A Computational Framework for Topological Operations

Michael Spevak, René Heinzl, Philipp Schwaha, and Siegfried Selberherr

Institute for Microelectronics, TU Wien,
Gusshausstrasse 27-29, 1040 Wien, Austria
spevak@iue.tuwien.ac.at

Abstract. We present a complete topological framework that is able to provide incidence traversal operations for various topological elements. This enables one to perform the necessary topological operations for several discretization schemes. A combination of incidence information combined with an archetype concept enables one to optimize traversal operations of inter-dimensional objects without explicitly storing them. Access to topological structures is provided using a generalized iterator concept.

1 Introduction

The field of scientific computing often imposes highly complex formulae with quantities on different topological elements. For example, some discretization schemes require the scalar solution to reside on vertices, while the projections of the vector-valued fluxes are stored on edges. Many applications require such a discretization of partial differential equations (PDE) as well as interpolation mechanisms and thus strongly depend on the base traversal mechanisms provided by the environment. It is quite common for a discretization scheme to require quantities originally associated with a vertex on an edge and vice versa. The projections of fluxes on edges also need to be assembled to truly vector-valued quantities associated with a vertex and an edge. In order to accomplish this, the required information has to be collected by traversing the local neighborhood of a vertex or an edge. To this date, data structures and algorithms are implemented in a heavily application and discretization scheme specific way, making their reuse practically impossible.

We present a set of base traversal operations that is sufficient for many applications. This approach results in a rigorous implementation of topological structures, which covers all types of topological elements such as vertices, cells and general inter-dimensional elements called faces. The expressiveness of source code is increased, because we do not need to explicitly write traversal algorithms for each of the elements, such as edge-cell traversal, because this information can be derived from a subset of highly optimized operations automatically.

The iterator concept allows to formulate algorithms based on this interface independently of the actual implementation of the topological data structure

B. Kågström et al. (Eds.): PARA 2006, LNCS 4699, pp. 781–790, 2007.
© Springer-Verlag Berlin Heidelberg 2007

including dimension and archetype. The consequent use of this interface leads to dimensionally and topologically independent formulations of algorithms, e.g. a finite volume discretization scheme can be formulated independently of the type of cell complex and the dimension.

2 Motivation and Related Work

The motivation for developing a topological framework is derived from the need for flexibility in high performance applications in the field of scientific computing, especially in Technology Computer-Aided Design (TCAD). With a growing number of different simulation tools [1,2,3,4,5] and various requirements on the underlying data structures the question arises which part of an application can be re-used, if it is properly implemented. The main aim of our work is to provide a library that offers the functionality common to many different kinds of simulation tools in the field of scientific computing. This is especially true for functions which are evaluated on a topological cell complex.

In the last decade many approaches towards implementing a general purpose simulation environment for the solution of partial differential equations have been taken. Most of the tools resulting from these attempts use topological structures which are specialized to a particular discretization scheme. This reduces resource use, but it comes at the cost of greatly diminishing the flexibility of topological traversal. As an example, the finite volume method does not require vertex-faces traversal. However, for some reasons it might be advantageous to implement discretization equations based on a mixed finite element/finite volume scheme which requires such traversal operations.

The major step towards a more flexible use of topological structures is presented in [6]. The grid algorithm library (GrAL) introduces the first generalized iterator concept [7] based on multi-dimensional data structures.

Most of the other environments completely veil the topological information by formalisms such as element matrices [8] and control functions [9]. Some commercial simulation tools, such as FEMLab, accept the input in the form of a final PDE. For this reason calculations which use non-standard traversal mechanisms are cumbersome or impossible to specify.

3 Framework and Interfaces

The main aim of the topological container interface is to provide mechanisms for construction, modification and traversal. The most important conceptual requirement for the topological data structure is the retrieval of incidence information. We define the incidence relation in the following manner:

$$\mathrm{inc}(a, b) \leftrightarrow a \subset b \vee a \supset b \quad (1)$$

where a and b denote different topological elements. In the following table (Fig. 1) we list all different methods of incidence which are possible between topological

elements of different dimension. It can be seen easily that the incidence relation of elements of the same dimension can be modeled by the equality relation. The first row shows all edges, faces as well as cells which are incident with the same base vertex. The first column shows vertices which are incident with one base edge, faces or cell.

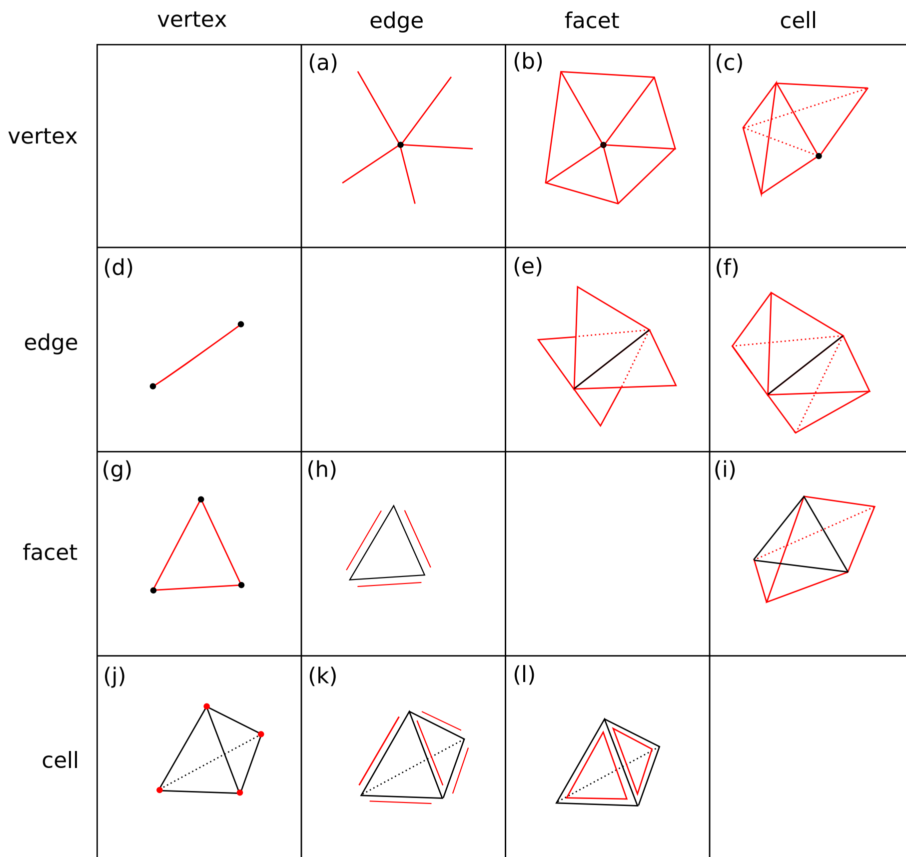

Fig. 1. Traversal methods induced by the incidence relation. horizontal: traversal schemes of the same base element. vertical: traversal scheme of the same traversal elements.

3.1 Topological Container Concept

The topological container covers the basic information of the topological cell complex. Our concept provides a subset of the required methods (Fig. 1) from which all further information can be obtained. According to the concepts of the standard template library (STL [10]), we provide iterators for the cells as well as the vertices of the cell complex. In analogy to the STL we use a formulation with `begin()` and `end()`. In order to obtain the base traversal mechanisms between

Name	Description	Requirements
vertex_iterator	iterator over vertices	iterator concept
vertex_begin(), vertex_end()	iterator range	const
cell_iterator	iterator over cells	iterator concept
cell_begin(), cell_end()	iterator range	const
cell_vertex	local traversal iterator	constructable with cell
vertex_cell	local traversal iterator	constructable with vertex

Fig. 2. Concepts for the topological base structure

vertices and cells we define traversal iterators. These traversal iterators perform the operations shown in Fig. 1, (c) and (j).

Due to the concept definition (Fig. 2) of the cell complex container we can formulate algorithms conveniently. As an example we present an algorithm which traverses all vertices as well as all cells.

Global Traversal

```
cell_complex_t cc;
cell_complex_t::vertex_iterator v_it;
for(v_it  = cc.vertex_begin();
    v_it != cc.vertex_end();  ++v_it)
{
  // do something on vertices
}

cell_complex_t::cell_iterator c_it;
for(c_it  = cc.cell_begin();
    c_it != cc.cell_end(); ++c_it)
{
  // do something on cells
}
```

A direct consequence of the use of the iterator ranges `vertex_begin()`, `vertex_end()` is that standard algorithms such as `for_each` are automatically supported.

3.2 Topological Elements and Handles

The data structures for single topological elements is kept to a bare minimum. In general, each topological data structure covers a so called handle in order to be distinguishable from other topological elements. Basically any type can be used for these handles, which allows to uniquely identify the element within all elements of the same dimension. The value of such handles itself does not have any semantic meaning apart from being equal. The only valid operation on handles is, therefore, the equality relation.

For inter-dimensional topological elements a unique identification can be found either via storing all the vertices or storing a cell and a local index which

determines the element within the cell. We discuss vertex based indexing in the following. A handle of an inter-dimensional element is uniquely constructed by the vertex handles, e.g. $h_F = h_{v_1} + n \cdot h_{v_2}$ where the following abbreviations are used:

- h_F: the handle used by a faces
- h_{v_i}: the handle from each vertex
- n: the number of vertices

For this reason the reconstruction of vertex information from the handle is straightforward. This covers incidence operations (Fig. 1 (d) and (g)).

3.3 Archetypes

As we only store incidence information between cells and vertices (Fig. 1 (c) and (j)) the incidence relation between inter-dimensional elements is still undefined. There are many methods to specify this kind of information, for instance to use containers and explicitly storing this information. Even though this is possible and can be used in cases where high performance is required, such methods result in a high memory consumption, because most of the information has to be duplicated.

In the following we take advantage of the fact that all elements within the container have the same shape (e.g. tetrahedral elements). If this is not the case (e.g. we have a small number of elements of different shape), we have to perform a dispatch operation in order to obtain the correct shape.

The concept that provides the internal structure of the cells within the cell-complex is called an `archetype`. The archetype [6] introduces local inter-dimensional elements within a cell. A topological 2-simplex, for example, consists of three vertices and three 1-faces (edges). The archetype can be shown either as simple graph or Hasse diagram (Fig. 3).

Using the archetype concept as well as the vertex on cell relation (Fig. 1 (j)) we can derive further traversal mechanisms (Fig. 1 (k) and (l)) as each cell is aware of its covered vertices.

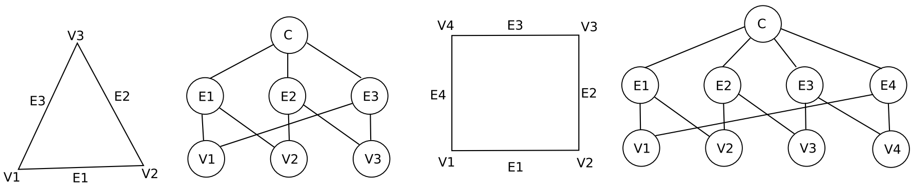

Fig. 3. The graph as well as the Hasse Diagram of the 2-simplex as well as a 2-cuboid archetype

3.4 Traversal Operations

Once the archetype as well as the vertex-cell incidence information is available, all incidence relations between arbitrary elements of the cell complex can be calculated. The topological framework stores vertex-cell incidence relations within the container which can be seen in Fig. 1 (c) and (j).

We present an example using the simplex archetype to introduce the computational operations that complete the traversal operations. All other relations ((a,b)(d,e,f,)(g,h,i)(k,l)) in Fig. 1 are taken care of by the framework. From topological handles of inter-dimensional elements *vertex on element* iteration can be obtained (Fig. 1 (d) and (g)). Due to the archetype information we obtain the element on cell traversal (Fig. 1 (k) and (l)). These base relations are sufficient for the construction of all other element-element incidence relations. The following algorithm gives a generic implementation for obtaining the traversal information via the base operations.

Incidence traversal

```
get all vertices which belong to the element (d) (g) (j)
get all cells which belong to these vertices (c)
get all n-dimensional sub-elements on those cells (j) (k) (l)
select the incident elements (by vertex-comparison)
```

We show the application of this algorithm using the example of faces on edge traversal (Fig. 4, Fig. 1 (e)). From the initial edge we obtain the vertices which are located on the edge using the basic traversal mechanism (d). Using the vertex-cell (c) information we obtain all cells which cover the vertices. We obtain all faces which are on these cells by method (k). From this set we select only the faces which are incident with the initial edge.

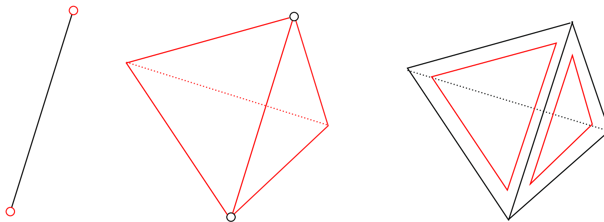

Fig. 4. Construction of the faces on edge traversal set. (left) The initial edge. (middle) The vertices on the initial edge. (right) Incident faces.

As all of these operations are local, we do not need to iterate all elements of a cell complex in order to perform an incidence traversal operation. This construction method is superior to an explicit search of topological elements within the complete cell complex for large meshes.

3.5 Generalized Iterator Concept

In order to allow an arbitrary number of nested traversal operations, which is often required in applications, it is necessary to have appropriate data structures for traversal in order to keep the program code as concise as possible. For this reason we use the random access iterator concept in order to provide access to the topological elements. We refine the iterator concept [7] for data structural convenience (Fig. 5). In contrast to the STL iterator concept, dereferentiation of this generalized iterator does not provide the data content but a handle, that is used as a key to access a property map or quantity [11] in order to obtain the contained data. Our iterator concept is a refinement of the random access iterator concept and introduces the following additional concepts.

Name	Description	Requirements
bool valid()	validity of iterator (not end)	const
iterator end()	past end of iterator validity	const
void reset()	set to the start point	const

Fig. 5. Concept refinement for the generalized iterator within the topological framework

To demonstrate the application of the `valid()` predicate we show a simple example of two nested iterations. We traverse from a base vertex to all incident edges and from these edges to incident vertices. (Fig. 1, (a) and (d))

The validity concept and its application

```
cell_vertex voc_it(cell);
while(voc_it.valid())
{
  vertex_edge eov_it(*voc_it);

  while(eov_it.valid())
  {
     //operations on edges
     ++eov_it;
  }
  ++voc_it;
}
```

In the first traversal state, a `cell` is used to perform an iteration over all its incident vertices. Each incident vertex `*voc_it` is available in the outer loop. The inner traversal loop is initiated using the actually traversed element of the outer loop. From this vertex we obtain all incident edges using the dereferentiation of the inner iterator.

In order to use standard algorithms of the STL, a data structure has to provide an initial as well as a terminal element. For this reason the `end()` function can be employed.

The iterator end functionality and its application

```
struct func_obj{
void operator()(cell_t cell) { cout << c << endl;}
};

cell_vertex voc_it(cell);
for_each(voc_it, voc_it.end(), func_obj());
```

As many of the iterators require a non-negligible time for construction, it is often more efficient to use an iterator twice rather than creating a second instance for the same purpose. For this reason we introduce the `reset` function that enables us to reset an iterator to the beginning of the iteration range.

Usage of the reset function

```
cell_vertex voc_it(cell);
for(; voc_it.valid(); ++voc_it)
{ // ... some operation
}
voc_it.reset();
for(; voc_it != voc_it.end(); ++voc_it)
{  // ... some operation
}
```

4 Generalized Data Access

The topological data structure itself, however, is not sufficient to perform complex calculations on such a structure. For this reason we provide means for the storage of values on the topological container. Each of the topological elements such as vertices or edges can be associated with one or more values.

In our approach, the *property map* concept [7] is adopted, which offers the possibility of accessing the quantities in a functional way by a mechanism called *quantity accessor*. The quantity accessor implementation also takes care of accessing quantities with different data locality, e.g., quantities on vertices, edges, faces, or cells. The quantity accessor is initialized with a domain. During initialization, the quantity accessor `quan` is bound to a specific domain with its quantity key. The `operator()` is evaluated with a vertex of the cell complex as argument and returns a reference to the stored value.

Quantity assignment

```
string key_quan  = "user_quantity";
quan_t quan = scalar_quan(domain, key_quan);

quan(vertex) = 1.0;
```

In the following code snippet a simple example of the generic use of this accessor is given, where a scalar value is assigned to each vertex in a domain. The

quantity accessor creates an assignment which is passed to the `std::for_each` algorithm.

Quantity assignment

```
string key_quan = "user_quantity";
quan_t quan = scalar_quan(domain, key_quan);

for_each(cc.vertex_begin(), cc.vertex_end(), quan = 1.0);
```

5 Conclusion

A computational mechanism was introduced for completing traversal operations obtained from the archetype's structure and a minimum of explicitely stored information. As a result we presented a framework that provides a complete means of topological traversal operations based the concept of archetypes.

Our framework presented here integrates well into existing software components such as the STL.

The use of clean, well defined iterator interfaces alongside generalized standard routines makes it possible to develop orthogonal and modular software. Futhermore the provided means are not only sufficient to build a homoegenous interface but can also be used to implement several discretization schemes.

References

1. Selberherr, S., Schütz, A., Pötzl, H.: MINIMOS—A Two-Dimensional MOS Transistor Analyzer. IEEE Trans. Electron Dev. ED-27(8), 1540–1550 (1980)
2. Halama, S., Pichler, C., Rieger, G., Schrom, G., Simlinger, T., Selberherr, S.: VISTA — User Interface, Task Level, and Tool Integration. IEEE J. Techn. Comp. Aided Design 14(10), 1208–1222 (1995)
3. Sabelka, R., Selberherr, S.: A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures. Microelectronics Journal 32(2), 163–171 (2001)
4. IμE: MINIMOS-NT 2.1 User's Guide. Institut für Mikroelektronik, Technische Universität Wien, Austria (2004), `http://www.iue.tuwien.ac.at/software/minimos-nt`
5. Binder, T., Hössinger, A., Selberherr, S.: Rigorous Integration of Semiconductor Process and Device Simulators. IEEE Trans. Comp.-Aided Design of Int. Circ. and Systems 22(9), 1204–1214 (2003)
6. Berti, G.: GrAL - The Grid Algorithms Library. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds.) Computational Science - ICCS 2002. LNCS, vol. 2331, pp. 745–754. Springer, Heidelberg (2002)
7. Abrahams, D., Siek, J., Witt, T.: New Iterator Concepts. Technical Report N1477 03-0060, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003)
8. Bangerth, W., Hartmann, R., Kanschat, G.: deal.II – A General Purpose Object Oriented Finite Element Library. Technical Report ISC-06-02-MATH, Institute for Scientific Computation, Texas A&M University (2006)

9. Rafferty, C.S., Smith, R.K.: Solving Partial Differential Equations with the Prophet Simulator (1996)
10. Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)
11. Heinzl, R., Spevak, M., Schwaha, P., Grasser, T.: Concepts for High Performance Generic Scientific Computing. In: Proc. of the 5th MATHMOD, vol. 1, Vienna, Austria (2006)