Science of Computer Programming 76 (2011) 324–346

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A generic topology library

René Heinzl *, Philipp Schwaha

Institute for Microelectronics, Gusshausstrasse 27-29, Vienna, Austria

ARTICLE INFO

Article history:
Received 21 February 2007
Received in revised form 7 September 2009
Accepted 14 September 2009
Available online 30 September 2009

Keywords:
Algebraic topology
Data structures
Traversal operations
Data access
Iterator categories
Fiber bundles
Partially ordered sets

ABSTRACT

Requirements in scientific computing emerge from various areas such as algebraic topology, geometrical algebra, and differential topology with different notations. Cell and complex properties are introduced in order to derive a common specification for properties of data structures. Only topological properties are used, thereby separating the actual data storage structure from the stored data. Several theoretical topological properties are introduced, and traversal capabilities which excel current implementations are presented and accompanied by selected examples.

This work focuses on extracting these necessary mathematical concepts and introduces generic programming concepts necessary to fully transfer the mathematical concepts. Not only theoretical contributions are presented, but they are also demonstrated by means of applications in scientific computing.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Scientific computing is understood here as the collection of mathematical concepts which aid in the treatment of problems emerging from different areas of science which rely on mappings to finite spaces. Considering not only several mathematical concepts but also the computational efficiency of software implementations of these numerical methods results in high overall complexity, because several kinds of topological cell complexes and their corresponding embeddings in arbitrary spatial dimensions as well as various solving strategies have to be considered during application development. The development of software for the numerical simulation of disparate physical phenomena, such as electromagnetic wave propagation, heat transfer and mechanical deformation, is not straightforward and remains complex even today.

To combine the necessary concepts, an efficient representation and manipulation of scientific computing's problem domain requires flexible and efficient data structure specification mechanisms supporting arbitrary dimensions and topological spaces. One prominent example of flexible data structure manipulation is the C++ STL and the separation of data structures and algorithms by means of iterators. Iterators have become key elements of modern programming because they enable the formulation of algorithms independently of the data structure, which in turn make data structures exchangeable. Up to now, this approach has focused on sequence, associative, and graph data structures. This great achievement of accessing all data structures in a minimal but concise way led to the conception of generic programming. A detailed analysis of generic programming from the perspective of category theory has already been published [30]. In contrast to the approach from category theory, the work presented here deals with the definition and characterization of data structures and their inherent topological structure.

* Corresponding author. Fax: +43 15880136099.
E-mail address: heinzl@iue.tuwien.ac.at (R. Heinzl).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.09.007

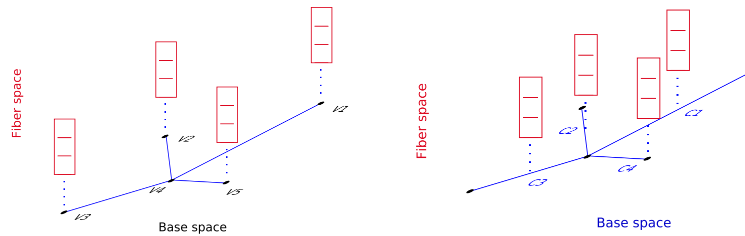


Fig. 1. Cell complex and associated data for 0-cells (left, vertices) and 1-cells (right, edges).

By using a topological characterization of data structures, two different parts of a classification scheme are introduced which can be summarized with the following expression and the distance between the topological space and the data type used:

topological space + data type = data structure

A necessary first step is to create a distinct categorization for different types for data structure. The approach presented here uses an extra notion of a topological cell concept. The dimension of a cell is then used to create a hierarchy for data structure classification. Another feature available by a cell concept is given by using a mapping from a topological space to partially ordered sets. Then generalized traversal operations can be derived enabling a common interface for different operations in a dimension and data structure independent way. On the basis of this formalization several properties of data structures (e.g., of the C++ STL) are derived and generalized to various other data structures (e.g., graphs, meshes, grids). Another directly derived concept is related to a container's topology, or more generally to the topology of a complex. This property enables a distinction to be made between global and local (dense/sequence and sparse/associative) structures. The final classification scheme is then summarized in Table 3.

The second part concerns the separation of data access and traversal [2] using the mathematical concept of fiber bundles [11]. This formal introduction enables a clean separation of the internal combinatorial properties of data structures and the mechanisms of data storage. On the basis of this concept it can be shown that application design can be greatly eased. Full backward compatibility with all existing data structures from the STL is achieved using an additional concept. Additionally, several issues of the currently used iterator categorization are studied and consistently related to the formalization given here.

The goal of the conversion of mathematical concepts presented is the minimization of the distance between the mathematical specification and the source code developed. This not only enables much faster application development but also reduces the possibility for errors. This formalized approach to topological properties and the traversal mechanism provides a new possibility of employing the functional programming paradigm which is emerging in C++ with the support of additional libraries [9,10]. Up to now, functional expressions have lacked the support of a unique interface for all different kinds of data structure iterations. By enabling a functional specification mechanism, stateless programming and hence parallelization of the final application become available.

A short example is given to introduce the application of generic algorithms by closely modeling the mathematical concepts. Discretized partial differential equations require physical quantities mapped to different locations in the abstractions of physical space. Using a finite volume discretization scheme [18] a generic Poisson equation $\mathrm{div}(\varepsilon\ \mathrm{grad}(\Psi)) = \varrho$ can be formulated in two spatial dimensions as

$$\sum_j D_{ij}\, A_{ij} = \varrho \quad D_{ij} = \frac{\Psi_j - \Psi_i}{d_{ij}} \frac{\varepsilon_i + \varepsilon_j}{2} \tag{1}$$

Quantities require different storage locations, such as D_{ij}, A_{ij}, d_{ij} on 1-cells (edges), and Ψ_i, ε_i on 0-cells (vertices). The value A_{ij} represents the area of the dual graph (Voronoi graph). 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. Fig. 1 depicts the different storage locations.

The direct transformation of each equation element can be observed clearly when considering the following source code:

Generic Poisson equation

```
value =
(   sum<edge>
    [
      diff<vertex>[ Psi ] * A / d  * sum<vertex>[eps] / 2
    ] - rho
)(vx);
```

The term `Psi` represents the desired solution quantity, `A` the Voronoi area, `d` the distance of two vertices, `rho` the right hand side, and `eps` a material property. The complex resulting from this mapping is completed by specifying the current vertex object `vx` at run-time.

The generic programming concepts described can be seen as an abstraction and unification of the following developments:

- The formalization of data structures is based on GrAL [5], introduced in Section 2.
- The separation of data access and traversal is derived from the cursor and property map concept [2] or, in a more general way, from the theory of fiber bundles [11,4].

Section 2 reviews current approaches, and explains and highlights current as well as possible future applications. Section 3 introduces necessary mathematical concepts for the underlying data structure classification hierarchy as well as giving a brief overview for order concepts in order to explain the topological traversal capabilities before introducing fiber bundles as an abstraction for data access. The library's set-up is introduced in Section 4. Examples ranging from 0-cell complexes up to 2-cell complexes are provided, accompanied by applications of the traversal mechanisms. References to related work as well as implementation details are also discussed. Section 5 then reviews a few example applications as well as performance benchmarks for examples from the field of TCAD along with comparisons to related work.

2. Historical overview and related work

The motivation for reviewing and developing a variety of concepts for scientific computing is derived from the need for flexibility in high performance applications. Therefore, the main related works consist of libraries which are compared to highlight the importance of a clean application design in the field of scientific computing under the necessary performance constraints. Significant advances toward a flexible and generic application design have been made and are briefly overviewed chronologically:

- 1993: first generic application for scientific visualization based on fiber bundles [20];
- 1995: multi-paradigm language run-time performance (C++) equal to Fortran [38];
- 1997: emergence of the generic programming paradigm in the shape of C++'s STL [26];
- 2002: generic data structure interface and generic traversal operations [6];
- 2002: the generic graph library [32] introduces the concept of property maps;
- 2003: concepts for the separation of traversal and data access [2].

By using these approaches the effort of application development in the area of scientific computing is already greatly reduced. In particular the formal and abstract interface specification of the generic programming approach, implemented by means of static parametric polymorphism in C++, offers a formal method of specification to the user, which the compiler can verify. Reusability and orthogonality of the libraries is thereby accomplished without sacrificing overall performance.

The main goals of the presented approach can be summarized as a complete formal transfer of mathematical concepts to implementations which support full interoperability by using only orthogonal base concepts. The following concepts can therefore be summarized:

- Data structure specifications for arbitrary dimensions based on cell complex properties.
  In the following, libraries are presented which are already specialized to specific dimensions. The approach presented offers concepts regarding cell and complex topology which enable cell complex specifications in arbitrary dimensions, while additionally offering interoperability to already existing and specialized approaches.
- Application of the fiber bundle approach outside the visualization area.
  STL and related iteration capabilities separate data structures and algorithms but suffer from overly restrictive definitions. The fiber bundle separation described offers backward compatibility to this iteration approach[1] but does not suffer from the extension constraint.
- Closed (incidence and adjacency) traversal operations.
  Iteration in related works is mostly integrated into the topological environments of the libraries. By using fiber bundles the topological index space (base space) is separated from the data storage space (fiber space). This not only resolves the cursor/property map issues but also enables topological traversal structures for cell complexes of arbitrary dimension.
- High overall application performance.
  Several generic library approaches have shown that abstract generic programming does not introduce an abstraction penalty. By using the techniques already established (meta-programming, small object optimization), mathematical concepts can be transferred directly into a generic library.

Starting from this overview, related works are presented in more detail. All of the related work libraries are a great achievement in the specific fields of scientific computing. But the disadvantage always remains that these libraries were not developed with interoperability in mind, e.g., using a CGAL algorithm with GrAL data structures.

A major step towards a flexible use of data structures was developed by the Boost Graph Library (BGL [32]). This library implements a generic interface to enable access to arbitrary graph structures but hides the details of the actual implementation. This approach was one of the first approaches in the field of non-trivial data structures with respect

[1] Backward compatibility is possible by setting the index space depth to zero.

to library interoperability. The interfaces make it possible for any graph library that implements these interfaces to be interoperable with the BGL. The approach is similar to the one taken by the C++ STL to ensure the interoperability of the various algorithms and containers. The property map concept [32] was also introduced. Unfortunately the BGL was designed for graphs only and neither lower nor higher dimensional data structures can be handled. All libraries which implement and interface of this type are interoperable with the BGL generic algorithms and are presented in Section 4.1.2.

The Grid Algorithms Library (GrAL [5]) was one of the first contributions towards the unification of data structures of arbitrary dimensions for the field of scientific computing. A common interface for grids was designed which facilitated dimensionally and topologically independent access and traversal. Mathematical concepts for topological spaces were introduced and applied to grids. Applications for solving PDEs were presented; however no concrete implementation was provided. Concepts for accessing data abstractly, like property maps, were also introduced and demonstrated in applications.

The Computational Geometry Algorithm Library (CGAL [14]) is another important collection of reusable components of a great number of geometrical algorithms and data structures, such as two- and three-dimensional modules for mesh generation, Voronoi diagrams or surface mesh simplification, in a generic, library-centered approach. The main contribution of CGAL is the concept of an algebraically parameterized kernel [29] which is responsible for the actual robustness of the implementation of the mathematical operations.

FiberLib2 was inspired by Butler's vector bundle data model [11] and OpenDX and is a reimplementation of a data model which was originally conceived [4] to visualize numerical data originating from general relativity. Since the mathematics of general relativity require explicit treatment of otherwise implicitly assumed properties of space and time, designing a data model to cover the issues of general relativity improves its generality. However, this library has so far remained restricted to the field of visualization.

The Sophus C++ library [17] aims to provide coordinate-free formulations.[2] This library implements grid components for sequential and parallel high performance computing. To achieve this, a field layer for numerical discretization schemes, a tensor layer for handling various quantities, and finally an application layer with solver interfaces were developed. However, this approach suffers from severe abstraction penalties and requires a code transformation tool [3].

The most important works related to the approach at hand are generic libraries. Theoretical works are briefly summarized which share mathematical concepts:

- An arbitrary cell specification was introduced [13] in the context of continuous functions related to Voronoi diagrams. This approach closely models the *n*-cell specification of the approach presented.
- Concepts of combinatorial maps or the extended notion of generalized maps [25] are part of the field of combinatorial topology, which can be seen as a direct precursor to the approach presented here.[3] Such maps already use the notion of abstract cell complexes to some extent but do not integrate various already existing data structures. Also no implementation in the field of high performance scientific computing is given. Incidence and adjacency relations were given but suffer from the separate cell topology concept presented here.
- The notion and concepts of cell complexes applied to data structures were already used to describe the structure of images [22,23]. On top of the topological classification of images, generic algorithms were described and efficiently applied. The reason for using images is related to the internal topological structure of images.[4] No extension for the treatment of general data structures was given.[5]

No further work was found that uses a formal data structure specification for arbitrary dimensions and a separation of cell and complex topology. Although the concept of fiber bundles has been already implemented in libraries, a fully orthogonal approach within the scope of high performance generic software components is still missing.

3. Mathematical properties of data structures

The following section introduces mathematical concepts of topology and order theory to formalize the combinatorial structure of cells and the global structure of cell complexes [12]. The combinatorial properties of a cell complex for characterizing different data structures of arbitrary dimensions and the internal structure of their cells are of particular interest to this work and are summarized in Table 3.

The mathematical concepts presented naturally introduce formal requirements for data structure interfaces and enable common traversal mechanisms. Here traversal means a generalized iteration, e.g., vertices on edges or cells on vertices. A complete introduction of all terms is available [5,21]. Fig. 2 depicts an overview of the terms used and their relation to each other.

The goal of this introduction of mathematical concepts is to demonstrate that data structures can be seen as a projection onto finite CW-cell complexes. Briefly, a cell can be described by the possibility of storing the user's data. A cell complex is

2 The GTL also supports coordinate-free formulations for PDEs by using differential forms which can be naturally integrated by the fiber bundle approach. A subsequent publication will include this formulation possibility as well.

3 The current version of GTL fully implements algebraic topology concepts like chain and co-chain concepts with arbitrary coefficient functions.

4 By using the terms introduced here, an image can be described by 0-cell and a global complex topology.

5 Today this set of operations can be described by homological and co-homological concepts using chains and co-chains. The chain and co-chain concepts are also supported by the GTL by calling *n*-skeletons chains and supporting integer or arbitrary coefficients for each *n*-cell.

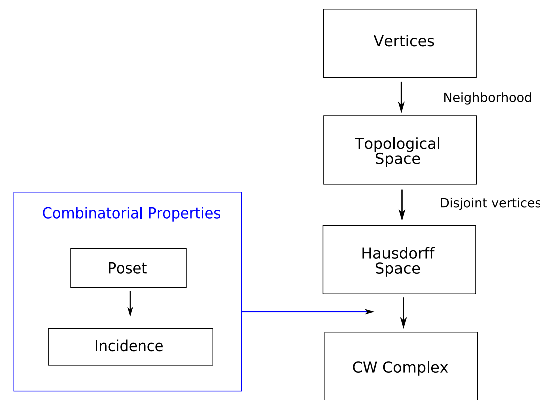


Fig. 2. Overview of mathematical concepts introduced in this section, starting with the most basic concept of a vertex which represents a single set with one element.

then a collection of well connected cells. On the basis of cell complexes several properties of data structures (e.g., of the C++ STL, graphs, meshes, grids) are overviewed and generalized. A common way to traverse these data structures based on their combinatorial properties is derived as well.

Afterwards a formal separation of traversal and data access by the means of the mathematical concept of fiber bundles is given. This separation enables a clean separation of the combinatorial properties of data structures and the mechanisms of data storage and access. Algorithms can operate independently of the used data structures.

3.1. Topology

The first part of this mathematical introduction is related to topological spaces. The formal definition of a topological space is given next:

Definition 1 (*Topological Spaces*)**.** A topological space $(X, \mathcal{T})$ consists of a set X and a family $\mathcal{T}$ of subsets of X such that:

- (T1) $\emptyset \in \mathcal{T}$ and $X \in \mathcal{T}$.
- (T2) $\mathcal{T}$ is closed under finite intersection.
- (T3) $\mathcal{T}$ is closed under arbitrary union.

The family $\mathcal{T}$ is called a topology on X, and the members of $\mathcal{T}$ are called open sets. These properties are later used to describe interdimensional elements for the complex such as edges of a cell.

In the following a basic set $X = \{a, b, c\}$ is given, where the subsets $\{a\}$, $\{b\}$, and $\{c\}$ model vertices, and one topology.

$$\begin{aligned}(X, \mathcal{T}) = \{&\emptyset,\\ &\{a\}, \{b\}, \{c\},\\ &\{a, b\}, \{a, c\}, \{b, c\},\\ &\{a, b, c\}\}\end{aligned}$$

This pair of the original set and the set of subsets generates the topological space. This particular topological space is used in several examples throughout this section.

The general definition for a topological space is very abstract and allows several topological spaces which are not useful in the field of data structures, e.g. a topological space $(X, \mathcal{T})$ with a trivial topology $\mathcal{T} = \{\emptyset, X\}$. So basic mechanisms of separation within a topological space are introduced.

Definition 2 (*Hausdorff Spaces*)**.** The topological space $(X, \mathcal{T})$ is said to be Hausdorff if, given $x, y \in X$ with $x \neq y$, there exist open sets U_1, U_2 such that $x \in U_1, y \in U_2$ and $U_1 \bigcap U_2 = \emptyset$.

A computer's data structures need the separation characteristics of a Hausdorff space. As soon as a topological space is a Hausdorff space the view of data structures correlates with this topological space. An important part of the characterization of data structures is the possibility of declaring a dimension for data structures. Here the dimension of an object, called a cell, is used, which usually accommodates a user's data.

Definition 3 (*N-Cell*)**.** A subset $c \subset X$ of a Hausdorff space X is an open cell if it is homeomorphic to the interior of the open n-dimensional ball $\mathbb{D}^n = \{x \in \mathbb{R}^n : |x| < 1\}$.

The number n is unique due to the *invariance of domain* theorem [5], and is called the dimension of c, whereas homeomorphic means that two or more spaces share the same topological characteristics [5]. The following list assigns terms corresponding to other areas of scientific computing:

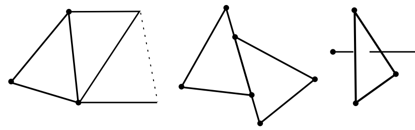


Fig. 3. Examples of violations of correct cell attachment. Left: cells of different dimensions (two 2-cells and one 1-cell). Middle: cells do not intersect at vertices. Right: intersection of cells.

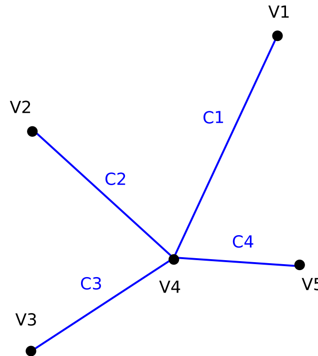


Fig. 4. Representation of a 1-cell complex with cells (X^1, edges, C) and vertices (X^0, V).

- 0-cell: point;
- 1-cell: edge;
- 2-cell: facet;
- n-cell: cell.

A collection of cells may form larger structures, so-called complexes, which are identified by the cell with the highest dimension. With the concepts already made known, the concept of a cell complex is introduced, which models the concept of a data structure directly.

Definition 4 (*CW-Complex*)**.** A CW-complex $\mathcal{C}$ is an inductively defined system of spaces $X^{(-1)} \subseteq X^{(0)} \subseteq X^{(1)} \subseteq \cdots \subseteq X^{(n)}$, constructed from pairwise disjoint open cells such that $X^{(n)}$ is obtained from $X^{(n-1)}$ by attaching adjacent n-cells to each $(n-1)$-cell where $X^{(-1)} = \emptyset$.

This CW-cell complex[6] with the underlying topological space guarantees that all interdimensional objects are connected in an appropriate manner. Although there are various possible attachments of cells, only one process results in a cell complex; see Fig. 3.

The respective subspaces $X^{(n)}$ are called the n-skeletons of the cell complex. From the viewpoint of the underlying topological space the skeletons are the subspaces of the topological space with corresponding topologies. Incidence and adjacency traversal[7] is directly related to the configuration of the space (complex topology) and the connectivity of the cells (cell topology).

For this work, the most important property of a CW-complex can be explained by the use of different n-cells and a consistent way of attaching $n-1$ cells to them. A convenient abbreviation is used to specify the finite CW-cell complex with its dimensionality, e.g., a 1-cell complex describes a one-dimensional finite CW-cell complex. An illustration of this type of cell complex is given in Fig. 4. As an example, adjacency traversal is possible by using `C1` as a starting edge, where the traversal operation returns `C2`, followed by `C3`, and `C4`.

For now, sufficient means have been introduced to handle the underlying topological space of data structures. Each subspace can be uniquely characterized by its dimension. A simple container can be classified as a 0-cell complex because of the absence of higher dimensional cells (edges). A graph can be classified as a 1-cell complex because of the existence of 1-cells (edges). All of these subspaces are connected appropriately by the concept of a finite CW-cell complex and summarized in Table 1.

A common interface for characterizing data structures is now available. A drawback of this characterization can be easily seen in the case of the 0-cell complexes. The topological space and subspaces only guarantee the correct connection of its subsets, but traversal mechanisms based on these concepts cannot be derived. Relations between subsets of the topological space are required to formalize the internal structure of cells (cell topology) and the structure of cell complexes (complex topology). Incidence and adjacency traversal can then be directly derived from this concept.

6 A CW-complex (closure-finite weak topology) with arbitrary cells.

7 Examples of incidence traversal can be given by vertex on edge traversal, and for adjacency traversal by vertex to vertex traversal.

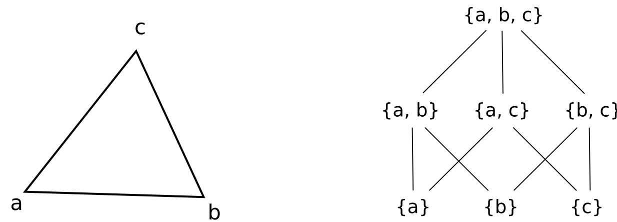


Fig. 5. Cell topology of a simplex cell in two dimensions.

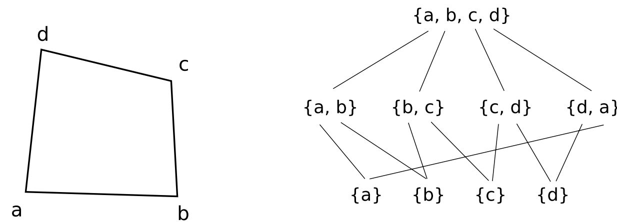


Fig. 6. Cell topology of a cuboid cell in two dimensions.

Table 1
Classification scheme based on the dimension of the cells.

Data structure	Cell dimension
Array	0
Singly linked list (SLL)/stream	0
Doubly linked list (DLL)/binary tree	0
Arbitrary tree	0
Graph	1
Grid	2
Grid	3

Table 2
Classification scheme based on the dimensions of cells and the cell topology.

Data structure	Cell dimension	Cell topology
Array	0	Simplex
SLL/stream	0	Simplex
DLL/binary tree	0	Simplex
Arbitrary tree	0	Simplex
Graph	1	Simplex
Grid	2,3,..	Polytope
Grid	2,3,..	Cuboid
Grid	2,3,..	Simplex

3.2. Cell topology

An important classification property already introduced is the dimension of a cell, e.g., 1-cells are collected in a 1-cell complex (graph), and 2-cells represent a grid or a mesh, but this leaves out the internal structure of a cell. The dimension of a cell is used to classify the whole complex, whereas the cell topology presented here represents the internal structure of a cell. This section uses a Hasse diagram to depict the internal structure, but other set/subset representations can also be used. A Hasse diagram depicts partially ordered sets (posets), where not all elements within a topological spaces can be compared, e.g., a 1-cell (edge) is not directly related to a 2-cell (triangle). By using the topology of a cell, a simplex cell is distinguishable from a cuboid[8] cell type in several dimensions as depicted in Figs. 5 and 6.

Next, interdimensional objects such as edges and facets can be identified, as can their relations to the cell. As can be seen, a complete traversal of all different objects is already determined by this structure. Vertex on edge as well as vertex on cell cases can be derived using the sets of the cell topology.

The topological characterization introduced in Table 1 can then be extended by including the cell topology and is given in Table 2. As can be seen, the 0-cell complex types do not differ at all. From dimension greater than 1, a distinction can be achieved by specifying the cell topology.

[8] The cuboid cell described here is a topological abstraction of cells with 2^n vertices, where n represents the dimension, e.g., a 2-cuboid cell represents a convex quadrilateral, and a 3-cuboid cell represents a regular convex hexahedron (cube).

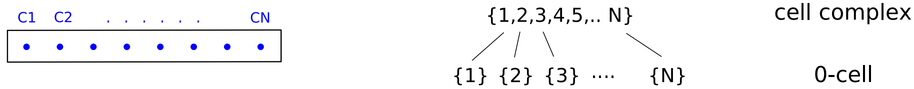


Fig. 7. Complex topology of an array.

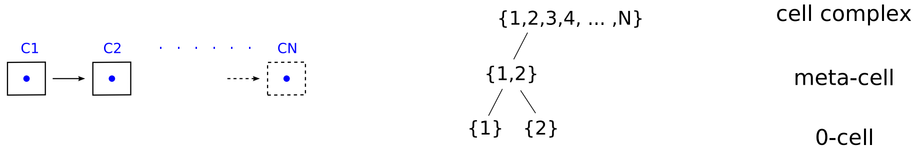


Fig. 8. Complex topology of a singly linked list.

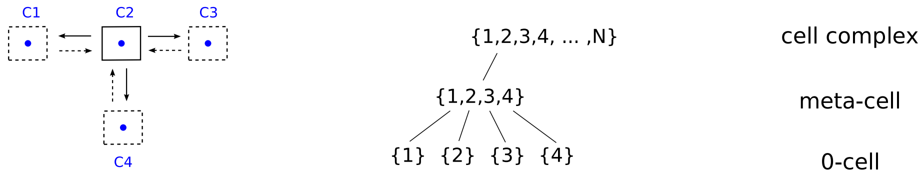


Fig. 9. Complex topology of a tree.

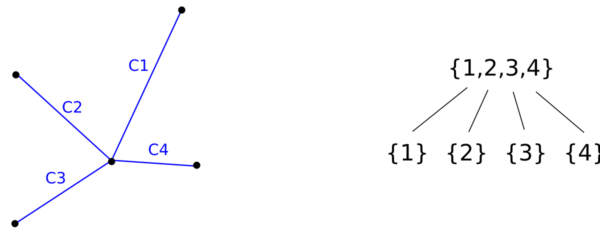


Fig. 10. Complex topology of a graph. Left: graph representation. Right: complex topology.

3.3. Complex topology

As the cell topology is used to describe the topology of a cell, this section deals with the structure of the cell complex, the complex topology. This concept finally creates the basic mechanisms required for identifying the important property of a unique classification for data structures.

In the following, several examples of different data structures within the context of 0-cell complex types are presented. The data structure first reviewed models the concept of an array. The internal topology can be described using a 0-cell, which means that no internal traversal is possible. The complex topology is based on a global property. In terms of the STL, this is called the `random access` property.

An array structure is shown[9] in Fig. 7; in contrast, Fig. 8 presents the structure of a singly linked list. Only locally neighboring cells can be traversed by this data structure which can be seen in the poset on the right. Furthermore the number of elements in the meta-cell[10] subset is bounded by a constant number, unique for each type of data structure in the 0-cell complex regime. The singly linked list uses two elements in the meta-cell, later stated as `local(2)` and called the `forward` concept in the STL.

The external topology of a tree structure is depicted, whereas the term `local(4)` is used for the characterization of the meta-cell (Fig. 9).

The next example already uses a cell dimension of 1. Fig. 10 introduces the complex topology of a graph with local properties. Vertices incident to an edge can be traversed as well as adjacent edges, as depicted in Fig. 10 on the right.

For dimensions greater than 1, Fig. 11 depicts the cell topology of a 2-simplex cell complex. As can be seen, the items of the related set and subsets are cells and meta-cells, in contrast to the sets and subsets of the cell topology diagrams. For this complex, the bottom nodes {1}, {2}, {3}, {4}, {5} and {6} represent the 2-cells. The rectangle in the figure marks the selected cell number.

The topology of the cell complex is only available locally because of the fact that the meta-cell set can have an arbitrary number of elements. In other words, there can be an arbitrary number of triangles in this cell complex attached to the innermost vertex. The final classification scheme uses the term `local` to represent this fact.

[9] The edges in the respective diagrams represent incidence relations and can be read from the top down.

[10] A meta-cell is used here as a subset which is not explicitly available within the data structure, but creates a distinct classification property.

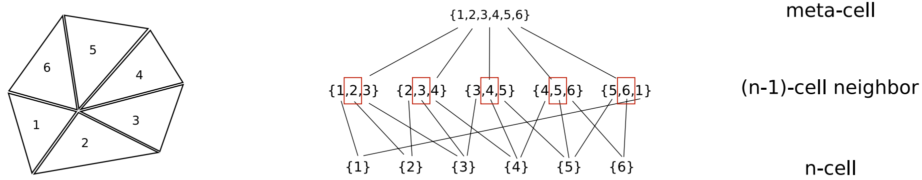


Fig. 11. Complex topology of a simplex cell complex.

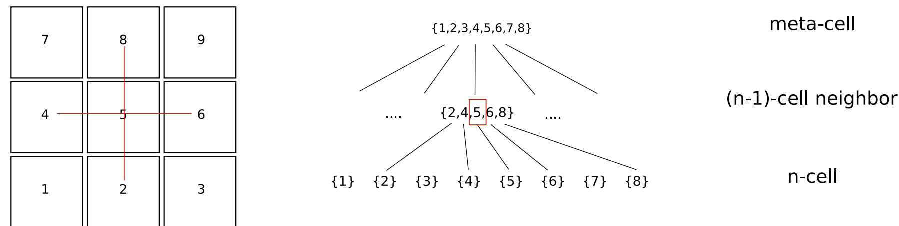


Fig. 12. Complex topology of a cuboid cell complex.

Table 3
The final topological classification scheme for various data structures of arbitrary dimension. The cell dimension is used to primarily classify data structures. Then the cell topology creates another distinct classification property for higher dimensional complex types. The complex topology provides the final property of classification for local and global storage (dense and sparse storage).

Data structure	Cell dimension	Cell topology	Complex topology
Array/vector	0	Simplex	Global
SLL/stream	0	Simplex	Local(2)
DLL/binary tree	0	Simplex	Local(3)
Arbitrary tree	0	Simplex	Local(4)
Graph	1	Simplex	Local
Regular grid	2	Cuboid	Global
Regular grid	2	Cuboid	Local
Irregular grid	2	Simplex	Local
Regular grid	3	Cuboid	Global
Regular grid	3	Cuboid	Local
Irregular grid	3	Simplex	Local

Fig. 12 presents the complex topology of a cuboid cell complex. The rectangle marks the main cell (5) under consideration. In comparison to the simplex cell complex, the depicted poset varies in one important fact: The number of elements in the meta-cell set is always constant throughout all of the cell complex.

This fact can be used to create an implicit data structure. All subsets of the cell complex can be created implicitly. On the basis of this property the topology of the cell complex can be used as a globally known property and the classification scheme uses the term `global` to represent this fact.

On the basis of this introduction, a formal, concise definition is derived, as summarized in Table 3. Section 4 yields a homogeneous interface for all different types of data structures based on this characterization. The cell topology uses a keyword to identify the internal object type while the complex topology uses the number of elements of the corresponding subspace (skeleton).

3.4. Theory of fiber bundles

This section deals with the separation of traversal of data structures and the corresponding data access. A mathematical model for this separation was introduced in 1992 [11] and most of the currently used data structures use this concept already but not to its full extent. Therefore, a brief introduction to the theory of fiber bundles and a concept to enable backward compatibility with established data structures is presented.

With the advent of the C++ STL, the separation of data structures and algorithms by the means of iterators has become one of the key elements for the generic programming paradigm, because the implementation complexity of algorithms and data structures is reduced significantly. However, iterators combine traversal and data access. The well-known example of a `std::vector<bool>` can actually be modeled by a random access container [2], but due to the currently used classification of iterators, this container cannot be used with respect to this classification type.

An alternative approach was suggested [2]: separating the traversal and data accessor into a cursor and property map concept. A possible application of this approach is demonstrated in the following 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
```

But this approach suffers from backward compatibility issues. The iterator approach cannot be used directly due to the explicitly indirect data access in another container pm.

On the basis of the formal classification introduced earlier (Section 3) the combination of traversal and data access is analyzed in more detail. The following list overviews the basic iterator traits [26]:

Iterator category	Specification	Property
Input/output		Data property
Forward	Local(2)	Topological property
Bidirectional	Local(3)	Topological property
Random access	Global	Topological property

As can be seen, there is a unique and distinguishable definition possible for all data structure properties. On the one hand, the backward and forward compatibility of the new iterator categories are a major problem [42] and are not included in the new C++0x standard [16]. On the other hand, problems are encountered if iterator categories are integrated into the introduced topological specification. In the following the replacements for the input and output traits are listed:

Iterator category	Specification	Property
Incrementable	Local(2)	Topological property
Single pass		Data property
Forward	Local(2)	Data and topological property

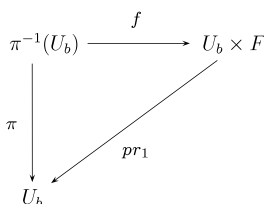
The combinatorial properties of the underlying spaces of these three categories are the same: a 0-cell complex with a local topological structure such as:

Data structure	Cell dimension	Cell topology	Complex topology
SLL/stream	0	Simplex	Local(2)

The old iterator properties have only used two different categories which specify the behavior of data, namely the input and output properties. Only the incrementable property can be described by a topological property; the other two categories are data dependent.

On the basis of these examples, the necessity of separating the topological structure from the attached data can be identified, which has already found an elegant solution in the form of the fiber bundle approach. The original contribution of this theory was given in Butler's vector bundle model [11] which is briefly reviewed here:

Definition 5 (*Fiber Bundle*)**.** A fiber bundle structure on a topological space E, with fiber F, consists of a map $\pi : E \to B$ such that each point of the topological space B has a neighborhood U_b for which there is a homeomorphism $f : \pi^{-1}(U_b) \to U_b \times F$ making the following diagram commute.



E is called the *total space*, B is called the *base space* and F is called the *fiber space*. In other words, this definition requires that a total space E can be written *locally* as the product of a base space B and a fiber space F. A direct application of these terms is provided in the following example:

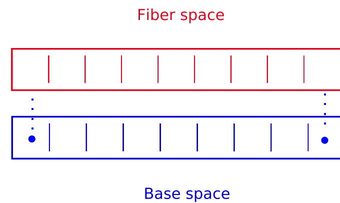


Fig. 13. A fiber space with a simple base space over a 0-cell complex.

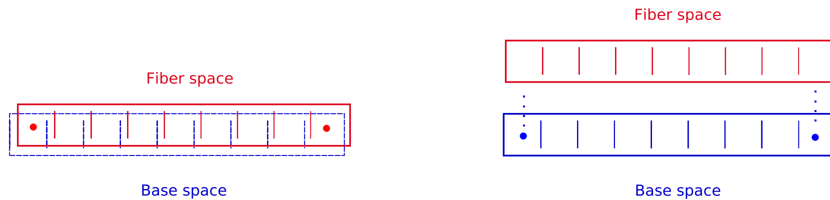


Fig. 14. Illustration of an index space depths of 0 (left) and 1 (right). The base space is modeled by a 0-cell complex.

Total space	Data structure
Base space	Cell complex (index space)
Fiber space	Attached data

The decomposition using a *base space* is enforced by a formal introduction of data structures. As an example, Fig. 13 depicts an array data structure as seen on the basis of the concept of fiber bundles. The indices of this array model the base space, which means that each 0-cell can be accessed through a global index. A simple fiber space, which preserves the neighborhood of the base space and contains the data of the array, is attached to each cell.

Employing the concept of fiber bundles, the base space is cleanly separated from the fiber space. For the base space of lower dimensional data structures, such as an array or a singly linked list, the only relevant information is the number of elements, the cardinality of the index space. On using a matrix property, this is a dense index space (in contrast to sparse index spaces). Therefore most of the data structures do not separate these two spaces. For backward compatibility with common data structures the concept of an index space depth is introduced.

As can be seen in Fig. 14, common data structures, such as arrays and lists, can be modeled with an index space depth of 0. The cursor and property map concept always model an index space depth of 1. An index space depth of 0 means that the index can be used as a value directly. An index space depth of 1 or greater means that the index value returned has to be used in another container, e.g. a fiber.

The advantage of this approach is similar to that of the cursor and property map [2] but differs in several details. The similarity is that the two properties can be implemented independently. But the fiber bundle approach equips the fiber space with much more structure, e.g., when storing more than a single value at a specific traversal 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. On the basis of the structure of the fiber space, vertical and horizontal subsets (slices) can be extracted. Another important property of the fiber bundle approach is that an equal (isomorphic) base space can be exchanged with another cell complex of the same dimension. Finally, the additional concept of an index space depth can be used, which provides full backward compatibility of all data structures in the same context as higher dimensional data structures, such as grids and meshes. The following table summarizes the common features and the differences, while examples are given in Section 4.

	Cursor and property map	Fiber bundles
Isomorphic base space	No	Yes
Traversal possibilities	STL iteration	Cell complex
Traverse base space	Yes	Yes
Traverse fiber space	No	Yes
Data access	Single value	Topological space
Fiber space slices	No	Yes
Backward compatibility	No	Yes

4. Generic topology library

This section not only gives an overview of the library, but also elucidates the transition from mathematical concepts to a library. It then proceeds to give examples for cell and complex type specifications which also encompass reviews of data structures in related libraries, which are classified according to their corresponding properties, thereby revealing

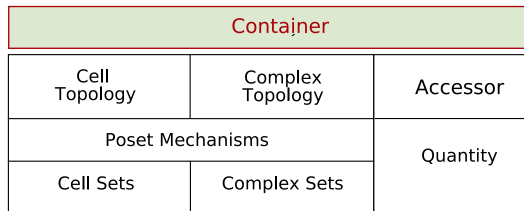


Fig. 15. Conceptual view of the GTL.

the common interface for several data structures. The library is based on layered and orthogonal concepts, which are schematically illustrated in Fig. 15. It follows from this design that each layer may be used separately.

The lowest layer represents mechanisms for handling sets. Most of this layer is implemented by means of several different data types. This layer also integrates a homogeneous access to compile time and run-time mechanisms. The next layer implements poset traversal mechanisms. Generic traversal operations use the information of the sets and specify the corresponding traversal objects. Cell and complex topology modules offer various access mechanisms based on the underlying set operations. Local and global properties of the complex topology part are implemented with their respective interfaces, as given in detail in Section 3.3. The layers on the left can be seen as the base space properties of the fiber bundle approach. The right side of the figure depicts the fiber space with the corresponding quantity and accessor layer. The implementation basically adheres the property map concept but differs in several details, as explained in Section 3.4. Finally, the container layer combines the final complex part with the data storage part, also considering the index space depth.

To guarantee an overall high performance and to avoid abstraction penalties, different times of evaluation have been made available. The cell and complex which provide all necessary information at compile time are completely optimized by the compiler and do not introduce any run-time overhead. Run-time specification is only necessary where topological information for cell or complex topology is not available at compile time, e.g., for polytope cell topologies or local complex topologies (meshes). The following table overviews the different mechanisms with their corresponding times of evaluation.

	Compile time	Run-time
Cell topology, simplex	Yes	No
Cell topology, cuboid	Yes	No
Cell topology, polytope	No	Yes
Complex topology, local	No	Yes
Complex topology, global	Yes	No

With the advent of the Fusion library [8] the interplay between compile and run-time is encapsulated cleanly. In the following the transition from the formal introduction to the notation made possible by the GTL is presented. Cells can be created by simply specifying their necessary topological properties: dimension and cell type.[11]

Cell type definition

```
cell_type<0, simplex>        cells_t;
cell_type<3, cuboid>         cellc_t;
```

Further specification for cell complexes is then possible by stating the cell types as well as the complex topology: global or local. As an example, an array, a doubly linked list, and a regular grid can be modelled concisely.

Complex type definition

```
complex_type<cells_t, global   >  complex1_t; //array
complex_type<cells_t, local<3> >  complex2_t; //DLL/tree
complex_type<cellc_t, global>     complex3_t; //regular grid
```

Data accessors model the fiber bundle approach and equip the data storage with additional structure such as base space index access, fiber space index access, and slice extraction. The container for the accessors can be chosen arbitrarily to satisfy the application requirements.

Data accessors for the container

```
value = fb::get_value(container)(base_handle, quan_key);
// or ..
value = fb::get_value(container)(*vertex_on_cell, quan_key);
```

11 Typedefs are suppressed due to space constraints. Suffixes are used to distinguish between the abstract concept type, e.g., `cell_type`, and the corresponding model type, e.g., `cells_t`. Both are types and hence carry a suffix to separate them from the instantiated object.

Here the `base_handle` (index) selects the respective fiber whereas the `quan_key` selects the corresponding fiber element. In contrast to the cursor and property map concept, the fiber bundle approach allows a traversal within the fiber space as well:

Fiber access

```
fiber = fb::get_fiber (container) (base_handle);
for (it = fiber.begin(); it != fiber.end(); ++it)
{  // ... }
```

Next, a slice can be extracted, which means that the complete data attached to a base space can be returned:

Slice access

```
slice = fb::get_slice (container)(quan_key);
for (it = slice.begin(); it != slice_end(); ++it)
{
  // ..
}
```

The following code snippet gives a simple example of the generic use of a data accessor similar to the property map concept, where a scalar value is assigned to each vertex in a domain. In the given example the data accessor is extracted from the container to enable functional access.

Data accessor

```
da_type da(container, key_d);

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

The container type is presented next, which combines the complex types (base space properties) with the data types (fiber space properties). Finally the `index_space_depth` property is used to introduce the distance of the data accessors from the base space.

Container type definition

```
typedef long                           data_t;
typedef container_type<complex1_t,
                       data_t,
                       indexspace<0> >    container_t;
container_t  container;
```

The following sections introduce several examples for cell complex types of different dimensions as well as a comparison of the GTL with the related works.

4.1. Data structure definition

By identifying data structures with finite topological spaces, or more specifically with cell complexes, concepts from topology, such as complex type, cell type, and the dimension, are available for common properties to be studied. This section illustrates the mathematical concepts introduced by giving examples of data structure equivalence of STL containers and GTL specifications. Data structure definitions of 0-cell complexes are compared to STL data structures. Then 1-cell complexes are used to compare BGL traversals and GTL traversals. It is shown that GTL and BGL are interoperable, which means that the same algorithms can be used. More complex traversal algorithms are then given for 2-cell complexes and compared to CGAL and GrAL algorithms. Not only can the algorithms from CGAL and GrAL be made available for the other libraries by the GTL, but it is demonstrated that GTL's traversal can be combined in arbitrary ways. Arbitrary dimensional cell complexes are then presented to highlight the unlimited topological specification possibilities.

4.1.1. The 0-cell complex

The 0-cell complex types are used, e.g., for the data structures within the C++ STL. The next code snippet presents an example of specification with the GTL whereas the tags *simplex* and *cuboid* stand for the cell topology type. The tags *global* and *local* specify the complex topology. GTL's internal representations are normally linked to STL data structures but can be changed easily.

STL data structure definitions

```
cell_type <0, simplex>          cells_t;

complex_type<cells_t, global   >  cx; //array
complex_type<cells_t, local<2> >  cx; //SLL/stream
complex_type<cells_t, local<3> >  cx; //DLL/tree
complex_type<cells_t, local<4> >  cx; //arbitrary tree
```

For cell complexes of this dimension the STL iterator traits may be used to classify the data structure as well:

STL data structure definitions, STL iterator traits

```
complex_type<cells_t, random_access> cx; //simplex, global
complex_type<cells_t, forward>       cx; //simplex, local<2>
complex_type<cells_t, bidirectional> cx; //simplex, local<3>
```

The equivalence of STL data structures and the GTL specification is illustrated in the following code snippet[12]:

Equivalence of data structures

```
typedef cell_type<0, simplex>                cells_t;
typedef complex_type<cells_t, random_access> complex_t;
typedef long                                 data_t;
typedef container_type<complex_t,
                       data_t,
                       indexspace<0> >       container_t;
container_t  container;

// is equivalent to

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

The separation of the topological structure and data specification can be observed clearly.

4.1.2. The 1-cell complex

This type of cell complex is usually called a graph. Fig. 10 presents a typical example. A cell of this type of cell complex is called an *edge*. Information on the incidence and adjacency of edges and vertices is available. The next code snippet presents a traversal (iteration) of all edges using mechanisms of the BGL.[13]

BGL traversal

```
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);
}
```

The same functionality can be accomplished with the GTL as is demonstrated in the following snippet of code. The `global` tag is used to highlight the global structure of the graph, which means that the internal data layout is prepared for dense graph storage. In terms of the BGL, an `adjacency graph` data structure is used. All BGL algorithms can then be used, with the GTL acting as an interface layer only.

12 The current version of GTL uses trait classes to remove unnecessary code.

13 For the BGL traversal snippet no GTL part is used.

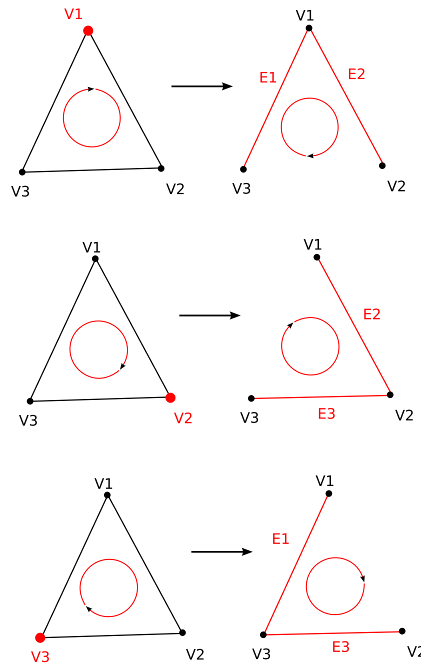


Fig. 16. Incidence relation and traversal operation.

GTL traversal

```
typedef cell_type <1, simplex>              cells_t;
typedef complex_type<cells_t, global>       complex_t;
typedef long                                data_t;
typedef container_type <complex_t,
                        data_t,
                        indexspace <1> >     container_t;

container_t container(number_of_vertices);

// cell initialization

container_t::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);
}
```

4.1.3. The 2-cell complex

This type of cell complex is usually called a grid. The following example specifies two-dimensional cell complexes; a simplex cell complex and a cuboid cell complex are depicted in Figs. 11 and 12 respectively.

Grid definitions

```
cell_type <2, simplex>             cells_t;
cell_type <2, cuboid>              cellc_t;
complex_type<cells_t, global>      cx_g;  //irregular grid
complex_type<cellc_t, local>       cx_l;  //regular grid
```

An illustration of the traversal mechanisms of the GTL is presented in the following code snippet. The traversal starts with an arbitrary cell of the 2-cell complex followed by a vertex on cell traversal, which is initialized with the selected cell of the complex. During this loop the currently evaluated vertex is used to initiate an additional edge on vertex traversal. A corresponding graphical representation is given in Fig. 16. The highlighted objects mark the currently evaluated objects.

In the first traversal state the vertex v1 is used and the traversal is performed over the incident edge, before the traversal continues with the remaining vertices.

More complex traversal

```
cell_iterator c_it = container.cell_begin();

vertex_on_cell_iterator voc_it   = container.voc_begin(*c_it);
vertex_on_cell_iterator voce_it  = container.voc_end(*c_it);

for (; voc_it != voce_it; voc_it.increment())
{
  edge_on_vertex_iterator eov_it  = container.eov_begin(*voc_it);
  edge_on_vertex_iterator eove_it = container.eov_end(*voc_it);

  for (; eov_it != eove_it; eov_it.increment())
  {   //operations on edges  }
}
```

The traversal can be used independently of the dimension or type of cell complex used. Only three different objects have to be guaranteed by the cell complex: vertices, edges, and cells. All cell complex types which support these three objects can be used for this traversal.

The next example, which implements the traversal of all facets related to a cell, compares the traversal mechanisms of the CGAL and the GTL.

CGAL traversal for the cell topology

```
// ..... Polyhedron P definition

for (Facet_iterator i = P.facets_begin();
     i != P.facets_end(); ++i)
{
  Halfedge_facet_circulator j = i->facet_begin();

  do
  { //operations on facet
  } while ( ++j != i->facet_begin());
}
```

GTL traversal for the cell topology

```
// container definition

cell_iterator c_it = container.cell_begin();

facet_on_cell_iterator foc_it   = container.foc_begin(*c_it);
facet_on_cell_iterator foce_it  = container.foc_end(*c_it);

for (; foc_it != foce_it; ++foc_it)
{      //operations on facet }
```

The next example compares the GrAL traversal with the traversal operations of the GTL.

GrAL traversal

```
typedef grid_types<Triang2D> gt;
Triang2D CC;

for( gt::CellIterator c(CC);!c.IsDone();++c)
{
  // loop over all vertices vc of cell c, print number of vc
  gt::VertexOnCellIterator vc;
  for(vc = (*c).FirstVertex(); ! vc.IsDone(); ++vc)
  {  // use *vc  }

  // loop over all neighbors
  for(gt::CellOnCellIterator cc(c); ! cc.IsDone(); ++cc)
  {   // use cc  }
}
```

GTL traversal

```
cell_type<2, simplex>          cell;    // triangle
// container definition
cell_iterator c_it = container.cell_begin();
cell_iterator ce_it = container.cell_end();

for (;  c_it != ce_it; ++c_it)
{
  vertex_on_cell_iterator voc_it   = container.voc_begin(*c_it);
  vertex_on_cell_iterator voce_it  = container.voc_end(*c_it);

  for (; voc_it != voce_it; voc_it.increment())
  {     // use *voc_it   }

  cell_on_cell_iterator coc_it   = container.coc_begin(*c_it);
  cell_on_cell_iterator coce_it  = container.coc_end(*c_it);

  for (; coc_it != coce_it; ++coc_it)
  {  // use *coc_it    }
}
```

The interfaces can be exchanged easily and each of these libraries can be used with the GTL interface. The 3-cell complex is similar if the simplex and cuboid object types are used. Therefore the corresponding figures and code snippets are omitted.

4.1.4. The ND cell complex

On the basis of the formal definition of finite cell complexes, higher dimensions can be used as well. The only extension to the GTL is the introduction of the cell and complex topology for higher dimensions. For the simple case of a four-dimensional hypercube, the cell topology can be calculated as in the other dimensions. The following code snippet presents the implementation of a 4-cell simplex complex.[14] The stored data are based on scalar values using a `long` representation.

Traversal with the approach presented here

```
typedef cell_type<4, simplex>                    cells_t;
typedef complex_type<cells_t, random_access> complex_t;
typedef long                                     data_t;
typedef container_t<complex_t, data_t, indexspace<1> >
                                             container_t;
```

5. Application and performance

This section presents several applications of the GTL, starting with traversal examples, some which exceed the possibilities provided by other libraries. Next, a performance analysis is given where the GTL is compared to native C and BGL examples. Finally, an application from the field of device simulation is given using a finite volume discretization scheme specified in a dimension and topology independent manner. By providing a concise interface to different kinds of data structures, different types of equation specification are made possible. In this way algorithms and equations can be specified independently of dimension or topological cell complex type. Here the functional programming paradigm becomes visible by using a partitioned index space and parallel assembly without source code modification [37,36].

5.1. Traversal

To introduce a simple example based on the mechanisms of the GTL a container is traversed,[15] the data accumulated and the result printed.

---

14 Due to compile time restrictions only up to 11 dimensions are currently available in GTL.

15 The `gtl::traverse` is a Phoenix function object which wraps the container iterators.

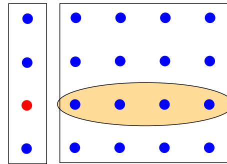


Fig. 17. Two-dimensional iteration. A two-dimensional field of vectors is traversed.

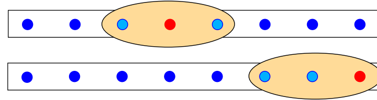


Fig. 18. Meta-cell iteration. In this iteration the two elements closest to a given element are traversed.

GTL traversal with accumulation

```
typedef container<2, random_access> topo_t;
typedef long                         data_t;
container_type<topo_t, data_t>       container;
container_type::data_accessor        da;
gtl::traverse<cell>
[
   std::cout << gtl::sum<vertex>(0.0)[da] << std::endl
](container) ;
```

To present the interoperability with STL containers, a standard vector is traversed and all of its elements are summed. This is still possible using standard algorithms as shown for comparison.

C++ STL and GTL traversal and accumulation

```
vector<int> vec;
std::accumulate(vec.begin(), vec.end(), ZERO, _1 + _2); //STL
gtl::sum<cell>[_1](vec);  // GTL
```

The following operations are not efficiently possible with standard algorithms but can be specified using functional environments like FC++ [27] and Phoenix [10]. An iteration over a two-dimensional field is performed as depicted in Fig. 17.

Two-dimensional traversal

```
vector<vector<int> > vec2;

std::accumulate(vec2.begin(), vec2.end(), 0,
 _1 +
 phoenix::accumulate(_2, 0.0, lambda()[_1 + _2])) // Phoenix

gtl::sum<cell> [ gtl::sum<cell>[_1] ](vec2);  // GTL
```

The following is an example which exceeds the power of available functional frameworks. A traversal operation over a container is used which accumulates the product of the values stored in the N elements, which are topologically closest to the initial element. The GTL groups these vertices into a meta-cell (Fig. 18)[16] as already presented in Section 3.3. The vertices within a `meta_cell<N>`, which is provided by the GTL, can then be traversed like vertices attached to a simplex or cube. This mechanism is used to calculate, e.g., higher order finite difference schemes.

GTL iteration with sum and product operation

```
vector<int> vec;

gtl::sum<cell>
[
   gtl::product<gtl::meta_cell<3> >[_1]
](vec);
```

Even though FC++ and Phoenix2 provide container access, such operations cannot be carried out in an easy manner without rewriting some components for this special case. Using the iteration data structures of the GTL, one can use arbitrary subsets for accumulation or iteration.

[16] Here, the meta-cell simply represents a subset of elements.

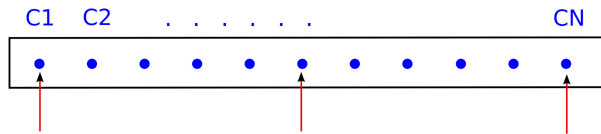


Fig. 19. Topological traversal of vertices.

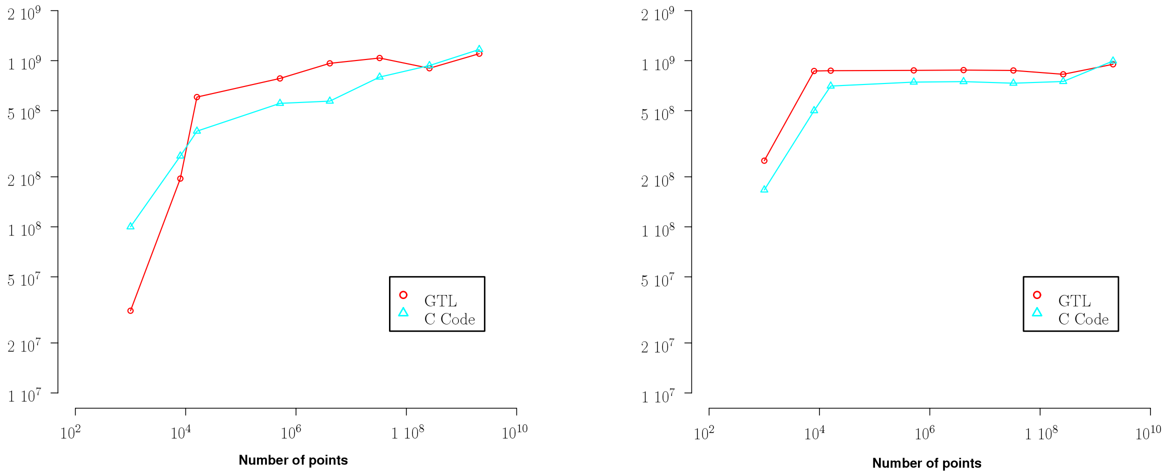


Fig. 20. 0-cell traversal on the P4 (left) and AMD64 (right); the units are iterations per second.

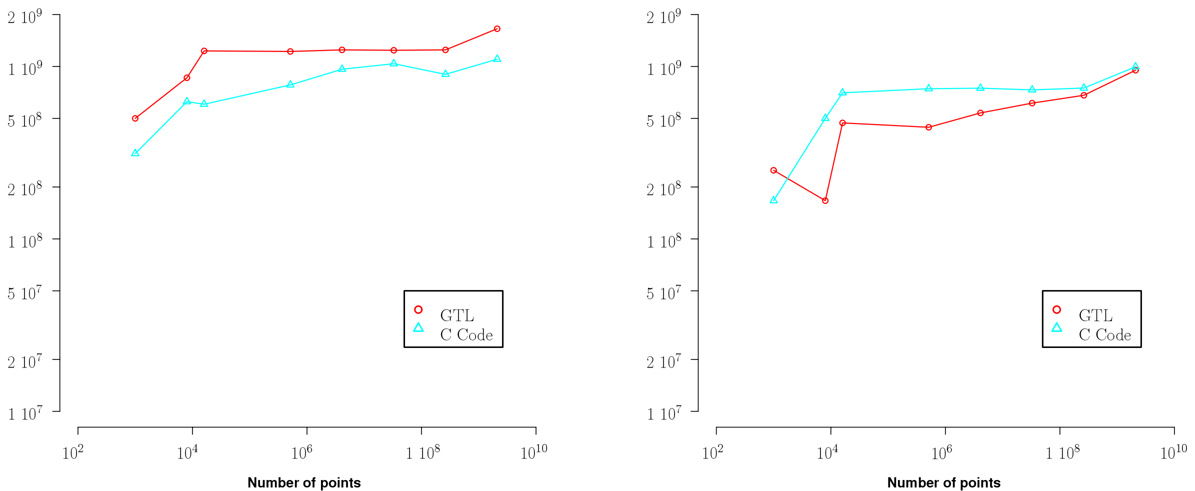


Fig. 21. 0-cell traversal on the G5 (left) and IBM (right); the units are iterations per second.

5.2. Performance analysis

The most important parts of how to achieve high performance in C++ can be summarized on the one hand by static parametric polymorphism [24,35,28] for global optimization with inlined function blocks, and on the other hand by lightweight object optimization [33] which enables allocation of simple objects to registers.

This has already been demonstrated in the field of numerical analysis, yielding figures comparable to those of Fortran [39,38,40], the previously undisputed choice for this kind of calculation. The C and Fortran languages do not offer techniques for a variable degree of optimization, such as controlled loop unrolling [34,24]. Such tasks are left to the compiler. Therefore, libraries have to use special techniques as employed by ATLAS [41] or have to rely on manually tuned code elements which have been assembled by domain experts or the vendors of the microprocessor architecture used.

To analyze the performance of the implemented GTL approach [19] on various computer systems, the maximum achievable performance is summarized in the following table, which shows the CPU types, the amount of RAM, and the compilers used. A balance factor (BF) is evaluated by dividing the maximum MFLOPS measured by ATLAS [41] by the maximum memory band width (MB) in megabytes per second measured with STREAM [1]. This factor states the relative cost of arithmetic calculations compared to memory access.

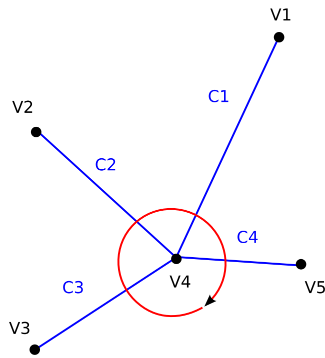


Fig. 22. Topological traversal of cells for a 1-cell complex.

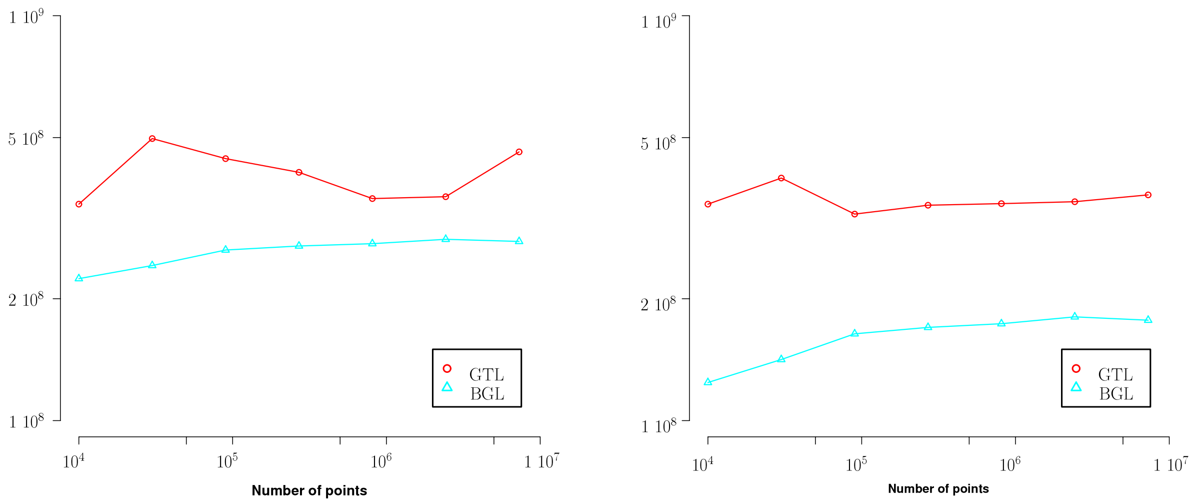


Fig. 23. Incidence traversal for the BGL and the GTL approach on the P4 (left) and AMD64 (right); the units are iterations per second.

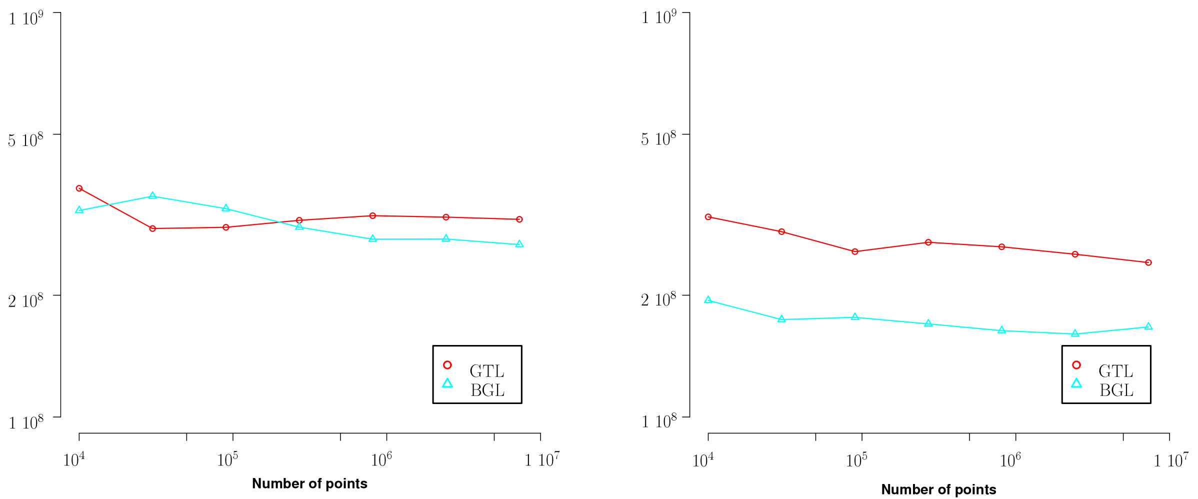


Fig. 24. Incidence traversal for the BGL and the GTL approach on the G5 (left) and IBM (right); the units are iterations per second.

Type	Speed	RAM	GCC	MFLOPS	MB/s	Balance factor
P4	1 × 2.8 GHz	2 GB	4.0.3	2310.9	3045.8	0.7
AMD64	1 × 2.2 GHz	2 GB	3.4.4	3543.0	2667.7	1.2
IBMP655	8 × 1.5 GHz	64 GB	4.0.1	16361.7	4830.4	3.4
G5	4 × 2.5 GHz	8 GB	4.0.0	24434.0	3213.3	7.6

The first analysis concerns to the topological traversal for different dimensional cell complex types. First a typical representative of a 0-cell complex, e.g., an array, is traversed. The C++ STL containers, such as a vector or list, are representatives and are schematically depicted in Fig. 19, where the points represent the cells on which data values are stored.

To investigate the abstraction penalty of the generic code a simple C implementation is used without any generic overhead which might be attributable to the GTL. The next code snippet presents the C source code for a three-dimensional 0-cell complex (cube) and is followed by the generic approach:

C approach for 0-cell traversal

```
for (i3 = 0; i3 < sized3; i3++)
  for (i2 = 0; i2 < sized2; i2++)
    for (i1 = 0; i1 < sized1; i1++)
    {     //operations, use i1, i2, i3     }
```

Generic approach for 0-cell traversal

```
vit1 = container.vertex_begin();
vit2 = container.vertex_end();
for (;vit1 != vit2; ++vit1)
{  //operations, use *vit1 }
```

Figs. 20–21 present results obtained on four different computer systems. As can be seen, the performances are comparable on different systems without incurring an abstraction penalty from the highly generic code when compared to a simple C implementation.

To provide a more complex example, the traversal mechanisms of the BGL data structures are compared to the approach given here. Fig. 22 presents a 1-cell complex type with an edge traversal mechanism. The source code snippets from Section 4.1.2 are used as the corresponding implementations.

Figs. 23–24 present the run-time results for edge on vertex traversal. The overall run-time behavior of the GTL is comparable, at times even superior, to that of the BGL.

5.3. TCAD Application

In this section, an example of a TCAD device simulation application is presented. The most basic model, the drift-diffusion model, consists of a set of coupled partial differential equations which need to be assembled and solved. In this case the assembly time is usually small compared to the time spent on the solution of the equation system. More sophisticated and complex models, such as the energy transport model presented or higher transport models, however, spend an increasing amount of time on equation assembly.

Discretized using finite volume schemes, it can be assembled in parallel by partitioning the traversal space due to the linewise entries into the matrix. Eq. (2) shows the energy flux equation for electrons, which is solved self-consistently with Poisson's equation and the current relations [31]. The following code snippet demonstrates the actual C++ code for the electron temperature `T_n` determined by Eq. (2) [15]:

$$\mathrm{div}\left(\alpha_n\,\mathrm{grad}(nT_\mathrm{n}^2) + \mathrm{grad}\,\varphi\; n\; T_\mathrm{n}\right) = -\mathrm{grad}\,\varphi\cdot\mathbf{J}_\mathrm{n} - \beta_n\, n(T_\mathrm{n} - T_\mathrm{Lattice}). \tag{2}$$

Each `sum` can be automatically parallelized by using the parallel STL, where the full matrix line is assembled in parallel by a vertex partitioning mechanism executed by spawned threads[17]:

```
(sum<edge>()
[ let(_x = Bern(edge_log<vertex>(T_n)) / T_n *
      sum<vertex>() [ phi ] +
      sum<vertex>() [ T_n ] )
   [
      alpha_n * T_n / Bern(edge_log<vertex>(T_n))  *
      sum<vertex>() [  Bern( _x )  *  n  *  T_n  ] *
      area / dist
   ]
]
+ sum<edge>()
[ sum<vertex>() [ phi ] / dist * J_n ]  * vol
  + beta_n * n * (T_n - T_lattice ) * vol
) (vx);
```

17 The `_x` represents Phoenix 2 local scope variables.

Table 4
Comparisons of the simulation times for drift-diffusion and hydrodynamic simulation of a pn-diode with GCC 4.2 on an AMD X4 Phenom 9600 (Quad-Core).

Example	Sequential (s)	2 threads (s)	4 threads (s)
DD	10.0	9.3	8.1
HD	15.2	10.5	8.8

Table 5
Comparisons of the simulation times for drift-diffusion and hydrodynamic simulation of a pn-diode with GCC 4.2 on an Opteron Cluster with 4xDual-Core 8222 SE.

Example	Sequential (s)	2 threads (s)	4 threads (s)	8 threads (s)
DD	11.6	10.1	9.1	8.3
HD	17.8	14.7	10.5	8.9

Several existing multi-thread libraries can be used orthogonally due to the separate partitioning of the vertex space (e.g., the Boost thread library [7]). This method can then be used for all finite volume codes.

To present the results obtained by using different machines and not altering the actual code parts, a benchmark[18] is given for a simple drift-diffusion (DD) and hydrodynamic (HD) simulation for a two-dimensional pn-diode with 4000 elements (Table 4). The bias voltage is stepped from −0.03 V to 1 V. For the actual benchmarks, a sequential implementation of a BiCGStab algorithm was used.

Table 5 reviews the benchmark results from four AMD 4xDual-Core Opteron Cluster 8222 SE 3 GHz with 2×32 GByte and 2×16 GByte RAM.

6. Conclusion

By using mathematical concepts derived from topology, data structure properties can be separated into cell and complex topologies. The separation of topological structure and data access has been studied by using concepts from the theory of fiber bundles. Iterator, cursor, and property map concepts are analyzed and a common data accessor with full backward compatibility has been presented.

A common specification for various data structure properties of arbitrary dimension and topological cell types then enables the efficient development of generic algorithms and applications in the field of scientific computing. The separation of base and fiber space eases the handling of more complex data sets by providing a common access mechanism. Additionally, traversal capabilities are available which exceed the capabilities of other libraries. This has been presented in the final section by presenting complex traversal capabilities as well as a performance comparisons of the GTL to other libraries.

Functional programming greatly benefits from common characterization of data structures and the common traversal properties due to inherent parallelization capabilities. A final application benchmark from the field of TCAD has been presented and hence it was shown that modern computer architectures can benefit greatly from the inherent parallel capabilities of the approach presented.

Acknowledgments

We want to thank Werner Benger, Franz Stimpfl, Michael Spevak, our colleagues, and especially the (unknown) reviewers for their valuable input and discussions, which significantly improved the readability and context of this paper. We also want to thank Prof. Siegfried Selberherr for the resources at the Institute of Microelectronics. This work was supported by the Austrian Science Fund FWF, project P19532-N13.

References

[1] Stream - Sustainable memory bandwidth in high performance computers. URL: http://www.cs.virginia.edu/stream/.
[2] D. Abrahams, J. Siek, T. Witt, New iterator concepts, Tech. Rep. N1477 03-0060, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003).
[3] O. Bagge, CodeBoost: A framework for transforming C++ programs, Master's thesis, University of Bergen, P.O. Box 7800, N-5020 Bergen, Norway, March 2003.
[4] W. Benger, Visualization of general relativistic tensor fields via a fiber bundle data model, Dissertation, Freie Universität Berlin (2004).
[5] G. Berti, Generic software components for scientific computing, Dissertation, Technische Universität Cottbus (2000).
[6] G. Berti, GrAL - The grid algorithms library, in: Proc. Computational Science ICCS, vol. 2331, Springer, London, UK, 2002, URL: http://www.math.tu-cottbus.de/ berti/gral]http://www.math.tu-cottbus.de/ berti/gral.
[7] Boost, Boost C++ Libraries. URL: http://www.boost.org.

[18] GCC 4.2, -O3 -march=k8 -mtune=k8.

[8] Boost, Boost Fusion 2.0. URL: http://www.boost.org/libs/fusion.
[9] Boost, Boost Lambda Library. URL: http://www.boost.org/libs/lambda.
[10] Boost, Boost Phoenix 2.0. URL: http://www.boost.org/libs/spirit/phoenix.
[11] D.M. Butler, S. Bryson, Vector bundle classes from powerful tool for scientific visualization, Computers in Physics 6 (1992) 576–584.
[12] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, Cambridge, 1990.
[13] H. Edelsbrunner, R. Seidel, Voronoi diagrams and arrangements, ACM 85 (6) (1985) 251–262.
[14] Computational Geometry Algorithms Library, URL: http://www.cgal.org.
[15] T. Grasser, T. Tang, H. Kosina, S. Selberherr, A review of hydrodynamic and energy-transport models for semiconductor device simulation, Proceedings of IEEE 91 (2) (2003) 251–274.
[16] D. Gregor, J. Willcock, A. Lumsdaine, Concepts for the C++0x standard library: Iterators, Tech. Rep. N2039=06-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (June 2006).
[17] M. Haveraaen, H. Friis, T. Johansen, Formal software engineering for computational modeling, Nordic Journal of Computing 3 (6) (1999) 241–270.
[18] R. Heinzl, Concepts for scientific computing, Dissertation, Technische Universität Wien, Austria (2007).
[19] R. Heinzl, P. Schwaha, M. Spevak, T. Grasser, Performance aspects of a DSEL for scientific computing with C++, in: Proc. of the POOSC Conf., Nantes, France, 2006.
[20] IBM Corporation, Yorktown Heights, NY, USA, IBM Visualization Data Explorer, 3rd ed. (Feb. 1993).
[21] K. Jänich, Topologie, Springer, Heidelberg, 2001.
[22] V. Kovalevsky, Finite topology as applied to image analysis, Computer Vision and Image Processing 161 (46) (1989) 141.
[23] J. Lachaud, Writing generic digital topology and geometry algorithms.
[24] L. Lee, A. Lumsdaine, Generic programming for high performance scientific applications, in: JGI '02: Proc. of the 2002 joint ACM-ISCOPE Conf. on Java Grande, ACM Press, New York, NY, USA, 2002.
[25] P. Lienhardt, Subdivisions of n-dimensional spaces and n-dimensional generalized maps, in: SCG '89: Proc. of the Fifth Annual Symposium on Computational Geometry, ACM, New York, NY, USA, 1989.
[26] M.H. Austern, Generic Programming and the STL: Using and Extending the C++ Standard Template Library, Addison-Wesley, Boston, MA, USA, 1998.
[27] B. McNamara, Y. Smaragdakis, Functional programming in C++ using the FC++ library, SIGPLAN 36 (4) (2001) 25–30.
[28] C.E. Oancea, S.M. Watt, Parametric polymorphism for software component architectures, in: Proc. of the Object-Oriented Programming Systems, Languages, and Applications Conf., ACM Press, New York, NY, USA, 2005.
[29] S. Pion, A. Fabri, A generic lazy evaluation scheme for exact geometric computations, in: Proc. of the Object-Oriented Programming Systems, Languages, and Applications Conf., Portland, OR, USA, 2006.
[30] G.D. Reis, J. Jarvi, What is generic programming? in: Proc. of the Object-Oriented Programming Systems, Languages, and Applications Conf., San Diego, CA, USA, 2005.
[31] P. Schwaha, M. Schwaha, R. Heinzl, E. Ungersboeck, S. Selberherr, Simulation methodologies for scientific computing, in: Proc. of the 2nd ICSOFT 2007, Barcelona, Spain, 2007.
[32] J. Siek, L.-Q. Lee, A. Lumsdaine, The Boost Graph Library: User Guide and Reference Manual, Addison-Wesley, 2002.
[33] J. Siek, A. Lumsdaine, Mayfly: A pattern for lightweight generic interfaces (July 1999). URL: http://citeseer.ist.psu.edu/siek99mayfly.html.
[34] J. Siek, A. Lumsdaine, The matrix template library: generic components for high-performance scientific computing, Computing in Science and Engineering 1 (6) (1999) 70–78.
[35] J.G. Siek, A. Lumsdaine, Concept checking: Binding parametric polymorphism in C++, in: Proc. of the First Workshop on C++ Template Programming, Erfurt, Germany, 2000.
[36] J. Singler, B. Kosnik, The libstdc++ parallel mode: Software engineering considerations, in: Proc. of IWMSE, Leipzig, Germany, 2008.
[37] J. Singler, P. Sanders, F. Putze, The Multi-Core Standard Template Library, in: Lecture Notes in Computer Science, vol. 4641/2007, Springer, Berlin, 2007, pp. 682–694.
[38] T.L. Veldhuizen, Expression templates, C++ Report 7 (5) (1995) 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.
[39] T.L. Veldhuizen, Using C++ template metaprograms, C++ Report 7 (4) (1995) 36–43. Reprinted in C++ Gems, ed. Stanley Lippman.
[40] T.L. Veldhuizen, D. Gannon, Active libraries: Rethinking the roles of compilers and libraries, in: Proc. of the SIAM Workshop on Obj.-Oriented Methods for Inter-Operable Sci. and Eng. Comp., OO'98, SIAM, 1998.
[41] R. Whaley, J. Dongarra, Automatically tuned linear algebra software, in: Proc. of the 1998 ACM/IEEE Conf. on Supercomputing, CDROM, IEEE Computer Society, 1998, cD-ROM Proc..
[42] M. Zalewski, S. Schupp, Changing iterators with confidence. A case study of change impact analysis applied to conceptual specifications, in: Proc. of the Object-Oriented Programming Systems, Languages, and Applications Conf., San Diego, CA, USA, 2005.