0.1 `EntitySequence` & `SequenceData`

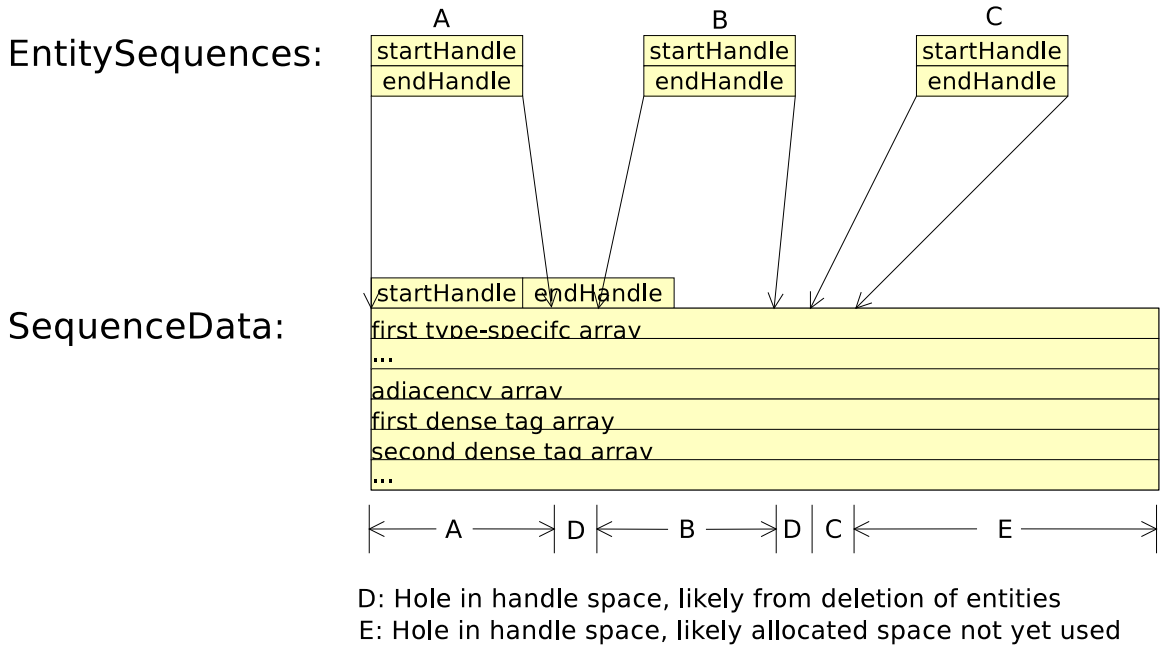

Figure 1: EntitySequences For One SequenceData

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 EntityType. TypeSequenceManager enforces the following four rules on its contained data:

1. No two SequenceData instances may overlap.

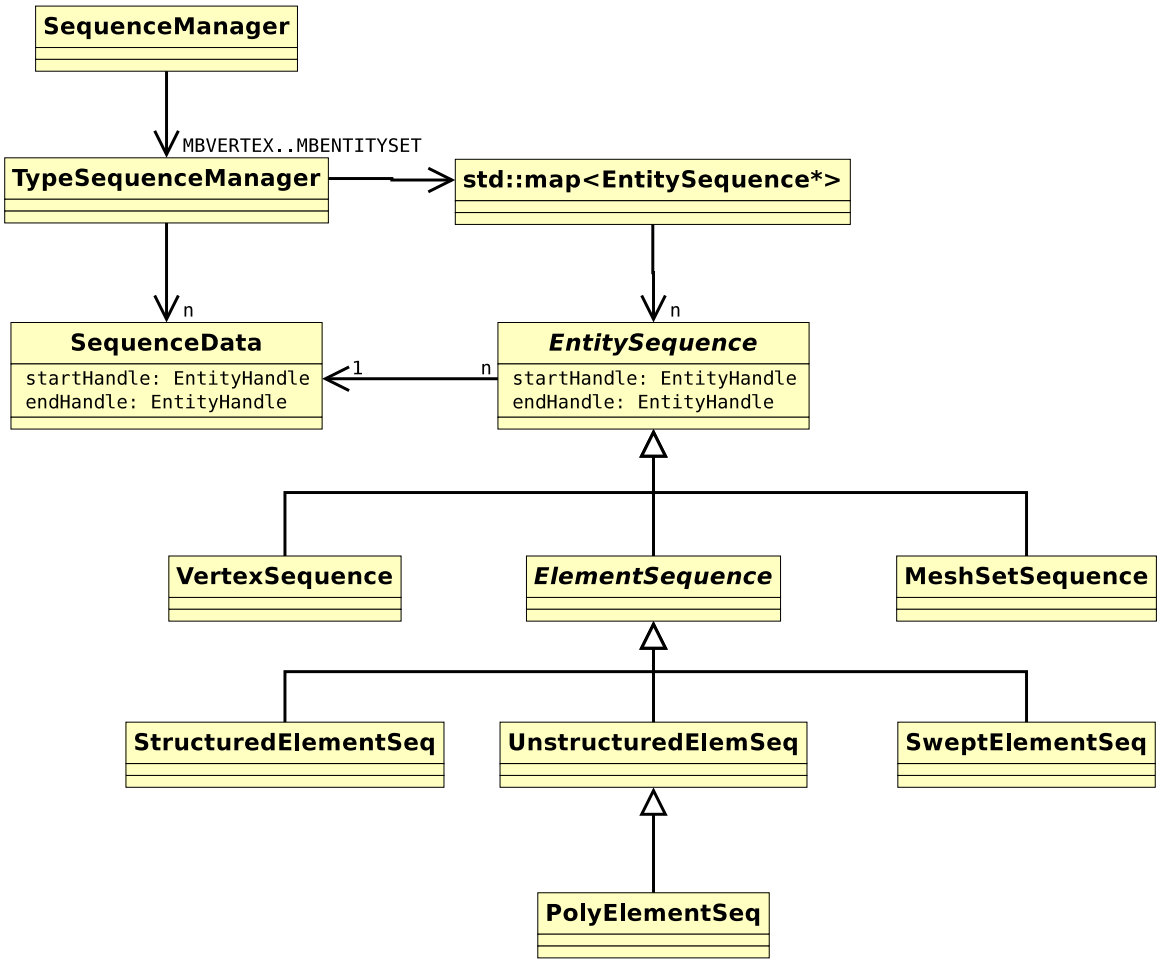

Figure 2: SequenceManager and Related Classes

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 EntityType. It also provides all type-specific operations such as allocating the correct EntitySequence subtype for a given EntityType.

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 Entity Sets

0.4.1 MeshSetSequence

The MeshSetSequence class is the same as most other subclasses of EntitySequence in that it utilizes SequenceData to store its data. A single array in the SequenceData is used to store instances of the MeshSet class, one per allocated EntityHandle. SequenceData allocates all of its managed arrays using malloc and free as simple arrays of bytes. MeshSetSequence does in-place construction and destruction of MeshSet instances within that array. This is similar to what is done by std::vector and other container classes that may own more storage than is required at a given time for contained objects.

0.4.2 MeshSet

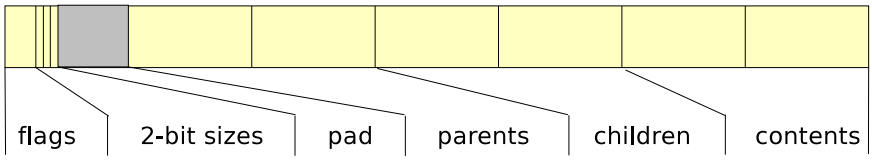

Figure 3: SequenceManager and Related Classes

The MeshSet class is used to represent a single entity set instance in MOAB. The class is optimized to minimize storage (further possible improvements in storage size are discussed later.)

Figure 3 shows the memory layout of an instance of the MeshSet class. The flags member holds the set creation bit flags: MESHSET_TRACK_OWNER, MESHSET_SET, and MESHSET_ORDERED. The presence of the MESHSET_TRACK_OWNER indicates that reverse links from the contained entities back to the owning set should be maintained in the adjacency list of each entity. The MESHSET_SET

and MESHSET_ORDERED bits are mutually exclusive, and as such most code only tests for the MESHSET_ORDERED, meaning that in practice the MESHSET_SET bit is ignored. MESHSET_ORDERED indicates that the set may contain duplicate handles and that the order that the handles are added to the set should be preserved. In practice, such sets are stored as a simple list of handles. MESHSET_SET (or in practice, the lack of MESHSET_ORDERED) indicates that the order of the handles need not be preserved and that the set may not contain duplicate handles. Such sets are stored in a sorted range-compacted format similar to that of the Range class.

The memory for storing contents, parents, and children are each handled in the same way. The data in the class is composed of a 2-bit 'size' field and two values, where the two values may either be two handles or two pointers. The size bit-fields are grouped together to reduce the required amount of memory. If the numerical value of the 2-bit size field is 0 then the corresponding list is empty. If the 2-bit size field is either 1 or 2, then the contents of the corresponding list are stored directly in the corresponding two data fields of the MeshSet object. If the 2-bit size field has a value of 3 (11 binary), then the corresponding two data fields store the begin and end pointers of an external array of handles. The number of handles in the external array can be obtained by taking the difference of the start and end pointers. Note that unlike std::vector, we do not store both an allocated and used size. We store only the 'used' size and call std::realloc whenever the used size is modified, thus we rely on the std::malloc implementation in the standard C library to track 'allocated' size for us. In practice this performs well but does not return memory to the 'system' when lists shrink (unless they shrink to zero). This overall scheme could exhibit poor performance if the size of one of the data lists in the set frequently changes between less than two and more than two handles, as this will result in frequent releasing and re-allocating of the memory for the corresponding array.

If the MESHSET_ORDERED flag is *not* present, then the set contents list (parent and child lists are unaffected) is stored in a range-compacted format. In this format the number of handles stored in the array is always a multiple of two. Each consecutive pair of handles indicate the start and end, inclusive, of a range of handles contained in the set. All such handle range pairs are stored in sorted order and do not overlap. Nor is the end handle of one range ever one less than the start handle of the next. All such 'adjacent' range pairs are merged into a single pair. The code for insertion and removal of handles from range-formatted set content lists is fairly complex. The implementation will guarantee that a given call to insert entities into a range or remove entities from a range is never worse than O(ln n) + O(m + n), where 'n' is the number of handles to insert and 'm' is the number of handles already contained in the set. So it is generally much more efficient to build Ranges of handles to insert (and remove) and call MOAB to insert (or remove) the entire list at once rather than making may calls to insert (or remove) one or a few handles from the contents of a set.

The set storage could probably be further minimized by allowing up to six handles in one of the lists to be elided. That is, as there are six potential 'slots' in the MeshSet object then if two of the lists are empty it should be possible

to store up to six values of the remaining list directly in the `MeshSet` object. However, the additional runtime cost of such complexity could easily outweigh any storage advantage. Further investigation into this has not been done because the primary motivation for the storage optimization was to support binary trees.

Another possible optimization of storage would be to remove the `MeshSet` object entirely and instead store the data in a 'blocked' format. The corresponding `SequenceData` would contain four arrays: flags, parents, children, and contents instead of a single array of `MeshSet` objects. If this were done then no storage need ever be allocated for parent or child links if none of the sets in a `SequenceData` has parent or child links. The effectiveness of the storage reduction would depend greatly on how sets get grouped into `SequenceData`s. This alternate storage scheme might also allow for better cache utilization as it would group like data together. It is often the case that application code that is querying the contents of one set will query the contents of many but never query the parents or children of any set. Or that an application will query only parent or child links of a set without every querying other set properties. The downside of this solution is that it makes the implementation a little less modular and maintainable because the existing logic contained in the `MeshSet` class would need to be spread throughout the `MeshSetSequence` class.