iRel: an interface for inter-interface associations

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, ???

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

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.

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.

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.

iRel Interface

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

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 types:

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.

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-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.

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.

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.

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.

Discussion

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

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.

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.

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.

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 re-populated 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.

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.

Document revisions

1/7/10:

- 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

1/27/10:

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

Interface revisions

1/27/10:

- add 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 (woo-hoo!)