0.1 EntitySequence & SequenceData

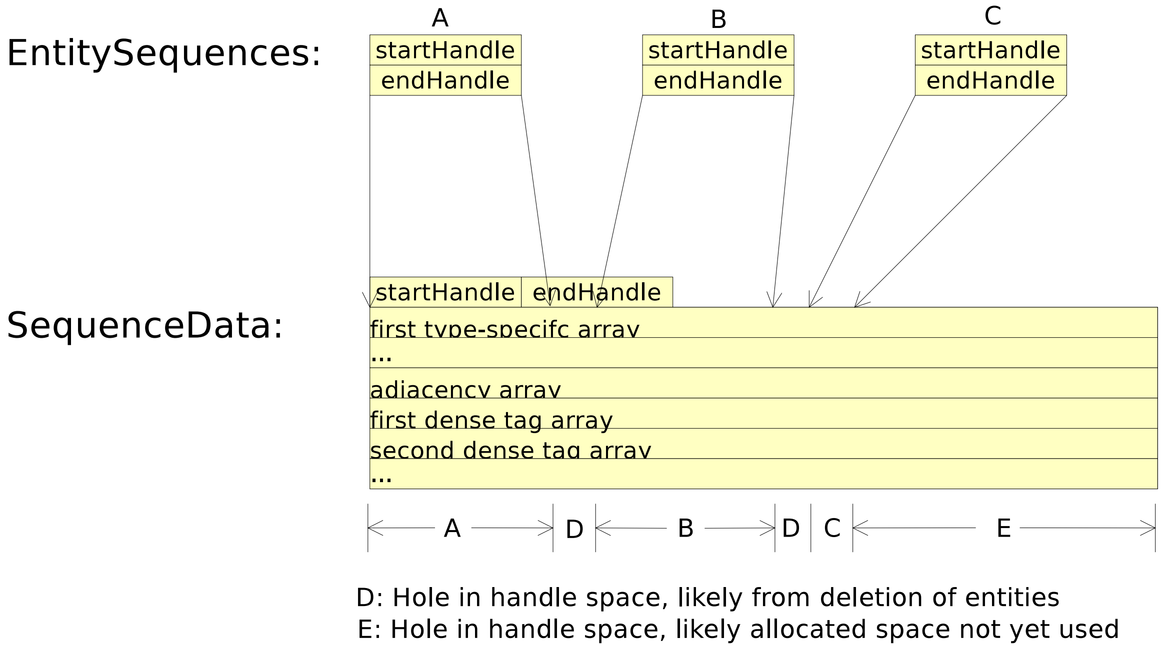

Figure 1: EntitySequences For One SequenceData

The SequenceData class manages as set of arrays of per-entity values. Each SequenceData has a start and end handle denoting the block of entities for which the arrays contain data. The arrays managed by a SequenceData instance are divided into three groups:

- Type-specific data (connectivity, coordinates, etc.): zero or more arrays.
- Adjacency data: zero or one array.
- Dense tag data: zero or more arrays.

The abstract EntitySequence class is a non-strict subset of a SequenceData. It contains a pointer to a SequenceData and the start and end handles to indicate the subset of the referenced SequenceData. The EntitySequence class is used to represent the regions of valid (or allocated) handles in a SequenceData. A SequenceData is expected to be referenced by one or more EntitySequence instances.

Initial EntitySequence and SequenceData pairs are typically created in one of two configurations. When reading from a file, a SequenceData will be created to represent all of a single type of entity contained in a file. As all entries in the

SequenceData correspond to valid handles (entities read from the file) a single EntitySequence instance corresponding to the entire SequenceData is initially created. The second configuration arises when allocating a single entity. If no entities have been allocated yet, a new SequenceData must be created to store the entity data. It is created with a constant size (e.g. 4k entities). The new EntitySequence corresponds to only the first entity in the SequenceData: the one allocated entity. As subsequent entities are allocated, the EntitySequence is extended to cover more of the corresponding SequenceData.

Concrete subclasses of the EntitySequence class are responsible for representing specific types of entities using the array storage provided by the SequenceData class. They also handle allocating SequenceData instances with appropriate arrays for storing a particular type of entity. Each concrete subclass typically provides two constructors corresponding to the two initial allocation configurations described in the previous paragraph. EntitySequence implementations also provide a split method, which is a type of factory method. It modifies the called sequence and creates a new sequence such that the range of entities represented by the original sequence is split.

The VertexSequence class provides an EntitySequence for storing vertex data. It references a SequenceData containing three arrays of doubles for storing the blocked vertex coordinate data. The ElementSequence class extends the EntitySequence interface with element-specific functionality. The UnstructuredElemSeq class is the concrete implementation of ElementSequence used to represent unstructured elements, polygons, and polyhedra. MeshSetSequence is the EntitySequence used for storing entity sets.

Each EntitySequence implementation also provides an implementation of the values_per_entity method. This value is used to determine if an existing SequenceData that has available entities is suitable for storing a particular entity. For example, UnstructuredElemSeq returns the number of nodes per element from values_per_entity. When allocating a new element with a specific number of nodes, this value is used to determine if that element may be stored in a specific SequenceData. For vertices, this value is always zero. This could be changed to the number of coordinates per vertex, allowing representation of mixed-dimension data. However, API changes would be required to utilize such a feature. Sequences for which the corresponding data cannot be used to store new entities (e.g. structured mesh discussed in a later section) will return -1 or some other invalid value.

0.2 TypeSequenceManager & SequenceManager

The TypeSequenceManager class maintains an organized set of EntitySequence instances and corresponding SequenceData instances. It is used to manage all such instances for entities of a single MBEntityType. TypeSequenceManager enforces the following four rules on its contained data:

1. No two SequenceData instances may overlap.

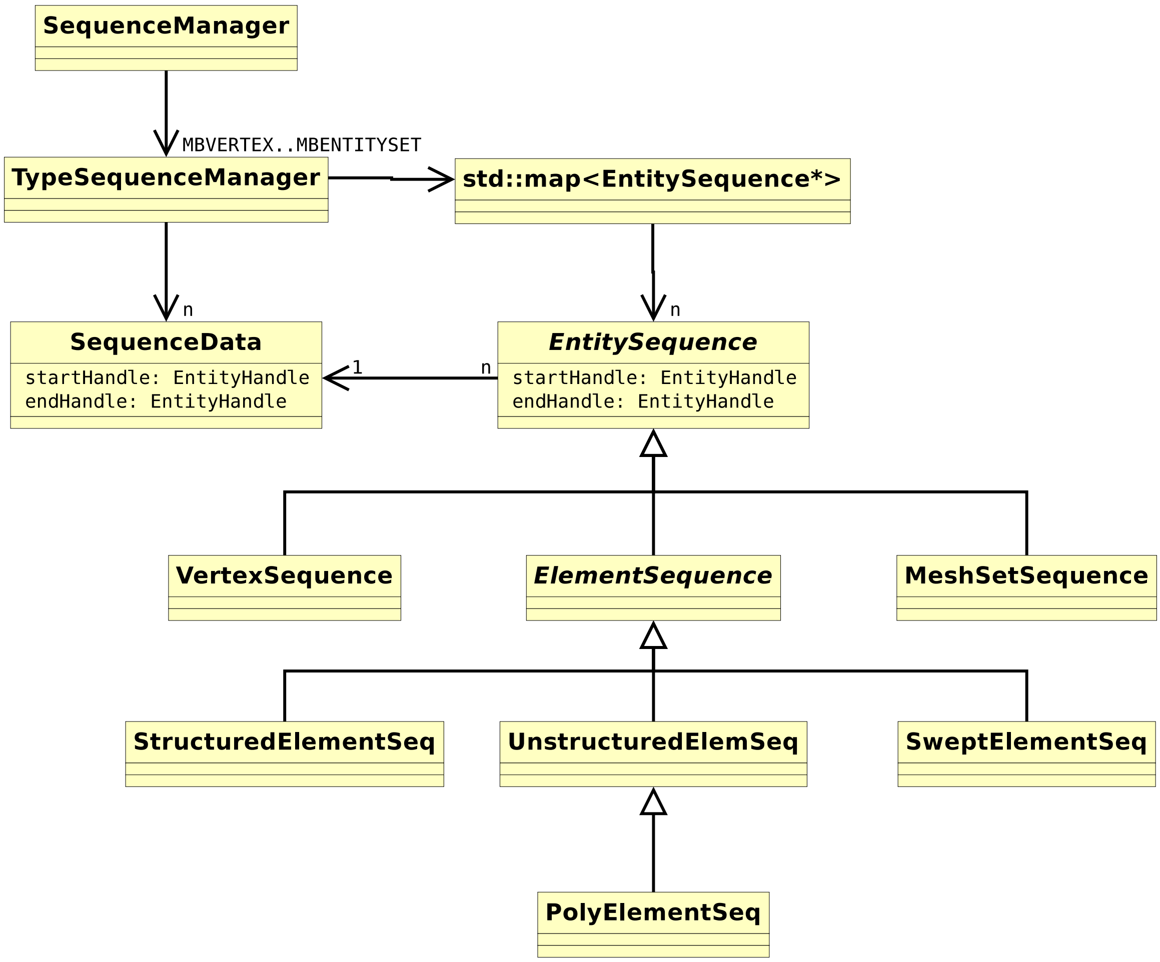

Figure 2: SequenceManager and Related Classes

2. No two `EntitySequence` instances may overlap.

3. Every `EntitySequence` must be a subset of a `SequenceData`.

4. Any pair of `EntitySequence` instances referencing the same `SequenceData` must be separated by at least one unallocated handle.

The first three rules are required for the validity of the data model. The fourth rule avoids unnecessary inefficiency. It is implemented by merging such adjacent sequences. In some cases, other classes (e.g. `SequenceManager`) can modify an `EntitySequence` such that the fourth rule is violated. In such cases, the `TypeSequenceManager::notify_prepended` or `TypeSequenceManager::notify_appended` method must be called to maintain the integrity of the data[1]. The above rules (including the fourth) are assumed in many other methods of the `TypeSequenceManager` class, such that those methods will fail or behave unexpectedly if the managed data does not conform to the rules.

`TypeSequenceManager` contains three principal data structures. The first is a `std::set` of `EntitySequence` pointers sorted using a custom comparison operator that queries the start and end handles of the referenced sequences. The comparison operation is defined as: `a->end_handle() < b->start_handle()`. This method of comparison has the advantage that a sequence corresponding to a specific handle can be located by searching the set for a "sequence" beginning and ending with the search value. The `lower_bound` and `find` methods provided by the library are guaranteed to return the sequence, if it exists. Using such a comparison operator will result in undefined behavior if the set contains overlapping sequences. This is acceptable, as rule two above prohibits such a configuration. However, some care must be taken in writing and modifying methods in `TypeSequenceManager` so as to avoid having overlapping sequences as a transitory state of some operation.

The second important data member of `TypeSequenceManager` is a pointer to the last referenced `EntitySequence`. This "cached" value is used to speed up searches by entity handle. This pointer is never null unless the sequence is empty. This rule is maintained to avoid unnecessary branches in fast query paths. In cases where the last referenced sequence is deleted, `TypeSequenceManager` will typically assign an arbitrary sequence (e.g. the first one) to the last referenced pointer.

The third data member of `TypeSequenceManager` is a `std::set` of `SequenceData` instances that are not completely covered by a `EntitySequence` instance[2]. This list is searched when allocating new handles. `TypeSequenceManager` also embeds in each `SequenceData` instance a reference to the first corresponding `EntitySequence` so that it may be located quickly from only the `SequenceData` pointer.

[1] This source of potential error can be eliminated with changes to the entity set representation. This is discussed in a later section.

[2] Given rule four for the data managed by a `TypeSequenceManager`, any `SequenceData` for which all handles are allocated will be referenced by exactly one `EntitySequence`.

The `SequenceManager` class contains an array of `TypeSequenceManager` instances, one for each `MBEntityType`. It also provides all type-specific operations such as allocating the correct `EntitySequence` subtype for a given `MBEntityType`.

0.3 Structured Mesh

Structured mesh storage is implemented using subclasses of `SequenceData`: `ScdElementData` and `ScdVertexData`. The `StructuredElementSeq` class is used to access the structured element connectivity. A standard `VertexSequence` instance is used to access the `ScdVertexData` because the vertex data storage is the same as for unstructured mesh.

0.4 `MeshSetSequence`

The representation of mesh sets within a `MeshSetSequence` results in significant complication of the code for working with the data model described in this document. Much common code for allocating handles, sequences, etc. could be moved from type-specific functions in `SequenceManager` to general purpose methods in `TypeSequenceManager`, where such general purpose methods would rely on factory/clone methods provided by `EntitySequence` implementations to handle tasks such as creating new sequences or new data instances. However, the current entity set representation requires that the `MeshSetSequence` know the type of any mesh sets (vector vs. range) when the corresponding handle is allocated. This necessitates an separate code path for entity sets for all handle allocation tasks. A new representation for entity sets that utilized a common, simple data structure as opposed to the current `std::vector` and `MBRange` storage mechanisms could defer the handling of the set type until a later time (after handle allocation), eliminating the special handling of entity sets during handle allocation.