief  Destroy a relation pair
   *
   * Destroy the relation pair corresponding to the handle input
      \param instance Interface instance
      \param rel Handle of relation pair to destroy
      \param *ierr Pointer to error value, returned from function
  */
void iRel_destroyRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  int *ierr);

/**\brief  Get relations containing specified interface
   *
   * Get relations containing the specified interface
      \param instance Interface instance
      \param iface Specified interface
      \param relations Pointer to array holding returned relations
             containing specified interface
      \param relations_allocated Pointer to allocated size of relations list
      \param relations_size Pointer to occupied size of relations list
      \param *ierr Pointer to error value, returned from function
  */
void iRel_findRelations(
  iRel_Instance instance,
```

```
  iBase_Instance iface,
  iRel_RelationHandle **relations,
  int *relations_allocated,
  int *relations_size,
  int *ierr);

  /**\brief
   *
   *
      \param instance Interface instance
      \param rel Relation handle being queried
      \param ent1 1st entity of relation being set
      \param ent2 2nd entity of relation being set
      \param *ierr Pointer to error value, returned from function
  */
void iRel_setEntEntRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle ent1,
  iBase_EntityHandle ent2,
  int *ierr);

void iRel_setEntSetRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle ent1,
  iBase_EntitySetHandle entset2,
  int *ierr);

void iRel_setSetEntRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle entset1,
  iBase_EntityHandle ent2,
  int *ierr);

void iRel_setSetSetRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle entset1,
  iBase_EntitySetHandle entset2,
  int *ierr);

  /**\brief  Set a relation between an entity and several entities
   *
   * Set a relation between an entity and several entities.  If either
      is a set and that side of the relation is 'both'-type, set relations
      for individual entities in that set too.
      \param instance Interface instance
      \param rel Relation handle being queried
      \param ent1 1st entity of relation being set
      \param switch_order If non-zero, ent1 is related with iface2 and
               ent_array_2 with iface1 of
               specified relation, otherwise vica versa
```

```
      \param ent_array_2 Entity(ies) to be related to ent1
      \param num_entities Number of entities in ent_array_2
      \param *ierr Pointer to error value, returned from function
  */
void iRel_setEntEntArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle ent1,
  int switch_order,
  iBase_EntityHandle *ent_array_2,
  int num_entities,
  int *ierr);

void iRel_setSetEntArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle entset1,
  int switch_order,
  iBase_EntityHandle *ent_array_2,
  int num_entities,
  int *ierr);

void iRel_setEntSetArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle ent1,
  int switch_order,
  iBase_EntitySetHandle *entset_array_2,
  int num_entities,
  int *ierr);

void iRel_setSetSetArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle entset1,
  int switch_order,
  iBase_EntitySetHandle *entset_array_2,
  int num_entities,
  int *ierr);

  /**\brief Set relations between arrays of entities pairwise,
   *        ent_array_1[i]<->ent_array_2[i]
   *
   * Set relations between arrays of entities pairwise,
      ent_array_1[i]<->ent_array_2[i].  If either array
      contains sets and that side of the relation is 'both'-type,
      set relations for individual entities in those sets too.
      \param instance Interface instance
      \param rel Relation handle being queried
      \param ent_array_1 1st array of entities of relation being set
      \param num_ent1 Number of entities in 1st array
      \param ent_array_2 2nd array of entities of relation being set
      \param num_ent2 Number of entities in 2nd array
      \param *ierr Pointer to error value, returned from function
```

```
  */
void iRel_setEntArrEntArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle *ent_array_1,
  int num_ent1,
  iBase_EntityHandle *ent_array_2,
  int num_ent2,
  int *ierr);

void iRel_setSetArrEntArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle *entset_array_1,
  int num_ent1,
  iBase_EntityHandle *ent_array_2,
  int num_ent2,
  int *ierr);

void iRel_setEntArrSetArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle *ent_array_1,
  int num_ent1,
  iBase_EntitySetHandle *entset_array_2,
  int num_ent2,
  int *ierr);

void iRel_setSetArrSetArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle *entset_array_1,
  int num_ent1,
  iBase_EntitySetHandle *entset_array_2,
  int num_ent2,
  int *ierr);

  /**\brief  Get entity related to specified entity and relation handle
   *
   * Get entity related to specified entity and relation handle.  Also
      returns whether the related entity is an entity or a set.
      \param instance Interface instance
      \param rel Relation handle being queried
      \param ent1 1st entity of relation being queried
      \param switch_order 1st entity is related to 1st interface (=0) or 2nd
             interface (=1) of relation pair
      \param *ent2 Pointer to entity related to ent1
      \param *ierr Pointer to error value, returned from function
  */
void iRel_getEntEntRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle ent1,
  int switch_order,
```

```
  iBase_EntityHandle *ent2,
  int *ierr);

void iRel_getEntSetRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle ent1,
  int switch_order,
  iBase_EntitySetHandle *entset2,
  int *ierr);

void iRel_getSetEntRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle entset1,
  int switch_order,
  iBase_EntityHandle *ent2,
  int *ierr);

void iRel_getSetSetRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle entset1,
  int switch_order,
  iBase_EntitySetHandle *entset2,
  int *ierr);

void iRel_getEntSetIterRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle ent1,
  int switch_order,
  iBase_EntityIterator *entset2,
  int *ierr);

  /**\brief  Get entities related to specified entity and relation
   *
   * Get entities related to specified entity and relation; returns
      entity sets or contained entities, depending on relation type
      (entity, set, or both).
      \param instance Interface instance
      \param rel Relation handle being queried
      \param ent1 1st entity of relation being queried
      \param switch_order ent1 is related with 1st (=0) or 2nd (=1) interface
             of this relation pair
      \param *ent_array_2 Pointer to array of entity handles returned from function
      \param *ent_array_2_allocated Pointer to allocated size of ent_array_2
      \param *ent_array_2_size Pointer to occupied size of ent_array_2
      \param *ierr Pointer to error value, returned from function
  */
void iRel_getEntEntArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle ent1,
```

```
  int switch_order,
  iBase_EntityHandle **ent_array_2,
  int *ent_array_2_allocated,
  int *ent_array_2_size,
  int *ierr);

void iRel_getSetEntArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle entset1,
  int switch_order,
  iBase_EntityHandle **ent_array_2,
  int *ent_array_2_allocated,
  int *ent_array_2_size,
  int *ierr);

  /**\brief  Get entities related to those in specified array and relation, pairwise
   *
   * Get entities related to those in specified array and relation, pairwise.
      Returns sets or entities, depending on relation type and entities in
      ent_array_1.
      \param instance Interface instance
      \param rel Relation handle being queried
      \param ent_array_1 Array of entities whose relations are being queried
      \param ent_array_1_size Number of entities in ent_array_1
      \param switch_order Entities in ent_array_1 are related with 1st (=0)
             or 2nd (=1) interface of this relation pair
      \param *ent_array_2 Pointer to array of entity handles returned from function
      \param *ent_array_2_allocated Pointer to allocated size of ent_array_2
      \param *ent_array_2_size Pointer to occupied size of ent_array_2
      \param *offset Pointer to offset array; (*offset)[i] is index into
             (*ent_array_2) of 1st relation of ent_array_1[i]
      \param *offset_allocated Pointer to allocated size of offset
      \param *offset_size Pointer to occupied size of offset
      \param *ierr Pointer to error value, returned from function
  */
void iRel_getEntArrEntArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle *ent_array_1,
  int ent_array_1_size,
  int switch_order,
  iBase_EntityHandle **ent_array_2,
  int *ent_array_2_allocated,
  int *ent_array_2_size,
  int **offset,
  int *offset_allocated,
  int *offset_size,
  int *ierr);

void iRel_getEntArrSetArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle *ent_array_1,
```

```
  int ent_array_1_size,
  int switch_order,
  iBase_EntitySetHandle **entset_array_2,
  int *entset_array_2_allocated,
  int *entset_array_2_size,
  int *ierr);

void iRel_getSetArrEntArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle *entset_array_1,
  int entset_array_1_size,
  int switch_order,
  iBase_EntityHandle **ent_array_2,
  int *ent_array_2_allocated,
  int *ent_array_2_size,
  int **offset,
  int *offset_allocated,
  int *offset_size,
  int *ierr);

void iRel_getSetArrSetArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle *entset_array_1,
  int entset_array_1_size,
  int switch_order,
  iBase_EntitySetHandle **entset_array_2,
  int *entset_array_2_allocated,
  int *entset_array_2_size,
  int *ierr);

void iRel_getEntArrSetIterArrRelation (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle *ent_array_1,
  int ent_array_1_size,
  int switch_order,
  iBase_EntityIterator **entiter,
  int *entiter_allocated,
  int *entiter_size,
  int *ierr);

  /**\brief  Infer relations between entities in specified pair of interfaces
   *
   * Infer relations between entities in specified pair of interfaces.  The
      criteria used to infer these relations depends on the interfaces in
      the pair, the iRel implementation, and the source of the data in those
      interfaces.
      \param instance Interface instance
      \param rel Relation handle being queried
      \param *ierr Pointer to error value, returned from function
  */
void iRel_inferAllRelations (
```

```
  iRel_Instance instance,
  iRel_RelationHandle rel,
  int *ierr);

  /**\brief  Infer relations and relation type between entities in specified
   * pair of interfaces
   *
   * Infer relations between entities in specified pair of interfaces, and the
   * relation type used by this iRel implementation.  The
   * criteria used to infer these relations depends on the interfaces in
   * the pair, the iRel implementation, and the source of the data in those
   * interfaces.
   *   \param instance Interface instance
   *   \param rel Relation handle created by implementation
   *   \param *ierr Pointer to error value, returned from function
   */
void iRel_inferAllRelationsAndType (
  iRel_Instance instance,
  iRel_RelationHandle *rel,
  int *ierr);

  /**\brief  Infer relations corresponding to specified entity and relation pair
   *
   * Infer relations corresponding to specified entity and relation pair.  The
      criteria used to infer these relations depends on the interfaces in
      the pair, the iRel implementation, and the source of the data in those
      interfaces.
      \param instance Interface instance
      \param rel Relation handle being queried
      \param entity Entity whose relations are being inferred
      \param is_set Entity is a regular entity (=0) or a set (=1)
      \param iface_no Entity corresponds to 1st (=0) or 2nd (=1) interface
             in relation pair
      \param *ierr Pointer to error value, returned from function
  */
void iRel_inferEntRelations (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntityHandle entity,
  int iface_no,
  int *ierr);

void iRel_inferSetRelations (
  iRel_Instance instance,
  iRel_RelationHandle rel,
  iBase_EntitySetHandle entity_set,
  int iface_no,
  int *ierr);

  /**\brief  Infer relations corresponding to specified entities and relation pair
   *
   * Infer relations corresponding to specified entities and relation pair.  The
      criteria used to infer these relations depends on the interfaces in
      the pair, the iRel implementation, and the source of the data in those
```

```
        interfaces.
        \param instance Interface instance
        \param rel Relation handle being queried
        \param entities Array of entities whose relation are being inferred
        \param entities_size Number of entities in array
        \param is_set Entities are regular entities (=0) or sets (=1)
        \param iface_no Entities correspond to 1st (=0) or 2nd (=1) interface
               in relation pair
        \param *ierr Pointer to error value, returned from function
    */
  void iRel_inferEntArrRelations (
    iRel_Instance instance,
    iRel_RelationHandle rel,
    iBase_EntityHandle *entities,
    int entities_size,
    int iface_no,
    int *ierr);

  void iRel_inferSetArrRelations (
    iRel_Instance instance,
    iRel_RelationHandle rel,
    iBase_EntitySetHandle *entity_sets,
    int entities_size,
    int iface_no,
    int *ierr);

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /* #ifndef __iRel_LASSO_HPP__ */
```