SAND2002-3616
Unlimited Release
Printed November 2002

SIERRA Framework Version 3: Core Services Theory and Design

H. Carter Edwards
Production Computing/SIERRA Architecture Department
Engineering Sciences Center
Sandia National Laboratories
Box 5800
Albuquerque, NM 87195-0827

Abstract

The SIERRA Framework core services provide essential services for managing the mesh data structure, computational fields, and physics models of an application. An application using these services will supply a set of physics models, define the computational fields that are required by those models, and define the mesh upon which its physics models operate. The SIERRA Framework then manages all of the data for a massively parallel multiphysics application.

Intentionally Left Blank

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

1D	one-dimensional
2D	two-dimensional
3D	three-dimensional
API	application programmer interface
CommSpec	communication specification
CPU	central processing unit
FEM	finite element method
PDE	partial differential equation

Intentionally Left Blank

1 Introduction

The purpose of this document is to describe the capabilities, abstractions, and conceptual design of the SIERRA Framework core services. This document **does not** describe the specific application programmer interface (API) or implementation of the core services. Documentation of the API appears as comments in the API header files that can be viewed through Doxygen-generated html pages.

1.1 Architecture of a SIERRA Application

All application codes that use the SIERRA Framework have the common and prescribed code architecture illustrated in Figure 1.1. The hierarchical decomposition illustrated in this figure defines several components of an application. These components have specific and well-defined roles in an application.

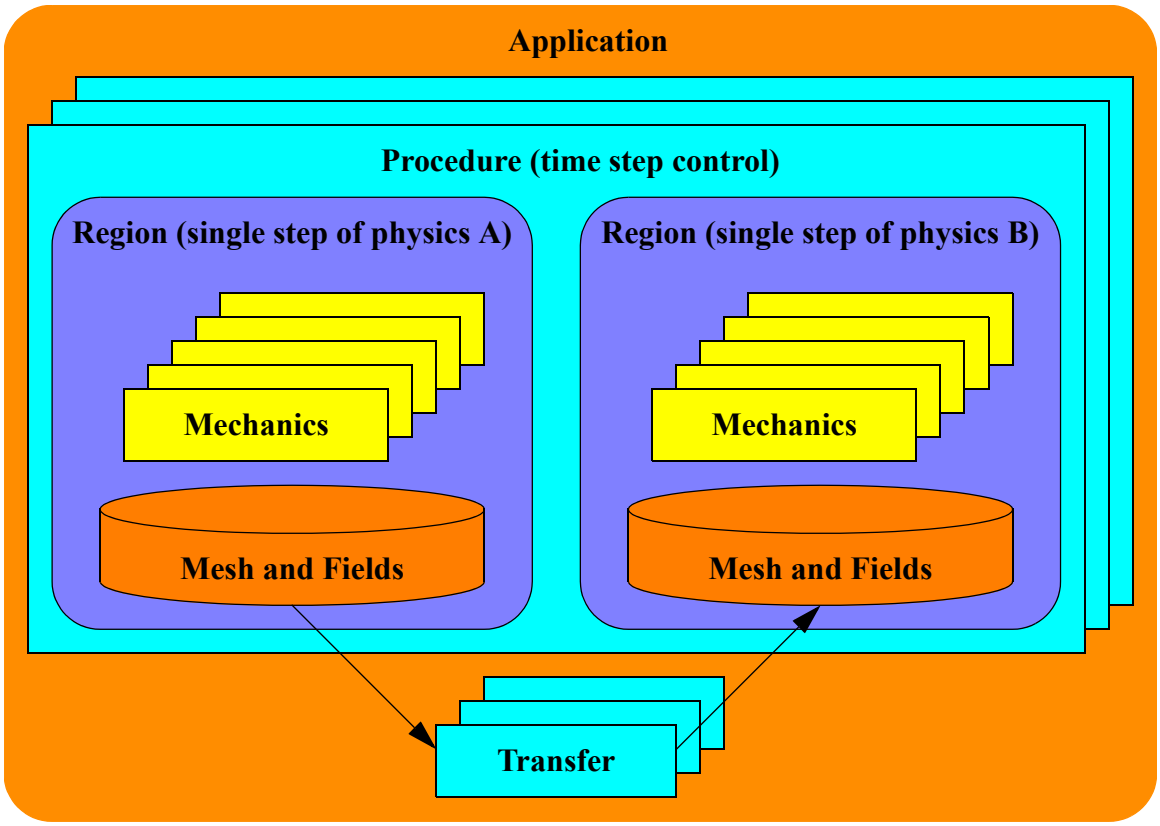


Figure 1.1. SIERRA application architecture.

Application's Procedures, Regions, and Mechanics

Each application code contains one or more **procedures** that are responsible for advancing the application through a sequence of time steps. Each procedure contains one or more **regions** that are responsible for advancing the physics that it models through a single time step. A region owns

an integrated set of **mechanics** that implement the region's physics models. A single mechanics typically implements a model for a single physics, e.g., a single partial differential equation (PDE).

Mesh and Fields

Each of the application's regions owns a **mesh** and computational **fields** required by its physics. A mesh defines a spatial discretization of a region's problem domain. Each of the region's mechanics will operate on the mesh or some subset of the mesh, e.g., a boundary. The computational fields are the independent, dependent, or scratch variables required by the region's mechanics that are defined on the mesh, e.g., coordinates of nodes in the mesh.

Transfers

A **transfer** is used to loosely couple two different regions by transferring computational fields between the regions' meshes. Transfers are an advanced capability of the SIERRA Framework and not part of the core services. Therefore, transfers are documented separately in the SIERRA Framework / Transfer Services documentation.

1.2 Execution of a SIERRA Application

The SIERRA framework executes an application in the following phases. Each phase is briefly described in this section.

1. Start-up of the parallel execution environment
2. Installation of the application's singletons
3. Processing of user input for configuration of the application's mechanics
4. Sequential processing of each of the application's procedures
    a. Construction of the meshes for the current procedure's regions
    b. Transfer of the previous procedure's region data to the current procedure's regions
    c. Execution of the current procedure
5. Termination of the application

Start-Up

Start-up of the parallel execution environment is managed by the SIERRA Framework core services; however, this start-up operation must be called immediately upon entry into the "main" code of the application. The SIERRA Framework start-up operation will initialize the application for parallel execution (currently via an implementation of the message-passing interface), process command-line arguments, and set up a parallel output stream for logging text information.

Installation of Singletons

A *singleton* is a software object that has one and only one instance of itself within a particular executable. An application's "main" code calls a "plug yourself in framework" subprogram for each of its singletons. Singletons primarily interact with the SIERRA Framework parser to configure the procedures, regions, and mechanics in the application according to an end user's specifications.

Processing of User Input

The SIERRA Framework parser reads the application user's input file synchronously on all processors. This synchronous read is accomplished by reading the input file on the "root" processor (typically processor #0) and broadcasting a sequence of input buffers to all other processors. The parser verifies that the input file is free of syntax errors and then calls upon the singletons to process the contents of the input file (see SIERRA Framework / Parser Services).

Sequential Processing of Procedures

The SIERRA Framework processes each procedure in the application once. First, a mesh is constructed for each of the procedure's regions. Construction of a mesh typically involves reading a mesh from a bulk mesh data input file and creating the mesh objects and their interconnections. Construction may also include the manufacture of mesh objects that did not appear in the bulk mesh data file, such as manufacturing of all faces on the surface of a solid body. Once a mesh has been created, it may have data transferred into it from the previous procedure's regions (see SIERRA Framework / Transfer Services). The procedure itself is then executed.

An application's procedure is expected to control the execution of its regions (Figure 1.1). Procedure execution control includes determining and performing multiple time steps, coordinating the execution of its loosely coupled regions, and subcycling the time stepping of individual regions as needed.

Termination

Upon normal termination of an application, the SIERRA Framework closes all remaining files, collects run-time statistics, and terminates the parallel execution environment.

1.3 Layered Set of Services

Other SIERRA Framework services such as user-input parsing, bulk mesh data input/output, interfaces to linear solvers, intermesh transfers, dynamic load balancing, and adaptivity are "layered on" the core services (Figure 1.2). An application primarily interacts with these "layered on" services through the SIERRA Framework core services. For example, an application informs the bulk mesh data input/output subsystem to read a particular mesh file and then that mesh file is loaded into the core services' mesh data structure without any further interaction with the application.

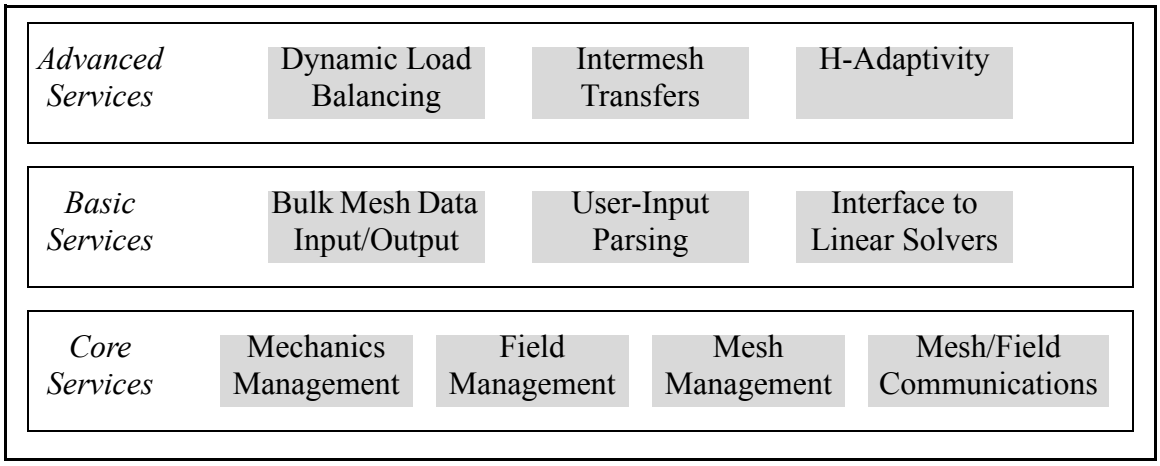


Figure 1.2. SIERRA Framework layered set of services.

1.4 Finite Element Method (FEM) Specializations

The SIERRA Framework Version 3 core services have specialized services for supporting applications that use the FEM on fully unstructured meshes. These services are aware of the potential existence of FEM master elements and FEM element topologies. This awareness requires that the *interface* to basic master element capabilities be part of the SIERRA Framework core services; however, the *implementation* of any particular master elements is not part of these core services.

The most elementary master element capability is the definition of the topology for an element; for example, a triangle "element" in a mesh is a two-dimensional (2D) entity that is defined by three vertices and has three edges. The SIERRA Framework core services are heavily dependent upon FEM element-topology information to efficiently manage the mesh data structure; therefore, this interface specification and implementation of element topology is part of the SIERRA Framework core services.

This bias for FEM master elements and element topologies does not constrain applications to use master elements. SIERRA Framework core services may be used without FEM master elements or element topologies. For example, a particle method may use the mesh and field data structure without supplying master elements or element topologies. It is also planned to extend these core services to support block-structured meshes.

1.5 Organization of this Document

The theory and abstractions expressed in the SIERRA Framework core services are presented in Section 2. This theory section includes a brief review of key concepts and terms in set theory. This review is required for the concise expression of the abstractions for the three major capabilities of the SIERRA Framework core services: mechanics management, mesh management, and field management.

The translation of these complex mathematics-based abstractions to software design is given in the remainder of this document. Numerous software-design decisions have been made with the objective of time and/or memory efficiency for complex interactions between mechanics, mesh, and fields. As such, the remaining sections of this document present the software design of the SIERRA Framework core services according to interactions between these capabilities.

2 Theory

It is the responsibility of the SIERRA Framework to manage the complexities of adaptive, multiphysics, computational mechanics applications that are used to solve large complex problems in massively parallel environments. In addition, the SIERRA Framework must be robust, efficient, scalable, extensible, and maintainable. It is essential to have a solid, mathematically based, theoretical foundation in order to successfully fulfill these responsibilities.

2.1 Fundamental Entities

The SIERRA Framework core services manage the fundamental parallel computational mechanics application entities of *processor*, *mechanics*, *mesh object*, and *field*. Complex, massively parallel multiphysics applications are managed by constructing sets of these entities and then defining subsets, set relations, and set topologies from these sets. These fundamental entities are defined as follows:

Processor

A processor is viewed as a resource that executes a single thread of instructions, has exclusive ownership of a block of memory, and can communicate with other processors. The set of processors that defines the parallel execution environment is assumed to be homogeneous and does not have any particular topology.

Mechanics

A *mechanics* is the software implementation of a model of a physics. An application contains a hierarchically organized set of coupled mechanics. The procedures and regions identified in Section 1.1 are specialized types of mechanics in this hierarchy.

Mesh Object

A *mesh* is a set of interconnected *mesh objects* that model the (spatially) discretized domain of a problem domain. Each mechanics operates on a subset of the mesh.

Field

A *field* is any independent, dependent, or scratch variable that is required by one or more mechanics to have a value associated with each mesh object upon which the mechanics operates. Each region in an application defines a set of fields on its mesh.

The sets of mechanics, mesh objects, and fields may have numerous subsets that are used to express an application's virtually unlimited possible heterogeneities in its physics and discretizations. The SIERRA Framework core services are responsible for managing these complex subsets, as well as the complex interactions between subsets of heterogeneous mechanics, mesh objects, and fields. These complex interactions are expressed and analyzed with the mathematics of set theory.

2.2 Brief and Selective Review of Set Theory

Numerous sets of objects appear in an application, as introduced in Section 1.1. The SIERRA Framework core services are responsible for managing sets of mechanics, mesh objects, and fields, as well as the many possible subsets required by an application. Well-defined abstractions and a consistent conceptual design are essential for fulfilling this responsibility; as such, the mathematics of set theory is used and briefly reviewed in this section.

- A *set* is any collection of individually identifiable *member* objects.
- Recall the usual definitions and symbols for membership, subsets, equality, inequality, intersection, union, null or empty set, *for all*, and *there exists*:
  $\in \quad \subset \quad \subseteq \quad = \quad \neq \quad \cap \quad \cup \quad \varnothing \quad \forall \quad \exists$.
- A set of sets is referred to as a *class*.
- A class $\boldsymbol{C}$ defines a *partition* for a set $\mathbf{A}$ if each member of the class $\boldsymbol{C}$ is a subset of $\mathbf{A}$, the union of all members of $\boldsymbol{C}$ is equal to $\mathbf{A}$, and the intersection of any two members of $\boldsymbol{C}$ is $\varnothing$.

$$\mathbf{A} \supset \mathbf{C}_i \in \mathbf{C}$$

$$\mathbf{A} = \bigcup_i \mathbf{C}_i$$

$$\varnothing = \mathbf{C}_i \cap \mathbf{C}_j, \forall i \neq j$$

- The product set of two sets $\mathbf{A}$ and $\mathbf{B}$, denoted $\mathbf{A} \times \mathbf{B}$, is the set of all ordered pairs $(\mathbf{a},\mathbf{b})$ where $\mathbf{a} \in \mathbf{A}$ and $\mathbf{b} \in \mathbf{B}$.
- A *relation* $\mathbf{R}$ from the set $\mathbf{A}$ to the set $\mathbf{B}$ is a subset of $\mathbf{A} \times \mathbf{B}$.
- The *domain* of a relation is the set of the first coordinates of its member pairs, and the *range* is the set of second coordinates of its member pairs.

$$\mathbf{domain}(\mathbf{R}) = \{\mathbf{a}:(\mathbf{a},\mathbf{b}) \in \mathbf{R}\}$$
$$\mathbf{range}(\mathbf{R}) = \{\mathbf{b}:(\mathbf{a},\mathbf{b}) \in \mathbf{R}\}$$

- The *converse* of a relation $\mathbf{R}$ is a relation from the range of $\mathbf{R}$ to the domain of $\mathbf{R}$, i.e., reversing the coordinates of the member pairs in $\mathbf{R}$, and is denoted $\mathbf{R}^c$.
- A relation $\mathbf{R}$ is *symmetric* if for each member pair $(\mathbf{a},\mathbf{b})$ of $\mathbf{R}$ the reversed pair $(\mathbf{b},\mathbf{a})$ is also a member of $\mathbf{R}$. Note that if a relation is symmetric then the relation is equal to its converse, $\mathbf{R} = \mathbf{R}^c$.
- A class $\boldsymbol{C}$ is a *topology* on set $\mathbf{A}$ if and only if $\boldsymbol{C}$ satisfies the following:
  a. $\mathbf{A}$ and $\varnothing$ are members of $\boldsymbol{C}$.
  b. The union of any number of sets in $\boldsymbol{C}$ is a member of $\boldsymbol{C}$.
  c. The intersection of any two sets in $\boldsymbol{C}$ is a member of $\boldsymbol{C}$.

- The pairing of a set **A** and a topology on **A**, (**A**,***C***), is a *topological space*.
- A nonempty class ***C*** of subsets of **A** *generates a topology* on **A** by (1) defining a superset of ***C*** that adds all sets that are pair-wise intersections of the members of ***C*** and then (2) expanding this superset to include all sets that are unions of any number of members of this superset.

2.3 Mechanics

A mechanics is a software implementation of a model of some arbitrary physics. This implementation performs one or more calculations with a set of fields defined over a subset of a mesh (i.e., a set of mesh objects). For example, a simple implementation of kinematics may visit each node in a finite element mesh, set the velocity to the previous value plus the acceleration scaled by time, and then update the position to the previous position plus the velocity scaled by time. In this example, the set of mesh objects consists of all nodes in the mesh, and the set of fields defines the position, velocity, and acceleration.

A single mechanics typically implements a single PDE (partial differential equation). This implementation defines the set of fields required by that PDE and provides a set of *mechanics algorithms* that perform the calculations. The implementation of a mechanics algorithm is simply a subprogram that is owned by the mechanics.

Tight Coupling Within a Region

A tightly coupled set of mechanics (e.g., coupled PDEs) is often required to model the physics of a region. Mechanics are coupled within a region whenever they share one or more fields. Coupling may be sequential, where the output of one calculation is the input to the next, or may be coupled hierarchically, such as an element calculation calling a nested material calculation. Mechanics may also be coupled in a region with an implicit time-stepping algorithm by contributing to a shared Jacobian.

Mechanics Instance

A mechanics may operate on several distinct subsets of mesh objects where the same calculation is applied to each subset but the parameters of that calculation (e.g., coefficients of the PDE) are different for each subset. The SIERRA Framework defines a separation of calculations from parameters, with mechanics responsible for supplying calculations and *mechanics instances* responsible for supplying parameters.

A mechanics owns a set of mechanics instances. Each mechanics instance is responsible for (1) identifying a distinct subset of mesh objects upon which the owning mechanics will operate and (2) supplying a set of parameters for that operation. For example, a mechanics instance identifies a subset of elements that model material “A,” and it supplies material properties for those elements. A mechanics instance *does not* supply algorithms or identify fields used by the mechanics.

Mechanics-Defined Usage Subsets and Instance Subsets

A mechanics operates on a region's mesh or some subset of a region's mesh. This subset contains one or more subsets that are associated with the mechanics instances owned by the mechanics. A mechanics also uses a subset of a region's fields. These mechanics-defined subsets of mesh objects and fields are referred to as usage subsets. Nested subsets of mesh objects associated with mechanics instances are referred to as instance subsets. This mechanics-defined hierarchy of mesh object subsets can be expressed as follows:

$$\textbf{UsageSubset}(\textbf{Mechanics}_i) = \{\textbf{MeshObject}_k \textbf{ : Mechanics}_i \textbf{ uses MeshObject}_k\}$$
$$\textbf{Mechanics}_i \textbf{ has } \{\textbf{MechanicsInstance}_{ij}\}$$

$$\textbf{InstanceSubset}(\textbf{MechanicsInstance}_{ij}) =$$
$$\{\textbf{MeshObject}_k \textbf{ : MeshObject}_k \textbf{ associated-with MechanicsInstance}_{ij}\}$$

$$\textbf{UsageSubset}(\textbf{Mechanics}_i) = \bigcup_j \textbf{InstanceSubset}(\textbf{MechanicsInstance}_{ij})$$

The set of usage subsets defines a class for a region's mesh. This usage class is not a partition of the mesh; a mesh object may and usually will be a member of more than one usage subset. Likewise, each set of instance subsets associated with a particular usage subset defines another class. For some mechanics, the associated instance class may be a partition of that mechanics' usage subset, and the mechanics' calculations may be optimized for this property. However, in general, an instance class will not be a partition of the associated mechanics' usage subset.

2.4 Mesh, Mesh Object, and Mesh Object Roster

A mesh is implemented as a set of interconnected mesh objects that model the discretized problem domain for a tightly coupled set of mechanics within a region. These mechanics may, at run time, modify the discretization of the problem domain. Dynamic modifications to a mesh may involve creating new mesh objects in the mesh, deleting existing mesh objects from the mesh, modifying mesh object connectivity within the mesh, or adding/removing mesh objects from/to mechanics-defined usage subsets and instance subsets.

Finite-Element Mesh Objects

The SIERRA Framework explicitly defines the following five types of mesh objects to support the FEM on unstructured meshes:

Node

A *node* mesh object represents a point in a one-dimensional (1D), 2D, or 3D discretization of a problem domain.

Edge

An *edge* mesh object represents a simple curve between two nodes in a 2D or 3D discretization of a problem domain. The specification for an edge must include its end-point nodes and may include additional interior nodes.

Face

A *face* mesh object represents a simple area enclosed by simply connected edges in a 3D discretization of a problem domain, e.g., a triangle or quadrilateral. The specification of a face must include a set of vertex nodes that are the end points for faces' boundary edges and may include additional boundary or interior nodes.

Element

An *element* mesh object represents a simple region of the same dimension as the discretization of the problem domain. For example, in a 3D discretization an element represents a simple volume, and in a 2D discretization an element represents a simple area. The specification of an element must include a set of vertex nodes and may include additional boundary or interior nodes.

Other / Constraint

Other mesh objects may be needed to model irregular connectivity in a discretized problem domain. For example, a problem domain with large deformation self-contact may define a constraint mesh object that connects a node to the face that the node should be prevented from penetrating. This fifth type is a "catch all" for representing irregular connectivity.

It is expected that Version 4 of the SIERRA Framework will support additional types of mesh objects, particularly mesh object types to support block-structured meshes. As such, the software design and implementation for mesh objects must be extensible.

Mesh object types define a partitioning of a region's set of mesh objects. A mesh object must be of a specific type. This type-defined partitioning of the set of mesh objects is used in the SIERRA Framework software design to optimize algorithmic access to mesh objects. Each type-specific subset of mesh objects is called a *mesh object roster*, sometimes referred to as simply a *roster*.

Parallel Distributed Mesh

The SIERRA Framework distributes a region's set of mesh objects among the processors of a distributed-memory parallel-processing environment. This distribution is defined by the following three classes of mesh objects:

- Each processor has a subset of mesh objects that reside on that processor. The set of these processor-resident subsets defines a processor-resident class.
- Each processor has a subset of mesh objects that it owns. The set of these processor-owned subsets defines a processor-owned class.
- Each processor has a subset of mesh objects that it shares with other processors. The set of these processor-shared subsets defines a processor-shared class.

A processor can only own mesh objects that reside on that processor; therefore, each processor-owned subset is a subset of the corresponding processor-resident subset. The processor-owned class is a partitioning of the mesh—each mesh object in a mesh is owned by exactly one processor. The processor-ownership partition is essential for coordinating global reduction operations on fields, such as a norm calculation, and for synchronizing modifications to a distributed mesh.

A pair-wise intersection of members from the processor-resident class defines subsets of mesh objects that are shared between pairs of processors. The set of these shared subsets defines a class that describes the global sharing of all mesh objects among all processors. Each processor also has a subset of shared mesh objects that is defined by the union of the mesh objects that the processor shares with all other processors.

$$\textbf{Resident}_{P_i} = \{\textbf{MeshObject}_k : \textbf{MeshObject}_k \textbf{ resides-on } P_i\}$$

$$\textbf{Owned}_{P_i} = \{\textbf{MeshObject}_k : \textbf{MeshObject}_k \textbf{ owned-by } P_i\}$$

$$\textbf{GlobalSharing} = \{\textbf{GlobalShared}_{P_i, P_j} = \textbf{Resident}_{P_i} \cap \textbf{Resident}_{P_j}\}$$

$$\textbf{LocalShared}_{P_i} = \bigcup_{P_j} \textbf{GlobalShared}_{P_i, P_j}$$

Generated Set-Topology for a Mesh

Six classes of mesh object subsets are defined: a partition of mesh objects by type (e.g., node, face, edge, element), a mechanics-defined usage class, the mechanics instance-defined instance class, a processor-resident class, a processor-owned class, and a processor-shared class. The SIERRA Framework uses the union of these classes to generate a set-topology on a region's mesh. This topology is significant in that any operation that is performed on the mesh will be applied to exactly one member of this topology.

Within this topology there exists a subclass **S** of the generated set-topology **T** with the following two properties. First, the subclass **S** is a partition for the mesh. Second, the intersection of any member of **S** with any member of **T** is either the null set or the same member from **S**, i.e., $S \cap T \in \{\varnothing, S\}\ \forall S \in \mathbf{S}$ **and** $T \in \mathbf{T}$. Each member of the partition **S** is a subset of mesh objects that are always operated on in the same manner. Thus a mechanics' calculation could be optimized to operate on members of **S** as a group, as opposed to operating on individual mesh objects.

Each member of **S** is a subset of mesh objects that have the same mechanics usage, mechanics instance association, and parallel operations. Each member of **S** is composed of a subset of mesh objects that are homogeneous with respect to the operations performed on them. Each

homogeneous subset (i.e., member of **S**) is called a *mesh object subroster*, sometimes referred to as simply a *subroster*.

Intramesh Connectivity Relations

Let $\mathbf{Mesh}_\alpha$ be a set of mesh objects that defines a particular mesh denoted α. Then the set of connections between pairs of mesh objects in the same mesh defines an intramesh connectivity relation for that mesh, e.g., $\mathbf{ConnectivityRelation}_\alpha \subset \mathbf{Mesh}_\alpha \times \mathbf{Mesh}_\alpha$. This connectivity relation is partitioned into an extensible class of *purpose subsets* that currently includes the following:

- The *uses* connectivity relation has domain members that use, or are dependent upon, range members. For example, an element has a uses connection with each of its nodes.
- The *used-by* connectivity relation has domain members that are used by range members. For example, a node may be used by several elements.
- Some mesh objects may be hierarchically partitioned. The *child* connectivity relation contains the parent-to-child connectivity.
- The *parent* connectivity relation contains the child-to-parent connectivity.
- The "other" or "auxiliary" connectivity relation contains mesh object connections that do not fit in one of the previous four subsets.

Another significant partitioning of the connectivity relation is defined by the subsets that have a given mesh object for the domain coordinate.

$$\{(\mathbf{m}_i,\mathbf{m}_j) : (\mathbf{m}_i,\mathbf{m}_j) \in \mathbf{ConnectivityRelation}_\alpha , \forall j\}_i$$

This class, for example, would include a subset that defines the connections from a given element to its nodes, edges, and faces. Members of this class may be intersected with members of the purpose class to define subsets such as all nodes that an element uses.

Intermesh Relations

The previous connectivity relation is defined with the same mesh for its domain and range. Similar intermesh relations are needed to define relationships between two different regions, e.g., $\mathbf{IntermeshRelation}_{\alpha,\,\beta} \subset \mathbf{Mesh}_\alpha \times \mathbf{Mesh}_\beta$. Such a relationship is needed to support loose coupling of two different regions.

2.5 Field and Field Registrar

A field is simply any variable that is used by a mechanics and has a value associated with each mesh object used by that mechanics. Each region owns a set of fields that are used by that region's

mechanics and associated with that region's mesh objects. This set of fields is partitioned according to mesh object type, i.e., five subsets for nodal fields, edge fields, face fields, element fields, and constraint fields. Each type-specific subset of fields is called a *field registrar*.

Field Types

The values of a field are of a specified field type. This field type may be a simple numeric type, a complex aggregation of values, or an array of a values of a specified field type. An aggregate field type is a collection of named values, each with its own field type; for example, a `struct` in the C or C++ language is an aggregate field type.

Field Value Relation

A field value is associated with the pairing of a field with a mesh object of the same type, e.g., pairing a node with a nodal field. This association defines the following field value relation for each type of mesh object:

$$\textbf{FieldValueRelation} \subseteq (\textbf{Mesh} \times \textbf{Field}) \times \textbf{FieldValues}$$

Not every pairing of a mesh object and field is required to have a value; therefore, the field value relation is a subset of the Cartesian product of its domain (mesh object–field pairs) and range (field values).

Existence of Field Values

The existence of a field value for a given mesh object–field pair is determined by the usage of that mesh object and field. If the mesh object and field are used by a mechanics, then a field value must exist; however, if the mesh object and field do not have a mechanics in common, then a field value should not exist. Thus a field value exists for a mesh object–field pair if and only if there exists a common mechanics-defined usage subset for the mesh object and the field.

$$\textbf{FieldUsageSubset}(\textbf{Mechanics}_i) = \{\textbf{Field}_k : \textbf{Mechanics}_i \textbf{ uses } \textbf{Field}_k\}$$

$$((\textbf{MeshObject}_j, \textbf{Field}_k), \textbf{FieldValue}) \in \textbf{FieldValueRelation}$$

$$\textbf{if and only if}$$

$$\exists\, \textbf{Mechanics}_i : \begin{cases} \textbf{MeshObject}_j \in \textbf{UsageSubset}(\textbf{Mechanics}_i) \textbf{ and} \\ \textbf{Field}_k \in \textbf{FieldUsageSubset}(\textbf{Mechanics}_i) \end{cases}$$

3 Usage Subsets, Instance Subsets, and Context

The set of mesh objects and the set of fields each has a class of usage subsets that is defined by the mechanics that operate on mesh objects and use the fields. This three-way interaction between mechanics, mesh, and fields is heavily exercised by an application—each execution of a mechanics algorithm iterates the mechanics-induced subset of mesh objects and uses the mechanics-induced subset of fields. It is critical to the performance of an application to have an efficient software implementation of this complex three-way interaction.

Usage subsets are only defined for mechanics that are in the same region as the mesh (see Figure 1.1). Mechanics that are further nested in this hierarchy do not define new usage subsets but instead inherit the usage subset that is defined for the root mechanics of the hierarchy.

The set of mesh objects has an additional class of instance subsets that is associated with the mechanics instances owned by the mechanics. Each mechanics instance is owned by a particular mechanics; therefore, each corresponding instance subset is a subset of the mechanics' usage subset. Instance subsets do not have an effect on fields; therefore, usage subsets and instance subsets are treated separately. Just as with usage subsets, it is critical to an application's performance to have an efficient software implementation of the two-way interaction between mechanics instances and mesh objects.

3.1 Cardinality Assumptions

The software design for usage subsets and instance subsets has assumptions regarding the cardinality of these classes of sets. These assumptions are based upon an informal survey of mechanics application developers and the planned capabilities for their applications. Minor adjustments to these assumptions (i.e., same order of magnitude) are possible with negligible effort. However, a major modification that would significantly alter the order of magnitude of the assumed cardinalities would require a redesign of the SIERRA Framework core services.

1. It is assumed that during the execution of an application the cardinality of the set of usage subsets has an upper bound of approximately 100 members.

2. It is assumed that during the execution of an application the cardinality of the set of instance subsets that are nested within a particular usage subset has an upper bound of approximately 10,000 members.

These upper bounds are an indication of the anticipated complexity of an application. The number of usage subsets is an indication of the heterogeneity of the mechanics that are available to a given region. Each distinct mechanics that is owned by a region defines a usage subset; therefore, there is an upper bound of approximately 100 such mechanics. The number of instance subsets is an indication of the complexity of the mesh, i.e., how many blocks of materials, boundary conditions, etc., are defined on the mesh.

3.2 Abstraction for Context

Usage subsets are coordinated SIERRA Framework core services of mechanics, mesh, and field management through a shared abstraction and software entity referred to as *context*. Each mechanics, mesh object, or field has an associated *context*.

Context

The *context* of a mesh object or field is a specification of how that mesh object or field is used within the application.

Contexts are defined by usage subsets. The set of usage subsets includes mechanics-defined usage subsets. Other usage subsets may be defined by nonmechanics considerations such as massively parallel partitioning (Section 2.4), master element usage or equivalence-use (Section 4.3), or the "condition" of a mesh object in a dynamically changed mesh (e.g., element death or h-adaptivity). Each context defines a particular subset of mesh objects or fields.

Efficient access to subsets of mesh objects (or fields) that are defined by intersections (or unions) of context subsets is also critical. For example, an inner product over a field may require processing of all nodes that are (1) used by a given mechanics, (2) owned by the local processor, and (3) are not "dead." Because such an inner product may be performed several times within a given time step, efficient iteration of this three-way intersection of subsets is essential.

3.3 Software Design for Context

Each mechanics, mesh object, and field has a context. This context is an array of boolean values (e.g., bit flags), where each entry in the array is associated with a distinct usage-subset within a particular **region** (Section 1.1). The length of this array is constant, fixed during compilation, and corresponds to the assumed upper bound on the number of usage subsets. This software design for context allows highly efficient operations on context objects and efficient memory utilization.

The membership of a given mesh object or field in a region's particular usage subset is determined by querying the appropriate entry in the context's array of boolean values. Such a query is only valid for mesh objects and fields that reside in the given region. The scope of a context is limited to a region, i.e., a context defined within one region is not applicable or even discernible within a different region. If the result of such a query is true, that mesh object or field is a member of the usage subset. If the result of the query is false, that mesh object or field is not a member of the usage subset.

$$\mathrm{i} = \textbf{UsageSubsetIndex}(\textbf{Mechanics}_j)$$

$$\textbf{Mechanics}_j \textbf{ uses } \textbf{MeshObject}_k \Leftrightarrow \textbf{context}(\textbf{MeshObject}_k)[\mathrm{i}] = \textbf{true}$$

$$\textbf{Mechanics}_j \textbf{ uses } \textbf{Field}_m \Leftrightarrow \textbf{context}(\textbf{Field}_m)[\mathrm{i}] = \textbf{true}$$

Evaluation for membership in intersections and unions of usage subsets is efficiently obtained through queries of the context array. If a mesh object is in the intersection of three usage subsets,

each of the three boolean entries is true. If a field is in the union of several usage subsets, at least one of the boolean entries is true. Such queries on fixed-length arrays of boolean values (i.e., bits) is very efficient in the C++ programming languages used by the SIERRA Framework.

3.4 Interaction with Mesh Objects

Recall that the set of mesh objects is partitioned into homogeneous subsets referred to as subrosters (Section 2.4). A necessary condition for a subroster to be homogeneous is that all mesh objects in that subroster have the same context. As such, any evaluations regarding membership in context-defined usage subsets may be made once for each subroster, and the result is applicable to all members of the subroster. Efficient iteration of a usage subset is performed by iterating subrosters, evaluating whether the subroster's context contains the usage-subset context, and, if so, iterating members of the subroster either by individual mesh objects or by bucket (Section 7.1).

Each mesh object in a particular subroster has an identical context. This context is associated with the subroster and is shared by all mesh objects that are members of the subroster. Context is one of several attributes associated with a subroster. These attributes are shared by all mesh objects that reside in the subroster.

3.5 Field Value Relation

Field values are associated with the pairing of a mesh object and a field defined for that type of mesh object (Section 2.5). This association may be thought of as a *field value relation*, where the domain is in the set of mesh object–field pairs and the range is the memory for field values.

$$\textbf{FieldValueRelation} \subset (\textbf{Mesh} \times \textbf{Field}) \times \textbf{FieldValues}$$

Note that the field value relation defined here is a subset of all possible members; thus not every mesh object–field pair has an associated field value.

A field value is only needed for mesh object–field pairs that are used by one or more of the same mechanics. If the subset of mechanics that uses a particular field does not operate on a given mesh object, there is no need for a field value to exist for that mesh object.

A simple software design for the field value relation would be to disregard the mechanics usage information and provide field values for all mesh object–field pairs. Such a simplistic approach could generate large numbers of field values that are not used in the application. For example, if a boundary condition defines a field on boundary nodes, setting aside values for all nodes would be a highly undesirable waste of memory.

The SIERRA Framework core services only allocate field values that will be used by a mechanics. The existence of a field value for a given mesh object–field pair is determined by comparing the context of the mesh object and field.

$$((\textbf{MeshObject}_k, \textbf{Field}_m)\ , \textbf{FieldValue}) \in \textbf{FieldValueRelation}$$

$$\textbf{if-and-only-if}$$

$$\exists i : (\textbf{context}(\textbf{MeshObject}_k)[i] = \textbf{true})\ \textbf{and}\ (\textbf{context}(\textbf{Field}_m)[i] = \textbf{true})$$

A field value will exist for a mesh object–field pair if and only if there exists some entry that is set in both the mesh object's context and the field's context. Since the context is represented as an array of bit flags, this condition is efficiently evaluated through the C++ bit-wise "and" operator.

```
if ( MeshObject.context() & Field.context() ) then
  value exists
else
  value does not exist
endif
```

Implications for Efficiency of Simple Algorithms

Each subroster is a homogeneous set of mesh objects that share a context. Therefore, the above evaluation for the existence of a field value is directly applicable to the existence of field values in a subroster's buckets (Section 7.1). Algorithms that access a particular field, such as a norm or inner product, are most efficiently implemented by iterating the subrosters that are compatible with that field.

```
foreach Subroster in Roster
  if ( Subroster.context() & Field.context() ) then
    foreach Bucket in Subroster
      access field value arrays
      perform computations over these arrays
    end
  endif
end
```

3.6 Predefined Usage Subsets

The SIERRA Framework core services predefines eight usage subsets and their associated contexts. These usage subsets primarily support distributed, dynamic mesh capabilities. Each of these usage subsets is only applicable to mesh objects. Membership in these usage subsets has significant implications for how the SIERRA Framework processes mesh objects.

Active and Inactive

These are two disjoint usage subsets of mesh objects that either should or should not be processed by a region's mechanics. The SIERRA Framework will only process mesh objects that are members of the active usage subset in workset algorithms, nodal linear-algebra utilities, and default iteration of a roster. Members of the inactive usage subset are mesh objects that must be

retained in the mesh data structure but should not be processed in normal calculations. Mesh objects are placed in the inactive usage subset when “killed” by the element-death capability or when refined by the h-adaptivity capability.

Globally Shared and Locally Owned

In a massively parallel distributed mesh data structure (Section 2.4), a given mesh object may reside on more than one processor. If that mesh object is shared by two or more processors, it is a member of the globally shared usage subset. If a mesh object is owned by the processor on which it resides, it is part of that processor’s locally owned usage subset. Note that each mesh object will be a member of exactly one processor’s locally owned usage subset.

Mesh objects are owned by exactly one processor so that reduction calculations or mesh-update operations may be efficiently and robustly implemented. In a reduction calculation, such as a norm or an inner product, only the owning processor contributes a term to the global reduction for a given mesh object. In a mesh-update operation, the owning processor is responsible for coordinating modifications to the mesh object.

Pending Create and Pending Delete

The SIERRA Framework core services support dynamic mesh modifications such as element death and h-adaptivity. These capabilities (and others) create new mesh objects and delete mesh objects that are no longer needed. Such modifications are handled in two phases: (1) local modifications and (2) global synchronization. Mesh objects created in the local modification phase are placed in the pending-create usage subset. Mesh objects that are to be deleted are placed in the pending-delete usage subset during local modifications. Global synchronization processes these two usage subsets to consistently update the distributed mesh data structure for the locally created and to-be-deleted mesh objects.

Exposed Boundary and Interblock Boundary

In a parallel distributed mesh it is difficult to determine whether the face of an element is on the boundary of the global mesh or merely on the boundary of the local processor’s subdomain. One of the SIERRA Framework core services is to manufacture and/or identify the faces on the exposed boundary of the global mesh and on boundaries between blocks of materials (Section 6.4). These faces are members of the exposed-boundary and interblock-boundary usage subsets, respectively.

3.7 Instance Subsets

Recall from Section 2.3 that an instance subset is a collection of mesh objects that are associated with a given mechanics instance. Also recall that each mechanics instance provides parameters to its associated mechanics and that these parameters are applicable to all mesh objects of the instance subset. For example, parameters provided by a mechanics instance could define material properties for a material mechanics or boundary values for a boundary-condition mechanics.

Instance subsets and mechanics instances are typically used to differentiate parts of a the system being modeled. These parts may be physically distinct, as in the differentiation between the rim and tire of a wheel; or these parts may be an artifact of a Computer-Aided Design (CAD) model, as in the modeling of a simple nail as a thin disk attached to a thin cylinder even though the nail is a single continuous material.

The number of instance subsets in a specific problem is directly proportional to the physical complexity of the system being modeled—more parts leads to more instance subsets. In contrast, the number of usage subsets is proportional to the complexity of the physics being modeled. Recall from Section 3.1 the assumption that the number of usage subsets has an upper bound of approximately 100 and the number of instance subsets has an upper bound O(10,000).

A final assumption regarding the cardinality of sets of subsets is that a given mesh object is a member of a relatively small number of instance subsets, e.g., O(10). This assumption is not a "hard limit" in the SIERRA Framework—a particular mesh object could be a member of more than O(10) instance subsets. However, the time required to determine whether a mesh object is a member of a particular instance subset is proportional to the number of instance subsets to which it belongs. In contrast, the software design for context and usage subsets (Section 3.3) allows membership of a mesh object in a particular usage subset to occur in a small fixed amount of time.

4 Mesh Heterogeneity and Master Element Usage

Master elements have three usages in finite element applications. First, master elements are used by mechanics to perform their calculations. Second, master elements define “templates” for mesh object connectivity. And third, master element properties may be associated with fields. The usages are complicated when an application uses a mixed FEM (finite element method), i.e., when the application uses more than one master element for its calculations on a given mesh object. Furthermore, a mesh may be heterogeneous. For example, a mesh may be a set of interconnected hexahedral, pyramid, and tetrahedral elements.

4.1 Master Element Families

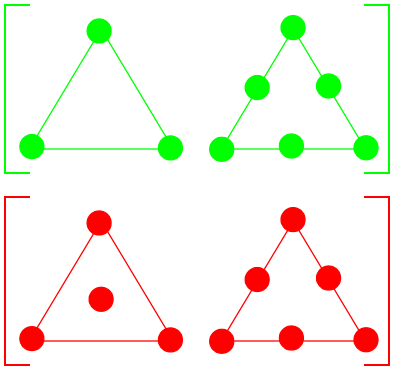
The complete set of master elements used by a region’s mechanics is maintained in the region’s field registrars. This set of master elements is accordingly partitioned according to mesh object type (i.e., elements, faces, edges, and nodes) such that each subset is maintained in the appropriate field registrar. Each type-specific subset of master elements is further partitioned into *master element families*.

A mechanics may use multiple master elements in its calculations. For example, a mechanics that solves the Stokes equation may use a linear master element for the pressure discretization and a quadratic master element for the velocity discretization. In this example, the linear master element may only “know” about an element’s vertex nodes, and the quadratic master element may “know” about both vertex nodes and edge nodes.

Topological Consistency

A master element family is defined by a set of master elements that are associated with the same subset of mesh objects. Each member of the family must be topologically consistent (Figure 4.1), i.e., the master element family must include at least one master element that is the topological superset of all other master elements in that family.

Two simple examples of sets of master elements are given in Figure 4.1. In this figure, three different triangular topologies of master elements are illustrated, with the dots representing nodes. The upper set is a master element family—the second element defines a superset of the first element’s nodes. The lower set is not a master element family—the second element does not include the center node of the first element, and the first element does not include the edge nodes of the second element.



Is a master element family. The second element defines a superset of the first element's connected nodes.

Is **NOT** a master element family. Neither element defines a superset of connected nodes.

Figure 4.1. Topological consistency of a master element family.

Text Labels for Family Members

Each field registrar may hold zero to many master element families. Typically, a master element family is defined for each shape element in the mesh, e.g., a master element family for hexahedrals and another family for tetrahedrals. Each member of a master element family is given a text label that is used by a mechanics to query specific master elements from the family. Note that if a master element family has only one member, the text label is extraneous.

The master element text labels must be unique within a family, but these labels should be replicated for each family. For example, if the 4-node linear and 8-node quadratic master elements within a tetrahedral family are labeled "LOW-ORDER" and "HIGH-ORDER," respectively, the 8-node linear and 27-node quadratic master elements within a hexahedral family should also be labeled "LOW-ORDER" and "HIGH-ORDER," respectively. Applications are strongly encouraged to follow this labeling convention so that their mechanics may be polymorphic with respect to the master element family—that the same mechanics could be applied to different element shapes simply by querying and using master elements.

Another Partitioning for Mesh Objects

A particular element has a shape, e.g., an element cannot be simultaneously a hexahedral and a tetrahedral. Each element (or face or edge or node) must be associated with no more than one master element family. For example, if a region's mechanics do not use edge master elements, an edge master element family is not needed, and the region's edges will not be associated with a master element family. Thus a mesh may be partitioned by a class of subsets where one member of the class is associated with "no master elements" and every other member of the class is uniquely associated with a master element family. This partitioning is denoted the *equivalence-use* partitioning (Section 4.3).

4.2 Field Dependency on Master Elements

Element shape and discretization heterogeneities over a region's mesh have an impact on the fields of a region. A field may be an array that is dimensioned according to some element

property, such as the number of nodes or number of terms in a numerical integration rule. If the element property that the field depends upon varies among elements, the field's array dimension must vary accordingly. For example, a field's type may be a 3D vector, and the field may be associated with the nodes of a master element. If the master element is a simple hexahedral, then this field will be dimensioned, using the FORTRAN convention, as **`field(3,8)`**. If the master element is a simple tetrahedral, the field will be dimensioned as **`field(3,4)`**.

The second dimension of some computational fields is allowed to vary with the master element (i.e., equivalence-use) partitioning, while all other properties (e.g., the field type) of the field remain the same. This variance is unambiguous since any given element must belong to exactly one member of this partitioning. A field that varies over this partitioning is said to have an *equivalent use* for the mesh objects within each subset of the partitioning.

4.3 Equivalence-Use

A partitioning of a mesh is defined by the set of subsets associated with master element families and one subset that is not associated with any master element family. This partitioning is the equivalence-use partitioning of a mesh. The set of master element families does not induce a partitioning of fields. A field may be associated with many equivalence-uses; however, it may have a different array dimension for each equivalence-use.

An equivalence-use defines usage subsets and is assigned a unique entry in a region's context (Section 3). Equivalence-use information is keyed in the SIERRA Framework by the associated index in the regions' array of context bit flags. Each mesh object of a given type belongs to exactly one equivalence-use usage subset for that type; therefore, the context of a mesh object will have exactly one equivalence-use context bit set. Thus the context of a mesh object may be queried for the associated equivalence-use.

The dimension of a field will vary with equivalence-use. Thus the dimension of a field may only be queried with respect to a particular equivalence-use. When accessing a field value associated with a mesh object, the array dimension of the field value (if it is uncertain) is obtained by first determining the equivalence-use of the mesh object and then querying the field for its dimension with respect to that equivalence-use. This two-step process is illustrated with the following pseudocode:

```
iEU = EquivalenceUseIndex of MeshObject.context()
nDim = Field.dimension( iEU )
```

4.4 Declaration of a Master Element Usage

A master element usage is declared in the field registrar of the associated type of mesh object, e.g., *element* master elements are declared in the element registrar, and *face* master elements are declared in the face registrar. Each master element usage is declared with an equivalence-use, a master element, a text label, and a mixed-method context. The mixed-method context is only needed (only useful) in mixed FEM applications that use multiple master elements with different topologies for calculations on the same mesh object. See Figure 4.1 for an example.

Association of Mechanics with Master Element Usage

A given mechanics object uses zero or one master element family in its calculations. If a particular physics is to be applied to different types of elements (e.g., hexahedrals and tetrahedrals), different mechanics objects (possibly of the same type) are created for each element type. Such a polymorphic set of same-physics mechanics can and should share a majority of their code. Design details for mechanics types appear in Section 9.

Recall that each mechanics that is directly nested within a region is associated with a unique entry in that region's context (Section 3). In addition, each mechanics has its own context object that is used to describe usage supersets to which the mechanics' usage subset belongs (Section 9). The equivalence-use that is associated with a master element family defines such a usage superset; thus the mechanics' own context object should include the equivalence-use context.

Multiple Master Element Families

Each declaration of a master element usage in its registrar must be associated with an equivalence-use. Each master element usage declared with the same equivalence-use becomes a member of the same master element family. Applications with heterogeneous elements will typically have a master element family for each type of element. When multiple master element families are present, the same set of text labels should be used to identify analogous members of each family.

Consider a mesh that contains hexahedral elements, tetrahedral elements, and pyramid-transition elements. The same physics is applied to each of the element types; however, different master elements are needed by the mechanics that implements the physics. In this example let the mechanics use a mixed method such that both quadratic and linear master elements are used (e.g., quadratic fluid velocities and linear pressures for Stokes). A total of six master elements divided among three master element families are used. Each master element family has two members, where the linear member is given the text label "Low-Order" and the quadratic member is given the text label "High-Order." Thus a single mechanics algorithm could be written so that it is polymorphic with respect to the master element family—the algorithm will work with any master element family that has declared two members with these labels.

Master Element Mixed-Method Context

If all master elements in a given master element family have the same topology (e.g., same number of nodes), the mixed-method context is not needed and would introduce unnecessary overhead. If the master element topologies are different, the mixed-method context associated with each master element usage will have a critical role in differentiating between the usage of nodes attached to the element. Details of this role are given in the two sections that follow.

If mixed-method contexts are used, it is recommended that they be reused among master element families in the same manner that text labels are reused.

4.5 Querying Master Element Usage

The set of master element usages has two levels of partitioning. The first level corresponds to the type of mesh object, e.g., element, face, edge, or node. Master element usages for a given mesh object type reside in the registrar of the corresponding type. The second level of partitioning corresponds to the master element families and the associated equivalence-use. Thus a given master element usage resides in a particular master element family that resides in a particular registrar.

A master element usage may be queried from the following information:

- The type of mesh object—so that the correct registrar is selected for the query.
- The equivalence-use—so that the correct master element family is selected within the registrar. An equivalence-use may be known in advance or may be obtained from the context of a mesh object, subroster, or mechanics.
- Selection of a particular member of the master element family. If the master element family has only one member, no additional information is needed. Otherwise, a specific member may be queried via its text label, context, or association with a field.

4.6 Context for Master Element Usage

Every master element usage is associated with an equivalence-use. This equivalence-use is associated with a unique entry in a region's set of context flags (Section 4.3). This equivalence-use context is used to differentiate between master element families and thus has a completely separate role from the mixed-method context. The role of the mixed-method context of a master element usage is to differentiate how nodes are used by a mixed method with multiple, topologically different master elements. If differentiation is not needed, the context parameter of master element usage is irrelevant and should not be used.

Each master element usage has an associated mixed-method context that is given during its registration. This context should have two entries set: (1) the entry associated with the mechanics that is using the master element and (2) a unique entry associated with only that master element usage. The second entry defines a usage subset that is unique for the combination of the mechanics and one of its master elements. For example, let some mechanics with two master element usages have an equivalence-use context entry E, mechanics context entry M, and two mixed-method context entries of A and B, as illustrated in Figure 4.2.

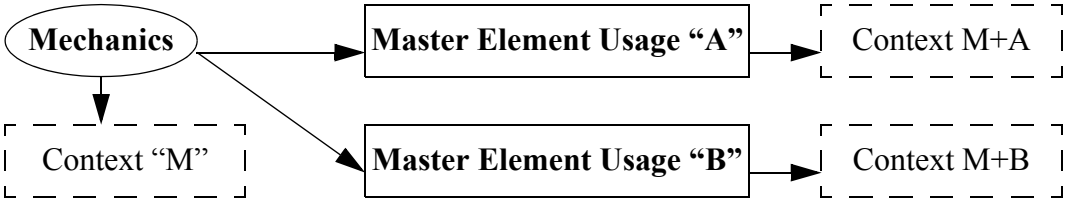


Figure 4.2. Example contexts for multiple master element usages.

Each element operated on by a mechanics is a member of the usage subset associated with the mechanics. As such, each element in this usage subset has the context entry M set to true. Because the nodes, edges, and faces attached to an element are also members of this usage subset, they also have context entry M set to true. At this point in our example there is no usage subset defined to differentiate between the nodes of the element.

The equivalence-use context E is deliberately absent in this illustration. The usage subset defined by an equivalence-use will include, and may be a superset of, the mechanics’ usage subset. Therefore, the usage subset defined by an equivalence-use cannot play a role in the differentiation of nodes that are already within the mechanics’ usage subset.

Imprinting Contexts for Master Element Usages

Continuing with the example, let the element be an eight-node quadrilateral as in Figure 4.3. The mechanics declares a master element usage “A” for the low-order element and a master element usage “B” for the higher-order element. All eight nodes of this element will have context entry “M” set to show that they are a member of the mechanics’ usage subset. Since the higher-order master element of usage “B” touches all nodes, all eight nodes will also have context entry “B” set. However, since the lower-order master element only touches the vertex nodes, only the vertex nodes have context entry “A” set. The final result in this example is that the vertex nodes have context M+A+B and the mid-edge nodes have context M+B.

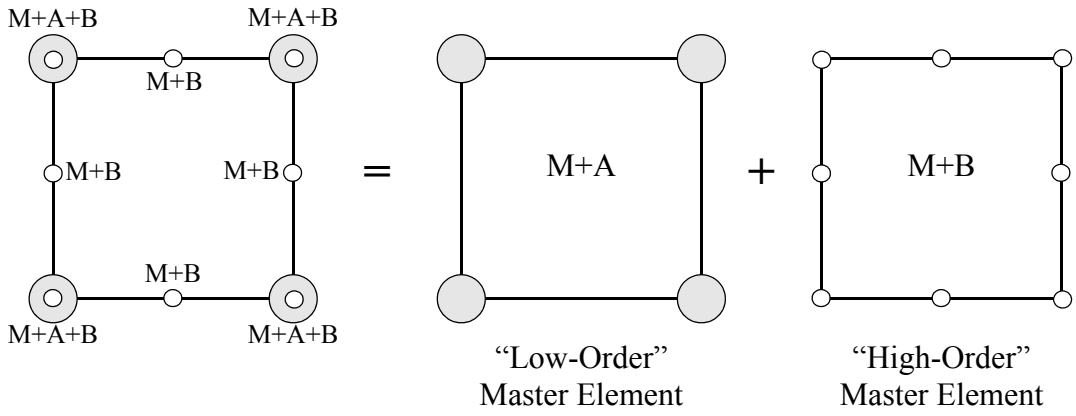


Figure 4.3. Imprinting contexts for master element usages.

Note that if the two master elements in this example were to have the same topology, or if there was only one master element, then the mixed-method contexts A and B would serve no purpose. Also note that in this particular example all nodes are imprinted with both context M and context B. Therefore, an implementation of this example could omit the extraneous context B.

Significance for Field Values

Recall that the context that is set on a mesh object, the element's nodes in this example, determines which field values will exist on that node (Section 3.5). A nodal field that has a context of M or B will have a field value associated with every node of this element. However, a field that has a context of A, but not M or B, will only have field values associated with the vertex nodes of this element. Thus a mixed FEM can declare nodal fields to exist only on specified nodes of an element.

4.7 Conditions for Simplifications

The master element usage example of the previous section identified four context entries: (1) the equivalence-use context E, (2) the mechanics context M, (3) the low-order master element usage context A, and (4) the high-order master element usage context B. In this example, the role of each context was described for an application using a mixed FEM. Simplifications that reduce the number of contexts may be made under certain conditions.

Omitting the Mixed-Method Context

If a master element family, i.e., the set of master elements applied to a given mesh object, consists of a single element, there are no subsets of nodes to differentiate. This condition is also present when all master elements in a family have the same topology. Under this condition a mixed-method context (A or B in the example) is useless and should be omitted. If under this condition a mixed-method context is present, the context will add unnecessary overhead to the application.

Reusing Mechanics Context for Equivalence-Use Context

An equivalence-use context is used to define a partitioning of mesh objects according to master element families. If some subset of mechanics defines a compatible partitioning, each mechanics context in this subset can be reused for an equivalence-use context. In the previous example, the mechanics context M could then be used for equivalence-use context E.

This partitioning condition typically occurs for the subset of mechanics that implement the primary physics of a region. This same subset of mechanics is applied to all of the region's elements and has been referred to as the *element mechanics* of a region. For example, a SIERRA-based explicit-dynamics application (presto) and quasi-statics application (adagio) have such a subset of mechanics that compute the internal forces for every element of the mesh. The element mechanics' context in these applications is reused for the equivalence-use context.

5 Fields and Field Registrars

Recall that a field is simply any variable that is used by one or more mechanics in a region and has a value associated with each mesh object used by any of those mechanics. The set of field specifications for a region is partitioned according to mesh object type, e.g., element fields, face fields, edge fields, and node fields. These type-specific subsets of field specifications are maintained within a region's associated field registrar. Note that this partitioning prevents a particular field from being associated with two different mesh object types, i.e., a particular field cannot have values for both nodes and elements.

A field specification may be dependent upon the set of master elements within a region, i.e., the dimension of a field may be based upon some master element property. Therefore, a region's set of master element usages is also maintained in the field registrars so that the coupling of field specifications to master elements may be more efficiently implemented.

Master element usages and field specifications are declared at run time when a region's set of mechanics is created and configured (Section 1.2). All declarations for master element usages and fields must be completed before a region's mesh is populated. This completion-before-use strategy allows the SIERRA Framework core services to "compile" the set of master element usages and field specifications into efficient internal data structures. These internal data structures are used to assign mesh objects to usage subsets (including equivalence-use) and efficiently manage memory for the field value relation (Section 2.5).

5.1 Field Specification

Each field specification includes the items below. The purpose and use of each item is described in the remainder of this section.

- association with a mesh object type (i.e., a member of which registrar)
- text name (e.g., "coordinates," "temperature")
- data type (e.g., integer, real, vector-3D, tensor-symmetric-3D, aggregate)
- association with one or more mechanics that use the field
- usage of the field with respect to time-stepping states (e.g., constant, independent, dependent, temporary)
- [optionally] association of the field with a master element (e.g., a single value, a value for each connected node, a value per quadrature point)

5.2 Declaring Fields (a.k.a. Field Registration)

A complete set of field specifications must be given when the field is declared. The association of the field with a particular mesh object type is defined by the registrar that holds the field

specification, e.g., a nodal field is declared in the nodal registrar. When a field is declared in a registrar, there are two possible results: (1) a new field specification is created or (2) an existing field specification is modified. The second result is a redeclaration or reregistration of the field. The implications of this redeclaration are described in Section 5.3.

Declaration of a "Simple" Field (Without a Master Element)

A declaration of a field requires (1) a text name for the field, (2) a data type, (3) the *state-use*, and (4) a context that associates the field with a mechanics. These are the minimum four pieces of information required to define a field. Declaration of a field that is associated with a master element is more complex and is described in the next subsection.

The text name of a field provides a means to label the field for mesh data input/output, query the field within the application, and share fields among two or more mechanics. Any field declaration that is associated with the same mesh object type and has the same text name is the same field. If several mechanics declare the same field, that field will be shared by those mechanics.

The data type for a field may be a simple numeric type; however, a data type could be an array or aggregate of other data types. The set of available data types includes basic numeric types, arrays of numeric types that are frequently used by mechanics (e.g., 3D vectors, tensors), and an extensible set of application-defined array or aggregate types (Section 5.4).

The state-use has two purposes. First, it defines the number of states for which the field is to have values. Second, it informs the SIERRA Framework of any special handling required by the field's values (Section 5.5).

The primary purpose for the context of a field is to determine for which mesh objects values for that field must exist (Section 3.5). Recall that a field value will exist for each mesh object with a context that has an entry in common with the field's context. For simple applications that do not use mixed FEMs, this context will be the context defined for the mechanics.

Each declaration of the same field, i.e., the declaration associated with the same mesh object type and with the same name, must have the same data type and state-use. However, the context for each declaration may be different. The different contexts obtained from each declaration are merged for the field. This redeclaration-merging operation (Section 5.3) will vary depending upon the set of equivalence-uses associated with the field's registrar.

Declaration of a Field with a Master Element

A field may be declared with a master element association. This association has two components: a master element from one of the field registrar's master element families and an association with a property of that master element. Each of these additional parameters has significant implications for the field and its values.

The specification for a master element family is an equivalence-use, and an equivalence-use is implemented as a context; therefore, the context of the field declaration must include the equivalence-use context to identify the desired master element family.

The master element properties associated with the field define a dimension for the values of the field. These properties include the master element's number of nodes, number of edges, number of faces, number of internal degrees of freedom, and cardinality of the integration rule. If a field is associated with the nodes of the master element, the field will have a value for each node of the master element. If a field is associated with the integration rule of the master element, the field will have a value for each term in the integration rule. (Note that in previous SIERRA Framework design documentation the terms of a master element's integration rule have also been referred to as the integration stations of the master element.)

All redeclarations of a field must have the same data type, state-use, and association with a master element property. A redeclaration can be associated with a different master element family and thus must have a different master element. If the redeclaration is with the same master element family, the same master element must also be used. A redeclaration with a different master element family has a different equivalence-use. The dimension of the field is allowed to vary with the equivalence-use; therefore, the property associated with the different master element may may have a different number.

5.3 Field Redeclaration (a.k.a. Reregistration)

The same field may be reregistered if and only if the name, data type, and use with respect to time stepping are the same. A field may be reregistered any number of times for two purposes: (1) to associate the same field with a different mechanics and (2) to associate the same field with a different master element. Reregistration with a different mechanics association indicates that the field is shared among the mechanics. Reregistration with a different master element may be used to define a different master element–dependent dimension for that field. For example, register a field for face mesh objects that are quadrilaterals and then reregister that same field for triangles.

When a field is reregistered with a different master element, that master element must also be from a different master element family. This constraint is imposed for compatibility between field dimensions and master element families. This compatibility is as follows:

- The dimension of a field may only vary with the equivalence-use.
- Reregistration of a field with a different master element may define a different dimension for the field.
- Each master element family is uniquely associated with an equivalence-use.
- Thus reregistration of a field with a different master element must also be with a different master element family to ensure a different equivalence-use.

The second or subsequent declaration of a field with the same mesh object type and text name constitutes a redeclaration of that field. A field redeclaration must have the same data type, state-use, master element, and association with a master element property. A field redeclaration must also have the same master element for a given master element family. A field redeclaration may have a different context and a different master element family. The master element family is specified by the equivalence-use that appears in the context.

Same Equivalence-Use (Same Master Element Family)

If the equivalence-use that appears in the context is the same, the master element must also be the same. The remainder of the context may be different. This type of redeclaration is handled by merging the field's context with the redeclaration context. Given that a context is simply an array of bits, this merge is a simple bit-wise union operation in C++:

```
Field.context() |= declaration_context ;
```

Note that the effect of merging the context is that the subset of mesh objects for which the field will have values is enlarged to include the usage subset associated with the declaration context. Because the declaration context is typically associated with a mechanics, redeclaration enlarges the scope of the field to include a new mechanics. In the end, the field is shared by all mechanics that declare it.

Different Equivalence-Use (Different Master Element Family)

Field properties such as the dimension and associated master element may vary with the equivalence-use. The equivalence-use of a field declaration is determined by which equivalence-use appears in the context. A given field declaration (or redeclaration) may have only one equivalence-use set in its context. This equivalence-use is used as a qualifier for the variable part of a field's properties: the dimension and the master element.

Recall that the set of equivalence-uses defines a partitioning for mesh objects, so a given mesh object will be associated with exactly one equivalence-use. If a mesh object–field pair has a field value, the master element and dimension of the field for that field value are well defined. **This variability allows a field to be polymorphic with respect to the kind of mesh object with which it is paired.** For example, the dimension of a field associated with the nodes of a master element would be eight for a linear hexahedral mesh object and four for a linear tetrahedral mesh object; however, to the SIERRA Framework this is the same field specification.

Three attributes of the field vary with the equivalence-use: (1) the associated master element, 2) the **`dimUse`** dimension, and 3) the context. An equivalence-use defines a partitioning of the associated set of mesh objects, so the context of a field defines a nested subset of mesh objects within the equivalence-use's partitioning subset. The context of a field varies with equivalence-use, and thus varies with master element family, to allow the mixed-method context of a master element usage (Section 4.6) to be consistently reused among master element families.

5.4 Array and Aggregate Declarations

A field type may be an aggregation of other field types. An example of this aggregation concept appears in the C (and C++) programming languages as a **`struct`**. Each aggregate field type is a set of other named fields. Each of these member fields has a field type, which may in turn be another aggregate. One application of this hierarchical organization of fields is to mimic the hierarchy of mechanics such that a nested mechanics uses similarly nested fields. For example, a material mechanics is nested inside an element mechanics, and specific material fields are nested within a single "material" field of an element. An application may extend the set of data types by

declaring new array or aggregate data types. Declaration of an array data type requires a type for the members of the array and a dimension. An aggregate data type is constructed by incrementally declaring new fields within as members of the data type.

A field registrar is an aggregate data type. Each new field declared within a field registrar becomes a member of that field registrar's aggregate data type. Field registrars are given special handling by the SIERRA Framework in that field values are automatically created for the associated type of mesh objects (e.g., elements, faces, edges, nodes).

Aggregate data types may be created in one of two ways: (1) as a stand-alone data type at a "global" scope and (2) embedded within the scope of another aggregate data type. An aggregate data type may be thought of as a **`struct`** in the C or C++ programming language. A struct may be declared at the global scope or may be declared within the scope of another **`struct`**, as in the illustrative code fragment in Figure 5.1.

```
struct field_registrar {
  double member_a ;
  int    member_b[30] ;
  struct type_x member_x ;
  struct {
    double member_c ;
    double member_d[3] ;
  } member_y[8] ;
};
```

Figure 5.1. Illustration for fields of aggregate data types.

This C or C++ language illustration **does not** represent the organization of storage for field values. This illustration has been given to describe the design for aggregate data types. The mapping from an aggregate data type to the organization of storage for field values is given at the end of this section.

The first two members listed in Figure 5.1, **`member_a`** and **`member_b`**, are simple fields declared within the field registrar. The **`member_x`** is a field of the aggregate data type **`type_x`**, where **`type_x`** was previously declared at the "global" scope. The **`member_y`** is a field of an aggregate data type that is embedded within the **`field_registrar`** aggregate data type. A field that is of an aggregate data type may also be declared with a dimension.

Declaring a Field of an Aggregate Data Type

In Figure 5.1 the **`member_x`** and **`member_y`** fields are of an aggregate data type. These fields, or any other fields of an aggregate data type, do not have a state-use (Section 5.5). The only fields that have a state-use are fields of simple types or arrays of simple types. This allows every "leaf" member of an aggregate-data-type hierarchy to specify its own state-use. Fields of an aggregate data type may be an array. The organization of field-value storage for such fields is described later.

Field Registrar Versus Aggregate Data Type

A field registrar is an aggregate data type; however, there are significant differences.

- Field registrars have master element usages but aggregate data types do not.
- Members of a field registrar may be associated with master elements, but members of any other aggregate data type are not.
- A field registrar cannot be embedded or used within another aggregate data type.
- Field registrars are always recognized by the SIERRA Framework, but aggregate data types are only recognized if they are nested within a field registrar.

Declaration of members within an aggregate data type is similar to declaration of simple fields (Section 5.2) in that a text name, data type, state-partition (for nonaggregate data types), and context are required. The member fields of an aggregate field (in Figure 5.1 **`member_c`** and **`member_d`** are member fields of the aggregate field **`member_y`**) are required to have a context that is compatible with the aggregate field. In the example of Figure 5.1 **`member_c`** and **`member_d`** must have contexts that are contained within the context of **`member_y`**.

Member Fields of an Aggregate Data Type and Equivalence-Use

The member fields of an aggregate data type may be declared for a subset of the equivalence-uses for which their parent field is declared. If this is the case, the nested field will only have a field value for those mesh objects that appear in the usage subset of the equivalence-use. Thus not only can the array dimensions of a field vary with equivalence-use, but the fields nested within an aggregate data type may vary as well. Another way to view this variability is that the array dimension of a nested field may be zero for a given equivalence-use.

5.5 State-Use, Multiple States, and Time Stepping

A field will be used in one of the following four ways with respect to the states of an application's multistate time-stepping algorithm:

State Field

The field has a value for each of N states where the values of the *new* $(N+1)$ state are updated at each time step while the values of the old $(N, N-1, ...)$ states are unchanged. The SIERRA Framework rotates the mapping between the state labels and the storage location at the end of each time step (Section 1.2).

Persistent Field

The field has a value that likely changes every time step.

Constant Field

The value of the field is set during initialization (Section 3) and remains unchanged thereafter.

Temporary Field

The field has a value that is ignored by the SIERRA Framework restart capability.

These state-uses are specified through predefined objects for the temporary, constant, and persistent uses and through an application-defined object for multistate-uses. Each field specification that is one of these state-uses will have a reference to the corresponding predefined object. An application may have multiple regions, each of which has a different time-stepping algorithm. Each region would then define a different state-use object that specifies the number of states required by that region.

(Note that in previous SIERRA Framework documentation this state-use property of a field was referred to as the field's data partition or datum partition.)

Field Values for Multiple States

Time-stepping algorithms typically require values for fields over a sequence of time planes. For example, updating to the next time plane of a field may require values from the previous time plane. The number of time planes required by the time-stepping algorithm defines the *number of states* for that field. If updating to the current time plane requires one previous time plane, the field has two states, a *new* state and an *old* state. When more than one previous time plane is required, the states are labeled as $N + 1$ for the new state to be computed and N, $N-1$, $N-2$, ... for the current and previous states in the sequence. The *new* and $N + 1$ labels are synonymous.

Each field is defined as a state field, persistent field, constant field, or temporary field. The SIERRA Framework services provide values for of the number of states for a state field. During a given time step it is assumed that the $N + 1$ value for these fields are updated, while the previous values (N, $N-1$, $N-2$, ...) are unchanged, as illustrated below.

```
field(N+1) = field(N) + function(field(N),field(N-1))
```

A persistent field is also assumed to be updated at each time step; however, it does not require values from previous states. A persistent field is typically computed from state fields. A constant field is initialized before the first time step and remains constant throughout all time stepping. Constant fields typically hold properties of a model that vary over the set of mesh objects. As such, constant fields are also referred to as model fields. Temporary fields refer to any other fields that an application may require.

Implications for the Restart Capability

A field's usage within an application (i.e. state, persistent, constant, or temporary) is significant for the SIERRA Framework restart capability. Fields that are temporary are ignored by the restart capability. Temporary fields are assumed to have auxiliary values that are used for intermediate calculations internal to a time step or derived quantities for output purposes. It is assumed that successful restart of an application does depend upon these values. Values for constant fields are unchanged after initialization, so the restart capability only outputs these fields once for a given restart file. Persistent and state values are output at each point in the application that a restart image is generated.

State-Field Updates and Rotation of Labels

A time-stepping algorithm typically requires a sequence of field values, where the cardinality of this sequence is referred to as the number of states. For example, a field with two states has a new and an old value, and a field with four states has values labeled *N + 1* or *new*, *N*, *N – 1*, and *N – 2*. Once a time-step update has completed, the "oldest" value in this sequence is no longer needed. Furthermore, the *new* value becomes the *old* value for the next time step, or the *N + 1*, *N*, and *N – 1* values become the *N*, *N – 1*, and *N – 2* values, respectively.

This relabeling of values could be accomplished by copying values from the newer state into the next older state (in oldest to newest order). For a large number of mesh objects and fields, such a copying operation is expensive. A negligible cost alternative to copying values is used by the SIERRA Framework: rotation of the labels for states.

Each state field has *K* values, where *K* is the number of states required for the time-stepping algorithm. These *K* values may be addressed with the integer values [1..*K*]. During a given time step *i*, the state labels are mapped to value addresses as follows:

$$\begin{bmatrix} N+1 \\ N \\ N-1 \\ N-2 \end{bmatrix}_{\mathbf{i}} \rightarrow \begin{bmatrix} 4 \\ 3 \\ 2 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} N+1 \\ N \\ N-1 \\ N-2 \end{bmatrix}_{\mathbf{i+1}} \rightarrow \begin{bmatrix} 1 \\ 4 \\ 3 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} N+1 \\ N \\ N-1 \\ N-2 \end{bmatrix}_{\mathbf{i+2}} \rightarrow \begin{bmatrix} 2 \\ 1 \\ 4 \\ 3 \end{bmatrix}$$

For each subsequent time step this mapping of state labels is rotated such that the value in the previous new state receives the label for the current old state.

5.6 Querying Declared Fields

A field registrar or any aggregate data type may be queried for its member fields. This query may be made by text name or by iteration through the set of all member fields. A field is implemented with a C++ object that contains all of the specifications that were given during the declaration or redeclarations of that field.

Persistence of Field Objects

The first declaration of a field creates a field object that is owned by the containing aggregate data type. Each subsequent declaration modifies the field object but does not invalidate it. Therefore, a reference (C++ pointer) to that field object may be obtained and used by a mechanics. This is the preferred approach to using declared fields.

Querying Properties of Field Objects

A field object may be queried for its text name, data type, dimension, context, state use, and associated master element property (if a direct member of a field registrar). The field properties of text name, data type, state-use, and association with a master element property are invariant with respect to equivalence-use, so queries for these properties may disregard equivalence-use. The

dimension, context, and associated master element vary with equivalence-use, so an equivalence-use must be supplied to query these properties. Recall that an equivalence-use may be obtained from a mesh object or a subroster, or it may already be known to the application.

If the data type of a field is an aggregate, that aggregate may be recursively queried for its member fields. These queries may iterate through the members or may retrieve a member by its text name.

6 Mesh Object Rosters and Subrosters

The SIERRA Framework Version 3 supports unstructured meshes that are defined by an interconnected set of node, edge, face, and element mesh objects (Section 2.4). Other types of arbitrary, nontopological mesh objects, such as contact constraints, are also included in this set. A coupled multiphysics application may have multiple meshes, where each mesh is owned by a region (Figure 1.1). Each region's set of mesh objects is hierarchically partitioned into type-specific *rosters* and then into homogeneous *subrosters*, as illustrated in Figure 6.1.

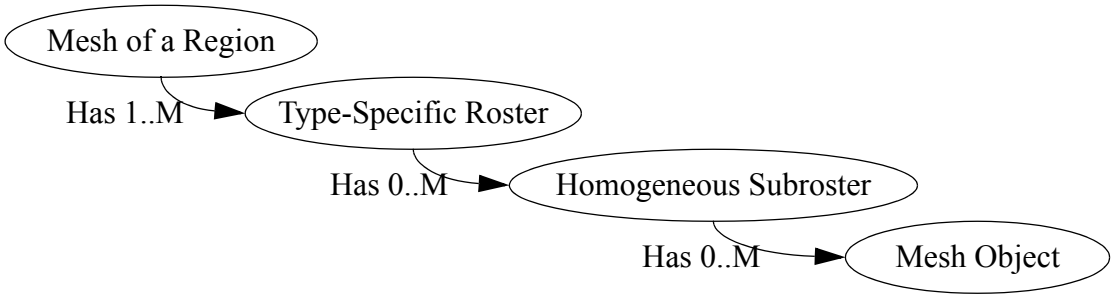


Figure 6.1. Mesh-partitioning hierarchy.

A roster is associated with a subset of mesh objects that are of the same type (e.g., node, edge, face, element, or other). The subset of mesh objects in a given subroster is homogeneous—all mesh objects in a subroster are members of the same usage subsets and instance subsets (Section 3), and are associated with the same master element family or equivalence-use (Section 4).

A roster is responsible for creating, storing, organizing, retrieving, and deleting mesh objects of its particular type. In a parallel-processing environment these capabilities are limited on a given processor to the subset of mesh objects that are resident on that processor. For example, a mesh object that is resident only on processors P1 and P2 cannot be retrieved on processor P0. Roster capabilities can be divided into two categories: mutating and nonmutating. The capabilities that mutate a roster, i.e. change the contents or organization of a roster, are described in Section 12. The organization and nonmutating capabilities of a roster are described in this section.

6.1 Hierarchical Partitioning

The objective for the mesh-partitioning hierarchy illustrated in Figure 6.1 is to provide a balance of execution-time and memory-space efficiency for applications. Design and analysis of this partitioning hierarchy is expressed through the mathematics of set theory.

Theory

A region's set of mesh objects is partitioned into homogeneous subsets where each subset is associated with, or owned by, a unique subroster. This subroster partitioning is defined by generating a set-topology (Section 2.2) on the region's set of mesh objects, as follows. Let **X** be a class of subsets of a mesh where the members of **X** include the roster subsets, usage subsets,

instance subsets, master element family (i.e., equivalence-use) subsets, and parallel distributed mesh subsets (Section 2.4). Generate a set-topology **T** on the mesh with **X**. The subroster subsets of the mesh are members of **T** such that the intersection of a subroster subset with any other member of **T** is either that subroster subset or the null set. Let **S** be the subclass of **T** whose members are the *subroster subsets*.

$$S \in \mathbf{S} \subset \mathbf{T} \Leftrightarrow S \cap T \in \{S, \varnothing\} \ \forall T \in \mathbf{T}$$

A roster is dynamic with respect to (1) the mesh objects it contains and (2) the partitioning of those mesh objects among subrosters. During the execution of an application, new mesh objects may be created, existing mesh objects may be moved between subrosters, and existing mesh objects may be destroyed. However, the set-topology **T** of a mesh is fixed before a mesh is populated (see Section 1.2) and does not change as mesh objects are created, moved, or destroyed. Creation and destruction of mesh objects merely change the contents of the subsets in **T**, but these processes do not change the class of subsets.

Time Efficiency

A given mechanics operates on a selected subset of a mesh; thus any given algorithm is required to select a subset of the mesh. This subset could be selected by interrogating each mesh object for inclusion or exclusion, which would be an $O(N)$ operation where N is the number of mesh objects in the mesh. If the same algorithm selects mesh objects by homogeneous subroster, with the selection result guaranteed to be the same for each member of the subset, the inclusion or exclusion operation is $O(M)$ where M is the cardinality of the subroster class **S**.

Space Efficiency

The association of mesh objects with subsets (e.g., usage subsets, instance subsets, master element family subsets) requires memory space in an implementation. Each member of a subroster of mesh objects has an identical association with subsets. Thus the memory space required to implement these mesh object–subset associations can be shared by all mesh objects in the subroster.

6.2 Mesh Object Properties

All mesh objects share the same implementation in the SIERRA Framework core services, e.g., a node and an element are implemented by the same C++ class and can only be differentiated by their associated data. Mesh object properties have values that are either associated directly with the mesh object or associated with a homogeneous subroster and indirectly with the subroster's mesh objects.

Unique Identifier (Mesh Object)

Each mesh object in a mesh is a member of a set and is therefore required to be uniquely identifiable (Section 2.2). Mesh objects within a mesh are uniquely identified by the type of the

mesh object and an integer value. The type of a mesh object identifies the roster in which the mesh object resides, and an integer value uniquely identifies the mesh object within that roster.

A mesh object's integer identifier is unique only within its roster. For example, a node and an element may both have an identifier of "1" but are uniquely identified within a mesh by a combination of their type and identifier (node-1 versus element-1). The integer identifier is unique and consistent among all processors of a dynamically changing parallel distributed mesh. For example, if a node is shared among several processors, the node has the same integer identifier on each processor.

Processor Ownership (Mesh Object)

A mesh object may be resident on more than one processor in a parallel distributed mesh; however, the mesh object is owned by exactly one processor on which it resides (Section 2.4). This unique processor ownership is known on each processor on which a mesh object is resident. Identification of the unique processor-owner is essential for correct and efficient reduction calculations over field values, e.g., norms and inner products. Knowledge of processor ownership is also essential for the SIERRA Framework to maintain global consistency of the mesh.

Connectivity (Mesh Object)

A mesh object is connected to zero-to-many other mesh objects as defined by the intramesh connectivity relation (Section 2.4). Software design details for mesh object connectivity appear in Section 8.

Field Values (Mesh Object)

Each mesh object is associated with zero to many field values as defined by the field value relation (Section 2.5). Each field value may be accessed through the pairing of a mesh object with a field. Software design details for field values appears in Section 7.

Usage Subsets and Context (Subroster)

A mesh object is a member of one to many usage subsets of a region. Most usage subsets are defined by mechanics. Other sources of usage subsets include equivalence-uses (master element families) and those that support distributed and dynamic mesh capabilities (Section 3.6). Membership in some of the dynamic-mesh usage subsets is mutually exclusive, as follows:

- The active and inactive subsets are mutually exclusive.
- The pending-create and pending-delete subsets are mutually exclusive. However, most mesh objects are not members of either of these subsets.

Each mesh object has an associated context that identifies the set of usage subsets for which the mesh object is a member (Section 3). The value of the context is identical for all mesh objects in a given subroster. Therefore, the context of a mesh object resides with the subroster and is shared by all mesh objects in that subroster.

Instance Subsets (Subroster)

The predominant number of subsets in the generator for the mesh topology consists of instance subsets. A mesh object is a member of zero to many instance subsets; however, a mesh object that does belong to any instance subset is unlikely ever to be used within an application. Each mesh object has an associated list of mechanics instances that identifies the instance subsets for which that mesh object is a member. This list is identical for all mesh objects in a given subroster; therefore, the list resides with the subroster and is shared by all mesh objects in that subroster.

Topology of the Master Element Family (Subroster)

The finite element topology of a mesh object is a frequently used property of a family of master elements (Section 4.1). This topology describes the expected connectivity for the mesh object (Section 8). This property could be obtained by first querying the context of the mesh object, then querying the master element family associated with that context, and finally obtaining the richest finite element topology of that master element family. Because of its frequent use, the finite element topology is included among the directly maintained mesh object properties. However, these properties are identical for all mesh objects in a given subroster, so the finite element topology resides with the subroster.

Summary of Mesh Object and Subroster Properties

The mesh object properties that are associated with individual mesh objects or shared by the mesh objects in a given subroster are summarized in Table 6.1.

Table 6.1. Mesh Object and Subroster Properties

Mesh Object Property	Subroster Property (Shared by Mesh Objects)
Unique Integer Identifier	Usage Subsets/Context
Owning Processor	List of Instance Subsets
Connectivity to Mesh Objects	Finite Element Topology
Field Values	

6.3 Iterating and Querying Rosters

A roster is a container for mesh objects of a given type. The ISO/ANSI C++ language standard defines a *container* as an object that stores other objects and controls the allocation and deallocation of those objects. An *iterator* is an object that is used to iterate over the members of a container. The mesh objects within a roster may be iterated or queried by their unique identifiers.

Parallel Distributed Mesh

A roster on a particular processor is a container for mesh objects that are resident on that processor. Iteration of the contents of a roster on a given processor will only access mesh objects that are resident on that processor. Furthermore, queries for mesh objects in a roster will only be successful on the processors for which the queried mesh object is resident. The SIERRA Framework will not automatically retrieve mesh objects among remote processors in response to iteration or queries of a roster.

Two-Level Iteration by Subroster

The mesh objects in a roster may be iterated directly or iterated by subroster. Iteration by subroster is a two-level looping process, where the outer loop is over the subrosters of the roster and the inner loop is over mesh objects in the current subroster. Recall that subrosters define a partitioning for the mesh objects in a roster. As such, iteration by subroster will only access each mesh object in a roster once.

The two-level iteration allows a mechanics calculation to apply its mesh-object selection code to select subrosters as opposed to individual mesh objects. This approach reduces the total number of operations performed to select the mechanics' appropriate subset of mesh objects from a roster. Thus it is recommended that a mechanics iterate mesh objects by subroster.

The code for selecting mesh objects may be a complex boolean expression involving several usage or instance subsets. For example, a calculation may select all mesh objects that are in the intersection of mechanics A and B usage subsets, in the locally owned usage subset, and not in the pending-create or pending-delete usage subsets.

```
foreach Subroster in Roster
  if  Subroster.context has Mechanics-A and
      Subroster.context has Mechanics-B and
      Subroster.context has Locally-Owned and
      Subroster.context does not have Pending-Create and
      Subroster.context does not have Pending-Delete then
    foreach MeshObject in Subroster
      perform computations on MeshObject
    end
  endif
end
```

Ordering of Subrosters and Mesh Objects

The iteration order of subrosters in a roster and mesh objects in a subroster is dynamic and arbitrary, but also deterministic. Ordering is dynamic in that any modifications to contents of the roster (creating, shifting between subrosters, or deleting) can and will reorder the mesh objects within the affected subrosters. Ordering is arbitrary in that mesh objects within a subroster are ordered and reordered to maximize the efficiency during iteration and update operations. An application should not attempt to anticipate the ordering of mesh objects within a given subroster. Ordering is deterministic in that an application may rely on the ordering of mesh objects to be the same given the same sequence of operations that is used to initially define the mesh set-topology

and then modify the mesh. Recall that the mesh set-topology is determined by the usage subsets and instance subsets.

The order of mesh objects within a subroster is arbitrary and subject to change if the roster is modified in any way. Arbitrary ordering allows efficient modification, creation, and deletion of mesh objects. However, reordering can and will cause a reduction operation to have numerical differences on the order of floating-point precision, as illustrated below. This numerical round-off is due to the reordering of summation terms in the reduction operation.

$$\left| \left(\sum_{i=1 \rightarrow n} a_i^2 \right) - \left(\sum_{j=n \rightarrow 1} a_j^2 \right) \right| < \varepsilon \mathbf{O}(n)$$

Querying by Unique Identifier

A roster may be queried for a mesh object that has a specified, unique integer identifier. If a mesh object with that identifier resides on the local processor, a valid iterator to that mesh object is returned from the query. If no such mesh object resides in the local processor's roster, an invalid iterator is returned. The result of this query is processor dependent.

Iteration by Unique Identifier

Mesh objects in a roster may be iterated in the order of the mesh objects' unique integer identifiers; however, it is strongly recommended that mechanics do not use this iteration technique. Iteration by identifier has no correlation with usage subsets or instance subsets. If a mechanics algorithm performs this iteration, each mesh object would have to be evaluated for its inclusion or exclusion in a mechanics' appropriate subset.

6.4 Manufactured Mesh Objects, Usage Subsets, and Connections

An initial set of mesh objects is read from a bulk mesh data input file into the region as described in Section 1.2. Mesh objects are assigned to subsets, connected to other mesh objects, and have field values allocated as they are read. Some applications require additional mesh objects, subsets, or connections that did not appear in the bulk mesh data input file. The SIERRA Framework provides the capability to manufacture the following frequently used additional subsets and connections:

1. faces and edges on the boundaries of blocks of elements
2. faces and edges in the interior of blocks of elements
3. subsets for exposed boundaries and interblock boundaries
4. the node-to-element relation which is the converse of the element-to-node relation
5. "ghost" elements on parallel domain decomposition boundaries

Manufactured mesh objects are created in the same rosters and subrosters, and will have have the same field-values capabilities, as mesh objects that are loaded from an input bulk mesh data file.

Mesh-manufacturing capabilities are requested through a set of flags on the region that must be set before the bulk mesh data file is read. These flags are typically set during the user-input parsing phase of an application (Section 1.2). The region's mesh-manufacturing flags and the results from setting them are given in Table 6.2.

Table 6.2. Region's Mesh-Manufacturing Flags

Region's Flags	Result If "True"
use boundary faces	manufacture faces on boundaries and generate exposed and interblock boundary subsets
use all faces	manufacture faces of all elements
use all edges	manufacture edges of all elements
use face edges	manufacture edges of all faces
support dynamic mesh	manufacture node-to-element relations

The exposed and interblock boundary subsets are two different usage subsets with context values defined by the SIERRA Framework. A face is in the exposed boundary subset if it has only one element connected to it in the global mesh. Note that a face may have only one element connected on a particular processor's subdomain; however, if it is also connected to a different element on another processor, the face is not exposed in the global mesh. A face that is connected to two elements from different blocks in the global mesh is in the interblock boundary subset. In addition, all edges and nodes belong to the same boundary subsets as any of their connected faces.

7 Field Value Relation and Buckets

Recall from Section 2.5 that the field value relation is a set-relation with a domain in the pairing of mesh objects and fields and a range in the value associated with that pairing. Recall also from Section 2.5 that a mesh object–field pair is a member of this relation if and only if there exists a mechanics that uses both the mesh object and the field. Finally, recall from Section 3.5 that this existence condition can be determined by comparing the context of the mesh object with the context of the field.

$$\textbf{FieldValueRelation} \subset (\textbf{Mesh} \times \textbf{Field}) \times \textbf{FieldValues}$$

$$((\textbf{MeshObject}_j, \textbf{Field}_j)\ , \textbf{FieldValue}) \in \textbf{FieldValueRelation}$$

$$\textbf{if-and-only-if}$$

$$\exists\ i\ \textbf{:}\ (\textbf{context}(\textbf{MeshObject}_j)[i] = \textbf{true})\ \textbf{and}\ (\textbf{context}(\textbf{Field}_k)[i] = \textbf{true})$$

7.1 Buckets for Efficiency

An implementation of the field value relation requires storage for the field values and for the association of mesh object–field pairs with those field values. Not all mesh object–field pairs have field values, so the implementation should not consume storage for unused field values. However, this selective existence of field values introduces both execution-time and storage overhead—conceptually similar to the overhead introduced by indexing arrays in numerous sparse matrix storage formats.

All mesh objects in a subroster have the same context; therefore, they have the same master element family, same associated field values, and same dimension for those field values (Section 4.2). Field values associated with a given subroster of mesh objects may be grouped into a contiguous block of storage referred to as a *bucket*. Thus the storage overhead is reduced from an O(N), per-mesh-object expense, to an O(M), per-subroster expense—conceptually similar to the reduced overhead of some block-sparse matrix storage formats. The execution-time overhead is likewise reduced **if** an application iterates subrosters (Section 6.3) and performs its calculations on field values per bucket as opposed to per mesh object.

Issues with Dynamic Mesh Modifications

An application may change a mesh so that mesh objects are added to or removed from a subroster. Changes to the membership of a subroster require the set of field values associated with the subroster's bucket to increase or decrease. This can lead to wasted time if a bucket's memory is frequently increased or wasted space if a bucket's memory is not reduced. The problem of wasted time and space is mitigated by allocating buckets of a fixed size and having a subroster use multiple buckets for the associated field values. An additional optimization is made by correlating the bucket size with the host systems' memory page and cache size.

Two-Level Iteration by Subroster and Bucket

The two-level iteration approach for mesh objects in a roster should be modified to iterate subrosters in the outer loop (as before) and buckets in the inner loop. Each bucket contains the field values associated with a subset of mesh objects residing in the subroster. These field values are contiguous arrays (Section 7.2) that may be passed to FORTRAN subroutines for highly optimized numerical calculations.

```
foreach Subroster in Roster
  if
Subroster is selected
    foreach Bucket in Subroster
      perform computations on Bucket
    end
  endif
end
```

In addition to arrays of field values, a bucket holds an array of C++ pointers to the mesh objects that are associated with the bucket. This array is needed by calculations that use mesh object properties that are not field values, e.g., the integer identifier or processor-owner of a mesh object.

7.2 Dimensions of Field Value Arrays

Field values are maintained in buckets as arrays. The dimension of a simple field-value array, using FORTRAN convention, is **`field(dimType,dimUse,numObjects)`**. The leading **`dimType`** dimension is the length of the array associated with the field's data type. If a data type is a simple scalar, this value is one. The trailing **`numObjects`** dimension is the number of mesh objects associated with the bucket. The **`dimUse`** value is obtained from the master element and specified master element property (Section 4.2). For example, if the master element has a $2 \times 2 \times 2$ Gauss quadrature integration rule and the associated master element property is the integration rule, use **`dimUse = 8`**.

Variant Dimension

This interior **`dimUse`** dimension may vary with the master element family associated with the homogeneous subroster of mesh objects. This dimension may be known to the mechanics *a priori*, e.g., the mechanics supports a particular master element family. If the field's association with a master element is known to the mechanics, the dimension may be determined by querying the master element (Section 4). This would be the case for a typical mechanics that is polymorphic with respect to the master element family. Finally, a truly generic calculation may query the field for its dimensions with respect to the equivalence-use of the subroster (Section 5.6).

Dimensions for Field Members of Aggregate Data Types

Recall from Section 5.4 and Figure 5.1 that a field may be a member of an aggregate data type. Aggregate data types may define members that are also aggregate data types with no imposed

limit on the depth of this hierarchy. However, an application developer should exercise some restraint for simplicity and clarity of the aggregate data types.

Each field that is a "leaf" in an aggregate-data-type hierarchy has field values that occupy a contiguous block of memory within a bucket. This contiguous block of memory is organized as a multidimensional array with dimensions defined by its parent fields in the hierarchy. Continuing with the illustration in Figure 5.1, the fields **member_a**, **member_b**, **member_c**, and **member_d** would have the following FORTRAN-organized array dimensions within a bucket:

```
member_a( 1,numObjects)
member_b(30,numObjects)
member_y::member_c(1,8,numObjects)
member_y::member_d(3,8,numObjects)
```

The leading-to-trailing dimensions correspond to beginning with the "leaf" field's dimension and appending a new dimension for each parent in the hierarchy. Each level in the hierarchy appends another dimension until at the field-registrar level the number of objects in the bucket is the final dimension.

This array organization is chosen so that any field value may be passed as an array to a FORTRAN77 computational kernel. Array-based computations, especially when using highly optimized FORTRAN77 subroutines, typically yield the best performance for an application.

7.3 Accessing Field Values

Field values may be accessed either directly from a bucket or indirectly through a mesh object. Indirect access through a mesh object will (behind the scenes) retrieve the bucket, access the field value array, and compute the offset into the array that corresponds to the mesh object. If field values are accessed from mesh objects for each mesh object in a subroster, this overhead is incurred for each mesh object. It is more efficient, and once again recommended, that a calculation directly access field values from buckets as opposed to indirectly accessing field values from mesh objects.

If a field has a state-use of temporary, persistent, or constant, the field value may be accessed from the pairing of a bucket or mesh object with the field. The access operation returns a C++ pointer to the storage of the field value, as in the following example. If the C++ pointer is for a specific data type (i.e., integer or floating point), type checking is performed.

```
Int * ival = bucket.data( iField );
Real * rval = mesh_object.data( rField );
```

If a field has multiple states, access to the field values requires selection of which state is to be accessed. This state specification is mapped to a particular memory location, where the mapping is updated at each time step (Section 5.5).

```
Int * ival = bucket.data( iField , STATE_NEW );
Real * rval = mesh_object.data( rField , STATE_OLD );
```

Volatility of Field Value Pointers

The mapping from states to storage location changes every time step. For this reason, all C++ pointers to field values of multistate fields should be treated as volatile between time steps. Furthermore, if the mesh is modified in any way, e.g., load balancing, element death, h-adaptivity, or simply changing a single mesh object's attributes, the storage location of any field value may change. As such, any C++ pointer to a field value should be treated as volatile with respect to such mesh modifications.

8 Mesh Object Connectivity

A mesh is a set of interconnected mesh objects. The interconnections among mesh objects in a given mesh is expressed by the intramesh connectivity relation (Section 2.4) associated with the mesh. This set-relation contains pairs of mesh objects from the same mesh. Each pair represents a connection between the two mesh objects, for example, the connection between an element and one of its nodes.

$$\mathbf{ConnectivityRelation}_{\alpha} \subset \mathbf{Mesh}_{\alpha} \times \mathbf{Mesh}_{\alpha}$$

8.1 Partitions of Mesh Object Connections

Several frequently used partitions are defined for the intramesh connectivity relation.

Domain Mesh Object Partition

The intramesh connectivity relation is partitioned into **N** subsets, where **N** is the number of mesh objects in the mesh. Each subset contains members that have a given mesh object as the domain coordinate. This partition is used to group members of the relation by domain mesh objects so that, given a domain mesh object, access to members is efficient.

$$\mathbf{m}_i , \mathbf{m}_j \in \mathbf{Mesh}_{\alpha}$$

$$\left\{ \{ (\mathbf{m}_i,\mathbf{m}_j) : (\mathbf{m}_i,\mathbf{m}_j) \in \mathbf{ConnectivityRelation}_{\alpha} , \forall j \}_i \right\}$$

Connection Type Partition

The intramesh connectivity relation is partitioned into five subsets that are associated with the type or purpose of the connection. The five subsets, introduced in Section 2.4, are summarized as follows:

1. The domain mesh object ***uses*** the range mesh object, e.g., an element uses a node. The domain mesh object may depend upon the existence of the range mesh object.
2. The domain mesh object is ***used-by*** the range mesh object. The domain mesh object typically exists to support the range mesh object.
3. The domain mesh object has a ***child*** which is the range mesh object.
4. The domain mesh object has a ***parent*** which is the range mesh object. The parent and child connections support hierarchical partitioning of mesh objects under h-adaptivity.
5. The domain mesh object has an ***auxiliary*** connection to the range mesh object. An auxiliary connection is defined when a uses, used-by, child, or parent connection is not applicable.

Range Mesh Object Type Partition

A final partition is defined based upon the mesh object type of the range mesh object. This final partitioning enables queries such as “given an element, generate the subset of used nodes.” This example query begins with a selection of a domain mesh object, “given an element.” The next clause in the query is the type of connection, “used.” The final clause is the type of range mesh object, “nodes.” This common query is for a subset defined by the intersection of a selected member from each of the three partitions.

8.2 Ordinal and Orientation

The intersection of a member from the domain mesh object partition, connection type partition, and range mesh object type partition defines a subset of mesh object connections. All members of such a subset have the same domain mesh object and several different range mesh objects. The objective of such a query operation is to obtain the range mesh objects from the resulting subset, for example, to query the set of nodes connected to a given element. However, in many situations the existence of the subset is not sufficient—a complete ordering of members is also required.

Requirement for Ordering

Connectivity between finite element mesh objects conforms to some element topology. For example, a hexahedral can be defined by an ordered set of eight vertices, and if the vertices are not ordered, then the hexahedral is ill defined. Each subset of mesh object relations that corresponds to the intersection of a member from the domain mesh object partition, a “uses”’ or “child” member from the connection type partition, and a member of the range mesh object type partition is given a complete ordering. This ordering corresponds to the ordering of connections defined by a mesh object topology.

Mesh Object Topology

Each mesh object may be given a topology that defines that mesh object’s required or expected connectivity to other mesh objects. The expected connectivity defines a template for the subset of range mesh objects that a domain mesh object uses. For example, a hexahedral mesh object may use eight nodes, twelve edges, and six faces. Calculations that involve connected mesh objects are more efficient if the members of such subsets appear in a known order. This order is defined by a mesh object topology.

A mesh object topology defines a template (or stencil) of required or expected “uses” connections between mesh objects. Furthermore, topological information includes ordering interdependencies for multiple mesh object connections. For example, the topology of a triangular mesh object requires connections to three vertices $\{v0, v1, v2\}$ and three edges $\{e0, e1, e2\}$. These subsets are interdependent such that each edge must use a particular pair of vertices, e.g., $e0 \leftrightarrow (v0,v1)$.

Orientation

Mesh objects are interconnected such that several different mesh objects may share the use of another mesh object. For example, the two triangles in Figure 8.1 share two nodes and one edge. In this example the two ordered subsets of triangle-uses-node connections are as follows:

$$\{(\mathbf{T1},\mathbf{N1})_0, (\mathbf{T1},\mathbf{N2})_1, (\mathbf{T1},\mathbf{N3})_2\} \textbf{ and } \{(\mathbf{T2},\mathbf{N4})_0, (\mathbf{T2},\mathbf{N3})_1, (\mathbf{T2},\mathbf{N2})_2\}$$

Each triangle also uses edge **E1** that has the ordered subset of edge-uses-node connections $\{(\mathbf{E1},\mathbf{N2})_0, (\mathbf{E1},\mathbf{N3})_1\}$. Each triangle expects to have an edge defined by vertex nodes **N2** and **N3**. However, triangle **T1** expects the 0^{th} node of this edge to be **N2**, and triangle **T2** expects the 0^{th} node of this edge to be **N3**. Thus the orientation of edge **E1** is compatible with the expectations of triangle **T1** and reversed with respect to the expectations of triangle **T2**.

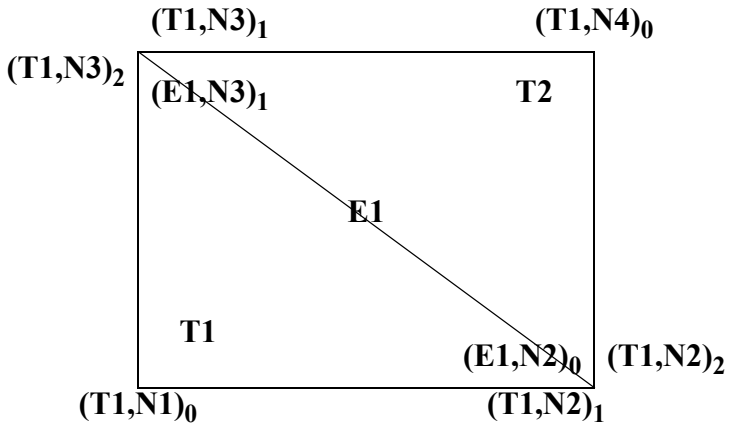


Figure 8.1. Example ordering of connection relations.

Each member of a particular domain mesh object's "uses" connection-relation subset has an orientation. This orientation indicates whether the range mesh object is oriented as the domain mesh objects expect or, if not, then how the range mesh object is actually oriented. Orientation of edges is simple: either it is expected or the reverse of what is expected. Orientation of faces on elements is more complex in that the face may be reversed and rotated with respect to the expected orientation.

8.3 Data Structure

Each member of the intramesh connectivity relation is an object that is owned by the member's domain mesh object, as illustrated in Figure 8.2. Thus a mesh object is a container for members of the connectivity relation that have that mesh object as their domain coordinate. This allows efficient access to subsets defined by fixing the domain mesh object, for example, accessing the nodes that an element uses.

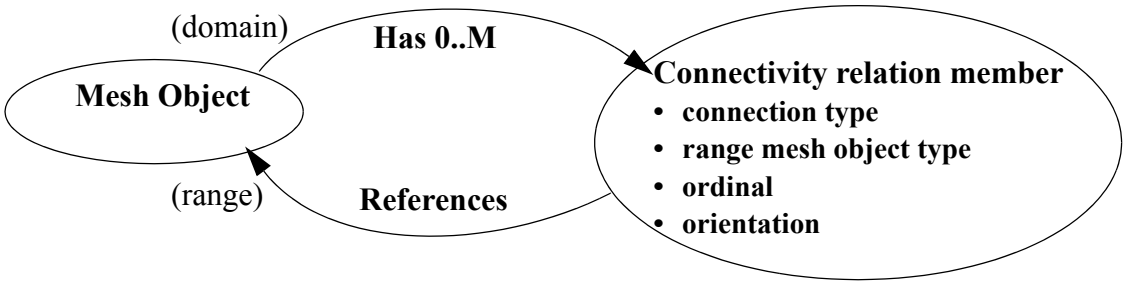


Figure 8.2. Connectivity relation objects.

Connectivity relation objects have attributes that denote their membership in the connectivity type partition, membership in the range mesh object type partition, ordering (ordinal with respect to expected), and orientation. The value of the ordinal corresponds to the ordering of the expected connections. For example, the mesh object topology for a hexahedral expects to have six faces. If only one of the six faces explicitly exists in the mesh, the element-to-face connectivity relation object will be assigned an ordinal as if all faces existed. Thus the ordinal would answer the question, Given an element and some face of that element, which face is it?

The value of the orientation attribute describes how a range mesh object is oriented with respect to the domain mesh object's expectation in a uses connection. This attribute could be determined as needed by comparing the nodes of the two mesh objects. Such a comparison operation requires execution time, so the SIERRA Framework determines the orientation once and saves this information in the orientation attribute.

Mesh Object as a Container for Relation Objects

A mesh object is a container for the subset of relation objects that have that mesh object as their domain coordinate. The members of this container may be iterated, searched, or modified by the application. This container capability orders its members first by range mesh object type and then by connection type. If desired, iteration and searching operations may be restricted to a type of range mesh object and also to a connection type.

Relation objects are initially created when the mesh is read from a bulk mesh data file (Section 1.2). New relation objects may be created, existing relation objects modified, or existing relation objects deleted. Existing relation objects are rarely modified; however, the presto SIERRA application has an operation that modifies relation objects to reorient selected elements to optimize some element calculations.

Converse Relation Objects (e.g., "Used-by" Connections)

The primary purpose of relation objects is to provide "uses" and "child" connection types. Each member of these connection-relation subsets has a corresponding converse that switches the domain and range mesh object to the "used-by" or "parent" connection type. Given a "uses" relation object, the converse "used-by" relation object may exist; however, it is not required to

exist. For example, the element-uses-node connections are always needed, but the node-*used-by*-element connections may not be needed. The SIERRA Framework allows an application to specify which "used-by" connections are needed—and if not needed, they are not maintained.

If a "used-by" relation object exists, the ordinal and orientation attributes of that relation object are obtained from the converse "uses" object. For example, a face-*used-by*-element relation object has ordinal and orientation attributes obtained from the corresponding element-uses-face relation object. Thus these attributes describe the range mesh object connection to the domain mesh object.

If a "used-by" relation object does not exist, an asymmetry exists in the relation-object data structure. This asymmetry could allow a mesh to be modified in such a way that it becomes inconsistent, with the inconsistency being undetectable. This possibility is mitigated by having each mesh object maintain a count of the number of relation objects for which it is the range coordinate and for which a corresponding converse relation object does not exist. This count enables a consistency check of the mesh to occur without having to maintain all converse relations.

8.4 Mesh Object Topology

A mesh object topology defines the expected "uses" connections from a particular type of mesh object to other mesh objects, and the correlation between these connections. A mesh object topology should be uniquely associated with the following set of properties:

1. The spatial dimension identifies the dimension of the problem domain in which the mesh object type is valid.

2. The topological dimension corresponds to the dimension spanned by the topology of the mesh object type (number of parametric coordinates). For example, in a 3D problem domain a hexahedral has three parametric coordinates, so its topological dimension is three; a quadrilateral face has two parametric coordinates, so its topological dimension is two; and quadrilateral shell also has two parametric coordinates, so its topological dimension is also two.

3. The number of vertices of the mesh object, when combined with the topological dimension, should identify the shape of the mesh object. Note that an edge has two "vertices" corresponding to its two endpoints.

4. The number of nodes that the mesh object uses must include at least one node per vertex; however, other nodes may also be used to define points in the interior of edges, faces, or elements. Furthermore, the nodes corresponding to vertices must be ordered before any other nodes.

A mesh object topology is an object that provides data to describe expected mesh object connectivity. For example, a simple mesh object topology for a quadrilateral face has four vertices, four nodes for the vertices numbered counterclockwise (v0,v1,v2,v3), and four simple

edges that are expected to be connected to vertex nodes (v0,v1), (v1,v2), (v2,v3), and (v3,v0), respectively.

Standard Mesh Object Topologies

The SIERRA Framework defines a set of standard mesh-object-topology objects for common finite element topologies. An application should use these objects whenever applicable, as opposed to creating a mesh-object-topology object that is a duplicate of an object in this set. Many operations compare the objects, as opposed to their contents, and will fail if a duplicate mesh-object-topology object is created and used.

The set of standard mesh-object-topology objects is extensible and should be extended as needed for new finite-element mesh object topologies. It is anticipated that the rate at which this set needs to be extended will decrease over time.

9 Creating and Using Mechanics Objects

The software design for mechanics includes the four types of interrelated objects illustrated in Figure 9.1. These types are *mechanics*, *mechanics algorithm*, *mechanics instance*, and *mechanics support*. Each type of object has a particular role in the implementation of an application's mechanics. A mechanics, as is referenced throughout this document, corresponds to the mechanics object identified in Figure 9.1.

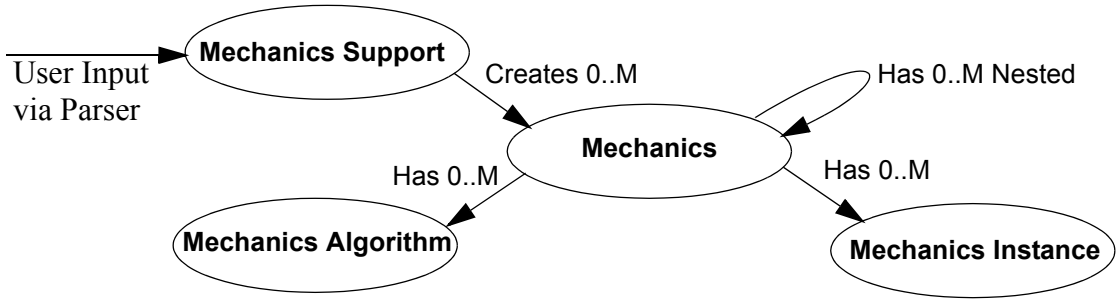


Figure 9.1. Mechanics components.

A mechanics owns a tightly coupled set of *mechanics algorithms*, for example, a PDE (partial differential equation) may be partitioned into several algorithms that are owned by a single mechanics. A mechanics may also own other *nested mechanics*, for example, a model for some physics may be the composition of a set of coupled PDEs. A mechanics and its algorithms define a set of calculations; however, they do not define the parameters for those calculations. The calculations of a particular mechanics may be applied to many different subsets of the mesh with different parameters; therefore, the parameters for these calculations are held in a set of *mechanics instances* that are owned by the mechanics. The fourth type of entity in Figure 9.1, *mechanics support*, is responsible for creating and configuring the mechanics needed by an application to solve the end user's problem. A mechanics support obtains the end user's problem specifications from a parsed input file.

9.1 Mechanics Algorithms

The role of a mechanics algorithm is to apply an operation to a subset of a mesh. This operation typically includes queries and modifications to field values associated with the subset of mesh objects. The mesh subset is defined by a set of mechanics instances that is owned by the same mechanics that owns the algorithm. A mechanics algorithm may also query or modify the parameters owned by each of the mechanics instances.

Each mechanics has one to many mechanics algorithms (having no algorithms is supported but would be of little or no use). The set of mechanics algorithms owned by a mechanics is dynamic, and this set is determined when a mechanics is configured by its mechanics support singleton. A particular type of mechanics will have some superset of available mechanics algorithms. The mechanics support singleton will select algorithms from this superset and register those

algorithms on the mechanics that it creates. Algorithms are selected to perform the simulation required by the application's end user.

Two forms of mechanics algorithms are supported: a *workset* algorithm and an "ordinary" algorithm. When either form of algorithm is called, it is given a list of the mechanics instances that it should use to perform its operation. Recall that the roles of a mechanics instance are (1) to identify an associated subset of mesh objects upon which the algorithm is to operate and (2) to supply a set of parameters for the algorithm's operation. An ordinary algorithm is responsible for iterating through this subset of mesh objects and querying or updating selected field values that are associated with those mesh objects. A workset algorithm is given additional support by the SIERRA Framework (Section 10) for simplified iteration of mesh objects and more efficient access to some field values.

Hierarchical Execution of Algorithms

An application's mechanics algorithms are executed hierarchically (Figure 9.2) according to the architecture of the application (Section 1.1). Recall that a procedure is responsible for managing the application's time stepping, a region is responsible for performing a single time step of the physics it models, and the set of mechanics nested within a region provides the implementation for a region's physics. Each mechanics within a region may also have nested mechanics that implement nested models, for example, a thermal mechanics may have a nested material mechanics.

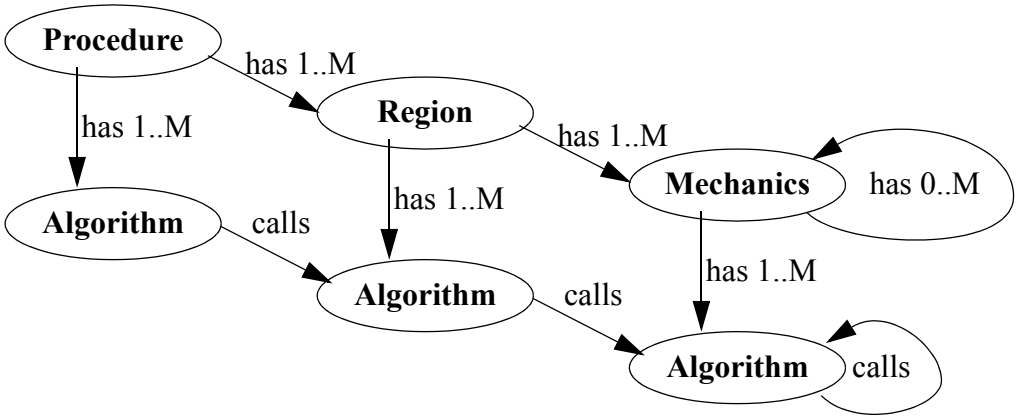


Figure 9.2. Mechanics and mechanics algorithm hierarchy.

Each layer of mechanics in the hierarchy is expected to have one or more mechanics algorithms. Mechanics algorithms are executed according to the hierarchical structure of the mechanics, e.g., a procedure's algorithms call its regions' algorithms and a region's algorithms call algorithms from the mechanics nested in the region. The SIERRA Framework initiates this hierarchical execution of mechanics algorithms by calling two mechanics algorithms that every procedure is assumed to have: *initialize* and *execute*.

A procedure's initialize algorithm is expected to initialize the procedure and its regions in preparation for time stepping. A procedure's execute algorithm is then expected to control the

subsequent sequence of time steps. It is recommended that each region and its internal mechanics also provide separate algorithms for initialization and subsequent execution. This separation is made so that the nominal time-stepping code is not cluttered with “if initial pass” code blocks.

Algorithm Interface

An algorithm executes to perform some calculation on behalf of the mechanics that owns it (see Figure 9.1). The algorithm is invoked to operate on a subset of mesh objects associated with mechanics instances of the owning mechanics using data provided by those mechanics instances. Thus an algorithm must have access to, or be given, the owning mechanics and the set of mechanics instances. This set of mechanics instances may be the complete set owned by the mechanics (see Figure 9.1) or may be a subset.

An algorithm is responsible for iterating over the subset of mesh objects associated with the mechanics instances, accessing field values (Section 7.3) associated with those mesh objects, and performing its specified calculations. Mesh object iteration and data access will be performed either directly on the mesh objects or through the workset capability (Section 10). The workset capability iterates mesh objects on behalf of the algorithm and provides the specified field values in contiguous arrays for efficient memory access.

9.2 Mechanics

The role of a mechanics is to aggregate the components of a model identified in Figure 9.1 and support a nested hierarchy of mechanics. For example, a mechanics that implements a thermal conductivity model may contain a nested mechanics that implements a temperature-dependent material model. The hierarchical nesting of mechanics is flexible and can be configured at run time in response to an application user’s inputs.

An individual mechanics is assumed to operate on mesh objects of a given type. For example, a thermal conductivity mechanics would be defined for linear hexahedral elements and another similar thermal conductivity mechanics (of the same type) would be defined for linear tetrahedrals. It is recommended that two such similar mechanics should share all of their algorithms and nested material-model mechanics; the similar mechanics should also have parameter and/or field specifications that only vary in their array dimensions (e.g., eight vertex quantities versus four vertex quantities).

Mechanics Types

A mechanics is an object of the mechanics type defined by the SIERRA Framework. Similarly, a region or procedure is an object of the region type or procedure type that is also defined by the SIERRA Framework. Note in Figure 9.2 that procedures and regions have a similar role in the hierarchical execution of algorithms as a mechanics. This role is sufficiently similar that the region type and procedure type are defined to be specializations of the mechanics type, as illustrated in Figure 9.3.

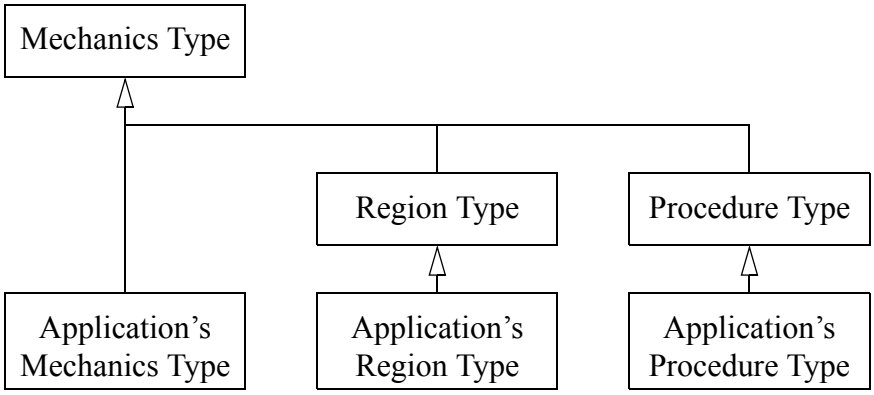


Figure 9.3. Mechanics-type inheritance hierarchy.

An application may also specialize the mechanics, region, or procedure types (illustrated in Figure 9.3) as needed to support its modeling needs. However, there are limitations on these application-defined specializations. Any new capabilities or data defined by the specialization are hidden from the SIERRA Framework. Such hidden data are inaccessible to the SIERRA Framework restart capability, are not reloaded during a restart operation, and thus become the responsibility of the application to reset this hidden data during a restart operation.

9.3 Mechanics Instances

The role of a mechanics instance is to (1) identify a particular subset of a mesh upon which the owning mechanics operates and (2) supply a set of parameters for that operation. For example, a mechanics instance is associated with the subset of a mesh that models material "A," and it also supplies material properties for material "A." Another example is a mechanics instance that identifies a particular boundary of a mesh and provides values for a flux boundary condition. A mechanics instance *does not* supply algorithms or identify fields used by the mechanics.

Parameter values are associated with the entire mechanics-instance subset and do not vary with individual mesh objects in that subset. For example, a subset of parameters may define material properties for a homogeneous block of elements. The parameters owned by a mechanics instance may be constant, such as fixed material properties, or may be updated by a mechanics algorithm. These updated parameters are automatically included within the SIERRA Framework restart capability.

Mechanics instances mirror the hierarchical nesting of the mechanics that own them, as illustrated in Figure 9.4. While procedures and regions are specialized types of mechanics, they do not have mechanics instances. A mechanics instance is used, in part, to identify a subset of a particular mesh. A procedure may have multiple regions, so it is excluded from holding references to particular subsets of a particular mesh and therefore does not need a mechanics instance. Similarly, a region is associated with the entire mesh and does not need a mechanics instance.

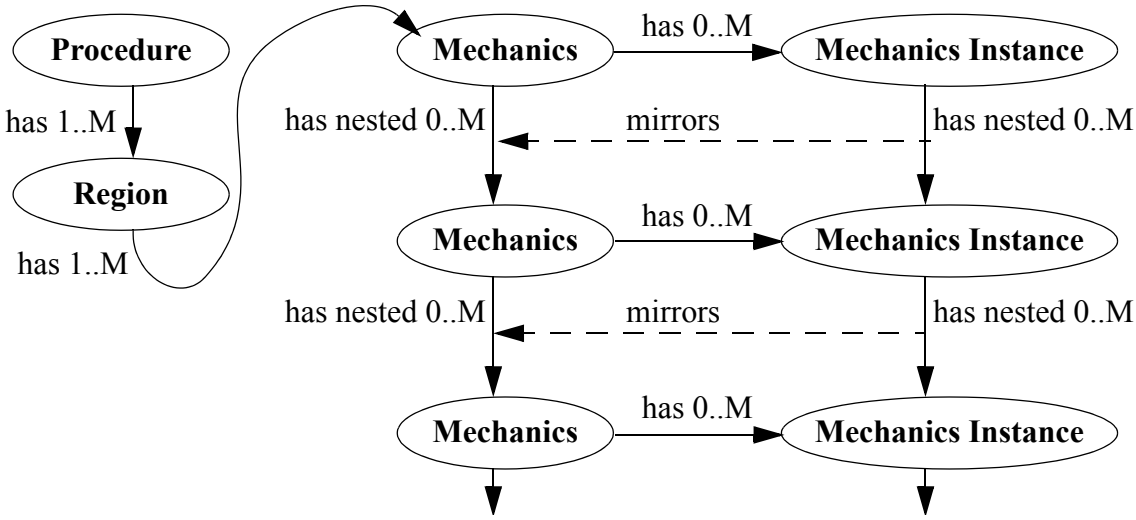


Figure 9.4. Mechanics-instance mirrored hierarchy.

Each mechanics that is nested immediately within a region has zero to many mechanics instances. If a mechanics owned by a region has a mechanics instance, each subsequent nested mechanics will have mechanics instances that mirror the mechanics hierarchy. This mirrored mechanics-instance hierarchy is automatically generated by the SIERRA Framework mechanics services.

Why Mechanics Versus Mechanics Instance

Mechanics and mechanics instances have distinct and defined roles in an application. These roles are separated into the algorithmic concerns for the mechanics and the data concerns of the mechanics instances. This separation of concerns is motivated by how algorithms and data are applied in an application.

It is assumed that an application will implement a moderate number (e.g., less than 100) of mechanics (i.e., physics models) immediately owned by a region. However, a particular mechanics at this level may be applied to thousands of distinct subsets of the mesh. The separation of concerns between algorithms and data reduces the memory requirements and improves efficiency when applying mechanics algorithms.

9.4 Mechanics Support (Singletons)

A mechanics support singleton is responsible for accepting the application user's input via the SIERRA Framework parser and generating mechanics objects as needed to solve the application user's problem. Each mechanics object generated by a particular mechanics support singleton is of a particular mechanics type. Each *type* of mechanics (e.g., thermal conductivity) is generated by exactly one mechanics support singleton in the application code.

Generating a mechanics object involves both creating and configuring a mechanics object. A mechanics object is created by dynamically allocating a new object of the mechanics support

singleton's mechanics type. Configuring the created mechanics object typically involves selecting a set of the mechanics algorithms and fields that the mechanics will need, selecting master elements that will be used, coordinating with other mechanics support singletons to set up a nested mechanics hierarchy, and defining the mechanics instances for the specific problem to be solved.

Installation of Mechanics Support Singletons

Mechanics support singletons must be installed in an application before the application user's input file can be parsed (Section 1.2).

Interaction with the Parser

Details of the parser's interface with singletons is described in SIERRA Framework / Parser Services; however, an overview is given in this section. Creation of mechanics includes both allocation and configuration of the mechanics object.

The SIERRA Framework parser processes commands from the application user's input file and routes information parsed from those commands to the designated singleton in the application. Interaction between a singleton and the parser has two parts: (1) designation of the singleton and input command pairs and (2) processing of parsed input commands by the designated singleton.

Each mechanics support singleton is installed in the application before the application user's input file is read (Section 1.2). To install a singleton object, the object must first be created and then the parser must be informed of the input file commands that will be handled by the singleton. Parsed input file commands are handled by a singleton with *command handler* routines. During installation a singleton gives the parser a set of input-file command identifier / command handler pairs.

Whenever the parser encounters a command matching the identifier, the parser will call the matching command handler with the parsed information. Command handlers for a particular mechanics support singleton may be called numerous times. During such a sequence of calls the singleton will create and configure mechanics according to information from the parsed commands.

9.5 Creating and Configuring Mechanics Objects

Each mechanics is an object of a particular mechanics type that is allocated by a mechanics support singleton and introduced into the application's mechanics hierarchy. Configuration of the mechanics has two major tasks: informing the SIERRA Framework of the fields that the mechanics needs (Section 10.5) and configuring the algorithms used by the mechanics. Configuration of a nonworkset algorithm is a simple registration of the algorithms's subprogram paired with a text label. After registration the subprogram will be referenced by the mechanics with the text label.

Configuration of a workset algorithm requires additional information regarding the type of mesh objects that are to be iterated, the connectivity of the iterated mesh objects to the mesh objects that will be accessed, and the list of fields that will be accessed on the iterated and connected mesh objects. This additional information is maintained in *workset registrars* that are part of the field management services (Section 10.5).

Registration of Fields

The algorithms of a mechanics will query and update field values associated with the set of fields and mesh objects that are used by the mechanics. The fields used by the mechanics must be registered with the SIERRA Framework through the field registrar (Section 5) and then "recalled" by the algorithms to access field values. Fields are associated with mechanics, i.e., made to be members of the mechanics' usage subset, by registering them with the mechanics' context (Section 3.2).

A given field may be registered by one or more mechanics, each with its own mechanics context. Such a reregistration places that field in each of the registering mechanics' usage subsets. However, each reregistration of a field must have compatible type and array dimensions.

Recommended Specialization of the Mechanics Type

It is recommended that an application specialize the Framework mechanics base class (see Appendix A: Mapping to C++ Classes) in order to save references to registered fields as member data. Each field registered by the mechanics will be used in some algorithm owned by the mechanics. These algorithms have immediate access to the mechanics and thus to any member data. Thus an algorithm may pass the required field specification from the mechanics to the access operation for field values (Section 7.3). If a reference to the field specification is not saved in a specialized mechanics type, that reference will have to be recovered by performing a search through the field registrar.

Registration of an Algorithm

An algorithm is registered within a mechanics (Figure 9.1) as either a workset algorithm (Section 10) or a nonworkset algorithm. In both cases algorithm registration requires the algorithm's implementation (e.g., a function) and a text label by which the algorithm is referenced. Registration of a workset algorithm also requires additional information: a workset stencil (Section 10.5) and a list of nested workset algorithms (Section 10.7) that the registered algorithm may call when it executes.

10 Workset Algorithms

Computational hosts with hierarchical memory (i.e., registers, cache, and main memory) are more efficient when data is "near" the CPU. For example, a calculation in which all data reside in the registers will be faster and more efficient than the same calculation in which all of its data reside in main memory. The goal of the SIERRA Framework *workset algorithm* capability is to keep as much as possible of an algorithm's required data in cache memory for as long as possible, i.e., minimize the number of cache misses during the execution of an algorithm.

Finite element algorithms typically use and update field values that are associated with elements and with mesh objects that are connected to elements. The access pattern for these algorithms is to iterate a set of elements and then, for each element, iterate the set of nodes several times, as illustrated in Figure 10.1. The set of values queried or updated for the designated (node, field) pairs are highly unlikely to be located near one another in computational memory; as such, the number of cache misses is likely to be large during the calculation.

```
foreach Element in { SetA }
  foreach Node connected to Element
    query Node.data(FieldX)
  end foreach
  some calculations
  foreach Node connected to Element
    Node.data(FieldY) = Node.data(FieldY) + somevalue(i)
  end foreach
  Element.data(FieldE) = someothervalue
end foreach
```

Figure 10.1. Example of access pattern for field values.

The workset algorithm capability performs the following operations on behalf of a mechanics algorithm:

- It manages a block of scratch memory, referred to as a workset, for use by the algorithm.
- It iterates the mechanics' usage subset of mesh objects.
- It iterates a specified subset of connected mesh objects, e.g., the nodes of each element in the usage subset.
- It gathers (prefetches) queried field values from mesh objects into arrays that reside in the workset.
- It scatters (overwrites) arrays in the workset to designated mesh-object field values.
- It assembles (sums) arrays in the workset into designated mesh-object field values.

10.1 Workset Scratch Memory

Each time a mechanics' workset algorithm is called, the algorithm is given a large block of scratch memory, referred to as a workset, for its data. A workset is sized to some predefined

fraction of the execution host's cache memory. This memory layout allows computationally intensive kernels to work on contiguous arrays of data that are concurrently in cache—a significant opportunity for efficient performance. This scratch memory is declared just before calling the algorithm and is reclaimed just after the algorithm returns. Therefore, all data in the workset are temporary.

An algorithm defines workset variables that are automatically mapped into the workset scratch memory. Each workset variable is a multidimensional array with the dimensions defined in Section 10.3. Fields that are inputs to an algorithm are automatically gathered from the iterated mesh objects' field values into workset arrays before the algorithm is called. Fields that are output from the algorithm are automatically scattered (assigned) or assembled (summed) from workset arrays to the iterated mesh objects' field values after the algorithm returns.

10.2 Automatic Iteration of Mesh Objects

The usage subset of a mechanics is automatically iterated on behalf of a mechanics' workset algorithm. As illustrated in Figure 10.1, this iteration performs an outer loop over mesh objects of a particular type and an inner loop over connected mesh objects. Each mesh object in the outer loop is iterated once. If some mesh object is connected to **N** of the mesh objects in the outer loop, the inner loop over connected mesh objects will access that mesh object **N** times, once for each connection.

Subsetting with Repeated Calls to a Workset Algorithm

The outer loop of the iteration follows the two-level subroster-bucket looping strategy recommended in Section 7. In this iteration strategy the iterated set of mesh objects is partitioned into subsets that correspond to the contents of a bucket. A mechanics' workset algorithm is repeatedly called to process each bucket-correlated subset. It is possible that the memory required for the workset variables associated with a bucket's subset would exceed the workset size. In this case, additional subsetting is performed until the desired workset size is attained.

Warning Regarding Parallel Processing

The subsetting of iterated mesh objects is performed independently on every processor. As such, the number of calls to workset algorithms is most likely different on each processor. Thus calls to workset algorithms are not synchronous, and parallel-processing communication operations must never be performed within these algorithms.

Coordination with Mechanics Instances

The subset of mesh objects that is associated with a particular subroster is homogeneous with respect to mesh object topology, use by mechanics, and association with mechanics instances (Section 6.1). When the workset capability calls a mechanics' workset algorithm for a "workset's worth" of mesh objects, it passes **all** of the mechanics instances that (1) are associated with the current subroster and (2) belong to the current mechanics. Typically this subset of mechanics

instances contains only one member; however, some mechanics may have mechanics instances that have overlapping subsets of associated mesh objects.

Alignment with Buckets

A bucket is a contiguous block of memory that holds field values in FORTRAN-compatible arrays, similar to a workset. As such, a mechanics' workset algorithm can directly operate on field values that reside in the current bucket of the workset iteration. If an algorithm coordinates its use of the workset with the current bucket, the number of required workset variables is reduced, in turn reducing the number of gather and scatter operations and also potentially increasing the maximum size of the mesh object subset that can be processed in a single workset. It is strongly recommended that a mechanics' workset algorithm use field values residing in the current bucket when possible.

10.3 Workset Variable Arrays

A workset variable is a multidimensional array that is resident in the workset scratch memory. It is expected that these arrays will be passed to optimized FORTRAN77 subroutines for numerical computations; therefore, workset variable arrays are dimensioned for use by FORTRAN77 code. Workset variable arrays can be dimensioned, using FORTRAN77 notation and semantics, as follows:

```
WSVarConformal(dimType, dimUse, numObjects )
WSVarPermuted( dimType, numObjects, dimUse )
WSVarConnected(dimType, numConn, numObjects)
```

- **`dimType`** is the dimension of the type of the variable, e.g., for a scalar `dimType = 1` and for a 3D vector `dimType = 3`.
- **`numObjects`** is the cardinality of the workset's current subset of mesh objects.
- **`dimUse`** is the number of scalars, vectors, or other type of values associated with the object.
- **`numConn`** is the number of mesh objects connected to the iterated mesh object, e.g., the number of nodes connected to an element.

The first workset variable in the above example, **`WSVarConformal`**, is dimensioned conformal to the dimensioning of a field value in a bucket (Section 7.2). A workset variable of this dimension should be a scratch variable that is not gathered from or scattered to field values. If such a variable is gathered or scattered, the algorithm is wasting execution time obtaining an exact copy of the current bucket's field value array.

The second workset variable in the above example, **`WSVarPermuted`**, has the **`dimUse`** and **`numObjects`** swapped with respect to the dimensioning of a field value in a bucket. Some mechanics' workset algorithms may prefer these dimensions in their calculations. For scratch variables these dimensions have no impact. For gathered and/or scattered variables these dimensions require a copy and permutation between field value arrays in a bucket and workset variable arrays.

The final workset variable in the above example, **`WSVarConnected`**, has the dimension **`numConn`** that corresponds to the expected number of connected mesh objects, e.g., number of nodes connected to an element in the workset. Field values may be gathered from the connected mesh objects and copied into the corresponding member of the array. Likewise, members of the workset array may be assembled, or summed into, field values of the connected mesh objects.

10.4 Gather, Compute, Scatter, and Assemble

The workset capability iterates the mechanics' usage subset of mesh objects and calls the mechanics' workset algorithm for a workset-compatible subset of these mesh objects. Immediately before calling the algorithm, a specified set of field values is gathered into a corresponding set of workset variable arrays. Immediately after calling the algorithm, a specified set of field values is either scattered to field values of the iterated mesh objects or assembled into field values of the connected mesh objects.

Each field value that is gathered, scattered, or assembled is associated with a specified state of the field. For example, field values of the old state are gathered, and field values of the new state are assembled. Arrays that reside within workset memory are temporary and are valid only during the execution of an algorithm; as such, these workset arrays never have a sense of state.

Field values of the iterated mesh objects may be gathered and/or scattered. However, a field value of the connected mesh objects may be either gathered or assembled, but not both. If a field value of a connected mesh object was gathered and assembled, the results of a workset algorithm would depend upon the order in which the elements are iterated. For example, if two elements share several nodes and the workset algorithm was to gather and then assemble the same nodal field values, the second element processed would gather modified values from the shared nodes. If in this example the order in which the elements are processed is reversed, the gathered nodal field values could be very different (orders of magnitude greater than numerical round-off). Furthermore, the update to the nodal file values cannot be a scatter operation—otherwise, the same order-dependency problem would occur with the last element processed setting the final value. Therefore, the only consistent update to field values for connected objects is an assemble operation.

Illustration of Gather, Scatter, and Assemble

The gather, scatter, and assemble operations are illustrated in Figure 10.2. This illustration uses pseudocode that is not intended to comply with any actual programming language. However, the array dimensions are intended to reflect FORTRAN77 semantics.

```
do n = 1,numObject
  // Gather from iterated mesh objects
  do j = 1,dimUse
    do i = 1,dimType
      WSVarConformal(i,j,n) = bucket.data(i,j,n)
      // or if perumuted:
      WSVarPermuted(i,n,j) = bucket.data(i,j,n)
    end do
  end do
  // Gather from connected mesh objects
  do k = 1,numConn
    do i = 1,dimType
    WSVarConnected(i,k,n) = element(n).node(k).data(i)
  end do
end do

// Call workset algorithm for the current subset

do n = 1,numObject
  // Scatter to iterated mesh objects
  do j = 1,dimUse
    do i = 1,dimType
      bucket.data(i,j,n) = WSVarConformal(i,j,n)
      // or if permuted:
      bucket.data(i,j,n) = WSVarPermuted(i,n,j)
    end do
  end do
  // Assemble to connected mesh objects
  do k = 1,numConn
    do i = 1,dimType
    element(n).node(k).data(i) += WSVarConnected(i,k,n)
  end do
```

Figure 10.2. Illustration of gather, scatter, and assemble operations.

10.5 Workset Stencil

The workset capability will manage workset memory, iterate a specified set of mesh objects, and perform gather, scatter, and assemble operations. The set of mesh objects iterated by a workset is specified by (1) a mechanics that owns the algorithm and (2) the mesh object topology (Section 8.4) of the iterated mesh objects. The mesh object topology provides the expected number of connected objects for the workset.

For example, one mechanics algorithm uses the workset capability to process simple quadrilateral elements, while another mechanics algorithm processes simple triangular elements. The first workset knows to expect four nodes per element and the second workset knows to expect three nodes per workset. The workset stencil for these two examples is simply the mesh object topology of the elements to be processed.

Homogeneity

A mechanics' workset algorithm processes a subset of mesh objects that is homogeneous with respect to mesh object topology. If a mechanics is required to process a heterogeneous subset of mesh object topologies, e.g., quadrilaterals and triangles, the mechanics must provide a workset algorithm for each mesh object topology. These algorithms could share the same calculations (e.g., subroutines), if those calculations are parameterized with respect to mesh object topology. It is recommended that algorithms be parameterized with respect to mesh object topology and any other master element property whenever possible.

Iterate Versus Root

The workset capability has an additional iteration algorithm that is available to a mechanics. The previously discussed algorithm iterates a topologically homogeneous set of mesh objects in an outer loop and iterates connected mesh objects in an inner loop. The additional algorithm also iterates mesh objects of a specified topology; however, it then traverses to some other connected mesh object before performing the inner loop. This other connected mesh object is referred to as the *root of the workset stencil*, while the iterated object in the outer loop is referred to as the *iterate of the workset stencil*.

A workset stencil is an identification of a mesh object topology for the iterate of the workset stencil and a mesh object topology for the root of the workset stencil. For a simple workset only one mesh object topology is supplied, the original simple iteration algorithm is used, and the iterate and root refer to the same mesh object. Illustrations of workset stencils are given in Figure 10.3.

A workset's inner loop always iterates mesh objects that are connected to the root. Therefore, all gather and assemble operations for connected mesh objects' field values are with respect to the root mesh object, not the iterate.

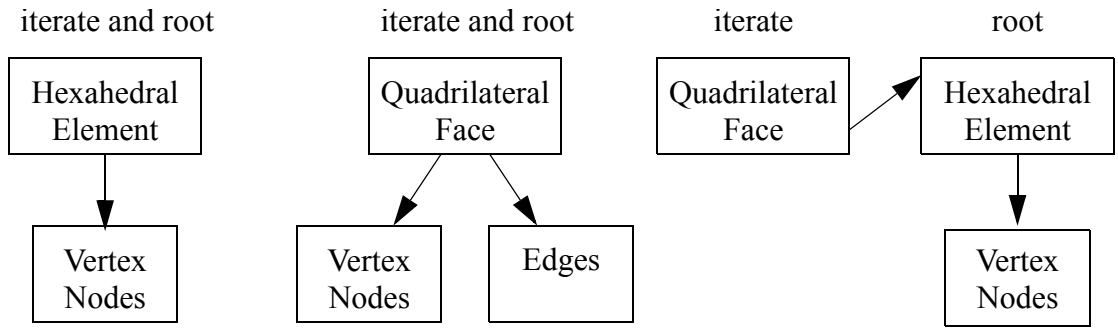


Figure 10.3. Examples of workset stencils.

Three different workset stencils are illustrated in Figure 10.3. The first (left) stencil specifies that the iterate is a hexahedral element and the inner loop will include nodes connected to those elements. The second (center) stencil specifies that the iterate is a quadrilateral face and the inner loop will include the nodes and edges connected to those faces. The last (right) stencil specifies

that the iterate is a quadrilateral face; the root of the stencil is each element connected to those faces; and the inner loop will include the iterated face, connected elements, and nodes connected to the elements.

Impact of Iterate-Root Workset Stencils

When the iterate and the root of a workset stencil are not the same mesh object, the workset capability will generate an entry in the workset for each unique (iterate, root) pair of mesh objects. If in the face-element stencil example (Figure 10.3) an iterate face is connected to two different root elements, the face will appear in the workset twice, once per root element. Similarly if two iterate faces are connected to the same root element, the element will appear in the workset twice, once per face.

Iteration of (iterate, root) pairs of mesh objects cannot align with buckets. Such a workset algorithm does not have a current bucket to access; therefore, all input data must be gathered into the workset. Likewise, all output data must be scattered or assembled from the workset.

Gather and scatter operations may be performed for either the iterate or the root of the workset. However, such operations are potentially hazardous. Consider a workset with faces for iterate mesh objects and elements for root mesh objects. If an element has two iterated faces, the element is processed twice. If workset data are scattered to the element's field values, the final scatter operation to the element overwrites all previous scatter operations. Finally, if the iteration was simply reordered, the results could change dramatically for a different final (face, element) pair.

10.6 Declaring a Workset Algorithm

Declaration of a workset algorithm has the following three steps:

1. Create the workset algorithm within a mechanics hierarchy.
2. Declare the workset's fields.
3. Declare the workset's gather, scatter, and assemble operations.

Creating a Workset Algorithm

Each workset algorithm is created as a member of a mechanics within a mechanics hierarchy (Section 9.5). Creation of a workset algorithm requires the subprogram that is to be called, a text label for the algorithm, a workset stencil, and a list of the nested algorithms that are called by the created workset algorithm. This information is "compiled" into an internal data structure that is used to set up the workset and call the subprogram.

The data structure for a workset algorithm includes a set of specifications for workset variables and for gather, scatter, and assemble operations. This set of specifications resides in a *workset registrar*. A workset registrar is similar to a field registrar in that it is a container for a set of workset variable specifications (versus a mesh object's field specifications). However, the variables declared in a workset registrar have temporary values that only exist while the workset

algorithm is called. A workset registrar is also a container for a set of the gather, scatter, and assemble declarations.

Declaring Workset Variables

Declaration of a workset variable is simpler than declaration of a field (Section 5.2). A workset variable declaration only requires a field type, array dimension, and text label. Workset variables are not explicitly associated with mesh objects; however, a workset variable may be indirectly associated with mesh objects through gather, scatter, or assemble operations. There is no need to associate workset variables with mechanics since the scope of a workset variable is limited to the mechanics that owns the workset algorithm. Because workset variables do not exist between time steps, such variables are temporary.

The declaration of a workset variable returns a C++ pointer to a workset variable object. This object is required to access the workset-resident data array associated with the workset variable. It is recommended that this pointer be saved as a private data member of the workset algorithm's owning mechanics object. These private data members must appear in the derived mechanics type of the mechanics. If a pointer to the workset variable object is not saved as recommended, the workset algorithm will be required to retrieve this object in a search-by-name operation. It is not recommended that a workset algorithm pay this execution-time overhead.

Declaring Gather and Scatter Operations for Iterate and Root

Gather and scatter operations for the iterate or root mesh object are declared by associating a mesh object field with a workset variable. The workset variable must be declared with the same field type and **`dimUse`** dimension as the associated mesh object field (see Section 10.4 and Figure 10.2). The declaration of a gather or scatter operation is also required to specify whether or not the data should be permuted as described in Section 10.4.

Declaring Gather and Assemble Operations for Connected Mesh Objects

Gather and assemble operations for the mesh objects connected to a workset's root mesh object are declared by selecting the kind of connected mesh object to be iterated (Figure 10.3) and associating a mesh object field with a workset variable. The kind of connected mesh object is selected by providing its mesh object topology. The workset variable must be declared with the same field type as the mesh object field.

A workset variable must also be declared with a dimension that is compatible with the connected mesh object. For a gather operation this dimension must be equal to the number of connected mesh objects as illustrated in Figure 10.3. For an assemble operation the dimension is either equal to the number of connected mesh objects or equal to one. If it is equal to the number of connected mesh objects, the summation is performed as in Figure 10.3. If the dimension is one, the same workset value is summed into each of the connected mesh-object field values, as illustrated in Figure 10.4.

The second assemble option illustrated in Figure 10.4 allows a workset algorithm to reduce the memory it requires for workset variables, if the same value is to be assembled to each connected mesh object. This optimization was introduced for a prototype application, and it is unknown if it is being used within a production application.

```
dimension WSVarConn(dimType,numConn,numObject)
dimension WSVarSame(dimType,1,numObject)
do n = 1,numObject
  // Assemble to connected mesh objects
  do k = 1,numConn
    do i = 1,dimType
    element(n).node(k).dataX(i) += WSVarConn(i,k,n)
    element(n).node(k).dataY(i) += WSVarSame(i,1,n)
  end do
end do
```

Figure 10.4. Illustration of options in assemble operations.

10.7 Nested Workset Algorithms

Workset algorithms may call other workset algorithms as illustrated in Figure 9.2. The hierarchical execution of workset algorithms is supported by merging the workset variables required by each nested workset algorithm into the topmost workset. Thus each nested workset algorithm shares the contiguous block of workset memory with the topmost workset algorithm.

A nested workset algorithm may have an associated workset registrar that is nested in a hierarchy beginning at the topmost workset registrar. Typically, a nested workset algorithm will have input and output data with respect to the calling workset algorithm. This input and output data is a shared workset variable, typically declared by the calling workset algorithm.

The workset capability supports nesting of workset algorithms by merging nested workset registrars into a single workset registrar with field members of aggregate data types. These aggregate data types are defined by recursively merging the nested workset registrars into the next higher-level workset registrar. The resulting workset-variable array dimensions are generated in the same manner as aggregate field-value array dimensions in buckets (Section 7.2). Gather and scatter operations are similarly merged from lower-level workset registrars into the next higher-level workset registrar.

11 Parallel Distributed Mesh

Scalability in a massively parallel distributed-memory environment requires that a region's set of mesh objects be distributed among the processor's memory space. The SIERRA Framework core services manage the parallel distribution of mesh objects for an application. Management of a parallel distributed mesh is described in three parts:

1. policies and distributed mesh sets, relations, and data structures
2. parallel operations that do not modify the distributed-mesh data structures
3. operations that modify the distributed-mesh data structures

11.1 Processor Subset Classes

Support for management of a massively parallel distributed mesh is based upon the specifications for several classes of subsets defined on each region's mesh and the relations between these classes. These classes are the result of analysis of required capabilities and policy decisions for the SIERRA Framework software design.

Processor-Resident Class

A mesh object resides in the memory space of one or more processors in the massively parallel distributed-memory environment. Given a mesh object, it is mandatory to identify the processors on which that mesh object resides. A processor-correlated domain decomposition of a mesh is typically used to assign mesh objects to processors.

A domain decomposition of a mesh typically assigns type-specific subsets of mesh objects to processors. The assignment of the remaining subset of mesh objects is induced by the connectivity of the mesh. For example, if a domain decomposition assigns elements to processors, the nodes, edges, and faces can be assigned to their attached element's processor. In this example, a node that is shared by four elements assigned to four different processors would be assigned to, and reside on, all four processors.

The subset of mesh objects that are assigned to a particular processor defines the resident subset of mesh objects for that processor (Section 2.4). The set of these subsets defines the processor-resident class for the mesh.

$$\mathbf{Resident}_{Pi} = \{\mathbf{MeshObject}_j : \mathbf{MeshObject}_j \textbf{ resides-on } Pi\}$$

$$\mathbf{ResidentClass} = \{\mathbf{Resident}_{Pi} \forall\ Pi\}$$

Processor-Owned Partition Class

A SIERRA Framework policy for parallel distributed meshes is that each mesh object is owned by exactly one of the processors on which it resides. An additional related policy is that operations must "favor" the use of the owned mesh object when appropriate. For example, the calculation of

a norm of a field performs a summation of terms contributed from mesh-object field values. In this example, a processor should contribute terms for mesh objects that are owned by that processor; otherwise, redundant terms would be contributed and an erroneous result would be generated.

The subset of mesh objects that are owned by a particular processor defines the processor's owned subset of mesh objects (Section 2.4). The set of these subsets defines the processor-owned class for the mesh. This processor-owned class is a partition for the mesh.

$$\mathbf{Owned}_{Pi} = \{\mathbf{MeshObject}_j : \mathbf{MeshObject}_j \textbf{ owned-by } Pi\}$$

$$\mathbf{OwnedClass} = \{\mathbf{Owned}_{Pi} \forall\ Pi\}$$

$$\mathbf{Owned}_{Pi} \subseteq \mathbf{Resident}_{Pi}$$

$$\mathbf{Owned}_{Pi} \cap \mathbf{Owned}_{Pj} = \varnothing \quad \forall\ Pi \neq Pj$$

Processor-Shared Class

Operations on a parallel distributed mesh must be coordinated for mesh objects that reside on more than one processor. Given a mesh object, this coordination requires that the membership of the mesh object in each processor-resident subset is known. This requirement is limited by the initial clause of "Given a mesh object" such that only the processors upon which a mesh object resides are required to have knowledge of that mesh object's membership in processor-resident subsets.

The following processor-shared class represents the required knowledge for all processors. However, a given processor is only required to have knowledge of the subclass $\mathbf{SharedClass}_{Pi}$ that is associated with that processor.

$$\mathbf{SharedClass} = \{\mathbf{GlobalShared}_{Pi,\,Pj} = \mathbf{Resident}_{Pi} \cap \mathbf{Resident}_{Pj} \quad \forall\ Pi \neq Pj\}$$

$$\mathbf{SharedClass} \supset \mathbf{SharedClass}_{Pi} = \{\mathbf{GlobalShared}_{Pi,\,Pj} \forall\ Pj\}$$

11.2 Processor-Resident Policies

A face and an edge may reside on more than one processor in a parallel distributed mesh. If a face or an edge is on a processor subdomain boundary, all processors that share that face or edge are aware of its global sharing. A face or an edge will be present in any processor's subdomain that includes all nodes of that face or edge. This policy can and does result in faces and edges being resident on a processor that does not have an element to attach to that face or edge, as illustrated in Figure 11.1.

Each mesh object in a parallel distributed mesh may be resident on more than one processor but is always owned by exactly one processor. The residence policy for edges and faces is that if all nodes of an edge or a face are resident on a processor, the edge or face will also be resident on that processor. This residence policy has implications for the ownership policy related to edges and

faces such that an edge or a face attached to one or more elements in the global mesh will be owned by a processor that also owns one of those elements.

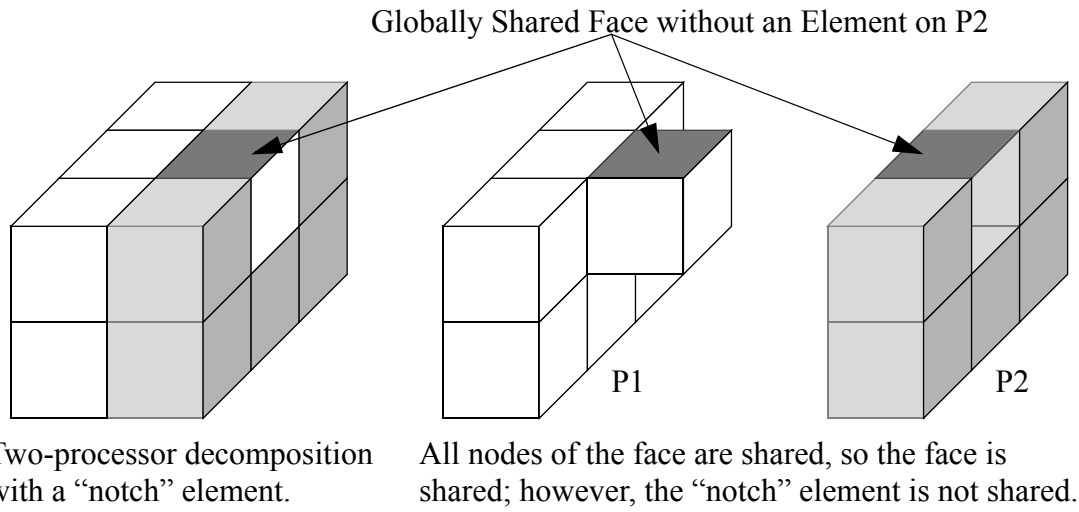


Figure 11.1. Parallel distributed mesh with “orphaned” face.

In the illustration presented in Figure 11.1, the residence policy caused a face to be resident on processor #2 without a connected element. However, the ownership policy would require that this face be owned by processor #1 since that is the only processor with an element connected to the face. A developer should be aware of the situation described and illustrated here and its impact on algorithms that process faces and their connected elements.

11.3 Intermesh Relation

An application may have multiple regions where each has its own mesh. These regions are typically coupled by transferring (see Figure 1.1) field values from a source mesh to a destination mesh. Such a transfer of data implies some form of intermesh connectivity between two meshes. Intermesh connectivity is complicated by the possibility that two mesh objects with an intermesh connection may not reside on the same processor.

Recall that the intramesh connectivity relation defined in Section 8 represents the connectivity between mesh objects within the same mesh. A similar intermesh connectivity relation is defined to represent connectivity between different meshes.

$$\mathbf{IntrameshRelation}_{\alpha,\,\beta} \subset \mathbf{Mesh}_{\alpha} \times \mathbf{Mesh}_{\beta}$$

The role and use of an intramesh relation are very different from those of an intermesh relation; as such, the software design and implementation are also very different.

Comparison with Intramesh Relation

The intramesh connectivity relation is used to represent the connectivity between mesh objects that are members of the same mesh and also are resident on the same processor. An intermesh connectivity relation is required to represent connectivity between mesh objects that may be members of different meshes and may be resident on different processors. These same-mesh and same-processor restrictions on an intramesh connectivity relation enable a highly optimized implementation that is not possible for the more general intermesh connectivity relation.

Similarities with Processor-Shared Class

An intermesh connectivity relation has similarities with the processor-shared class. A member in an intermesh connectivity relation must represent the connection between two mesh objects that may reside on different processors. If a mesh object is a member of a processor-shared subset $\mathbf{Shared_{Pi,Pj}}$, the mesh object resides on Pi and Pj, and some connection must exist between the two incarnations of the same mesh object. In an intermesh connection, the mesh objects are different and may reside on different processors; in a shared subset, the mesh object is the same and does reside on different processors.

The abstractions for the intermesh relation and the processor-shared subset have a common, more general abstraction of a *mesh object communication relation* described in Section 11.4. This abstraction is specialized to represent the intermesh relation and processor-shared subset abstractions.

11.4 Mesh Object Communication Relation (a.k.a. Communication Specification)

A mesh object communication relation, also referred to as a *communication specification* (CommSpec), contains generalized connections between mesh objects that may reside on different processors. These relations and their specializations are used to support all of the SIERRA Framework's mesh-dependent communication operations.

Definition

Members of a CommSpec are defined by the following expression:

$$\mathbf{CommSpec}_{\alpha,\,\beta} \subset \left\{ \begin{array}{l} ((\mathbf{MeshObject}_m, \mathrm{Pi}), (\mathbf{MeshObject}_n, \mathrm{Pj})) : \\ \qquad \mathbf{MeshObject}_m \in \mathbf{Resident}_{\mathrm{Pi}} \subseteq \mathbf{Mesh}_\alpha \\ \quad \mathbf{and} \quad \mathbf{MeshObject}_n \in \mathbf{Resident}_{\mathrm{Pj}} \subseteq \mathbf{Mesh}_\beta \end{array} \right\}$$

The domain and range coordinates of a CommSpec member are themselves a pairing of a mesh object with one of the processors upon which that mesh object resides. These domain and range coordinate pairings identify a processor-specific incarnation of a mesh object. Thus a CommSpec represents the connectivity between processor-specific incarnations of mesh objects.

Each CommSpec has a domain mesh and a range mesh, identified as $\mathbf{Mesh}_{\alpha}$ and $\mathbf{Mesh}_{\beta}$ in the previous definition. Every mesh object appearing in the domain coordinate must be a member of the domain mesh. Likewise, every mesh object appearing in the range coordinate must be a member of the range mesh.

Send and Receive Subsets of a CommSpec

A given processor only needs to have knowledge of the members of a CommSpec that include that processor, i.e., all members whose domain or range coordinate includes that processor. Furthermore, a processor needs two subsets, one in which that processor appears in the members' domain coordinate and another in which that processor appears in the range coordinate. The first subset defines a set of mesh objects for which the processor will send information. The second subset defines a set of mesh objects for which the processor will receive information.

A processor's send and receive subsets of a CommSpec are partitioned according to the range processor and domain processor, respectively. The members of this partition identify specific messages that must be exchanged and the mesh objects associated with each message. The following relationship defines the mesh objects associated with a message from Pi to Pj:

$$((\mathbf{MeshObject}_{m},Pi),(\mathbf{MeshObject}_{n},Pj)) \in \mathbf{CommSpec}_{\alpha,\,\beta}$$

$$\mathbf{Message}_{Pi,\,Pj} \sim \{((\mathbf{MeshObject}_{m},Pi),(\mathbf{MeshObject}_{n},Pj)) : \forall m, n\}$$

Symmetric CommSpec for Processor-Shared Class

The processor-shared class (Section 11.1) of a mesh can be represented by a symmetric CommSpec that is constructed as follows:

$$\mathbf{CommSpec}_{\alpha,\,\alpha} \subset \left\{ \begin{array}{l} ((\mathbf{MeshObject}_{m},Pi),(\mathbf{MeshObject}_{m},Pj)) : \\ \quad \mathbf{MeshObject}_{m} \in \mathbf{GlobalShared}_{Pi,\,Pj} \subseteq \mathbf{Mesh}_{\alpha} \end{array} \right\}$$

$$\mathbf{MeshObject}_{m} \in \mathbf{GlobalShared}_{Pi,\,Pj} \Leftrightarrow \mathbf{MeshObject}_{m} \in \mathbf{GlobalShared}_{Pj,\,Pi}$$

therefore

$$\mathbf{CommSpec}_{\alpha,\,\alpha} = \mathbf{CommSpec}^{C}_{\alpha,\,\alpha}$$

Note that each member of this CommSpec has the same mesh object in both its domain coordinate and range coordinate. However, the domain and range coordinates refer to a particular incarnation of that mesh object on a specific processor. Symmetry is guaranteed by construction since mesh object membership in the subset $\mathbf{GlobalShared}_{\mathbf{Pi,Pj}}$ guarantees membership in the subset $\mathbf{GlobalShared}_{\mathbf{Pj,Pi}}$.

General CommSpec for Intermesh Connectivity Relation

The CommSpec for an intermesh connectivity relation is constructed as follows:

$$\mathbf{CommSpec}_{\alpha,\,\beta} \subset \left\{ \begin{array}{l} ((\mathbf{MeshObject}_m, Pi),(\mathbf{MeshObject}_n, Pj)) : \\ \quad \mathbf{MeshObject}_m \in \mathbf{Resident}_{Pi} \subseteq \mathbf{Mesh}_\alpha \quad \textbf{and} \\ \quad \mathbf{MeshObject}_n \in \mathbf{Resident}_{Pj} \subseteq \mathbf{Mesh}_\beta \quad \textbf{and} \\ \quad (\mathbf{MeshObject}_m, \mathbf{MeshObject}_n) \in \mathbf{IntermeshRelation}_{\alpha,\,\beta} \end{array} \right\}$$

This CommSpec defines complete connectivity of the parallel distributed mesh for all incarnations of a mesh object in the domain and range. However, this degree of completeness may not be required by a mechanics algorithm. For example, if the use of the intermesh connectivity relation is to copy a field value from the domain mesh object to the range mesh object, the domain mesh object only needs to appear once in the CommSpec. The reduced CommSpec would be generated with only the owned incarnations of mesh objects appearing in the domain.

$$\mathbf{CommSpec}_{\alpha,\,\beta} \subset \left\{ \begin{array}{l} ((\mathbf{MeshObject}_m, Pi),(\mathbf{MeshObject}_n, Pj)) : \\ \quad \mathbf{MeshObject}_m \in \mathbf{Owned}_{Pi} \subseteq \mathbf{Mesh}_\alpha \quad \textbf{and} \\ \quad \mathbf{MeshObject}_n \in \mathbf{Resident}_{Pj} \subseteq \mathbf{Mesh}_\beta \quad \textbf{and} \\ \quad (\mathbf{MeshObject}_m, \mathbf{MeshObject}_n) \in \mathbf{IntermeshRelation}_{\alpha,\,\beta} \end{array} \right\}$$

Restriction to Mesh Object Type

Each communication operation on a parallel distributed mesh is typically performed for a type-specific subset of the mesh, e.g., for the shared nodes of a mesh. The implementation of CommSpecs is tailored to this typical usage such that the mesh objects appearing in the members' domain coordinate is restricted to the same mesh object type (e.g., node, edge, face, element). The mesh objects appearing in the members' range coordinate are likewise restricted; however, the mesh object type for the domain and range may be different.

11.5 Distributed Data Structure for CommSpec

The members of a CommSpec, $((\mathbf{MeshObject}_m, Pi),(\mathbf{MeshObject}_n, Pj))$, identify a domain mesh object, a domain processor upon which the domain mesh object is resident, a range mesh object, and a range processor upon which the range mesh object is resident. The domain and range processors (Pi and Pj) are typically not the same; as such, the referenced domain and range mesh

objects are resident on different processors. Therefore, the data members of a CommSpec must be managed within a distributed data structure.

Software Design

Each member of a CommSpec is translated into software as two pairs: (1) a reference to the domain mesh object paired with the identifier for the range processors and (2) a reference to the range mesh object paired with the identifier for the domain processor. Members are split in this manner in order to associate *source* (domain) mesh objects with *destination* (range) processors and *destination* (range) mesh objects with *source* (domain) processors. This association supports sending and receiving messages—information generated from domain mesh objects is sent to the range processor, and information received from the domain processor is incorporated into the range mesh object.

$$((\textbf{domain-MeshObject}_m,\textbf{domain-}Pi),(\textbf{range-MeshObject}_n,\textbf{range-}Pj)) \rightarrow$$
$$(\textbf{domain-MeshObject}_m,\textbf{range-}Pj)\ ,\ (\textbf{range-MeshObject}_n,\textbf{domain-}Pi)$$
$$\textbf{where}$$
$$(\textbf{domain-MeshObject}_m,\textbf{range-}Pj)\ \ \textbf{resides-on}\ Pi$$
$$(\textbf{range-MeshObject}_n,\textbf{domain-}Pi)\ \ \textbf{resides-on}\ Pj$$

The association of domain mesh objects with range processors and range mesh objects with domain processors is necessary but not sufficient. The association between the domain mesh object and range mesh object must also be maintained. This association cannot be explicitly expressed in the data structure when the domain and range processors are different, which is typically the case.

CommSpec-member domain mesh objects are associated with range mesh objects as follows. The members of a CommSpec are partitioned into processor-pair subsets such that each subset has a given domain and range processor. The corresponding data members of $\textbf{CommSpec}_{(Pi,Pj)}$ that

$$\textbf{CommSpec}_{(Pi,Pj)} = \{((\textbf{MeshObject}_m,Pi),(\textbf{MeshObject}_n,Pj))\forall m, n\}$$

reside on the domain processor Pi and range processor Pj are maintained in arrays with conformal ordering. For example, if $(\textbf{domain-MeshObject}_m,\textbf{range-}Pj)$ is the k^{th} entry of this array on domain processor Pi, then $(\textbf{range-MeshObject}_n,\textbf{domain-}Pi)$ must also be the k^{th} entry in the corresponding array on range processor Pj.

Storage Reduction for Symmetric CommSpec

A communication specification is symmetric if it is equal to its own converse and meets the following condition:

$$((\textbf{MeshObject}_m, Pi),(\textbf{MeshObject}_m, Pj)) \in \textbf{SymmCommSpec}$$

if-and-only-if

$$((\textbf{MeshObject}_m, Pj),(\textbf{MeshObject}_m, Pi)) \in \textbf{SymmCommSpec}$$

If a CommSpec is symmetric, then the set of processor-resident data members satisfy the following condition:

$$((\textbf{MeshObject}_m, Pi),(\textbf{MeshObject}_m, Pj)) \in \textbf{SymmCommSpec}$$

if-and-only-if

$$((\textbf{MeshObject}_m, Pj),(\textbf{MeshObject}_m, Pi)) \in \textbf{SymmCommSpec}$$

$$\rightarrow$$

$$(\textbf{domain-MeshObject}_m, \textbf{range-}Pj) \ \textbf{resides-on}\ Pi$$

if-and-only-if

$$(\textbf{range-MeshObject}_m, \textbf{domain-}Pj) \ \textbf{resides-on}\ Pi$$

Note from the above condition that the resident data members for the domain and range arrays are identical—a $(\textbf{MeshObject}_m, Pj)$ data member exists in the domain array if and only if it also exists in the range array. Thus only a single array of CommSpec data members is required to represent both the domain array and the range array.

11.6 Field-Value Global Assemble Operation

The most heavily used communication operation for a typical parallel distributed mesh is the assembly of field values from each incarnation of a shared mesh object. This assembly operation, also referred to as a parallel swap-add, is defined as follows:

$$\textbf{let} \qquad \textbf{Value}_{m,\,k,\,Pi} = \textbf{FieldValue}(\textbf{MeshObject}_m, \textbf{Field}_k) \ \textbf{on}\ Pi$$

$$\textbf{let} \qquad \textbf{P}(\textbf{MeshObject}_m) = \{Pj : \textbf{MeshObject}_m \in \textbf{Resident}_{Pj}\}$$

$$\textbf{global-assembly:} \quad \textbf{Value}_{m,\,k,\,Pi} = \sum_{Pj \in \textbf{P}(\textbf{MeshObject}_m)} \textbf{Value}_{m,\,k,\,Pj} \qquad \forall Pi$$

The CommSpec for processor-shared mesh objects provides all information necessary to define the structure of the messages.

The assembly operation is required to consistently give identical results on all processors. This cannot be guaranteed unless the summation loop is in exactly the same order on all processors and for every invocation of the global assemble operation. If this loop is not performed in the same order, a summation with more than two terms may result in numerical round-off differences among processors. Such differences can lead to serious run-time problems, as they have in some non-SIERRA Framework–based applications.

11.7 Copying Mesh Objects Between Processors

The SIERRA Framework supports copying mesh objects between processors. This parallel operation is currently used by the dynamic load balancing capability, intermesh transfer capability, and element ghosting capability. Copying a mesh object is divided into two parts: (1) copying its subset membership information (Section 3) and intramesh connectivity information (Section 8) and (2) copying the mesh object's field values. When a mesh object is copied to a processor it becomes a member of that processor's resident subset, $\mathbf{Resident}_{Pi}$.

The dynamic load balancing capability uses the mesh-object copying operation as follows. First, a dynamic mesh-partitioning algorithm is used to determine on which processor each mesh object should reside for the subsequent calculations to be load balanced. Next, each mesh object that is not already on the desired processor is copied to that processor. Finally, copied mesh objects are deleted from their original processor. The processor-resident and processor-shared mesh object subsets are updated accordingly.

Mesh objects that are copied between processors of the same mesh, or from a source mesh to a different destination mesh, retain their global identifier. If the destination processor of the copy operation has an existing mesh object with the given global identifier, the copied mesh object is compared to an existing mesh object. If the two versions of the mesh object are incompatible, an error is generated. If the copied and existing mesh object are compatible, their connectivity information is merged and existing field values are overwritten.

12 Dynamic Mesh Modifications

Mesh objects may be created, deleted, or have their attributes changed during the execution of an application. Use of these dynamic-mesh-modification capabilities must conform to a set of guidelines to guarantee correct results. These guidelines and their consequences are described in this section.

12.1 Local Modifications and Global Synchronization

In a parallel distributed mesh, dynamic mesh modifications are performed in two phases. First, each processor independently modifies its resident (local) subset of the global mesh. Once all local modifications are complete, the processors synchronize the modifications that were made on processor boundaries.

Each of the three capabilities for dynamic mesh modification (i.e., creating, deleting, or changing attributes) must be globally synchronized. Local mesh-modification operations may be mixed together, e.g., both creating mesh objects and changing attributes in one operation. However, each global synchronization operation must be called after all local modifications are completed.

Mesh objects are not deleted during the local modification phase. Instead, the mesh objects are marked for deletion and then actually deleted during the global synchronization phase. This mark-and-then-delete strategy is used to preserve processor-boundary information until global synchronization can coordinate the deletion of a shared mesh object.

Global synchronization of mesh modifications must be performed for each modified roster. If a roster is not modified, synchronization is not required for the roster. If multiple rosters are simultaneously modified (e.g., both nodes and elements), these rosters must be synchronized as follows:

- Global synchronization of rosters for creating mesh objects must be in the following order: node, edge, face, element. This order ensures that created mesh objects are synchronized after the mesh objects they use, e.g., a created element is synchronized after a created node that it uses.
- Global synchronization of rosters for deleting mesh objects must be in the following order: element, face, edge, node. This order ensures that deleted mesh objects are synchronized before mesh objects they use, e.g., a deleted element is synchronized before a deleted node that it uses.
- Global synchronization of rosters for changing attributes of mesh objects may be in any order.

The Inconsistent Transient State

The first local dynamic-mesh-modification operation performed by the application places that mesh in a globally inconsistent state. While in this inconsistent state the “regular” Framework

services, such as support for workset algorithms, will not perform correctly. Global consistency of the mesh must be restored via synchronization before "regular" Framework services can be used.

Global Synchronization: Framework Operations

Global synchronization of dynamic mesh modifications by the SIERRA Framework includes several parallel operations:

- Global synchronization in the creation of mesh objects resolves potentially duplicated mesh objects newly created on processor boundaries, determines unique global identifiers for newly created mesh objects, synchronizes the usage-subset membership of shared mesh objects, and updates the CommSpecs for shared mesh objects.
- Global synchronization in the deletion of mesh objects updates the CommSpec for shared mesh objects that are marked for deletion and then deletes all marked mesh objects in the roster.
- Global synchronization of changes to the attributes of mesh objects takes the global union of shared mesh-object contexts, mechanics instances, and input/output instances.

12.2 Synchronization of Created Mesh Objects

Each newly created mesh object is marked as "pending creation" and is assigned an arbitrary and globally inconsistent identifier on the local processor. A newly created mesh object is assigned to usage and instance subsets, may be connected to other existing mesh objects or to newly created mesh objects, and may have field values set. Global synchronization of pending-create mesh objects must resolve the creation of mesh objects that should be shared among processors, select globally unique identifiers, synchronize usage-subset membership, and update the communication specification for shared mesh objects.

Resolution of Newly Created Shared Mesh Objects

A newly created mesh object that may be shared with one or more other processors is created in one of three states:

1. No other processor has created that mesh object.
2. One or more other processors have also newly created a compatible mesh object.
3. One or more other processors already have an existing compatible mesh object.

Resolution of a potentially shared newly created mesh object first determines whether a compatible mesh object exists on another processor. A mesh object is compatible or represents the same mesh object if it is topologically identical. For example, two face mesh objects are compatible if they are defined by the same nodes (vertex and, if present, mid-edge nodes) and are compatibly attached to elements. For faces in 3D and edges in 2D, this "compatibly attached to elements" clause means that if the face is attached to two different elements, then those elements

must be attached to opposite sides of the face. Evaluation of this element-compatibility condition is complicated in the presence of topologically degenerate element-shells. In this situation the two faces of a shell will have the same nodes but be associated with different sides of the shell.

For nodes, the determination of compatibility is more complex. It is assumed that any dynamically created node that could be shared among processors is associated with the interior of an edge, face, or element. These nodes are referred to as child nodes of the associated mesh object. Newly created nodes are resolved by matching the topological description of their parent mesh objects among processors.

If it is determined that a newly created mesh object already exists on another processor, the newly created mesh object is updated to have the existing global identifier. Otherwise, one processor from the subset of processors that could share the mesh object is designated to coordinate the selection of a global identifier for that mesh object. In either situation, all instances of the shared mesh object are synchronized to have the same usage-subset membership.

Mesh objects that are newly created globally are assigned globally unique identifiers with the following algorithm:

1. The number of globally unique mesh objects needing global identifiers is determined. Note that each newly created shared mesh object has a processor that has been designated to determine the global identifier for that mesh object.

2. The existing set of global identifiers is analyzed to determine if there are any “holes” in the contiguous numbering of the set. For example, the set {1, 2, 3, 5, 6, 8} has holes of {4, 7}. These “holes” in the set of used identifiers are selected for newly created mesh objects before values that are greater than the current maximum value. This selection policy leads to the re-use of global identifiers that had been assigned to previously deleted mesh objects. This selection policy was implemented, as opposed to a simpler policy of selecting the next largest value, so that the identifier would not “overflow” when mesh objects are cyclically created and deleted.

3. Globally unique identifiers are selected by the designated processors and assigned to the newly created mesh objects. These designated processors then inform any processors sharing these mesh objects of the assigned global identifier. Thus all instances of each mesh object are assigned the same global identifier.

4. Finally, the processor-owner of each newly created mesh object is selected, and the “pending creation” marking is removed from all instances of these mesh objects.

12.3 Synchronization of Deleted Mesh Objects

Each processor may mark one or more mesh objects as “pending deletion.” A mesh object that has been marked is not immediately deleted as this would corrupt processor-sharing information. When an application’s mark-for-deletion process is complete, a globally synchronized delete operation is performed. This delete operation first removes the pending-delete mesh objects from the processor-sharing communication specification. If all instances of a mesh object have been

marked as pending deletion, the influence of that mesh object is also removed from the mesh. Finally, the marked instances of a mesh object are deleted.

De-imprinting

A mesh object's "influence" in a mesh is most readily illustrated by the imprinting of an element on its nodes (Figure 4.3). If the elements that have imprinted on a node are deleted, the imprinted contexts are no longer valid and must be removed from the node. This "de-imprinting" is complicated when nodes or other mesh objects are shared between processors and the imprinting elements are not similarly shared. In this case, an imprinting element of a node may be deleted on the local processor, but another processor may have a different element that is not shared and has the same imprinting context.

12.4 Synchronization of Mesh Object Attributes

Shared mesh objects may be independently imprinted on different processors. The results of processor-independent imprinting operations must be synchronized between processors. This synchronization is accomplished by assigning a shared mesh object to each usage and instance subset that is identified on each sharing processor. This assignment is the union of the usage and instance subsets from the sharing processors.

References

1. Taylor, Lee M., Harold Carter Edwards, and James R. Stewart. *Functional Requirements for SIERRA Version 1.0 Beta*. SAND99-2587. Albuquerque, NM: Sandia National Laboratories' Engineering Sciences Center, 1999.

2. Edwards, H. Carter, and James R. Stewart. "SIERRA, a Software Environment for Developing Complex Multiphysics Applications." In *Computational Fluid and Solid Mechanics. Proc. First MIT Conf.,* Cambridge, MA, 2001, edited by K. J. Bathe, 1147–1150. Oxford, UK: Elsevier, 2001.

3. Edwards, H. Carter, James R. Stewart, and John D. Zepper. "Mathematical Abstractions of the SIERRA Computational Mechanics Framework." In *Proc. 5th World Congress Comp. Mech.*, Vienna, Austria, July 2002, edited by H. A. Mang, F. G. Rammerstorfer, and J. Eberhardsteiner.

Appendix A: Mapping to C++ Classes

The theory and design abstractions described in this document are implemented in the C++ programming language, primarily as C++ classes. The mapping from these abstractions to C++ classes implemented in SIERRA Framework Version 3 is given in this appendix. Future versions of the SIERRA Framework may alter this implementation such that the design-to-implementation mapping presented here is not compatible. However, it is anticipated that the theory and design of the core framework will change more slowly than the implementation.

Table A.1. Mapping to SIERRA Framework Version 3

Section	Abstraction	C++ Class
Section 1.1	Procedure	`Fmwk_Procedure`
Section 1.1	Region	`Fmwk_Region`
Section 1.1, Section 2.3, and Section 9.2	Mechanics	`Fmwk_Mechanics`
Section 1.1	Transfer	`Xfer_Transfer`
Section 2.3 and Section 9.3	Mechanics Instance	`Fmwk_MechanicsInstance`
Section 2.4	Mesh Object	`Fmwk_MeshObj`
Section 2.4, Section 11.3	Intramesh Connectivity Relations	`Fmwk_CommSpec`
Section 2.4 and Section 8	Intermesh Connectivity Relations	`Fmwk_MeshObj::Relation`
Section 2.5 and Section 5.1	Field Specification	`Fmwk_DatumVariable`
Section 2.5	Field Type	`Fmwk_DatumSpec`
Section 3.2	Context for Usage Subsets	`Fmwk_Context`
Section 4.1	Master Element	`Elem_MasterElem`
Section 4.1 and Section 8.4	Topology of a Master Element	`Fmwk_MeshObjTopology`
Section 4.4	Master Element Usage	`Fmwk_MeshObjRegistrar::MasterElemUsage`
Section 6	Mesh Object Roster	`Fmwk_MeshObjRoster`
Section 6	Mesh Object Subroster	`Fmwk_MeshObjSubroster`

Table A.1. Mapping to SIERRA Framework Version 3 (Continued)

Section	Abstraction	C++ Class
Section 6.2	Globally Unique Identifier	`Fmwk_Id`
Section 7.1	Buckets of Field Values	`Fmwk_DatumBucket`
Section 10	Worksets	`Fmwk_WorksetRegistrar` `Fmwk_Workset`

Distribution

External

Texas Institutute for Computational and Applied Mathematics
University of Texas at Austin
Austin, TX 78712
Attn: J. Tinsley Oden

Lawrence Livermore National Laboratories
LLNL L-95
P.O. Box 808
Livermore CA 94551
Attn: Evi Dube

Internal

1	MS 0841	9100	T. C. Bickel
1	MS 0835	9140	J. M. McGlaun
5	MS 0835	9141	S. N. Kempka
5	MS 0835	9142	J. S. Peery
10	MS 0827	9143	J. D. Zepper
1	MS 0824	9110	A. C. Ratzel
1	MS 0826	9113	W. L. Hermina,
1	MS 0834	9114	J. E. Johannes
1	MS 0836	9115	E. S. Hertel
1	MS 0847	9120	H. S. Morgan
1	MS 0824	9130	J. L. Moya
1	MS 0828	9133	M. Pilch
1	MS 0847	9211	S. A. Mitchell
1	MS 1110	9214	D. E. Womble
1	MS 0819	9231	E. A. Boucheron
1	MS 0139	9900	M. O. Vahle
1	MS 0835	9141	S. W. Bova
1	MS 0835	9141	R. J. Cochran
1	MS 0835	9141	S. P. Domino
1	MS 0835	9141	M. W. Glass
1	MS 0835	9141	R. R. Lober
1	MS 0835	9141	A. A. Lorber
1	MS 0835	9141	P. A. Sackinger
1	MS 0835	9141	J. H. Strickland

1	MS 0835	9141	S. R. Subia
1	MS 9217	8920	C. J. Aro
1	MS 9042	8728	C. D. Moen
1	MS 0826	9113	D. R. Noble
1	MS 0826	9114	E. S. Piekos
1	MS 0834	9114	M. M. Hopkins
1	MS 0834	9114	P. K. Notz
1	MS 0825	9115	J. L. Payne
1	MS 0838	9116	R. E. Hogan
1	MS 0828	9133	K. J. Dowding
1	MS 0847	9133	W. R. Witkowski
1	MS 0316	9233	C. C. Ober
1	MS 0316	9233	T. M. Smith
1	MS 0316	9233	R. Hooper
1	MS 0847	9142	M. K. Bhardwaj
1	MS 0847	9142	M. L. Blanford
1	MS 0847	9142	A. S. Gullerud
1	MS 0835	9142	J. D. Hales
1	MS 0847	9142	M. W. Heinstein
1	MS 0847	9142	S. W. Key
1	MS 0847	9142	W. S. Klug
1	MS 0847	9142	J. R. Koteras
1	MS 0847	9142	N. K. Crane
1	MS 0847	9142	J. A. Mitchell
1	MS 0835	9142	K. H. Pierson
1	MS 0847	9142	V. L. Porter
1	MS 0847	9142	T. J. Preston
1	MS 0847	9142	G. M. Reese
1	MS 0847	9142	T. F. Walsh
1	MS 0807	9338	B. H. Cole
1	MS 0847	9214	K. F. Alvin
1	MS 9217	9214	M. F. Adams
1	MS 0847	9126	J. Jung
1	MS 9405	8726	R. E. Jones
1	MS 0847	9211	M. S. Eldred
1	MS 0827	9143	K. M. Aragon
1	MS 0827	9143	K. N. Belcourt
1	MS 0827	9143	D. M. Brethauer
1	MS 0827	9143	K. D. Copps
20	MS 0827	9143	H. C. Edwards
1	MS 0827	9143	C. A. Forsythe

1	MS 0827	9143	M. E. Hamilton
1	MS 0827	9143	J. R. Overfelt
1	MS 0827	9143	J. S. Rath
1	MS 0827	9143	G. D. Sjaardema
10	MS 0827	9143	J. R. Stewart
1	MS 0827	8920	A. B. Williams
1	MS 1111	9215	K. D. Devine
1	MS 0819	9231	K. H. Brown
1	MS 0819	9231	K. G. Budge
1	MS 0819	9231	S. P. Burns
1	MS 0819	9231	D. E. Carrol
1	MS 0819	9231	R. R. Drake
1	MS 0847	9226	S. J. Owen
1	MS 9018	8945-1	Central Technical Files
2	MS 0899	9616	Technical Library
1	MS 0612	9621	Review & Approval Desk for DOE/OSTI