INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING
Int. J. Numer. Meth. Engng 2005; **64**:1529–1556
Published online 1 September 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/nme.1440

A compact adjacency-based topological data structure for finite element mesh representation

Waldemar Celes[1], Glaucio H. Paulino[2,*,†] and Rodrigo Espinha[1]

[1]*Tecgraf/PUC-Rio—Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rua Marquês de São Vicente 225, Rio de Janeiro, RJ 22450-900, Brazil*
[2]*Department of Civil and Environmental Engineering, University of Illinois at Urbana-Champaign, Newmark Laboratory, MC-250, 205 North Mathews Avenue, Urbana, IL 61801-2397, U.S.A.*

SUMMARY

This paper presents a novel compact adjacency-based topological data structure for finite element mesh representation. The proposed data structure is designed to support, under the same framework, both two- and three-dimensional meshes, with any type of elements defined by templates of ordered nodes. When compared to other proposals, our data structure reduces the required storage space while being 'complete', in the sense that it preserves the ability to retrieve all topological adjacency relationships in constant time or in time proportional to the number of retrieved entities. *Element* and *node* are the only entities explicitly represented. Other topological entities, which include *facet*, *edge*, and *vertex*, are implicitly represented. In order to simplify accessing topological adjacency relationships, we also define and implicitly represent oriented entities, associated to the use of facets, edges, and vertices by an element. All implicit entities are represented by *concrete types*, being handled as *values*, which avoid usual problems encountered in other reduced data structures when performing operations such as entity enumeration and attribute attachment. We also extend the data structure with the use of 'reverse indices', which improves performance for extracting adjacency relationships while maintaining storage space within reasonable limits. The data structure effectiveness is demonstrated by two different applications: for supporting fragmentation simulation and for supporting volume rendering algorithms.

KEY WORDS: topological data structure; reduced representation; implicit entities; finite element mesh

*Correspondence to: Glaucio H. Paulino, Department of Civil and Environmental Engineering, University of Illinois at Urbana-Champaign, Newmark Laboratory, 205 North Mathews Avenue, Urbana, IL 61801, U.S.A.
†E-mail: paulino@uiuc.edu

Contract/grant sponsor: NASA-Ames; contract/grant number: NAG 2-1424
Contract/grant sponsor: National Science Foundation; contract/grant number: CMS-0115954
Contract/grant sponsor: CAPES

Received 17 January 2005
Accepted 9 March 2005

1. INTRODUCTION

The usual element-node mesh representation consists of a table of nodes and a table of elements, with the corresponding node incidence. This data structure, although very simple and sufficient for several finite element applications, is not adequate in an adaptive analysis environment due to the lack of topological information [1–4]. Indeed, several finite element applications [5–10], such as fragmentation simulation, require a data structure that provides adjacency information among the mesh topological entities.

A topological representation is also required for certain visualization techniques. Direct volume rendering, for instance, based on projective methods, requires a visibility ordering of the cells (elements) in a mesh [11–13]. Visibility ordering algorithms need access to topological information to construct the adjacency graph for the cells in a mesh.

Topological data structures represent a model by means of a set of abstract topological entities. For three-dimensional (3D) models, the main topological entities usually comprise: *region*, *face*, *edge* and *vertex*. If the model represents a finite element mesh, each region corresponds to an element. For two-dimensional (2D) models, there are usually *face*, *edge* and *vertex* (with a face corresponding to an element in a 2D mesh). For a topological representation to be considered sufficient, it must contain enough information to retrieve all adjacency relationships among its topological entities, without any errors or ambiguities [14, 15].

Conversely, the storage requirement for representing topological entities and their adjacency relationships can arise as a crucial problem, especially for the representation of finite element meshes where topological data are more dominant than geometric data [16]. Various data structures have been proposed to represent finite element meshes. In order to select an appropriate representation, we should consider storage space, querying and editing efficiency, and implementation complexity. A data structure that explicitly stores all topological entities, and all the adjacency relationships among them, tends to perform queries very efficiently, but demands high, sometimes prohibitive, storage space (to store all the information). Moreover, editing tasks may demand high computational efforts because several adjacency relationships have to be updated [1–4].

In this paper, we propose a new compact adjacency-based topological data structure for representing finite element meshes. The proposed data structure explicitly represents only two entities: *elements* and *nodes*. Other topological entities, which include *facet*, *edge*, and *vertex*, are implicitly represented. This data structure also defines and implicitly represents oriented entities, associated to the use of facets, edges, and vertices by an element. When compared to other representations, our data structure reduces the required storage space while preserving the ability to retrieve topological adjacency in constant time or in time proportional to the output size, i.e. the time needed to retrieve an adjacency is proportional to the number of retrieved neighbour entities. The proposed data structure was designed to support, under the same framework, both 2D and 3D meshes, with elements of any order (T3, T6, Q4, Q8, Tetra4, Tetra10, Hexa8, etc.). It can be extended to support any type of element, as long as the element is defined by templates of ordered nodes, which include, for example, all Lagrangian-type elements.

We assume that the mesh domain is *manifold*—however, the mesh itself is non-manifold [17]. This means that the external boundary of a 3D mesh must have 2-manifold topology; therefore, each edge on the boundary is shared by exactly two boundary faces. Accordingly, for 2D models, the external boundary must have 1-manifold topology, with each vertex on the

boundary having exactly two boundary edges connected to it. On the other hand, if the entities in the interior of the model are considered, for 3D models, an edge represents a non-manifold condition since it is shared by more than two facets (and similarly for a vertex in 2D models). An extension to handle non-manifold domains, although not straightforward, is possible and is presently under consideration.

The remainder of this paper is organized as follows: Section 2 describes previous work on topological data structures for representing various models. Section 3 describes in detail the compact adjacency-based data structure that is introduced for finite element mesh representation. Section 4 presents an analysis of the proposed data structure and, in Section 5, some concluding remarks are drawn.

2. RELATED WORK

Several researches have been conducted on the development and implementation of topological data structures for representing various models. A significant proposal for representing non-manifold objects is the radial-edge data structure, presented by Weiler [14, 15], which stores the radial ordering of faces around an edge. An important concept of the radial-edge data structure is the distinction between a non-oriented entity (e.g. an edge) and the oriented use of that entity (e.g. an edge-use). The oriented entities represent the use of a specific topological entity in an adjacency relationship, such as the use of an edge composing the boundary of a face. Weiler [14, 15] has shown that representing the use of an entity is unambiguous and produces more straightforward access algorithms. A well known and widely used variation of that data structure is Mäntylä's half-edge data structure, limited to manifold objects [17].

The radial-edge data structure has been specialized by McMains *et al.* [18] to polygonal meshes. They proposed a data structure (called loop edge-use data structure) with constant storage space for each vertex and edge-use. The set of edge-uses incident to a vertex is retrieved by following single fields (references to next edge-uses of the given vertex) stored in the edge-uses.

Lee and Lee [16, 19] have presented a more compact data structure, called partial entity structure, as a variation of the radial-edge data structure. Their representation replaces the entity-uses with partial topological entities, which are light oriented entities used to describe the non-manifold conditions at vertices, edges, and faces. For the sake of conciseness, Silva and Gomes [20] have opted for not representing oriented entities at all. An entity's orientation is computed by consistently inducing a topological orientation. The result is a very compact data structure for polygonal meshes designed for the efficient management of multi-resolution meshes.

De Floriani *et al.* [21] have introduced the triangle-segment data structure for representing non-manifold, non-regular triangle meshes. Their data structure only explicitly represents triangles and vertices, defining four adjacency relationships between these two entities. The triangle–triangle relation implicitly provides a cyclic list of all triangles around a given edge (not explicitly represented). A vertex stores a reference to one triangle, selected arbitrarily, incident to it. Based on this reference, one can retrieve all other adjacent triangles.

Previous works have directly applied data structures for solid modelling in the development of an integrated process of fracture simulation. Wawrzynek and Ingraffea [22] have used the winged-edge data structure for 2D models while Martha *et al.* [23] have used the radial-edge

data structure for representing 3D models. By using the same topological data structure for representing both the geometric model and the finite element mesh, they have obtained a highly integrated framework where mesh entities are related to the corresponding geometric entities, which leads to a *mesh classification* relationship [1]. However, the use of a general topological data structure for finite element mesh representation imposes a prohibitive cost of storage space for large models.

The finite element method imposes several constraints on how to make the domain discretization by a set of finite elements. A data structure for representing finite element meshes can rely on such constraints to achieve a more compact and adequate topological data structure. Some topological data structures have been specifically designed for attending the needs of particular mesh generation algorithms [24, 25] and analysis applications [26]. Beall and Shephard [1] have discussed the need of topology-based mesh representations for finite element applications in a more general form and presented three different proposals. They considered the representation of 3D meshes, composed by regions, faces, edges, and vertices. In the first proposal, all downward one-level adjacencies are explicitly stored: each region (element) has a list of bounding faces; each face has a list of bounding edges, and so on. This data structure also stores the upward adjacencies: each vertex (node) has a list of incident edges; each edge has a list of incident faces, and so on. This full representation is capable of retrieving the adjacency relationships efficiently, but imposes a high cost on storage space. Maintaining the data structure's integrity is also not simple due to the large number of references that must be kept consistent. Another proposed implementation, the 'circular adjacency representation,' keeps the downward adjacencies but discards the upward one-level adjacencies. Instead, each vertex has a list of all incident regions. The other upward one-level adjacencies can then be derived. Beall and Shephard [1] have also presented a reduced data structure, where the internal faces and edges are not stored explicitly. These entities are represented implicitly and are created when necessary. In general, the use of reduced data structures is very interesting, but there are a few caveats. The main problem consists of ensuring the consistency of implicit entities. An operation that modifies the model may change or invalidate implicit entities.

Garimella [2] has presented a comparison among different data structures for mesh representation, from complete ones to several flavours of reduced ones. The comparison is based on storage space and computational efficiency to perform topological operations. More recently, Remacle *et al.* [3, 4] have proposed a different approach in which, instead of using a particular data structure, they have presented an algorithm-oriented mesh database. Their proposal uses a dynamic mesh representation by extracting from a database an appropriate representation for specific needs whenever necessary. Recently, a mesh-oriented database, called MOAB, has also been presented for creating, storing and accessing finite element mesh data [27]. Pandolfi and Ortiz [9] have proposed a topological data structure for quadratic tetrahedral mesh to support fragmentation simulation. They have opted for using a redundant topological data, thus ending up with a structure inadequate for large models.

Cignoni *et al.* [28] have proposed an adjacency-based data structure for edge-based multiresolution of tetrahedral meshes. For representing a mesh level, they proposed a data structure where only tetrahedra and vertices are explicitly stored. Each vertex keeps a pointer to one tetrahedron incident to it, and each tetrahedron keeps pointers to its four vertices, pointers to the adjacent tetrahedra, and reverse indices providing the relative position of the tetrahedron with respect to each of the other adjacent tetrahedra.

We propose a new reduced adjacency-based topological data structure for representing different types of finite element meshes. Our data structure explicitly stores elements and nodes, in a way similar to Cignoni *et al.* [28] who have stored tetrahedra and vertices. We extend the representation to finite elements including any element type defined by templates of ordered nodes. Our data structure implicitly represents facets, edges, and vertices. In addition, we define oriented entities, representing the use of facets, edges, and vertices by the elements. As stated by Weiler [14, 15], the representation of oriented entities greatly simplifies accessing topological adjacency relationships. In our data structure, as these oriented entities are implicitly represented, they do not impose any additional storage requirements.

3. REDUCED ADJACENCY-BASED DATA STRUCTURE

For finite element applications, *node* (*N*) and *element* (*E*) are the two most important topological entities. The mesh itself is described by means of nodes and elements, and even a simple finite element simulation needs access to both fundamental entities. For that reason, in the proposed data structure, these two entities are the only ones that are *explicitly* represented, and for conciseness, all other topological entities are *implicitly* represented.

Besides node and element, the data structure defines and implicitly represents *facet* (f), *edge* (e), and *vertex* (v). By definition, a facet represents an interface between two elements, or the interface between an element and the domain boundary. For 3D models, a facet corresponds to an entity of dimension 2, while, for 2D models, it corresponds to an entity of dimension 1. Despite the model's dimension, each facet has a set of bounding edges (for 2D models, there is only one edge for each facet). An edge is bounded by two vertices, and a vertex represents an element-corner node. In our data structure, for quadratic or higher-order elements, there are no vertices corresponding to the mid-side nodes.

One can argue that the existence of both facet and edge for 2D models is redundant; however, the use of this set of topological entities has brought important benefits for the design and implementation of the data structure. It has allowed the conception of a unified framework for 2D and 3D models: the same algorithm (i.e. *same code*) can be applied to any model, despite its dimension. As an example, in Section 4.3, we use one unique procedure, for both 2D and 3D models, to insert cohesive elements along bulk element interfaces for fragmentation simulation. It is important to note that such redundancy does not impose any additional memory cost because the entities are implicitly represented.

Here we define the concepts of *entity-uses* (as oriented entities) and *entity-mates*. An oriented entity, namely *facet-use* (fu), *edge-use* (eu), or *vertex-use* (vu), represents the use of a topological entity in the adjacent group of an element-based adjacency relationship. Thus, a facet-use, an edge-use, or a vertex-use represents the use of a facet, an edge, or a vertex by an element, respectively. The concepts of entity and entity-use are related to the concept of *entity-mate*: given an entity-use, mates are all the other uses that have the same associated entity. We note that each facet has two associated facet-uses, one for each interfacing element (see Figure 1), except for the boundary facets, which have only one associated use. For 3D models, each edge has a radially ordered group of edge-uses, one for each incident element, and each vertex has an unordered group of vertex-uses, also one for each incident element. Accordingly, for 2D models, each edge has two associated edge-uses and each vertex has a radially ordered group of vertex-uses.

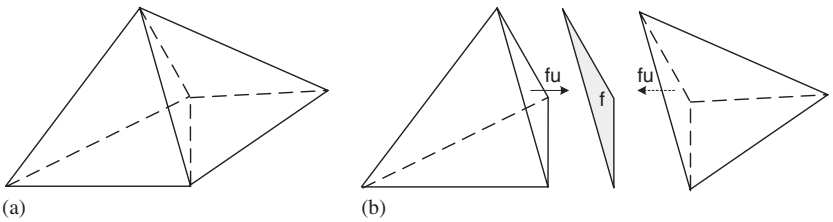
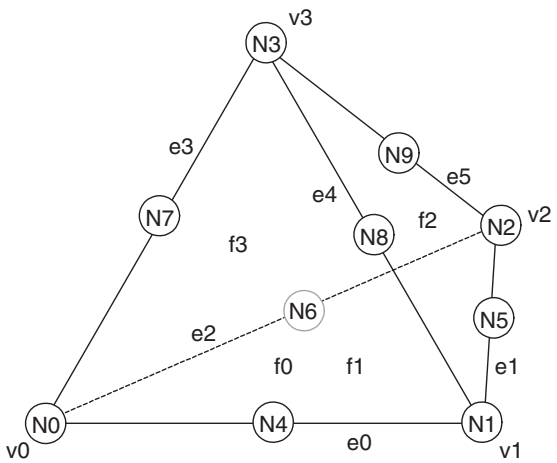


Figure 1. Entity and entity-uses exemplified by the facet entity for a 3D configuration. Each facet (non-oriented entity) has two associated facet-uses (oriented entities), one for each interfacing element: (a) two interfacing tetrahedra; and (b) illustrative separation between the two elements showing the facet and its associated facet-uses.




```
f0: <v0,v1,v3>            e0: [v0,v1]      v0: {v1,v2,v3}
f0: <e0,e4,e3>            e0: [e4,e2]      v0: {e0,e2,e3}
f0: <f1,f2,f3>            e0: [f0,f1]      v0: {f0,f1,f3}
f0: <N0,N4,N1,N8,N3,N7>   e0: [N0,N4,N1]   v0: {N0}
```

Figure 2. Element template for quadratic tetrahedral element. For each facet (f), edge (e), and vertex (v) of an isolated element, the template provides access to the adjacent vertices (v), edges (e), facets (f), and nodes (N). For instance, facet $f0$ is bounded by vertices $v0$, $v1$, and $v3$; edge $e0$ is bounded by vertices $v0$ and $v1$; and vertex $v0$ is adjacent to vertices $v1$, $v2$, and $v3$. Similar relationships are inferred with respect to edges, facets, and nodes.

Each finite element in isolation is composed by a set of local facets, edges, and vertices. Each local entity is labelled by an 'id' (identification number). The local topology of each type of finite element is known in advance and defines the *element template* [1, 3, 4]. Therefore, within an element, we have direct access to all adjacency relationships relating local topological entities. These local topological entities are mapped to the corresponding entity-uses of an element. Figure 2, shows a template for a quadratic tetrahedral element and illustrates the local adjacency relationships for a few topological entities.

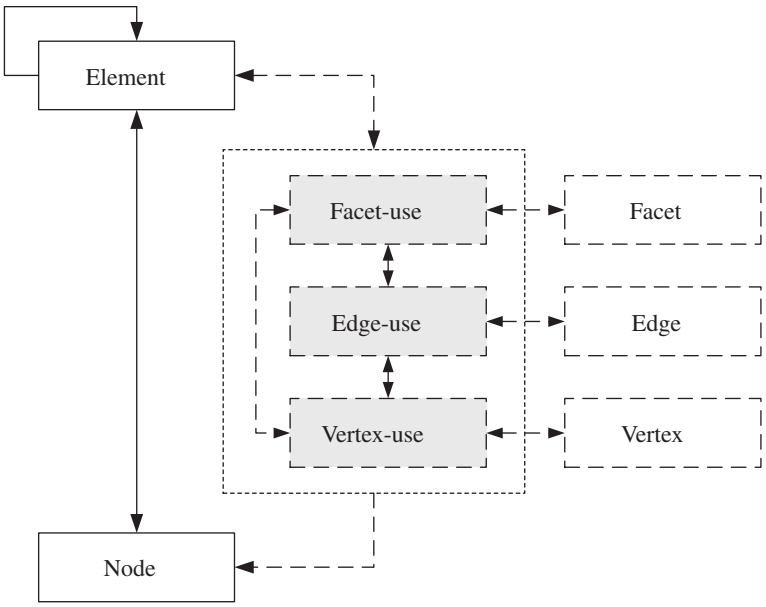


Figure 3. Simplified diagram of the novel data structure. Solid lines represent explicit entities and dashed lines represent implicit entities. Solid arrows represent explicit references, i.e. the ones that are directly stored; and dashed arrows represent implicit references, i.e. the ones that are derived based on element templates.

3.1. A novel representation of topological entities

The schematic diagram, shown in Figure 3, illustrates entity representation in the data structure. For each element, we store *references* to its nodes and *references* to the adjacent elements that share a facet with the current element. For each node, we store its co-ordinates and a *reference* to an incident element. This representation has the advantages of being very compact and of having constant space storage for nodes and elements of the same type. Therefore, if we have a model with all elements of the same type, we can store the nodes and elements in arrays and can use indices to reference them. In this work, we have opted for implementing elements and nodes as abstract types, with each actual finite element (T3, T6, Tetra4, etc.) representing a specialization of the element abstract type. The entities are arranged in double linked lists for easy inclusion and removal operations. Thus, in this particular implementation, the *references* are made by means of pointers.

Facet, edge, vertex and their associated uses are retrieved, as required, on-the-fly. Their representations have short lifetime, because mesh editing operation may invalidate implicit entities. We have therefore opted for representing these short-living entities as *concrete types*. A concrete type is similar to the built-in types of a programming language because they are treated as values [29]. Provided that the storage spaces used by these entities are small, this approach is quite appropriate because we can manipulate the entities without invoking dynamic memory allocation.

Each entity-use (facet-use, edge-use, or vertex-use) is represented by a pointer to the associated element (E_i) and the local 'id' that identifies the corresponding local entity within the element, (E_i, id). Facets, edges, and vertices are represented by one of its uses, as follows:

- Facet (f) is represented by anyone of its two facet-uses (fu);
- Edge (e) is represented by the edge-use (eu) associated with an incident element;
- Vertex (v) is represented by the vertex-use (vu) associated with the element pointed by the corresponding corner node.

For facilitating the implementation of editing and querying functions, for quadratic (or higher order) edges, the representative edge-use is the one associated with the element pointed to by its mid-side nodes. For linear edges, the chosen edge-use is anyone among all uses of the edge.

For uniquely representing each implicit entity and for easing their enumeration, within the element representation, we also store *bit flags* that indicate which facets, edges, and vertices reference the given element (more precisely, reference an use of the given element). Thus, given an element, we easily extract the representations of the implicit entities that reference it. Also, given a node, we easily have access to the corresponding vertex (for corner node) or edge (for mid-side node).

Previous works have represented implicit faces and edges by their bounding nodes [1–4], but those representations are ambiguous for certain analysis applications. A fragmentation simulation, for instance, needs to duplicate an edge, without duplicating their bounding nodes, in order to insert a cohesive element [7–9], thus ending up with two distinct edges with the same bounding nodes. By representing the edges by corresponding edge-uses, the proposed data structure naturally handles such occurrences. Within the cohesive finite element, we have distinct edge-uses (the local 'id' is different) and, consequently, distinct representation for the edges.

3.2. The role of entity-uses

An important concept of the proposed data structure is the introduction of oriented entities for mesh representation, thus distinguishing between a non-oriented entity (e.g. a facet) and the oriented use of that entity (e.g. a facet-use). In the context of the proposed data structure, the oriented entities represent the use of a specific topological entity by an element. The entity-uses play an important role in the design, implementation and application of the proposed data structure:

- In virtue of entity-uses, each element is handled in isolation and has complete local description of its boundaries. The local topological entities are directly mapped to the corresponding entity-uses. As a consequence, the entity-uses have natural implicit representation, based on the element templates.
- As remarked previously, facet, edge, and vertex are defined as non-oriented entities. This definition greatly simplifies implicit-entity management because one does not have to keep track of orientation changes when the mesh undergoes modifications.
- Representing the entity-uses has produced more straightforward access algorithms. In order to retrieve all elements adjacent to a facet, edge or vertex, it suffices to retrieve all the associated uses.
- The client applications (e.g. finite element analyses) can choose the best topological entity where to store specific attributes. For instance, a natural choice for attaching the normal

vector of an element's facet is the facet-use, because this orientation is intrinsically related to the use of the facet by the corresponding element. In the absence of a facet-use, one would have to attach the normal vector to the actual facet, together with an additional attribute to make unambiguous the element to which the normal orientation is referred to.

3.3. Implicit entity issues

Although facets, edges, and vertices are not explicitly represented, the client applications have to be able to access them in a transparent way. A first issue regarding implicit entities is related to the problem of entity enumeration. The application should be able to efficiently enumerate implicit entities. The proposed data structure is capable of traversing all implicit entities in time proportional to the number of elements. For enumerating all facets, edges, or vertices without duplication, we traverse all the elements and, for each element, we collect all facets, edges, or vertices which reference the current element. This is done by checking the bit flags described above.

Beall and Shephard [1] have pointed out other potential problems related to handling implicitly represented entities. One major problem is that modifying the mesh may change the representation of an entity that was not directly modified. In our data structure, the implicit entities remain valid while their referenced elements remain valid. In order to keep track of the lifetimes of elements, we can add a reference counter to their representations. Then, each time a representation of an implicit entity is retrieved (or released), we can increase (or decrease) the corresponding counters. By doing that, instead of deleting removed elements, we can indicate they are no longer valid. Each time an implicit entity is accessed, we can check if it remains valid (i.e. if the referenced elements remain valid).

The need of associating data with implicitly represented entities is another major problem. It is not possible to store data in implicit entities. In order to overcome this limitation, we have two options. A first option consists in using the referenced entities (elements or nodes, for vertices) to store the data, in a way similar to that proposed by Beall and Shephard [1]. As a second option, we can use a hash table in order to associate implicit entities (key values) with their corresponding data. This second option is quite appropriate for the proposed data structure since implicit entities are represented by values (concrete types).

3.4. Adjacency relationships

In order to express the adjacency relationships, we have adopted a terminology similar to the one used by Weiler [14, 15]: {.} means unordered group, [.] ordered group, and ⟨.⟩ cyclically or radially ordered group. There are a total of 25 adjacency relationships relating the 5 represented entities (element, `E`; node, `N`; facet, `f`; edge, `e`; vertex, `v`) in the data structure. We have classified these relationships in four different groups.

Group 1: The first group comprises the four relations regarding only nodes and elements:

- `E[N]`: associates an element with its defining nodes; the resulting group of nodes is ordered with respect to the element's incidence. This is the basic adjacency relationship in a finite element mesh.
- `E[E]`: associates an element with all its adjacent elements; two elements are considered adjacent if they share a common facet. The resulting group is ordered according to the local facet numbering.

- `N{E}`: associates a node with all its incident elements; the resulting group has no specific order.
- `N{N}`: associates a node with all nodes of its incident elements; the resulting group has no specific order.

Group 2: The second group relates the adjacency among implicit entities and explicit entities. The six defined relations are:

- `f[E]`: associates a facet with its two ordered interfacing elements.
- `f⟨N⟩`: associates a facet with all its bounding nodes, cyclically ordered.
- `e⟨E⟩`: associates an edge with all its incident elements, radially ordered.
- `e[N]`: associates an edge with its two (or more, for quadratic or higher order edges) ordered bounding nodes.
- `v{E}`: associates a vertex with its unordered group of all incident elements.
- `v{N}`: associates a vertex with its corresponding node.

Group 3: The third group relates relationships in the reverse order, from explicit to implicit entities. The six defined relations are:

- `E[f]`: associates an element with its ordered group of bounding facets.
- `E[e]`: associates an element with its ordered group of bounding edges.
- `E[v]`: associates an element with its ordered group of bounding vertices.
- `N{f}`: associates a node with its unordered group of all incident facets.
- `N{e}`: associates a node with its unordered group of all incident edges (for mid-side nodes, there is only one edge).
- `N{v}`: associates a node with its corresponding vertex.

Group 4: Finally, the fourth group includes relationships that involve only implicit entities. We have defined these nine relations in a way similar to Weiler [14, 15]:

- `f{f}`: associates a facet with the unordered group of all adjacent facets that share an edge with the referenced facet.
- `f⟨e⟩`: associates a facet with its cyclically ordered group of bounding edges.
- `f⟨v⟩`: associates a facet with its cyclically ordered group of bounding vertices.
- `e⟨f⟩`: associates an edge with its radially ordered group of incident facets.
- `e{e}`: associates an edge with an unordered group of all edges that bound an adjacent facet and share a vertex with the referenced edge.
- `e[v]`: associates an edge with its two ordered bounding vertices.
- `v{f}`: associates a vertex with its unordered group of all incident faces.
- `v{e}`: associates a vertex with its unordered group of all incident edges.
- `v{v}`: associates a vertex with its unordered group of vertices that represent the other end of the incident edges.

Remark
The adjacency relationships `N{E}`, `N{N}`, `N{f}`, `N{e}`, `v{E}`, `v{f}`, `v{e}`, and `v{v}` result in a cyclically ordered group for 2D models. In this case, this cyclic order can be naturally obtained from the data structure.

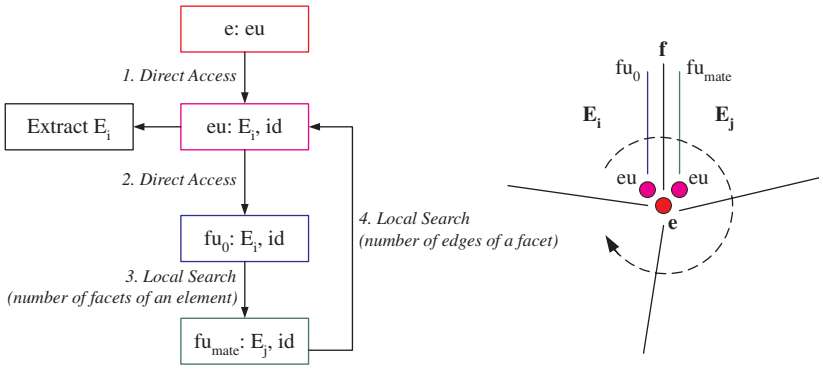


Figure 4. Procedure to retrieve relation e⟨E⟩ (elements adjacent to an edge). The colours in the diagram (left) correspond to those in the cross-section figure (right). The numbers 1–4 refer to the steps in the retrieval procedure.

3.5. Data structure completeness

A data structure is considered *complete* if all adjacency relationships can be derived from the stored data [14, 15]. We now demonstrate that the proposed data structure is complete.

Relations E[N] and E[E] are directly stored in the data structure. In order to retrieve the relation N{E}, we identify two different cases: (1) if N corresponds to a corner node, deriving this relation is the same as deriving relation v{E}; (2) if N corresponds to a mid-side node, it is equivalent to relation e⟨E⟩. Based on N{E} and E[N], we can derive N{N}: it suffices to avoid outputting duplicated nodes, which can be done by masking visited nodes. Relations E[f], E[e], and E[v] are directly provided by the element templates: given an element, we can enumerate its corresponding entity-uses (fu: facet-use, eu: edge-use, or vu: vertex-use) and, for each use, we have to search, among the entity-mates, the one used to represent the entity (facet, edge, or vertex). More precisely, we have to find which element is referenced by the entity, thus being equivalent to extracting f[E], e⟨E⟩, or v{E}. Relations N{f}, N{e}, and N{v} can be derived based on N{E} and on E[f], E[e], and E[v], respectively.

Relation f[E] is directly retrieved. From the facet representation we have access to the referenced element and then, using the local 'id', to the adjacent element. Relations f⟨N⟩, e[N], and v{N} are retrieved by first accessing the associated use (facet-use, edge-use, or vertex-use) from the entity representation and then obtaining the bounding nodes given by the element templates.

Relations e⟨E⟩ and v{E} deserve more attention because, as mentioned above, they are used to derive N{E}, and consequently N{N}, which are important adjacency relationships for finite element applications, such as fragmentation simulations [7, 9]. Relations e⟨E⟩ and v{E} are illustrated in Figures 4 and 5, respectively. Retrieving all elements around an edge (e⟨E⟩) corresponds to finding out all edge-uses associated with the given edge. From an edge, we can access a first edge-use from its representation. Given an edge-use, we can get the next edge-use. This is done by accessing an adjacent facet-use (from the element's template) and finding the corresponding facet-mate in the adjacent element (by means of a local search in

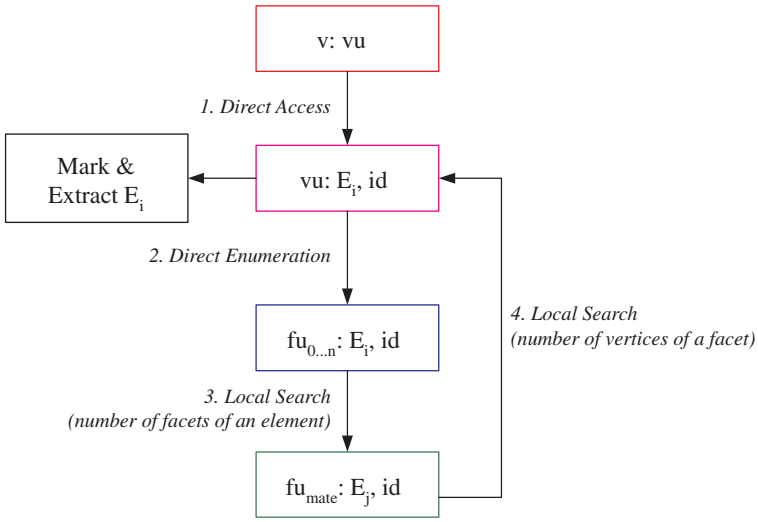


Figure 5. Procedure to retrieve relation `v{E}` (elements adjacent to a vertex). Similarly to Figure 4, the colours are associated to the procedure steps. The numbers 1–4 refer to the steps in the retrieval procedure.

the adjacent element's facets). Then, in order to find the next edge-use, we perform another local search among the edge-uses that bound the facet-mate. This procedure is repeated until the original element (i.e. the first retrieved element) is again reached, resulting in a radially ordered set of elements for 3D models. Figure 4 schematically illustrates this procedure, along with a cross-section of four elements, radially ordered, sharing a common edge. However, if the edge is not an interior edge, instead of reaching the original element again, the procedure will reach the domain's boundary. In that case, in order to retrieve the remaining adjacent elements, we have to go back to the original element and continue the search by accessing the other facet-use adjacent to the first edge-use.

Retrieving all elements around a vertex (`v{E}`) is similar and corresponds to finding all vertex-uses associated to a vertex. In this case, we need to mask the outputted elements in order to avoid duplications. From the given vertex, we access a first vertex-use (from its representation). We then enumerate all adjacent facet-uses (via element template) for accessing the facet-mates. For each reached facet-mate, we have to perform a local search in order to find the corresponding vertex-use. This procedure is repeated until all adjacent elements are visited, as illustrated in Figure 5.

Finally, we can show that the adjacency relationships involving implicit entities are also extracted from the stored data. Relation `f{f}` is derived based on `f⟨e⟩` and `e⟨f⟩`. Relation `f⟨e⟩` is extracted using the element templates. Given a facet, we access one of its uses and then get all the adjacent edge-uses by means of the template. For each edge-use, we get the associated edge. Relation `f⟨v⟩` is retrieved in a similar way: we get a facet-use, then the set of vertex-uses and then the associated vertices. For retrieving `e⟨f⟩`, we first find all edge-uses of an edge and then, via template, get the adjacent facet-uses. Relation `e{e}` is

derived based on e⟨f⟩ and f⟨e⟩. Relation e[v] is extracted based on element templates, in a way similar to f⟨e⟩. Extracting v{f} and v{e} corresponds to accessing all the vertex-uses associated to the given vertex and then getting, respectively, the adjacent facet-uses and edge-uses of each vertex-use, by means of the template. At last, v{v} is derived based on v{e} and e[v].

3.6. Extended representation with reverse indices

As described in the previous section, all defined adjacency relationships can be extracted in a time proportional to the number of retrieved entities. However, some procedures require local searches among the entity-uses within an element. In order to improve the efficiency on how to retrieve these relationships, we have extended the element representation to store reverse indices that allow direct access to entity-mates (facet, edge, or vertex).

Reverse indices were also used in the adjacency-based data structure proposed by Cignoni *et al.* [28]. At each element, we can store, for each element's facet-use, the local position (numeric 'id') of the corresponding facet-mate with respect to each adjacent element. Thereby, given a facet-use, we have direct access to its mate, thus avoiding the local search among all facet-uses of the adjacent element.

In order to stress performance gain, we propose to extend the use of reverse indices by also storing, for each local facet within an element, the position where the first facet's vertex is mapped into the corresponding facet-mate. Figure 6 illustrates these reverse indices.

In Figure 6, we have two interfacing elements: the third facet (f2, indexing begins with zero) of the left element interfaces the second facet (f1) of the right element. Associated with the third facet of the left element, there are two indices stored: {1, 2}. The first index, 1, indicates that the corresponding facet-mate is the second facet (f1) of the right element. This allows direct access to the facet-mate. The second index, 2, indicates that the first vertex of facet f2 (node 11 from the template) appears as the third vertex of the corresponding facet-mate (f1 of the right element). This second index plays a role similar to the reference to the next edge-use stored in the data structure proposed by McMains *et al.* [18], in a way that, using the element template, it allows, from an edge-use or vertex-use, direct access to the next edge-use or vertex-use, respectively. Conversely, associated with the second facet of the element on the right, we have stored the indices {2, 2}, providing access to the entity-mates in the element on the left.

We can thus extend each element representation by adding, for each local facet, reverse indices with respect to the adjacent element in the following manner:

- one facet-mate index;
- one vertex position within the facet-mate.

The explicit representation of reverse indices does not impose a severe penalty on the storage space because these indices represent: a local facet 'id' within an element, and a local vertex 'id' within the facet. Therefore, each index can be stored in 4 bits, ranging from 0 to 15 (e.g. the hexahedral element has 6 facets and each facet has 4 vertices). We then use an additional one byte per facet to store the reverse indices. Each element then stores, besides the incident nodes and the adjacent elements, the bit flags indicating which implicit entities make reference to the element and the reverse indices. The code excerpts in Appendix A illustrate the representation of different types of elements.

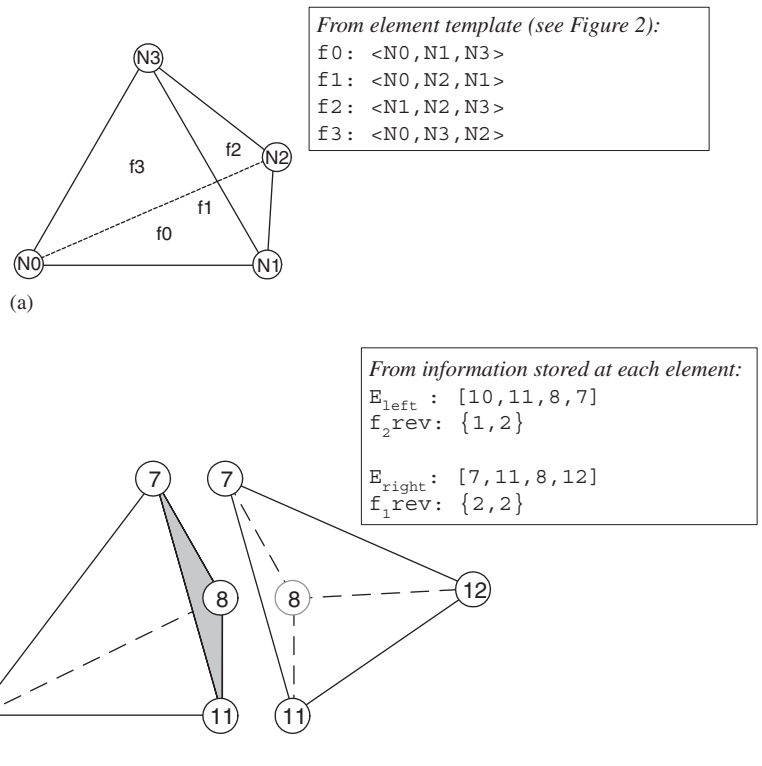


Figure 6. Extended element representation with 'reverse indices:' (a) excerpt of element template for linear tetrahedron; and (b) two interfacing element for a 3D mesh with their corresponding reverse indices. Besides the nodal incidence for each element, there are two stored indices. The first index indicates the corresponding facet-mate in the adjacent element. The second index indicates where the first vertex of the current facet appears in the corresponding facet-mate. For instance, the facet `f2` of the left element (in grey) interfaces with the facet `f1` of the right element (first index). The first node of `f2` on the left element (node 11) appears as the third (indexing begins from zero) node in `f1` on the right element (second index).

4. DATA STRUCTURE ANALYSIS

The proposed data structure is complete (see Section 3.5) in the sense that it is possible to retrieve from it all adjacency relationships relating topological entities. We will now analyse the data structure regarding the required storage space and computational effort for retrieving topological information. We also illustrate the use of the proposed data structure in two different applications: one for fragmentation simulation and one for volumetric visualization.

4.1. Storage space

Garimella [2] presented a methodology for calculating the storage cost of mesh representation and analysed 10 different mesh representations for both linear tetrahedral and hexahedral meshes.

Table I. The storage requirements per node of the data structures in numbers of entities ($\mathscr{E}$) and numbers of connections ($\mathscr{C}$).

Mesh representation	Classical Elem-node	Garimella's [2]		Proposed data structure	
		Reduced	Full	Normal	Extended
Tetrahedral	$6\mathscr{E} + 20\mathscr{C}$	$6\mathscr{E} + 43\mathscr{C}$	$25\mathscr{E} + 143\mathscr{C}$	$6\mathscr{E} + 46\mathscr{C}$	$6\mathscr{E} + 51\mathscr{C}$
Hexahedral	$2\mathscr{E} + 8\mathscr{C}$	$2\mathscr{E} + 16\mathscr{C}$	$8\mathscr{E} + 48\mathscr{C}$	$2\mathscr{E} + 16\mathscr{C}$	$2\mathscr{E} + 18\mathscr{C}$

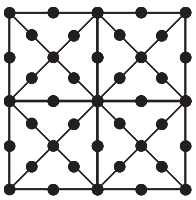
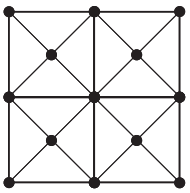
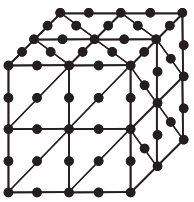
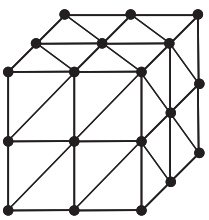
For storage space comparison, we have opted to use the metric proposed by Garimella [2], which accounts for the number of represented entities ($\mathscr{E}$) and the number of stored connections ($\mathscr{C}$), i.e. references to other entities. Often, each connection uses the space of a single pointer (a word, 32 bits in general). For this comparison, the term $\mathscr{C}$ accounts only for the basic adjacency information, not including the pointers needed to arrange the entities in linked lists (as mentioned, arrays could be used in the place of lists). The bit flags are also stored in 32 bits (the hexahedral element requires 26 bits: 6 for facets, 12 for edges, and 8 for vertices). For the extended version with reverse indices, we need an extra byte for each facet. Due to memory alignment, the effective size of the element representation is then extended to be word-aligned. Thus, for the data structure with reverse index, we assume two extra connections are needed for storing the indices of the 6 facets of a hexahedron. Therefore, without reverse indices, we need an additional connection for storing the bit flags, for both tetrahedral and hexahedral meshes. Besides that, for the extended version, we need one or two extra connections for storing the indices, for tetrahedral or hexahedral meshes, respectively. Table I compares the proposed data structures (the normal version and the extended version with reverse indices) with the classical element-node mesh representation (which only stores the incidence of each element), and two data structures selected from Garimella [2]: the most complete one, which was originally proposed by Beall and Shephard [1], and the most reduced one, where only regions and vertices are explicitly stored. The storage requirement is shown in a *per-node* basis, using the estimates presented by Beall and Shephard [1], for the relative number of different entities in tetrahedral and hexahedral meshes with linear elements. Thus, for a tetrahedral mesh with n nodes, we consider the existence of $5n$ elements in average; similarly, for a hexahedral mesh, we consider n elements.

The proposed data structure, without the reverse indices, requires storage space comparable to the most reduced topological data structures for finite element mesh representation presented by Garimella [2], while preserving the ability to directly retrieve all adjacency information. By extending the representation with reverse indices, we achieve a more efficient data structure, while maintaining the storage space within reasonable limits.

4.2. Computational efficiency

The data structure has been implemented using C++. In order to test the data structure performance, we have measured the elapsed time to extract topological information on four different data sets. For reference, the tests were done using the gcc 3.2 compiler with a Linux kernel 2.4 operating system, running on an Intel Pentium 4, 2.53 GHz. Table II presents the descriptions

Table II. Description of data sets used on performance tests.

Data set	Description	Mesh pattern
T6_256 × 256 262 144 elements 525 313 nodes 393 728 facets 393 728 edges 131 585 vertices	2D regular grid decomposed into four quadratic triangular elements per square.	
T3_256 × 256 262 144 elements 131 585 nodes 393 728 facets 393 728 edges 131 585 vertices	2D regular grid decomposed into four linear triangular elements per square.	
T*ET*10_32 × 32 × 32 196 608 elements 274 625 nodes 399 360 facets 238 688 edges 35 937 vertices	3D regular grid decomposed into six quadratic tetrahedral elements per voxel.	
NASA's Blunt Fin [30] 224 874 elements 40 960 nodes 456 506 facets 272 591 edges 40 960 vertices	3D regular grid decomposed into six linear tetrahedral elements per voxel.	

of the data sets used on the computational tests. The reported elapsed times were taken by averaging a large number of simulations (e.g. around 30).

We have first tested the performance to traverse all the entities represented by the data structure: element, node, facet, edge, and vertex. The goal of this test is to measure the time needed to enumerate the implicit entities compared to the time to enumerate elements and nodes, which are explicitly represented. It is important to note that each time an implicit entity is accessed, its corresponding value (concrete type) is returned. Thus, enumerating all entities is an operation that does not impose any memory allocation or entity creation. Conceptually, the entities, although not explicitly stored, always exist. When required, the entity representations are returned. As expected, the experimental results have demonstrated that the time needed to traverse all the facets, edges, or vertices is proportional to the time needed to traverse all the elements, as shown in Table III.

Table III. Elapsed time (in seconds) on entity enumerations. Each time reported is the average of a large number of simulations for each specific task.

Data set	Elements	Nodes	Facets	Edges	Vertices
T6_256 × 256	0.023	0.024	0.023	0.023	0.022
T3_256 × 256	0.019	0.006	0.020	0.020	0.019
TET10_32×32 × 32	0.032	0.013	0.039	0.040	0.038
NASA's Blunt Fin	0.018	0.002	0.019	0.019	0.018

Table IV. Elapsed time (in seconds) on topological information retrievals. The extended representation uses the concept of reverse indices. Each time reported is the average of a large number of simulations for each specific task.

	Representation w/o reverse indices		Extended representation		Gain of extended representation (%)	
Data set	e⟨E⟩	v{E}	e⟨E⟩	v{E}	e⟨E⟩	v{E}
T6_256×256	0.13	0.15	0.11	0.14	8	8
T3_256×256	0.10	0.16	0.09	0.14	8	8
TET10_32×32 × 32	0.31	0.28	0.23	0.24	27	12
NASA's Blunt Fin	0.32	0.30	0.20	0.26	37	14

In order to test the performance of the data structure on extracting topological information, we have considered two topological operations: to retrieve all elements adjacent to a given edge and to retrieve all elements adjacent to a given vertex. To measure the elapsed time to retrieve such information, all edges (or vertices) of the model are traversed and, for each edge (or vertex), the adjacent elements are retrieved. Table IV shows the measured times to retrieve the adjacent elements of all edges or vertices in the models investigated. The elapsed time for retrieving a single adjacency relationship is negligible. As shown in Table IV, for the Blunt Fin data set [30] with the reverse indices, it took 0.20 s to retrieve 272 591 e⟨E⟩ adjacency relationships and 0.26s to retrieve all 40960v{E} relationships. As can be noted, for 3D models, the use of reverse indices has brought a gain of about 30% in performance for retrieving the adjacency e⟨E⟩. For retrieving the adjacency v{E}, the gain decreases to 12%. For 2D models, the gain is less significant: around 8% for both $e\langle E\rangle$ and $v\{E\}$. Thus we have chosen to use the extended representation (with reverse indices) because it offers a unified framework with advantages either in 3D or 2D representations.

4.3. Application: fragmentation simulation

A major motivation in the development of the new data structure is the treatment of both intrinsic and extrinsic cohesive zone models (CZMs) for fracture/fragmentation simulations [7–10] within a unified framework. The insertion of cohesive elements along bulk element interfaces may require the duplication of nodes. Whenever a node is duplicated, element connectivity has to be updated. Whether a node has to be duplicated or not depends on the topological classification of the fractured facet [9, 10], based on retrieval of adjacency relationships. Duplication of finite

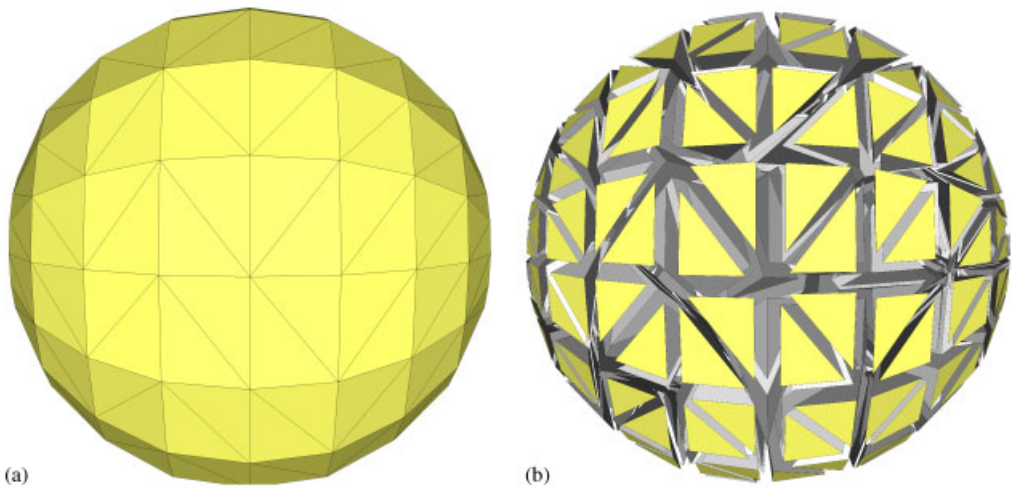


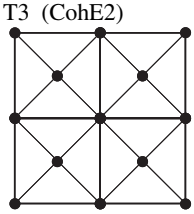
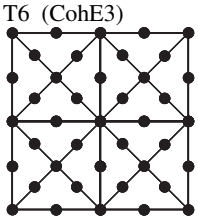
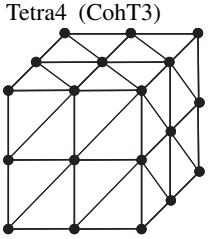
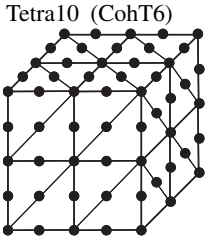
Figure 7. Insertion of cohesive elements along all facets of the model, in a random order: (a) original model; and (b) resulted model (for illustration purpose, we impose a separation in between each pair of cohesive element's facets).

element corner-nodes implies duplication of vertices and duplication of mid-side nodes implies duplication of edges in the topological data structure. The original set of vertex-uses or edge-uses is distributed between the two split vertices or edges, respectively. As a result of the proposed data structure, cohesive elements can be dynamically inserted in constant time.

Support for representing cohesive zone (CZ) elements was added to the data structure. For instance, for 2D meshes, two possible types of cohesive elements are: the cohesive element with linear edge (CohE2) and the cohesive element with quadratic edge (CohE3). Similarly, for 3D tetrahedral meshes, we have: the cohesive elements with linear face (CohT3) and the cohesive element with quadratic face (CohT6). In order to test the efficiency of the underlying topological data structure together with the correctness of the proposed topological procedure to insert cohesive elements, we decouple the topological framework from the mechanics simulation and insert, in a random order, cohesive elements at all the facets of a given mesh. The random order in which the cohesive elements are inserted results in arbitrarily complex crack patterns during the simulation process. At the end, each node of the mesh is used by only one bulk element. This test allows verification that the resulting topological entities (nodes, elements, facets, edges, and vertices) are the expected ones. Figure 7 illustrates the original and the final configuration of a spherical model represented by a tetrahedral mesh, which is similar to the cubic model presented by Pandolfi and Ortiz [9]. For illustrative purpose, in the final configuration, we impose a separation in between each pair of cohesive element's facets.

To address scalability of the proposed data structure, we have applied the above test, illustrated by Figure 7, to various meshes using different discretizations. Table V shows the time required to insert the cohesive elements along all internal facets in the corresponding models. Figure 8 plots the elapsed time against the number of cohesive elements inserted for the quadratic tetrahedral meshes. As expected, we have achieved a linear variation, which is due to local topological operations for each insertion.

Table V. Elapsed time (in seconds) for inserting cohesive elements at all the facets of the models. Each time reported is the average of 10 simulations for each specific model, inserting the elements in different random orders.

Initial mesh type	Grid dimension	Initial # nodes	Final # nodes	# bulk elems.	# CZ elems.	Elapsed time (s)
T3 (CohE2) 	100×100	20 201	120 000	40 000	59 800	0.6
	300×300	180 601	1 080 000	360 000	539 400	5.4
	500×500	501 001	3 000 000	1 000 000	1 499 000	15.7
T6 (CohE3) 	100×100	80 401	240 000	40 000	59 800	0.7
	300×300	721 201	2 160 000	360 000	539 400	6.5
	500×500	2 002 001	6 000 000	1 000 000	1 499 000	18.8
Tetra4 (CohT3) 	$10 \times 10 \times 10$	1331	24 000	6 000	11 400	0.4
	$30 \times 30 \times 30$	29 791	648 000	162 000	318 600	14.7
	$50 \times 50 \times 50$	132 651	3 000 000	750 000	1 485 000	72.3
Tetra10 (CohT6) 	$10 \times 10 \times 10$	9261	60 000	6 000	11 400	0.5
	$30 \times 30 \times 30$	226 981	1 620 000	162 000	318 600	16.9
	$50 \times 50 \times 50$	1 030 301	7 500 000	750 000	1 485 000	81.4

As an actual engineering example, we have tested the new data structure for supporting the implementation of fracture simulation on 2D models using an intrinsic CZM approach. All the details of the explicit finite element analysis are given by Zhang and Paulino [8]. The specific problem under consideration refers to the experiments by Kalthoff and Winkler [31] in which a plate with two edge notches is subjected to an impact by a projectile, as illustrated by Figure 9. Notice that, due to symmetry considerations, only half of the specimen is modelled.

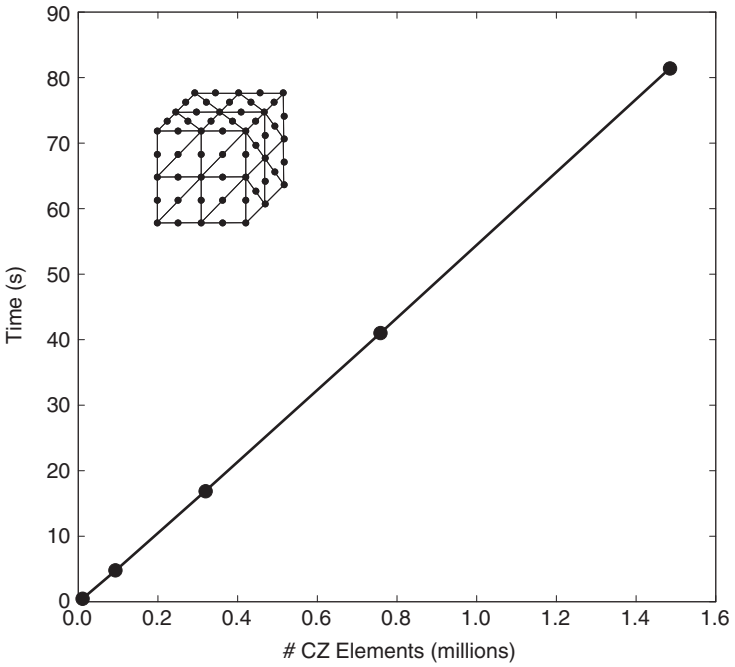


Figure 8. Plot of time vs # CZ elements for quadratic tetrahedral mesh.

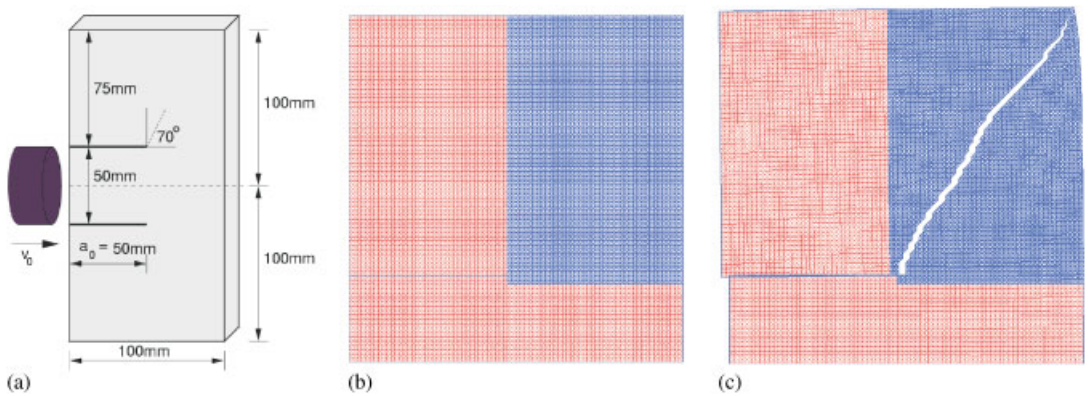


Figure 9. Simulation of the Kalthoff and Winkler [31] experiment in which a pre-cracked is subjected to an impact by a projectile [8]: (a) actual experimental configuration; (b) original mesh employing symmetry; and (c) deformed mesh illustrating spontaneous mixed-mode crack propagation.

This is a mixed-mode dynamic crack propagation problem, where the CZ elements allow crack initiation and turning of crack path to occur spontaneously without predefining the crack path. The brittle fracture mode is illustrated by Figure 9(c), with an approximate propagation angle

Table VI. Finite element mesh discretization for the Kalthoff and Winkler experiment [31].

	Cohesive elements	
Meshes	Before insertion	After insertion
# nodes	51 601	93 262
# elements (T6)	25 600	25 600
# CZ elements	—	15 622

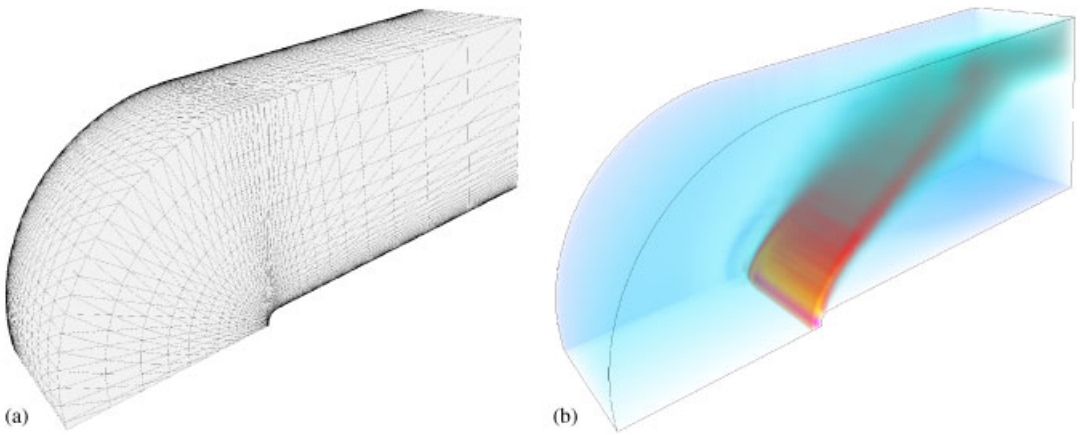


Figure 10. Image of the NASA's Blunt Fin data set [30]: (a) external view of the tetrahedral mesh; and (b) volume visualization showing the internal density distribution.

of 72°, which agrees well with the experimental prediction of 70°. The corresponding finite element mesh statistics used in this example is given by Table VI.

In the above example, the 2D quadratic CZ elements are naturally handled by the data structure, unambiguously representing distinct edges, including those with the same bounding nodes. Moreover, the new data structure offers a convenient and robust interface to CZMs, either intrinsic, e.g. References [6–8], as illustrated above, or extrinsic, e.g. References [9, 10].

4.4. Application: volumetric visualization

The proposed data structure has also been tested for supporting visualization algorithms and applied to the NASA's Blunt Fin data set [30]. Particularly, we have used the data structure to implement volumetric visualization of unstructured meshes. Direct volume rendering was achieved by cell-projecting each tetrahedral element using the graphics hardware [13]. In order to correctly handle transparency, cell-projection algorithms require visibility-ordering of cells (elements) because cells must be rendered in back-to-front (or front-to-back) order. To achieve this ordering, we have implemented the meshed polyhedra visibility ordering (MPVO) algorithm [11] on the top of the data structure. The associated adjacency graph for the cells is implicitly

represented by the adjacency relationship among elements. Figure 10 shows an image obtained by rendering the density distribution of NASA's Blunt Fin data set [30].

5. CONCLUSION

We have developed a complete topological data structure for finite element mesh representation. When compared to previous proposals [2], the present data structure reduces the required storage space, while preserving the ability to directly retrieve all adjacency information. Such reduced memory requirement is achieved by explicitly representing only two basic topological entities: element and node. All other topological entities (facet, edge, and vertex) are implicitly represented and efficiently created upon request by using concrete types. Inspired by Weiler's work [14, 15], we define oriented topological entities to facilitate retrieval of topological adjacency relationships. These oriented entities represent the use of facet, edge, or vertex by an element and do not impose any additional storage requirements, as they are also implicitly represented. A noteworthy aspect of the data structure is its extension by means of 'reverse indices'. Such extended data structure improves performance for extracting adjacency relationships, while maintaining storage space within reasonable limits. Experimental results have demonstrated the data structure efficiency for retrieving adjacency topological information. The results have also demonstrated that the proposed data structure scales linearly with the number of entities in the model.

The present data structure includes support for fragmentation simulation by fully decoupling the topological framework from the physics. The simulation requires insertion of cohesive elements by duplicating an edge without duplicating their bounding nodes [7–9] (which leads to two distinct edges with the same bounding nodes). A limitation of other reduced data structures [1–4] is the inability to handle such situation. Based on the set of defined topological entities, the proposed data structure naturally handles such occurrence.

The data structure effectiveness has been illustrated by its use for supporting both simulation (fracture/fragmentation) and visualization applications, dealing with different types of finite element meshes. The data structure also has potential applications in many other areas in solid or fluid mechanics. A few applications of particular interest include adaptive mesh refinement [32], support for non-linear finite element analysis [33] including graded material systems [34], and 3D fracture analysis using discrete approaches [35]. Automatic adaptive meshing requires extension of the data structure to treat non-manifold boundary representations. Some of these applications are currently under consideration by the authors [36].

APPENDIX A: DATA STRUCTURE IMPLEMENTATION

The proposed data structure has been implemented in C++. More precisely, aiming portability and efficiency, we have opted for using a small sub-set of the C++ language, avoiding using features such as complex templates, dynamic casts, and multiple inheritances. In order to facilitate its integration with existing FEM applications, the data structure functionality is also exposed by a C interface.

This appendix presents part of the code used for defining the classes. The main concern here is to present the associated data together with a brief description of the functionality provided by each class.

A.1. Model

The mesh is represented by the *Model* class. Each model provides access to all nodes and elements. We have opted for storing the nodes and elements in doubly linked lists in order to ease the implementation of entity insertion and removal operations. One can also store the nodes and elements in arrays, thus allowing access from entity 'ids'.

The model class provides methods for inserting and removing nodes and elements. It also provides methods for enumerating the model entities, despite they are explicitly or implicitly represented. One can extend the model class to manage hash tables to store attributes associated with the implicit entities (facets, edges, and vertices).

```
class Model {
     Node*       first_node;    /* pointer to a first node */
     Element*    first_elem;    /* pointer to a first element */
     ...
};
```

A.2. Node

The mesh nodes are represented by the *Node* class. Each node stores its numeric *id*, its co-ordinates, and a reference to an adjacent element. We also have opted for representing a generic pointer to hold client attributes. The code excerpt below also shows pointers to build the linked list. From a given node, we have access to the corresponding vertex, for corner node, or edge, for mid-side node.

```
class Node {
     int         id;         /* node id */
     double      x, y, z;    /* node coordinates */
     Element*    elem;       /* pointer to an adjacent element */
     void*       attrib;     /* element client attribute */
     Node*       next;       /* pointer to next node in model */
     Node*       prev;       /* pointer to prev node in model */
     ...
};
```

A.3. Element

Element is an abstract class that represents the mesh elements. This abstract class is responsible for providing the element interface and for holding data that are common to any type of element: the element *id*, the bit-vector for flagging implicit entities that reference the element, and a generic pointer to hold client attribute (plus the list links). Each class representing specific types of finite elements inherits from this abstract class and stores the node incidence and references to the adjacent elements. For the extended version, we also have to store the reverse indices. The specialized classes provide all information related to the element template. It is possible, for instance, to access all facet-uses adjacent to a given vertex-use within an element or, by using the reverse indices, to access the position of a given vertex-use in a facet-use that is adjacent to it. The excerpt code below illustrates the representation of different type of 2D and 3D finite elements, including the representation of cohesive elements used in fragmentation simulation.


```
class Element {
    int             id;       /* element id */
    unsigned int    flags;    /* referencing entity flags */
    void*           attrib;   /* element client attribute */
    Element*        next;     /* pointer to next elem in model */
    Element*        prev;     /* pointer to prev elem in model */
    ...
};

/* Linear triangular element */
class T3 : public Element {
   Node*          nodes[3];  /* pointers to adjacent nodes */
   Element*       adj[3];    /* pointers to adjacent elements */
   unsigned char  rev[3];    /* reverse indices of facet-uses */
   ...
};

/* Quadric triangular element */
class T6 : public Element {
   Node*          nodes[6];  /* pointers to adjacent nodes */
   Element*       adj[3];    /* pointers to adjacent elements */
   unsigned char  rev[3];    /* reverse indices of facet-uses */
   ...
};

/* Linear tetrahedral element */
class Tetra4 : public Element {
   Node*          nodes[4];  /* pointers to adjacent nodes */
   Element*       adj[4];    /* pointers to adjacent elements */
   unsigned char  rev[4];    /* reverse indices of facet-uses */
   ...
};

/* Quadratic tetrahedral element */
class Tetra10 : public Element {
   Node*          node[10];  /* pointers to adjacent nodes */
   Element*       adj[4];    /* pointers to adjacent elements */
   unsigned char  rev[4];    /* reverse indices of facet-uses */
};

/* Cohesive element with quadratic segment boundary */
class Coh2E3 : public Element {
   Node*          nodes[6];  /* pointers to adjacent nodes */
   Element*       adj[2];    /* pointers to adjacent elements */
   unsigned char  rev[2];    /* reverse indices of facet-uses */
   ...
};
```


A.4. Oriented entities

The data structure defines and implicitly represents oriented entities, namely facet-use, edge-use, and vertex-use. These entities represent the use of facets, edges, or vertices by elements, respectively. Each oriented entity is represented by a concrete class that stores a reference to the using element and the local *id* of the corresponding entity (facet, edge, or vertex) within that element. The classes provide methods to access the next entity-use, thus allowing traversing all associated uses and, consequently, all adjacent elements.

```
class FacetUse {
    Element*  E;        /* owner element */
    int       id;       /* local facet id */
    ...
};

class EdgeUse {
    Element*  E;        /* owner element */
    int       id;       /* local edge id */
    ...
};
class VertexUse {
    Element*  E;        /* owner element */
    int       id;       /* local vertex id */
    ...
};
```

A.5. Facet, edges and vertices

The implicit entities, facets, edges, and vertices, are represented by the concrete classes *Facet*, *Edge*, and *Vertex*, respectively. Each class stores a representative use. For quadratic edges, the representative use is the one associated with the element pointed by the mid-side node. For vertices, the representative use is the one associated with the element pointed by the corresponding corner node. For facets and linear edges, any use may be chosen as the representative.

Any implicit entity provides methods to return the corresponding oriented entity. We can also extract adjacency information. For instance, we can query if a given entity belongs to the model boundary. For facets, it suffices to check if either interfacing element is void. For edges and vertices, it requires accessing all incident facets to check the boundary condition.

```
class Facet {
    FacetUse fu;        /* representative use */
    ...
};


class Edge {
    EdgeUse eu;         /* representative use */
    ...
};
```



```
class Vertex {
   VertexUse vu;        /* representative use */
   ...
};
```

APPENDIX B: NOMENCLATURE

B.1. Topological entities

E	element, representing any type of finite element defined by templates of ordered nodes (for both 2D and 3D cases)
N	node, representing element-corner node or mid-side node
f	facet, representing the interface between two elements (for 3D, it is a two-dimensional entity; for 2D, it is a one-dimensional entity)
e	edge, representing a one-dimensional entity bounded by two vertices
v	vertex, representing a zero-dimensional entity associated to an element-corner node
fu	facet-use, representing the use of a facet by an element
eu	edge-use, representing the use of an edge by an element
vu	vertex-use, representing the use of a vertex by an element

B.2. Adjacency relationships

$\varphi[\alpha]$	ordered group of α adjacent to φ, with φ and α representing any topological entities
$\varphi\langle\alpha\rangle$	cyclically or radially ordered group of α adjacent to φ, with φ and α representing any topological entities
$\varphi\{\alpha\}$	unordered group of α adjacent to φ, with φ and α representing any topological entities

Examples

$E[N]$	ordered group of nodes adjacent to an element (group of nodes that defines the element)
$e\langle E\rangle$	radially ordered group of elements adjacent to an edge
$v\{E\}$	unordered group of elements adjacent to a vertex

B.2. Storage space

$\mathscr{E}$	memory space to store an explicit entity, either element (E) or node (N)
$\mathscr{C}$	memory space to store a connection (a reference to an entity)

ACKNOWLEDGEMENTS

Paulino gratefully acknowledges the support from NASA-Ames, Engineering for Complex Systems Program, and the NASA-Ames Chief Engineer (Dr Tina Panontin) through Grant NAG 2-1424. He also acknowledges additional support from the National Science Foundation (NSF) under Grant CMS-0115954 (Mechanics & Materials Program). Espinha was financially supported by the Brazilian

agency CAPES (*Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*). Both Celes and Espinha would like to thank the Tecgraf laboratory at PUC-Rio, which is mainly funded by the Brazilian oil company Petrobras. The authors thank Ms Zhengyu Zhang for her help with the fracture/fragmentation example, which is described in Reference [8]; and Dr Alok Sutradhar for very useful suggestions.

REFERENCES

1. Beall MW, Shephard MS. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering* 1997; **40**:1573–1596.
2. Garimella RV. Mesh data structure selection for mesh generation and FEA applications. *International Journal for Numerical Methods in Engineering* 2002; **55**:451–478.
3. Remacle J-F, Karamete BK, Shephard MS. Algorithm oriented mesh database. *Proceedings 9th International Meshing Roundtable*. Sandia National Laboratories: Albuquerque, NM, 2000; 349–359.
4. Remacle J-F, Shephard MS. An algorithm oriented mesh database. *International Journal for Numerical Methods in Engineering* 2003; **58**:349–374.
5. Owen SJ, Shephard MS. Editorial: special issue on trends in unstructured mesh generation. *International Journal for Numerical Methods in Engineering* 2003; **58**:159–160.
6. Paulino GH, Jin Z-H, Dodds Jr RH. Failure of functionally graded materials. In *Comprehensive Structural Integrity*, Karihaloo B, Knauss WG (eds), vol. 2(13). Elsevier: Amsterdam, 2003; 607–644.
7. Jin Z, Paulino GH, Dodds Jr RH. Cohesive fracture modeling of elastic–plastic crack growth in functionally graded materials. *Engineering Fracture Mechanics* 2003; **70**:1885–1912.
8. Zhang Z, Paulino GH. Cohesive zone modeling of dynamic failure in homogeneous and functionally graded materials. Special issue on inelastic response of multiphase materials. *International Journal of Plasticity* 2005; **21**:1195–1254.
9. Pandolfi A, Ortiz M. Solid modeling aspects of three-dimensional fragmentation. *Engineering with Computers* 1998; **14**:287–308.
10. Pandolfi A, Ortiz M. An efficient adaptive procedure for three-dimensional fragmentation simulations. *Engineering with Computers* 2002; **18**:148–159.
11. Williams PL. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics* 1992; **11**(2):103–126.
12. Comba J, Klosowski JT, Max N, Mitchell JSB, Silva CT, Williams PL. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum* (*Proceedings of EUROGRAPHICS'99*) 1999; **18**(3):369–376.
13. Weiler M, Kraus M, Merz M, Ertl T. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics* 2003; **9**(2):163–175.
14. Weiler K. The radial edge structure: a topological representation for non-manifold geometric boundary modeling. In *Geometric Modeling for CAD Applications*, Wozny MJ, McLaughlin HW, Encarnação JL (eds). 1988; 3–36.
15. Weiler K. Topological structures for geometric modeling. *Ph.D. Thesis*, Rensselaer Polytechnic Institute, New York, 1986.
16. Lee SH, Lee K. Partial entity structure: a compact non-manifold boundary representation based on partial topological entities. *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications* 2001; 159–170.
17. Mäntylä M. *An Introduction to Solid Modeling*. Computer Science Press: Rockville, MD, 1988.
18. McMains S, Hellerstein JM, Séquin CH. Out-of-core build of a topological data structure from polygon soup. *Solid Modeling* 2001; 171–182.
19. Lee SH, Lee K. Partial entity structure: a compact boundary representation for non-manifold geometric modeling. *Journal of Computing and Information Science in Engineering* 2001; **1**(4):356–365.
20. Silva FGM, Gomes AJP. Adjacency and incidence framework—a data structure for efficient and fast management of multiresolution meshes. *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques*, Melbourne, Australia, 2003; 159–166.
21. De Floriani L, Magillo P, Puppo E, Sobrero D. A multi-resolution topological representation for non-manifold meshes. *Solid Modeling* 2002; 17–21.

22. Wawrzynek PA, Ingraffea AR. Interactive finite element analysis of fracture processes: an integrated approach. *Theoretical and Applied Fracture Mechanics* 1987; **8**:137–150.
23. Martha LF, Wawrzynek PA, Ingraffea AR. Arbitrary crack representation using solid modeling. *Engineering with Computers* 1993; **9**:63–82.
24. Löhner R. Some useful data structures for the generation of unstructured grids. *Communications in Applied Numerical Methods* 1988; **4**:123–135.
25. Carey GF, Sharma M, Wang KC. A class of data structures for 2-D and 3-D adaptive mesh refinement. *International Journal for Numerical Methods in Engineering* 1988; **26**:2607–2622.
26. Hawken DM, Townsend P, Webster MF. The use of dynamic data structures in finite element applications. *International Journal for Numerical Methods in Engineering* 1992; **33**(9):1795–1811.
27. Tautges T, Ernst C, Merkley K, Meyers R, Stimpson C. MOAB, a mesh-oriented database. *Sandia National Laboratories Report SAND2004-1592*, 2004, Sandia National Laboratories, Albuquerque, New Mexico (http://cubit.sandia.gov/MOAB).
28. Cignoni P, De Floriani L, Magillo P, Puppo E, Scopigno R. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics* 2003; **10**(1):29–45.
29. Stroustrup B. *The C++ Programming Language*. Reading, Massachusetts, Addison-Wesley: Reading, MA, 1997.
30. Hung CM, Buning PG. *NASA's Blunt Fin Data Set*. Available at NAS, 2004, www.nas.nasa.gov/Research/Datasets/Hung/index.shtml.
31. Kalthoff JF, Winkler S. Failure mode transition at high rates of shear loading. In *International Conference on Impact Loading and Dynamic Behavior of Materials*, Chiem CY, Kunze HD, Meyer LW (eds). 1987; 185–195.
32. Paulino GH, Menezes IFM, Neto JBC, Martha FL. A methodology for self-adaptive finite element analysis—towards an integrated computational environment. *Computational Mechanics* 1999; **23**(5–6):361–388.
33. Lages EN, Paulino GH, Menezes IFM, Silva RR. Nonlinear finite element analysis using an object-oriented philosophy—application to beam elements and to the Cosserat continuum. *Engineering with Computers* 1999; **15**(1):73–89.
34. Kim J-H, Paulino GH. Isoparametric graded finite elements for nonhomogeneous isotropic and orthotropic materials. *ASME Journal of Applied Mechanics* 2002; **69**(4):502–514.
35. Walters MC, Paulino GH, Dodds RH. Stress intensity factors for surface cracks in functionally graded materials under mode-I thermomechanical loading. *International Journal of Solids and Structures* 2004; **41**(3–4): 1081–1118.
36. Celes W, Paulino GH, Espinha R. Efficient handling of implicit entities in reduced mesh representations. Special issue on mesh-based geometric data processing for engineering design and analysis. *Journal of Computing and Information Science in Engineering* 2005 (in press).

