A Generic Topology Library

René Heinzl
Christian Doppler Laboratory
Gusshausstrasse 27-29
Vienna, Austria
heinzl@iue.tuwien.ac.at

Michael Spevak
Institute for Microelectronics
Gusshausstrasse 27-29
Vienna, Austria
spevak@iue.tuwien.ac.at

Philipp Schwaha
Christian Doppler Laboratory
Gusshausstrasse 27-29
Vienna, Austria
schwaha@iue.tuwien.ac.at

ABSTRACT

We present a generic topology library that is based on topological space and combinatorial properties. A notation is introduced whereby data structures can be described by their topological cell dimensions and internal combinatorial properties. A common interface for different types of data structures is presented. Various issues of iteration of these data structures can be explained from the topological properties. Using this multi-dimensional topology library we introduce new possibilities for functional programming in the field of scientific computing.

1. INTRODUCTION

In this work we investigate internal topological and combinatorial properties of data structures and the effect on their interfaces. Generic interfaces to data structures have proven to be highly successful means of generic programming. With the great achievement of accessing all data structures in a minimal but concise way, generic programming has emerged. A detailed analysis of generic programming is given in [1], where this topic is introduced from a theoretical point of view, namely category theory. A lot of insight is gained through this approach and a solid base has been achieved with this theory. Our work deals with the basic nature of topological spaces related only to data structures and is based on GrAL [2]. This is not as general as the category theory approach, but the basic features and issues are exposed.

Usually programmers have to know the **specific properties** of data structures to achieve the best performance of an algorithm. A simple example is the iteration and data access within a `std::vector`, which is constant, whereas the insertion or deletion uses linear time. This is relevant for the actual run-time behavior of all implemented algorithm applied to it. Closely related to this issue is the fact that the C++ STL algorithms use the most basic iteration mechanism for the access to data structures, the `forward` iterator mechanism. The optimal way of iteration of containers can often not be achieved, because linear iteration is simply not optimal [1], such as traversing a `std::map` or higher-dimensional topological structures, e.g., `boost::graph` from the Boost Graph library [3]. We introduce (Section 4.1) a unified data structure definition, where only the dimension and the combinatorial properties of topological spaces are specified. This can also be accomplished automatically at compile-time, based on requirements of algorithms.

Modern application design requires the utilization of data structures in several dimensions. Especially the field of scientific computing uses different topological elements to discretize partial differential equations (PDE). Various approaches are available such as the STL containers, the BGL, and for grids the GrAL [2]. However, a **standardized interface** to these data structures is missing. We introduce a basic interface (Section 4.2.3) for different dimensions of data structures based on topological and combinatorial properties.

A major issue of generic programming is the treatment of **data structure iteration and data access** [4], but the upcoming C++0x standard does not yet include this insight [5]. Therefore we use the property map concept [4], which is presented in Section 5 to utilize an extra data space. Briefly, the combination of iteration and access leads to a miscategorized algorithm specialization.

Our search for a general data structure library for the needs of scientific computing has shown that the topological structures of different STL containers and BGL mechanisms can be abstracted and generalized to a multi-dimensional generic topology library (GTL). We do not only separate the data access and iteration [4], but also provide a formal description of the underlying topological space with emphasis on the combinatorial properties:

topological space + data type = data structure

With a formalization of the topological properties and the iteration mechanism this approach renders a new possibility of the functional programming paradigm (Section 7) which is emerging in C++ [6, 7]. Up to now, functional expressions lack the support of a unique interface for all different kinds of data structure iteration. As we present in a generic discretization library for the discretization of various partial differential equations (GDL [8]), the full power of functional programming is revealed with consistent topological data structure. Note, the GTL is not restricted to applications for scientific computing, simple iterations can be specified elegantly as well.

2. MOTIVATION

Our motivation for developing generic libraries is derived from the need in high performance applications in the field of scientific computing, especially in Technology Computer

Aided Design (TCAD). Briefly, TCAD deals with the assembly of large equation systems by utilizing discretized partial differential equations from different fields of physics. All types of PDEs (elliptic, parabolic, hyperbolic) have to be considered for the various types of problems from the fields of semiconductor simulation [9]. Different grid types and dimensions of topological elements, linear and nonlinear solvers with their associated numerical issues have to be considered during application development and demand great care to ensure high software quality while also addressing performance issues.

Our institute has a long history in developing such applications [10, 11, 12, 13, 14]. In early years only one- and two-dimensional data structures were used, due to the limitations of computer resources. The imperative programming paradigm was sufficient for this type of task [11]. With the improvement of computer hardware and the advent of the object-oriented programming paradigm, the shift to more complex data structures was possible. More complexity is added when modeling requires a change of the underlying topological data structure, usually from regular to irregular grids. Additional complexity is introduced by changes in the solver mechanisms or through the use of different types of data, e.g., vectorial or tensorial data [15]. The most drastic changes usually result from a change in the discretization scheme, or the mathematical problem formulation itself that is derived from PDEs [9, 16].

The main motivation for the GTL was the circumstance that a detailed analysis of the tools developed at our institute has shown the following distribution between the amount of source code for data structures and algorithms:

Name	Year	DS	Algorithm	Reference
`MINIMOS`	1980	60 %	40 %	[9]
`S*AP`	1989	60 %	40 %	[12]
`MINIMOS-NT`	1996	70 %	30 %	[13]
`ELSA`	1999	70 %	30 %	[17]
`WSS`	2000	90 %	10 %	[14]

Most of these applications use data structures such as `list` and `array` as well as triangles, quadrilaterals, tetrahedra, cuboids, each with their own different access and storage mechanisms, and iteration operations. Although these tools use the C++ STL to some extent, the overall application design is not based on generic libraries. For this reason, the number of source lines is growing quickly due to the complex requirements of two and three-dimensional problems. The currently used applications exceed the limit of maintainability greatly.

This was the start for our own analysis related to data structures and different programming paradigms in TCAD. Our analysis then revealed that, up to now, none of the investigated libraries (BGL, GrAL) can be used directly. For lower-dimensional applications (0D, 1D) the libraries suffer from higher-dimensional information, such as incidence or adjacence. Applications, based on libraries, which use different types of grids (triangles, tetrahedra, cubes) were always outperformed by manually tuned applications. However, for the field of scientific computing, it is essential to abstract from the iteration mechanism, dimensionality, and type of the underlying cell complex.

3. FORMAL SPECIFICATION

This section introduces the basic notation of topological spaces and cell complexes in our approach. In Figure 1 we present an overview of the terms used.

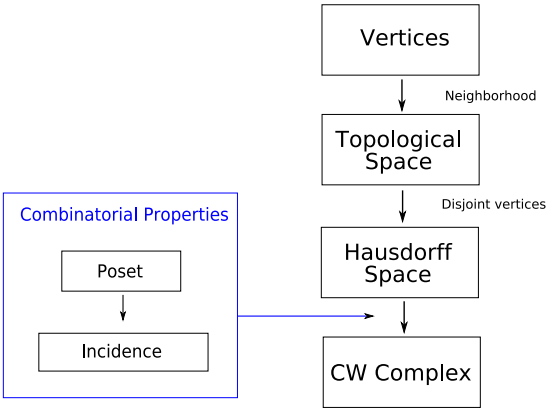

Figure 1: Basic mathematical formalism.

Of particular interest are the combinatorial properties of a CW-complex to characterize different data structures of arbitrary dimensions. Hence, we introduce the formal specification of a CW-complex [18] first. A complete introduction of all terms is available in [2, 18].

Definition: ***CW-Complex*** $\mathcal{C}$***, [18]***
A pair $(\mathcal{T}, \mathcal{E})$, with $\mathcal{T}$ a Hausdorff space and a decomposition $\mathcal{E}$ into cells is called a CW-Complex, iff the following axioms are satisfied:

- mapping function: for each n-cell $c \in \mathcal{E}$ a continuous function $\Phi_e : D^n \rightarrow \mathcal{T}$ exists, which transforms D^n homeomorphically onto a cell c and S^{n-1} in the union of maximal $(n-1)$ dimensional cells. D^n represents an n-dimensional ball and S^{n-1} represents the $n-1$ cell complex.
- finite hull: the closed hull(c) of each cell $c \in \mathcal{E}$ connects only with a finite number of other cells.
- weak topology: $A \subset \mathcal{T}$ is open, iff each $A \cap \text{hull}(c)$ is open.

An n-cell describes the cell with the highest dimension:

- zero-dimensional (0D) cell complex: vertex
- one-dimensional (1D) cell complex: edge
- two-dimensional (2D) cell complex: triangle

For this work, the most important property of a CW-complex can be explained by the usage of different n-cells and the consistent way of attaching sub-dimensional cells to the n-cells. This fact is covered by the mapping function. From now, we use an abbreviation to specify the CW-complex with its dimensionality, e.g., a 1-cell complex describes a one-dimensional CW-complex. An illustration of this type of cell complex is given in Figure 2.

In the regime of data structures the requirements of a CW-complex, the finite hull and weak topology, are always satisfied due to the finite structure. The underlying topology of a CW-complex used in computer data structures is always generated from the power set $\mathcal{P}(X)$. For this reason, the topological space cannot be used directly to characterize the different data structural properties. An example is the topological space of a random access container specified by

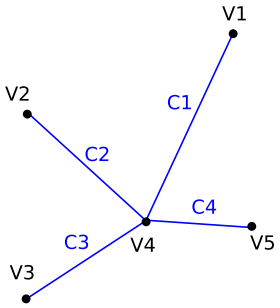

Figure 2: Representation of a 1-cell complex with cells (edges, C) and vertices (V).

the following code line:

```
std::vector<int> container(3);
```

The topological space $\mathcal{T}$ is described by the power set which models the arbitrary access of this container.

$$\mathcal{T} = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\}$$

For this reason we introduce the concept of a topological neighborhood [18].

Definition: ***Neighborhood***
A subset $A \subseteq X$ of a topological space $\mathcal{T}$ is a neighborhood of an element $p \in X$, iff it contains an element O of $\mathcal{T}$ that contains p.

$$A \subseteq X \text{ neighborhood of p} \iff \exists \mathcal{O} \in \mathcal{T} : \text{p} \in \mathcal{O}, \mathcal{O} \subseteq \mathcal{A}$$

A *base of neighborhoods* at $p \in X$ is a set of neighborhoods of p such that every neighborhood of p contains one of the base neighborhoods. We introduce the notion of *bn* which describes the number of elements of the base of neighborhoods. Different data structures can be uniquely characterized by this number. To illustrate this term we present the following list data structure:

```
std::list<int> container(4);
```

$\mathcal{T}$ is also described by the power set but the base of neighborhood can be used to characterize the list. The following sub-set of the topology represents the base of neighborhood of the list:

$$\mathcal{T}_i = \{\{0,1\}, \{1,2\}, \{2,3\}, \{3,4\}\}$$

Next, we introduce the combinatorial properties of a cell complex. These properties are responsible for the internal layout of data structures, as well as for the iteration mechanisms of these data structures.

With the assumption of cell complexes and the base of neighborhoods we introduce the following term:

Definition: ***Adjacence and Incidence***
Given two sets $a, b \in \mathcal{T}$, we define a binary adjacence relation $\mathcal{R}_{\text{adj}}(a,b)$ with the following properties:

$$\mathcal{R}_{\text{adj}}(a,b) : \iff a \cap b \neq \emptyset$$

As a special case of adjacence we define the incidence relation $\mathcal{R}_{\text{in}}(a,b)$:

$$\mathcal{R}_{\text{in}}(a,b) : \iff a \cap b = a \vee b$$

The incidence relation gives the possibility of an iteration of a topological spaces, using only the definition of a base of neighborhoods which separates the combinatorial properties of our underlying topological spaces.

To define higher-dimensional cell complexes, a mechanism is introduced which handles the internal structures of cells. The topological space of, e.g., a triangular grid is described by the vertex on cell information. The number of elements of a sub-set does not give any information about the internal structure of this element. The sub-set $\mathcal{T}_j = \{1,2,3,4\}$ can describe a tetrahedron in three dimensions or a quadrilateral in two dimensions. In order to be able to distinguish these different element, we introduce the concept of a poset:

Definition: ***Poset, [19]***
A poset ($\mathcal{S}$,<) is a finite set $\mathcal{S}$, together with a partial order relation.

In the case of a cell complex, the partial order relation is described by incidence. A Hasse diagram can be used to visualize the poset of a cell. Any two elements are connected by a line, if they are comparable.

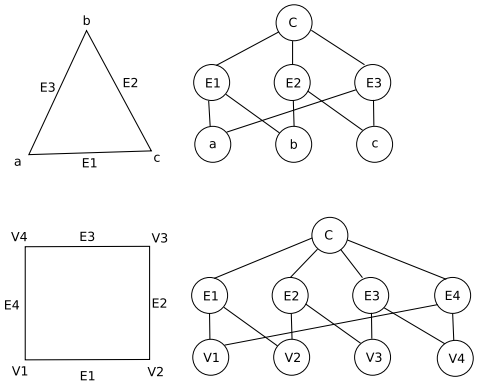

Figure 3: A Hasse-diagram for a triangle cell (top) and a quadrilateral cell (bottom).

With the Hausdorff property of the CW-complex we can uniquely characterize cells or faces by their set of vertices. We define $\{a,b,c\}$ as the element which exactly contains the vertices a, b, c.

Another important property is the locality of the cell complex. Two different properties can be distinguished, which represent the arbitrary and the iterative access of data structures.

Definition: ***Global Cell Complex***
A cell complex $\mathcal{C}$ which is homeomorphic to the following combinatorial structure of cells [2], where i_d represents the dimensional ticks:

$$\{[i_1, i_1+1] \times .. \times [i_d, i_d+1] \mid 0 \leq i_j \leq m_j\}$$

is called a cell complex with *global* properties. Here the topological incidence relation is apparent from the fact that global information is explicitly available. This property is important because of the fact, that a global cell complex describes the *random access* container types.

Definition: *Local Cell Complex*
Conversely, a cell complex which cannot be described globally is called a local cell complex.

In scientific computing, neighborhood information of a local cell complex has to be stored explicitly. Due to the non-trivial construction of instances of cell complex types, we refer to literature [20]. Related to data structures, a local cell complex models different types of lists, trees, or maps.

4. GENERIC TOPOLOGY LIBRARY

In this section we introduce the basic idea of the underlying cell complex for data structures. The classification of each data structure is using the dimension of the cells. Figure 4 shows a 0-cell complex. In this special case, cells and vertices are identical. No neighborhood information is given, only the cells are depicted. This topological structure covers most of the STL data structures. The differences between each of the data structures such as `std::vector` and `std::list` can be found in the *base of neighborhoods* and the *incidence relation* or, in other words, in their combinatorial properties.

The internal mechanism and utilization of the internal structure of the data structure is not possible due to the 0-cell complex, which means that no higher incidence or adjacence (see Section 3) is available directly. No data can be stored on edges or cells easily.

Generic algorithms cannot always use the internal structure of, e.g., `std::map` or `boost::graph` without modification. Copying a map or graph could be much more efficient, if the algorithms were aware of different internal topologies of data structure, such as the tree structure of a map.

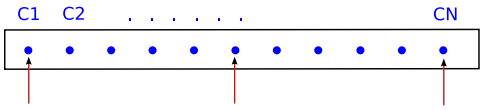

Figure 4: Iteration over cells within a 0-cell complex.

As already mentioned, applications designed in the field of scientific computing need higher-dimensional data structures as well as higher-dimensional iteration operations. Consider, for example, a 1-cell complex (Figure 5) and a 2-cell complex (Figure 6).

In the case of the 1-cell complex, the basic concept of *incidence* is mostly covered. There are only edges and vertices, and most of the operations on these two elements can be implemented with basic methods. For higher-dimensional cell complexes, e.g., a 2-cell complex, the incidence relation becomes more complex. There are various permutations of incidence relations which all lead to a different iteration. All vertices connected to a triangle, or all edges which are part of the triangle can be traversed. Also adjacent iteration can be derived easily.

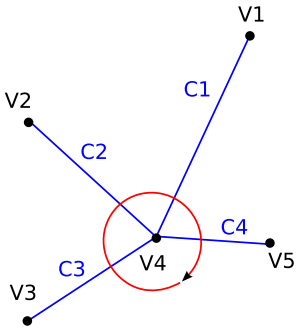

Figure 5: Iteration over edges for a 1-cell complex.

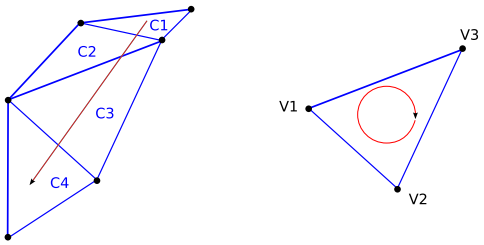

Figure 6: Iteration over cells and incident vertices of a cell for a 2-cell complex.

4.1 Topological Properties of Data Structures

We can now show, based on the formal definitions in Section 3, that we can derive a consistent categorization of different data structures and therewith a homogeneous interface which does not restrict the dimensionality or iteration mechanism of the data structures. In the following table we characterize common data structures with their combinatorial properties. The used terms are:

- dim: dimension of the cell complex
- locality: refers to the local or global combinatorial properties of the underlying space
- bn: represents the number of the elements of the base of neighborhoods of the cell

SLL stands for single-linked-list whereas DLL means double-linked-list. A global defined cell complex does not require a base of neighborhood due to the fact, that the neighborhood is implicitly available.

data structure	dim	locality	bn
array/vector	0	global	
SLL/stream	0	local	2
DLL/binary tree	0	local	3
arbitrary tree	0	local	4
graph	1	global	
regular grid	2	global	
irregular rid	2	local	4
regular grid	3	global	
irregular grid	3	local	5

The next code snippet presents our topological data structure definition. The first number stands for the actual dimension, the tags *global* and *local* stand for the combinato-

rial property, and the final number specifies the number of elements of the base of neighborhoods.

```
topology<0, global  >  topo;  // array
topology<0, local, 2>  topo;  // SLL/stream
topology<2, global  >  topo;  // regular grid
topology<2, local, 4>  topo;  // irregular grid
```

For a 0-cell complex the STL iterator traits can be used to classify the data structure easily:

```
topology<0, random_acess>   topo;  // global
topology<0, forward>        topo;  // local, 2
topology<0, bidirectional>  topo;  // local, 3
```

Based on this formulation, an automatic mechanism is possible to derive optimal data structures based on the requirements of algorithms.

To show the implementation with the GTL and equivalence of the data structure compared to the STL vector a simple code snippet is presented:

Equivalence of data structures

```
typedef topology<0, random_access> topo_t;
typedef long                       data_t;

typedef cell_t<topo_t, data_t>     container_t;
container_t  container;

// is equivalent to

std::vector<data_t>       container;
```

Here, the separation of the topological structure specification can be clearly observed.

4.2 Finite Cell Complexes

This section deals with the analysis of the data structures from the STL and BGL and generalize these expressions to arbitrary-dimensional data structures. We show that all different data structures model a common interface and each dimension can use specializations to obtain the best performance.

4.2.1 The 0-Cell Complex

A typical representative of a 0-cell complex is the topological structure of a simple array. The C++ STL containers such as `vector` and `list` are representatives and are schematically depicted in Figure 7. The points represent the cells on which data values are stored.

Figure 7: Representation of a 0-cell complex with a topological structure equivalent to a standard container.

Iteration and data access is used simultaneously in the basic iterator concept of the STL. The next code snippet presents these facts, where the forward iteration `++it` is used to traverse the cells. The `*it` is used to access the value attached to the cell at position `it`.

C++ STL approach

```
std::vector<int>           container;
std::vector<int>::iterator it;

it = container.begin();
++it;                 //topological traversal
int value = *it;      //data access
```

On the one hand side, the iterator concept is one of the key elements of the C++ STL. It separates the actual data structure access from algorithms. Thereby the implementation complexity is significantly reduced. On the other hand side, it combines iteration and data access. The improvements of separating the iteration and data access are outlined with a cursor and property map concept [4]. A possible application of this approach is demonstrated in the next code snippet:

Separated iteration and data access

```
vector<bool>           container;
vector<bool>::iterator it;
property_map           pm(container);

it = container.begin();
++it;                  //iteration
bool value = pm(*it);  //data access
```

The `std::vector<bool>::iterator` can be modeled by a random access iterator [4], whereas the data access returns a temporary object which can be used efficiently [21] with modern compilers. Additionally, this mechanism offers the possibility of storing more than one value corresponding to the iterator position. This feature is especially useful in the area of scientific computing, where different data sets have to be managed, e.g., multiple scalar or vector values on a vertex, face, or cell.

Based on the formal classification of Section 3 we analyze the combination of iteration and data access in more detail. The following list overviews the basic iterator traits [22]:

- input/output
- forward
- bidirectional
- random access

As we have seen, there is a unique and distinguishable definition possible for all of these data structural properties. On the one hand side, the backward and forward compatibility of the new iterator categories are a major problem [23]. On the other hand side, problems are encountered, if we integrate the iterator categories into our topological specification. In the following the replacement for the input and output traits is listed:

- incrementable
- single pass
- forward

The combinatorial property of the underlying space of these three categories is the same: a 0-cell complex with a local topological structure, defined by the following code snippet:

```
topology<0, local, 2>  tp;
```

The old iterator properties have only used two different categories which specify the data behavior, namely the input and output property.

The difference between these three categories can be described by:

- incrementable: this is a topological property only
- single pass: this is a data property only
- forward: this combines the incrementable and single pass properties

Only the incrementable property can be described by a topological property, whereas the other two categories are data dependent.

4.2.2 The 1-Cell Complex

This type of cell complex is usually called a graph. Figure 2 presents a typical example. A cell of this type of cell complex is called an *edge*. Incidence and adjacence information is available between edges and vertices.

We give examples on simple algorithms based on graphs using the BGL. The BGL implements comprehensive and high performance graph treatment capabilities including the associated adjacence and incidence relation. Iteration and data access are separated by the already mentioned cursor and property map concept [3]. The next code snippet presents an iteration using mechanisms of the BGL. In this algorithm all edges are traversed.

BGL iteration

```
typedef adjacency_list<vecS, vecS> Graph;
Graph gr(number_of_points);

// edge initialization

edge_iterator eit, eit_end;

for (tie(eit, eit_end) = edges(gr);
     eit != eit_end; ++eit)
{
  test_source1 += source(*eit, gr);
  test_source2 += target(*eit, gr);
}
```

With the GTL the same functionality can be accomplished as demonstrated in the following code snippet. The `global` keyword is used to highlight the global structure of the graph, which means, that the internal data layout is prepared for a dense graph storage.

GTL iteration

```
typedef topology<1, global> topo_t;
topo_t topo(number_of_vertices);

// cell initialization

cell_on_vertex_it covit, covit_end;

for (tie(covit, covit_end) = cells(topo);
     covit != covit_end; ++covit)
{
  test_source1 += source(*covit, topo);
  test_source2 += target(*covit, topo);
}
```

4.2.3 The ND Cell Complex

We extend the 0-cell and 1-cell complex types to arbitrary-dimensional cell complexes. In this work we restrict the topological spaces to the most important to scientific computing: the local (Figure 8) and the global cell complex (Figure 9). Based on our cell complex types the following cell types are available:

- 0-cell: vertex
- 1-cell: edge
- 2-cell: triangles, quadrilaterals
- 3-cell: tetrahedra, cubes

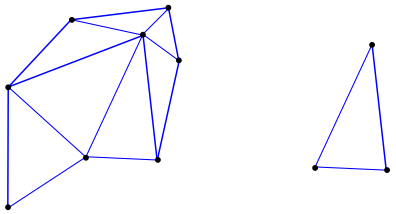

Figure 8: Local cell complex (left) and a cell representation (right). Vertices are marked with black circles.

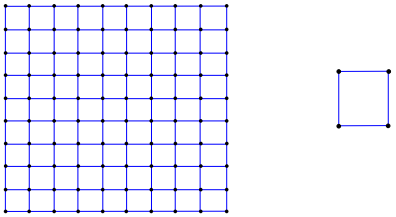

Figure 9: Global cell complex (left) and a cell representation (right).

The following code snippet presents the implementation of an arbitrary topology with the structure of a local 2-cell complex. The stored data is based on scalar values using a `double` for representation.

Iteration with our approach

```
typedef topology<2, local, 4>          topo_t;
topology_traits<topo_t>::iterator      it;

typedef data<scalar, double>           data_t;
data_t                                 data;
data_traits<data_t>::value             value;

it = topo.vertex_begin();

++it;                    // iteration
value = data(*it);       // access
```

The next example presents an iteration mechanism starting with an arbitrary cell iterator evaluated on a cell complex, which is an instance of a topological cell complex. Then a vertex on cell iterator is initialized with a cell of the complex. The iteration is started with the `for` loop. During this loop an edge on vertex iterator is created and initialized with the evaluated vertex. This edge iterator starts the next iteration. The corresponding graphical representation is given in Figure 10. The necessary *valid()* mechanism models a *circulator* concept [24]. The objects marked depict the currently evaluated objects. In the first iteration state the vertex `v1` is used and the iteration is performed over the incident edge, then the iteration continues with the remaining vertices.

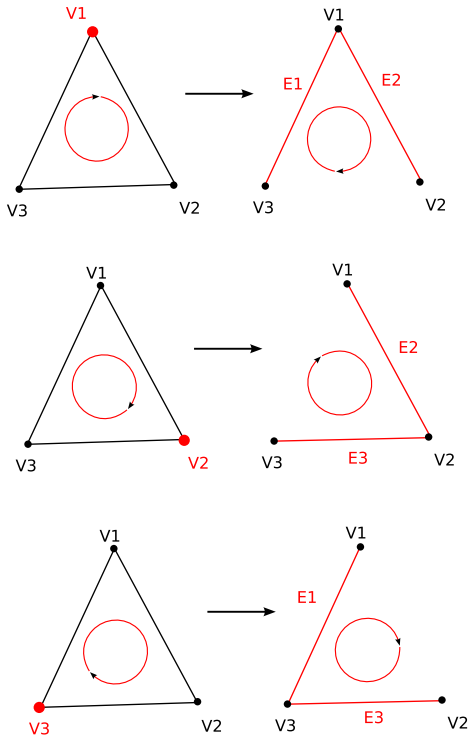

Figure 10: Incidence relation and iteration mechanism.

A more complex iteration

```
cell_iterator ce_it = topo.cell_begin();

vertex_on_cell_iterator vocit(*ce_it);
for(; vocit.valid(); ++vocit )
{
  edge_on_vertex_iterator eovit(*vocit);

  for (; eovit.valid(); ++eovit )
  {
    //operations on edges
  }
}
```

As can be seen, the iteration mechanism can be used independently of the used dimension or type of cell complex. The iteration is initialized with a cell iterator only. Three different objects have to be assured by the cell complex: vertices, edges, and cells. All cell complex types which support these three objects can be used for this iteration.

5. DATA ACCESS

We use the *property map* concept by a functional access mechanism called *data accessor*. The data accessor implementation also takes care of accessing data sets with different data locality, e.g., data on vertices, edges, facets, or cells. This locality is specified by the given key `key_d`. During initialization the data accessor `da` is bound to a specific cell complex with that key. The `operator()` is evaluated with a vertex of the cell complex as argument. The next code snippet presents this assignment briefly.

Data assignment

```
string key_d  = "user_data";
data_t da = scalar_data(topo, key_d);

da(vertex) = 1.0;
```

In the following code snippet, a simple example of the generic use of this accessor at run-time is given, where a scalar value is assigned to each vertex in a domain. The data accessor creates an assignment which is passed to the `std::for_each` algorithm.

Data assignment

```
da_t da = scalar_data(topo, key_d);

for_each
(
 topo.vertex_begin(),
 topo.vertex_end(),
 da = 1.0
);
```

Another example is given, where the data accessor is combined with the topological structure to completely specify a container. The data accessor can be used independently.

Equivalence of data structures

```
typedef topology<0, random_access> topo_t;
typedef long                       data_t;
container_t<topo_t, data_t>        container;

// is equivalent to

std::vector<data_t>        container;
```

6. GTL ARCHITECTURE

The GTL is based on a layered concept, which means that the iteration mechanism and data access mechanisms are orthogonal (Figure 11). The lowest layer represents the concepts for cell, vertex, and the poset information. The other part of the lowest layer implements the data storage. It can be observed that data can be handled independently of the topological information and iteration. The second layer provides the incidence relation and the data accessor mechanisms.

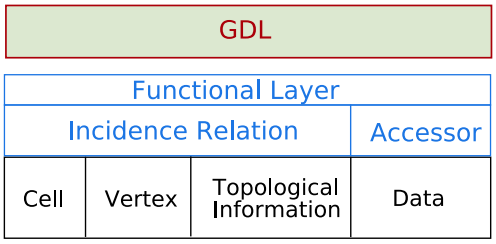

Figure 11: Conceptual view of the GTL.

The highest level in the GTL is based on meta and functional programming for a convenient usage of the different iteration mechanisms. To illustrate these mechanisms different examples are presented. The first snippet shows a simple functional iteration:

Functional iteration

```
typedef topology<2, random_access>  topo_t;
typedef long                        data_t;
container_t<topo_t, data_t>         container;

gtl::iterate<vertex_on_cell>
[
  std::cout << _1 << std::endl
](container);
```

With the GDL, different algorithms can be used as well as presented in the next example. Here different topological containers can be traversed and the data is accumulated and printed.

GTL iteration with GDL mechanisms

```
typedef topology<2, random_access>  topo_t;
typedef long                        data_t;

container_t<topo_t, data_t>         container;
container_t::data_accessor          da;

gtl::iterate<cell>
[
   std::cout <<
   gdl::sum<vertex_on_cell>(0.0)[da(_1)]
   << std::endl
](container);
```

7. OUTLOOK

The field of scientific computing requires an efficient notation of equation systems, has to construct equations, and has to abstract from the iteration mechanisms of different underlying objects. Various algorithms in the field of scientific computing only depend on the combinatorial properties of the underlying space. Using only combinatorial information results in more stable algorithms.

By providing a concise interface to different kinds of data structures, a new type of equation specification is made possible. In this way algorithms and equations can be specified independently from dimension or topological cell complex types.

To show the requirement for the equation specification we use a simple equation system resulting from a self-adjoint PDE type. Figure 12 presents a local patch of a 1-cell complex on which the equation is evaluated.

The data `Aij` represents the area of the dual graph (Voronoi graph). Using a finite volume discretization scheme [9] a generic Poisson equation $\mathrm{div}(\varepsilon\ \mathrm{grad}(\Psi)) = \varrho$ can be formulated in two spatial dimensions as:

$$\sum_j D_{ij}\ A_{ij} = \varrho \tag{1}$$

$$D_{ij} = \frac{\Psi_j - \Psi_i}{d_{ij}}\ \frac{\varepsilon_i + \varepsilon_j}{2} \tag{2}$$

D_{ij} stands for the projection of the dielectric flux onto the cell/edge c_i that connects the vertices v_i and v_j. The direct transformation of each equation element can be observed clearly when considering the following source code:

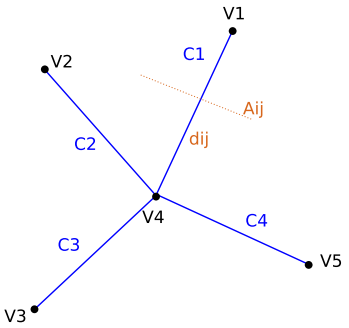

Figure 12: Cell complex with corresponding data.

Generic Poisson equation

```
value =
(
  gdl::sum<vertex_edge>
  [
     gdl::diff<edge_vertex>
     [
       Psi(_1)
     ] * A(_1)/d(_1) *
     gdl::sum<edge_vertex>[epsilon(_1)] / 2
   ] - rho(_1)
)(vertex);
```

The term `Psi` represents the distributed data set, `A` the Voronoi area, `d` the distance of two points, `rho` the right hand side, and `epsilon` some material property. It is important to stress that all data sets have to be evaluated in their right data locality, that is `Psi`, `epsilon`, and `rho` on vertices and `A`, `d` on the incident edges. The example uses the unnamed function object `_1` only. The data accessor implementation handles the correct access mechanism. The GDL implements mechanisms to derive the correct data locality of each unnamed object. An in-depth discussion is given in [8] The complex resulting from this mapping is completed by specifying the current vertex object `vertex` at run-time.

8. CONCLUSION

We have shown that the specific properties of different data structures can be specified by means of topological and combinatorial properties. An automatic derivation of optimal data structures based on the requirements of algorithms is possible.

Based on the topological properties the iterator traits can be derived automatically from combinatorial properties of the corresponding data structure. A concise iteration mechanism for different dimension is presented which includes the STL containers as well as higher-dimensional cell complex types. Different issues of the currently used iterator mechanism can be easily explained.

The full power of functional programming is revealed, when it performs different types of topological traversal with our approach.

9. ADDITIONAL AUTHORS

Siegfried Selberherr, email: selberherr@iue.tuwien.ac.at

10. REFERENCES

[1] G. D. Reis and J. Jarvi, "What is Generic Programming?" in *Library Centric Sofware Design, OOPSLA*, San Diego, CA, USA, October 2005.

[2] G. Berti, "Generic Software Components for Scientific Computing," Dissertation, Technische Universität Cottbus, 2000.

[3] J. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[4] D. Abrahams, J. Siek, and T. Witt, "New Iterator Concepts," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. N1477 03-0060, 2003.

[5] D. Gregor, J. Willcock, and A. Lumsdaine, "Concepts for the C++0x Standard Library: Iterators," ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, Tech. Rep. N2039=06-0109, June 2006.

[6] *Boost Lambda Library*, Boost, http://www.boost.org. [Online]. Available: http://www.boost.org

[7] *Boost Phoenix*, Boost, 2004, http://spirit.sourceforge.net/. [Online]. Available: http://spirit.sourceforge.net/

[8] M. Spevak, R. Heinzl, P. Schwaha, T. Grasser, and S. Selberherr, "A Generic Discretization Library," in *Library Centric Sofware Design, OOPSLA*, Portland, OR, USA, October 2006.

[9] S. Selberherr, *Analysis and Simulation of Semiconductor Devices*. Wien–New York: Springer, 1984.

[10] S. Selberherr, A. Schütz, and H. Pötzl, "MINIMOS—A Two-Dimensional MOS Transistor Analyzer," *IEEE Trans. Electron Dev.*, vol. ED-27, no. 8, pp. 1540–1550, 1980.

[11] S. Halama, C. Pichler, G. Rieger, G. Schrom, T. Simlinger, and S. Selberherr, "VISTA — User Interface, Task Level, and Tool Integration," *IEEE J.Techn. Comp. Aided Design*, vol. 14, no. 10, pp. 1208–1222, 1995.

[12] R. Sabelka and S. Selberherr, "A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures," *Microelectronics Journal*, vol. 32, no. 2, pp. 163–171, 2001.

[13] 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.

[14] T. Binder, A. Hössinger, and S. Selberherr, "Rigorous Integration of Semiconductor Process and Device Simulators," *IEEE Trans.Comp.-Aided Design of Int. Circ. and Systems*, vol. 22, no. 9, pp. 1204–1214, 2003.

[15] W. Benger, "Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model," Dissertation, Freie Universität Berlin, 2004.

[16] P. A. Markowich, C. Ringhofer, and C. Schmeiser, *Semiconductor Equations*. Wien-New York: Springer, 1990.

[17] A. Sheikholeslami, E. Al-Ani, R. Heinzl, C. Heitzinger, F. Parhami, F. Badrieh, H. Puchner, T. Grasser, and S. Selberherr, "Level Set Method Based General Topography Simulator and its Applications in Interconnect Processes," in *Intl. Conf. on Ultimate Integration of Silicon*, Bologna, Italy, July 2005, pp. 139–142.

[18] K. Jänich, *Topologie*. Heidelberg: Springer, 2001.

[19] A. J. Zomorodian, "Topology for Computing," in *Cambridge Monographs on Applied and Computational Mathematics*, 2005.

[20] J. R. Shewchuk, "Delaunay Refinement Mesh Generation," Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1997.

[21] J. Siek and A. Lumsdaine, "Mayfly: A Pattern for Lightweight Generic Interfaces," in *Pattern Languages of Programs*, July 1999.

[22] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[23] M. Zalewski and S. Schupp, "Changing Iterators with Confidence. A Case Study of Change Impact Analysis Applied to Conceptual Specifications," in *Library Centric Sofware Design, OOPSLA*, San Diego, CA, USA, October 2005.

[24] A. Fabri, "CGAL- The Computational Geometry Algorithm Library," 2001, citeseer.ist.psu.edu/fabri01cgal.html. [Online]. Available: citeseer.ist.psu.edu/fabri01cgal.html