Concepts for High-Perfomance Scientific Computing

René Heinzl, Philipp Schwaha, Franz Stimpfl, and Siegfried Selberherr

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

Abstract. During the last decades various high-performance libraries were developed written in fairly low level languages, like FORTRAN, carefully specializing codes to achieve the best performance. However, the objective to achieve reusable components has regularly eluded the software community ever since. The fundamental goal of our approach is to create a high-performance mathematical framework with reusable domain-specific abstractions which are close to the mathematical notations to describe many problems in scientific computing. Interoperability driven by strong theoretical derivations of mathematical concepts is another important goal of our approach.

1 Introduction

This work reviews common concepts for scientific computing and introduces new ones for a timely approach to library centric application design. Based on concepts for generic programming, e.g. in C++, we have investigated and developed data structures for scientific computing. The Boost Graph Library [11] was one of the first generic libraries, which introduced concept based programming for a more complex data structure, a graph. The actual implementation of the Boost Graph Library (BGL) is for our work of secondary importance, however, we value the consistent interfaces for graph operations. We have extended this type of concept based programming and library development to the field of scientific computing. To give a brief introduction we use an example resulting from a self-adjoint partial differential equation (PDE), namely the Poisson equation:

$$\operatorname{div}(\varepsilon \operatorname{grad}(\Psi)) = \rho$$

Several discretization schemes are available to project this PDE into a finite space. We use the method of finite volumes. The resulting equations are given next, where A_{ij} and d_{ij} represents geometrical properties of the discretized space, ρ the space charge, Ψ the potential, and ε the permittivity of the medium.

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

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

J. Filipe et al. (Eds.): ICSOFT/ENASE 2007, CCIS 22, pp. 89–100, 2008.

An example of our domain specific notation is given in the following code snippet and explained in Section 4:

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

Generic Poisson Equation

As can be seen, the actual notation does not depend on any dimension or topological type of the cell complex (mesh) and is therefore dimensionally and topologically indepent. Only the relevant concepts, in this case, the existence of edges incident to a vertex and several quantity accessors, have to be met. In other words, we have extended the concept programming of the standard template library (STL) and the generic programming of C++ to higher dimensional data structures and automatic quantity access mechanisms.

Compared to the reviewed related work given in Section 2, our approach implements a domain specific embedded language. The related topological concepts are given in Section 3, whereas Section 4 briefly overviews the used programming paradigms. In Section 5 several application examples are presented. The first example introduces a problem of a biological system with a simple PDE. The second example shows a nonlinear system of coupled PDEs, which makes use of the linearization framework introduced in Section 4.1, where derivatives are calculated automatically.

For a successful treatment of a domain specific embedded notation several programming paradigms are used. By object-oriented programming the appropriate iterators are generated, hidden in this example in the expression `vertex_edge` and `edge_vertex`. Functional programming supplies the higher order function expression between the `[` and `]` and the unnamed function object `_1`. And finally the generic programming paradigm (in C++ realized with parametric polymorphism or templates) connects the various data types of the iterators and quantity accessors.

A significant target of this work is the separation of data access and traversal by means of the mathematical concept of fiber bundles [5]. The related formal introduction enables a clean separation of the internal combinatorial properties of data structures and the mechanisms of data access. A high degree of interoperability can be achieved with this formal interface. Due to space constraints the performance analysis is omitted and we refer to a recent work [6] where the overall high performance is presented in more detail.

2 Related Work

In the following several related works are presented. All of these software libraries are a great achievement in the various fields of scientific computing. The **FEniCS** project [9], which is a unified framework for several tasks in the area of scientific computing, is a great step towards generic modules for scientific computing.

The **Boost Graph Library** is a generic interface which enables access to a graph's structure, but hides the details of the actual implementation. All libraries which implement this type of interface are interoperable with the BGL generic algorithms. This approach was one of the first in the field of non-trivial data structures with respect to interoperability. The property map concept [11] was introduced and heavily used. The **Grid Algorithms Library, GrAL** [4] was one of the first contributions to the unification of data structures of arbitrary dimension for the field of scientific computing. A common interface for grids with a dimensionally and topologically independent way of access and traversal was designed.

Our Developed Approach, the Generic Scientific Simulation Environment (GSSE [8]) deals with various modules for different discretization schemes such as finite elements and finite differences. In comparison, our approach focuses more on providing building blocks for scientific computing, especially an embedded domain language to express mathematical dependencies directly, not only for finite elements. To achieve interoperability between different library approaches we use concepts of fiber bundle theory to separate the base space and fiber space properties. With this separation we can use several other libraries (see Section 3) for different tasks. The theory of fiber bundles separates the data structural components from data access (fibers).

Based on this interface specification we can use several libraries, such as STL, BGL, GrAL, and accomplish high interoperability and code reuse.

3 Concepts

Our approach extends the concept based programming of the STL to arbitrary dimensions similar to GrAL. The main difference to GrAL is the introduction of the concept of fiber bundles, which separates the base mechanism of application design into base and fiber space properties. The base space is modeled by a CW-complex and algebraic topology, whereas the fiber space is modeled by a generic data accessor mechanism, similar to the cursor and property map concept [3].

Table 1. Comparison of the cursor/property map and the fiber bundle concept

	cursor and property map	fiber bundles
isomorphic base space	no	yes
traversal possibilities	STL iteration	cell complex
traversal base space	yes	yes
traversal fiber space	no	yes
data access	single data	topological space
fiber space slices	no	yes

3.1 Theory of Fiber Bundles

We introduce concepts of fiber bundles as a description for data structures of various dimensions and topological properties.

- Base space: topology and partially ordered sets
- Fiber space: matrix and tensor handling

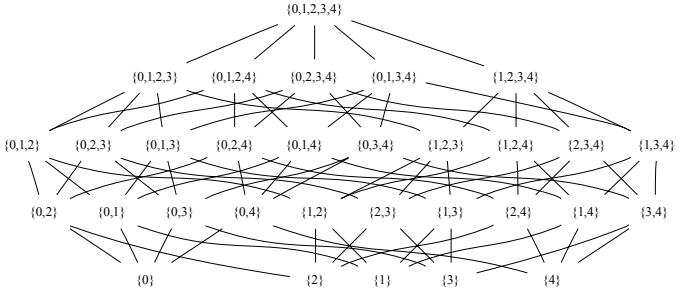

Fig. 1. Cell topology of a simplex cell in four dimensions

Based on these examples, we introduce a common theory for the separation of the topological structure and the attached data. The original contribution of this theory was given in Butler's vector bundle model [5], which we compactly review here:

Definition 1 (Fiber Bundle). *Let E, B be topological spaces and $f : E \rightarrow B$ a continuous map. Then (E, B, f) is called a **fiber bundle**, if there exists a space F such that the union of the inverse images of the projection map f (the fibers) of a neighborhood $U_b \subset B$ of each point $b \in B$ are homeomorphic to $U_b \times F$, whereby this homeomorphism has to be such that the projection pr_1 of $U_b \times F$ yields U_b again.*

E is called the *total space*, B is called the *base space*, and F is called the *fiber space*. This definition requires that a total space E can locally be written as the product of a base space B and a fiber space F. The decomposition of the *base space* is modeled by an identification of data structures by a CW-complex [7]. As an example Figure 2 depicts an array data structure based on the concept of a fiber bundle. We have a simple fiber space attached to each cell (marked with a dot in the figure), which carries the data of our array.

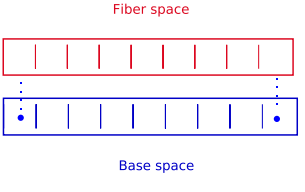

Fig. 2. A fiber bundle with a fiber space over a 0-cell complex. A simple array is an example of this type of fiber space.

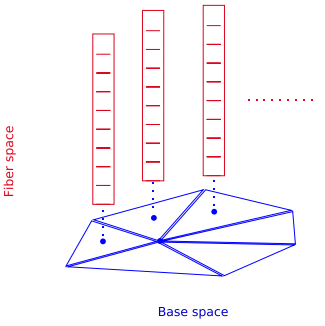

Fig. 3. A fiber space over a 2-simplex cell complex base space. An example of this type of fiber space is a triangle mesh with an array over each triangle.

Figure 3 depicts a fiber bundle with a 2-cell complex as base space. For the base space of lower dimensional data structures, such as an array or single linked list, the only relevant information is the number of elements determined by the index space. 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 used [7].The advantages of this approach are similar to those of the cursor and property map [3], but they differ in several details. The similarity is that both properties can be implemented independently. However, the fiber bundle approach equips the fiber space with more structure, e.g., storing more than one value corresponding to the traversal position as well as preservation of neighborhoods. 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 vertices, faces, or cells. 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. An overview of the common topological features and differences for various data structures are presented in Table 1.

Table 2. Classification scheme based on the dimension of cells, the cell topology, and the complex topology

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
grid	2,3,4,..	cuboid	global
mesh	2,3,4,..	simplex	local

3.2 Topological Interface

We briefly introduce parts of the interface specification for data structures and their corresponding iteration mechanism based on algebraic topology. A full reference can

be found in [7]. With the concept of partial ordered sets and a Hasse diagram we can order and depict the structure of a cell.

As an example the topological structure of a three-dimensional simplex is given in Figure 4.

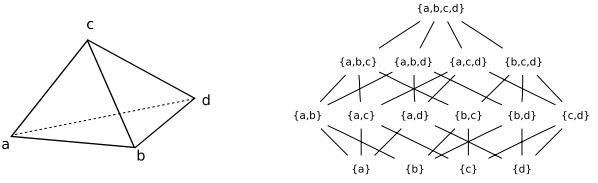

Fig. 4. Cell topology of a 3-simplex cell

Inter-dimensional objects such as edges and facets and their relations within the cell can thereby be identified. The complete traversal of all different objects is determined by this structure. We can derive the vertex on edge, vertex on cell, as well as edge on cell traversal up to the dimension of the cell in this way. Based on our topological specification arbitrary dimensional cells can be easily used and traversed in the same way as all other cell types, e.g., a 4-dimensional simplex shown in Figure 1.

Next to the cell topology a separate complex topology is derived to enable an efficient implementation of these two concepts. A significant amount of code can be reduced with this separation. Figure 5 depicts the complex topology of a 2-simplex cell complex where the bottom sets on the right-hand sides are now the cells. The rectangle in the figure marks the relevant cell number.

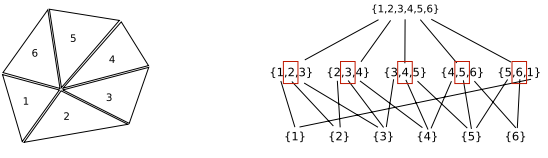

Fig. 5. Complex topology of a simplex cell complex

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

A formal concise definition of data structures can therewith be derived and is presented in Figure 2. The complex topology uses the number of elements of the corresponding subsets.

```
complex_t<cells_t,global  > cx; //{1}
complex_t<cells_t,local<2>> cx; //{2}
complex_t<cells_t,local<3>> cx; //{3}
complex_t<cells_t,local<4>> cx; //{4}
```

STL Data Structure Definitions

Here {1} describes an array, {2} a stream or a singly linked list, {3} a doubly linked list or a binary tree, and finally {4} an arbitrary tree. To demonstrate the equivalence of the STL data structures and our approach we present a simple code snippet (the typedefs are omitted due to space constraints):

```
cell_t<0, simplex>                cells_t;
complex_t<cells_t, global>        complex_t;
container_t<complex_t,  long>     container;
// is equivalent to
std::vector<data_t>               container;
```

Equivalence of Data Structures

3.3 Data Access

In the following code snippet a simple example of the generic use of a data accessor similar to the property map concept is given, where a scalar value is assigned to each vertex. The data accessor implementation also takes care of accessing data sets with different data locality, e.g., data on vertices, edges, facets, or cells. The data accessor is extracted from the container to enable a functional access mechanism with a key value which can be modeled by arbitrary comparable data types.

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

Data Accessor

Several programming paradigms are used in this example which are presented in detail in the next section, especially the functional programming, given in this example with the `da = 1.0`.

4 Programming Paradigms

Various areas of scientific computing encourage different programming techniques, even demands for several programming paradigms can be observed:

- Object-oriented programming: the close interaction of content and functions is one of the most important advantages of the object-oriented programming
- Functional programming: offers a clear notation, is side-effect free and inherently parallel

– Generic programming: can be seen as the glue between object-oriented and functional programming

Our implementation language of choice is C++ due to the fact, that it is one of the few programming languages where high performance can be achieved with various paradigms.

To give an example of this multi-paradigm approach, a simple C++ source snippet is given next.

```
std::for_each(v.begin(),v.end(),
    if_(arg1 > 5)
    [
       std::cout << arg1 << std::cout
    ]
);
```

Multiple Paradigms

The object-oriented programming paradigm is used to create the iterator capabilities of the container structures of the STL. This paradigm is not used anywhere else in our approach. Functional programming is used to create function objects which are passed into the generic `for_each` algorithm. In this example the notation of the Boost Phoenix 2 [2] library is used to create a functional object context, marked by the `[` and `]`. The generic paradigm uses the template mechanism of C++ to bind these two paradigms together efficiently. A more complex example is given in the following expression. Here a cell complex of arbitrary dimension is used and the vertex to vertex iteration is expressed.

```
gsse::for_each_vertex(domain
    result=gsse::add<vertex_vertex> [  quan  ]
);
```

Complex Functor

The same paradigms as in the example before can be seen, but in this case, a complex topological traversal is used instead of simple container traversal. The topological properties of the GSSE are demonstrated twofold: on the one side, the topological concept programming allows the implementation of a dimensionally independend algorithm. On the other side, different data structures of library approaches can be used, which fullfill the basic requirements of the required topological concept.

This GSSE algorithm sums up the potential values of all vertices adjacent to a vertex. The data accessor `quan` handles the storage mechanism for the value attached to a vertex. Here the interaction of programming paradigms related to the base space and fiber space can be seen clearly. The base space traversal is built with the generic programming paradigm, whereas the fiber space operation is implemented by means of functional programming. A lot of difficulties with conventional programming can be circumvented by this approach. Functional programming enables great extensibility due to the modular nature of function objects. Generic programming and the corresponding

template mechanisms of C++ offer an overall high performance. In addition arbitrary data structures of arbitrary dimensions can be used. The only requirement is that the data structure models the required concept, in this case a vertex to vertex information.

4.1 Automatic Linearization

A calculation framework is used where derivatives are implicitly available and do not have to be specified explicitly. This enables the specification of nonlinear differential equations in a convenient way. The elements of the framework are truncated Taylor series of the following form $f_0 + \sum_i c_i \cdot \Delta x_i$. To use a quantity x_i within a formula we have to specify its value f_0 and the linear dependence $c_i = 1$ on the vector $\mathbf{x}$ of quantities. This non-trivial and highly complex scenario yields itself exceptionally well to the application of the functional programming paradigm. In general, all discretization schemes which use line-wise assembly based on finite differences as well as finite volumes can be handled with the described formalism. Basic operations on Taylor series can handle truncated polynomial expansions. In the following we specify our nonlinear functionals using linearized functions in upper case letters. All necessary numerical operations on these data structures can be performed in a straight forward manner, e.g. multiplication:

$$F = f_0 + \sum_i c_i \cdot \Delta x_i \,, \qquad G = g_0 + \sum_i d_i \cdot \Delta x_i \tag{3}$$

$$F \otimes G = (f_0 \cdot g_0) + \sum_i (g_0 \cdot d_i + f_0 \cdot c_i) \cdot \Delta x_i \tag{4}$$

Having implemented these schemes we are able to derive all required functions on these mathematical structures. This means that we have a consistent framework for formulas in the following sense: if A is the linearized version of function $\mathcal{A}$ at x_0, we obtain $\partial A/\partial_{x_i} = \partial \mathcal{A}/\partial_{x_i}\,|_{x_0}$ around the point of linearization. Figure 6 shows the multiplication of two truncated expansions $F = 3 + \Delta x_1$, and $G = 3 + \Delta x_3$. As a result we obtain $F \otimes G = 12 + 4\Delta x_1 + 3\Delta x_3$.

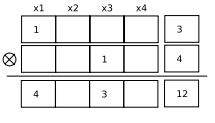

Fig. 6. The multiplication of two Taylor series

By implementing only the linear (or higher order polynomial) functional dependence of equations on variables around $\mathbf{x}$ we reduce the external specification effort to a minimum. Thus, it is possible to ease the specification with the functional programming approach, while also providing the functional dependence of formulas.

5 Application Design

In the following we briefly review a few applications based on the introduced concepts with their respective paradigms.

5.1 Biological System

Electric phenomena are common in biological organisms such as the discharges within the nervous system, but usually remain within a small scale. In some organisms, however, the electric phenomena take a more prominent role. Some fish species, such as *Gnathonemus petersii* from the family of Mormyridae [12], use them for detection of their prey. The up to 30 cm long fish actively generates electric pulses with an organ located near its tail fin (also marked in Figure 7). More information can be found in [10].

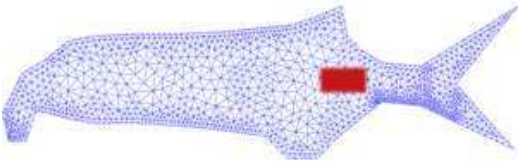

Fig. 7. Discretized domain of a fish with a red marked electrically active organ

For this case we derive the equation system based on a quasi-electro-statical system directly from the corresponding Maxwell equations. The charge separation of the electrically active organ which is actively taking place within parts of the simulation domain also has to be taken into account. We use the conservation law of charge and the divergence theorem (Gauss's law) and finally get:

$$\partial_t \left[\mathrm{div}(\varepsilon\, \mathrm{grad}(\Psi))\right] + \mathrm{div}(\gamma\, \mathrm{grad}(\Psi)) = P \tag{5}$$

Equation 5 is discretized using the finite volume discretization scheme. The high semantic level of the specification is illustrated by the following snippet of code:

```
linearized_equ=sum<vertex_edge>(_v)
  [
    orient(_v,_1) * sum<edge_vertex>(_e)
    [
      equ_pot * orient(_e,_1)
    ]
    * (area  / dist ) * (gamma * deltat + eps)
  ] + vol * ( (P * deltat) + rho)
```

This source snippets presents most of the application code which has to be developed. In addition, only a simple preprocessing step which creates the necessary quantity accessors is required.

The simulation domain is divided into several parts including the fish itself, its skin, that serves as insulation, the water the fish lives in, and an object, that represents either an inanimate object or prey. The parameters of each part can be adjusted separately. A result of the simulation is depicted in the following figure:

5.2 Drift-Diffusion Equation

Semiconductor devices have become an ubiquitous commodity and people expect a constant increase of device performance at higher integration densities and falling prices.

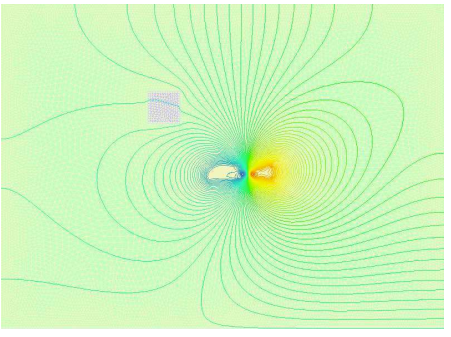

Fig. 8. Result of a simulation with a complete domain, the fish, and a ideally conductor as response object

To demonstrate the importance of a method for device simulation that is both easy and efficient we review the drift diffusion model that can be derived from Boltzmann's equation for electron transport by applying the method of moments [8]. Note that this problem is a nonlinear coupled system of partial differential equations where our linearization framework is used. This results in current relations as shown in Equation 6. These equations are solved self consistently with Poisson's equation, given in Equation 7.

$$\mathbf{J}_\mathrm{n} = \mathrm{q}\, n\, \mu_\mathrm{n} \operatorname{grad} \Psi + \mathrm{q}\, D_\mathrm{n} \operatorname{grad} n \tag{6}$$

$$\operatorname{div}(\operatorname{grad}(\varepsilon\, \Psi)) = -\rho \tag{7}$$

The following source code results from an application of the finite volume discretization scheme:

```
linearized_eq_t  equ_pot , equ_n;
equ_pot = (sum<vertex_edge>
             [
                diff<edge_vertex>[pot_quan]
             ] + ( n_quan - p_quan + nA - nD ) *
              vol * q / (eps0 * epsr)
           )(vertex);
equ_n = (sum<vertex_edge>
          [
           diff<edge_vertex>
           ( -n_quan*Bern( diff<edge_vertex>[ pot_quan / U_th] ),
             -n_quan*Bern( diff<edge_vertex>[-pot_quan / U_th] )
           )* (q * mu_h * U_th)
          ])(vertex);
```

Drift-Diffusion Equation

To briefly present a simulation result we provide Figure 9 which shows the potential in a pn diode at different stages of a nonlinear solving procedure. The leftmost figure shows the initial solution, while the rightmost depicts the final solution. The center

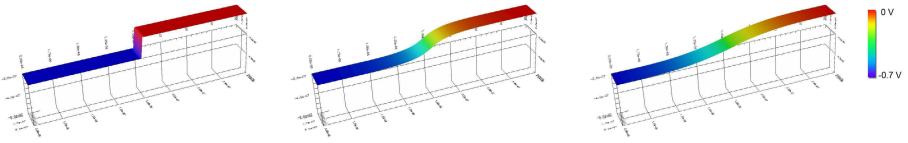

Fig. 9. Potential in a pn diode during different stages of the Newton iteration. From initial (left) to the final result(right).

image shows an intermediate result that has not yet fully converged. The visualization of the calculation is available in real time, making it possible to observe the evolution of the solution, which is realized by OpenDX [1].

References

1. DX, IBM Visualization Data Explorer. IBM Corporation, Yorktown Heights, 3rd edn., NY, USA (1993)
2. Phoenix2, Boost Phoenix 2. Boost C++ Libraries (2006), http://spirit.sourceforge.net/
3. Abrahams, D., Siek, J., Witt, T.: New Iterator Concepts. Technical Report N1477 03-0060, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003)
4. Berti, G.: Generic Software Components for Scientific Computing. Doctoral thesis, Technische Universität Cottbus (2000)
5. Butler, D.M., Bryson, S.: Vector Bundle Classes From Powerful Tool for Scientific Visualization. Computers in Physics 6, 576–584 (1992)
6. Heinzl, R., Schwaha, P., Spevak, M., Grasser, T.: Performance Aspects of a DSEL for Scientific Computing with C++. In: Proc. of the POOSC Conf., Nantes, France, pp. 37–41 (2006a)
7. Heinzl, R., Spevak, M., Schwaha, P., Selberherr, S.: A Generic Topology Library. In: Library Centric Sofware Design, OOPSLA, Portland, OR, USA, pp. 85–93 (2006c)
8. Heinzl, R.: Concepts for Scientific Computing. Dissertation, Technische Universität Wien (2007)
9. Logg, A., Dupont, T., Hoffman, J., Johnson, C., Kirby, R.C., Larson, M.G., Scott, L.R.: The FEniCS Project. Technical Report 2003-21, Chalmers Finite Element Center (2003)
10. Schwaha, P., Heinzl, R., Mach, G., Pogoreutz, C., Fister, S., Selberherr, S.: A High Performance Webapplication for an Electro-Biological Problem. In: Proc. of the 21th ECMS 2007, Prague, Czech Rep. (2007)
11. Siek, J., Lee, L.-Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, Reading (2002)
12. Westheide, W., Rieger, R.: Spezielle Zoologie. Teil 2: Wirbel- oder Schädeltiere. Elsevier, Amsterdam (2003)