From Formula to File

Coupling Complementary Generic Scientific Software Components

Werner Benger
Center for Computation & Technology
300 Frey CSC
LA70803, Baton Rouge, USA
werner@cct.lsu.edu

René Heinzl
Institute for Microelectronics
Gusshaussstrasse 27/E360
A-1040 Wien, Austria
heinzl@iue.tuwien.ac.at

Georg Ritter
Institute for Astro- and Particle Physics
Technikerstrasse 25
A-6020 Innsbruck, Austria
georg.ritter@uibk.ac.at

ABSTRACT

Genericity and performance are usually perceived as contradictory directions in software development. The frequently voiced opinion "special problems require special solutions" hinders overcoming the fears of abstraction penalties. With the complexity of state-of-the art problems such isolated approaches often tend to be dead ends. Generic approaches for scientific simulation models have become imperative: to improve the accuracy for a specific model we need to interface alternative implementations of a numerical scheme; to couple a-priori distinct simulation models in a Grid environment we need to utilize a common denominator for data exchange; last but not least for visualization we need to cover a broad range of input data stemming from diverse sources. In this article we present a novel approach to cover a wide range of application domains based on the mathematical model of fiber bundles, C++ template meta programming and multi-paradigm programming within one language. This approach allows the implementation of mathematical formulae as close to their abstract notation as possible, while still providing code performance close to or even exceeding a native FORTRAN reference implementation. The proposed approach covers finite elements, finite volumes and finite differences within the same environment, and through a mathematically founded data structure it canonically maps to a file format suitable for HPC usage.

Categories and Subject Descriptors

D.2.12 [**Software**]: Software Engineering—*Interoperability*; D.2.13 [**Software**]: Software Engineering—*Reusable Software*; E.1 [**Data**]: Data Structures; J.2 [**Computer Applications**]: Physical Sciences and Engineering

Keywords

numerical simulation, data model, file format, visualization, interoperability, generic programming

SPM2007 2007, Beijing, China

1. INTRODUCTION

Very early scientific software often consists of a monolithic single applications, dedicated to solve a fairly special problem. Applications of this kind perform exceptionally well, as they are written by domain experts and fine tuned manually. But extending their field beyond the initial set of problems requires a considerable amount of maintenance by the developing group. Extending a single application also is combined with a high investment of time to learn its internals, which often requires in depth knowledge. Frequently, programmers thus opt for recreation of their applications because this alternative is faster and less likely to fail. However, this approach does not scale as soon as multiple single applications are required to operate together. Such interoperability is inevitable in particular in the fields of

- visualization,
- multi-physics applications and
- workflows within grid environments.

The extensibility of applications hugely depends on the ease of re-usability of their components. The issue of finding an application-independent data model, constituted by the underlying data structure and the operations possible on it, was recognized and has led to various approaches but no final solution has yet emerged - a good discussion on this topic can be found in [12].

Mandatory to the success of a common data model and the re-usability of its operative components are its

i. applicability (what is the range of the model?),

ii. simplicity (what is the learning time of the model?) and

iii. performance (what is the execution time of the implemented model?).

While for single applications performance is most important, applicability beyond the certain problem is of minor relevance, and simplicity nearly ignored at all, the situation is different once interoperability aspects come in. In that case, simplicity is of importance similar to the the one of performance, since a developer, usually under pressure to provide quick results, rarely takes the burden of interfacing a more complex model beyond his current needs. The fear of decreased performance as compared to isolated solutions,

frequently termed as "abstraction penalty", is often used to argue against generic approaches in general, although the actual fear seems to be mostly against simplicity. However, utilizing modern programming paradigms applicability and performance have become orthogonal aspects while still maintaining an impressive level of simplicity, which will be demonstrated below.

1.1 Previous work

The concept of a "data model", denoting a data structure with a set of operations on them, was first considered by F. Codd when defining relational databases [19]. Since then, data models have been sought as well for scientific simulation and visualization purposes, e.g.[25, 32, 41, 42, 37]. The vector bundle data model of D.Butler [18] is an early approach introducing a generalized model for the specific, but still wide range of scientific data. It was particularly successful as it inspired the implementation of OpenDX [53] and the Sets and Fields approach of the ASCI project [20].M. Rumpf et.al. [43] explored a functional description of arbitrary meshes for visualization purposes. The Sophus C++ library [26] aims at coordinate-free formulations. This approach allows to exchange different implementations of the same mathematical concepts through components that are organized into four layers: *1. mesh layer* (implements grids for sequential and parallel HPC machines), *2. scalar field layer* (numerical discretization schemes such as finite differences and finite elements, including partial derivatives), *3. tensor layer* (coordinate systems, matrices and vectors and general differentiation operators) and the *4. application layer* (solvers for PDEs). However, this approach suffers from severe abstraction penalty and requires a code transformation tool [7].

With the current advancements of compiler technology in C++, template meta-programming [55] allows to move various operations from runtime to compile-time. G. Berti [10] utilized such generic programming techniques to implement algorithms operating on scientific data independently from their actual memory layout. Generic programming is also an essential concept of the Computational Geometry Algorithms Library [1], which allows to exchange the representation of the kernel objects. Recently, a data model based on a generalization of fiber bundles was patented as a sheaf data model [16]. The concept of fiber bundles also led to a data model called the FiberLib [8], which was implemented as an extension to the commercial visualization software Amira [52]. Its design concepts will be reviewed in this paper, as well as the coupling of its successor, FiberLib2, with the Generic Scientific Simulation Environment (GSSE) [28], a collection of generic algorithms which was recently developed independently at the TU Vienna.

1.2 Organization of the Article

This article discusses a novel approach to form a complete roadmap from an mathematical formula toward a complete, self-describing file format of output data.

In section 2 we state the problem from a theoretical point of view and discuss a systematic approach to achieve both interoperability among applications as well as re-usability of software components when building applications. It will be shown that both aspects complement each other. Section 3 proposes the paradigms that are considered to be crucial for the success of this approach. Section 4 discusses the tools that we developed to tackle these issues, while section 5 demonstrates the discussed methods via a concrete application example.

2. THEORETICAL BACKGROUND

2.1 Physical Modeling

Our motivation for solid, mathematical, and physical modeling is derived from the need in high performance applications in the field of scientific computing, especially in Technology Computer Aided Design (TCAD). Briefly, TCAD deals with the assembly of large equation systems by utilizing discretized partial differential equations from different fields of physics. All types of PDEs (elliptic, parabolic, hyperbolic) have to be considered for the various types of problems from the fields of semiconductor simulation [46]. The great diversity of physical phenomena present in semiconductor devices themselves and in the processes involved in their manufacture make the field of TCAD extremely challenging. Each of the phenomena can be described by differential equations of varying complexity.

Boltzmann's equation for electron transport in semiconductors is the basis for many calculations for device simulations:

$$\frac{\partial}{\partial t} f + \vec{v} \cdot \operatorname{grad}_r f + \vec{F} \cdot \operatorname{grad}_k f = \frac{\partial}{\partial t} f|_{\text{collisions}} \tag{1}$$

Here f is the distribution function and v the velocity of the charge carriers, while $\vec{F}$ denotes the force of an electric field on these particles.

Due to the complexity of Boltzmann's equation, several techniques have been developed which result in various, simpler models. One of these is the drift diffusion model, which can be derived from (1) by applying the method of moments [46]. Therewith PDEs of parabolic as well as elliptic type are coupled [35]. Due to the highly non-linear behavior of these equations, special discretization schemes using different shape functions are employed [45]. This results in current relations as shown in (4). These equations are solved self-consistently with Poisson's equation, also given in (4).

While for the Poisson equation a linear interpolation is used, the Scharfetter-Gummel discretization [45] leads to a non-linear interpolation scheme, which is described by:

$$J_{n,ij} = \frac{q\,\mu_n\,U_{\text{th}}}{d_{ij}} \left(n_j B(\Lambda_{ij}) - n_i B(-\Lambda_{ij}) \right) \tag{2}$$

$$\Lambda_{ij} = \frac{\Psi_j - \Psi_i}{U_{\text{th}}} \qquad B(x) = \frac{x}{\mathrm{e}^x - 1} \tag{3}$$

The J represents the current flow, whereas q is the charge density, μ_n the mobility for electrons, n the carrier density, U_{th} the thermal voltage, and Ψ the potential.

The final drift-diffusion equations are summarized by:

$$\begin{aligned} \operatorname{div}(\varepsilon \operatorname{grad}(\Psi)) &= \mathrm{q}\,(n - p - C) \\ \operatorname{div}(\mathbf{J}_{\mathrm{n}}) - \mathrm{q}\,\partial_t n &= \mathrm{q}R \\ \mathbf{J}_{\mathrm{n}} &= \mathrm{q}\,n\,\mu_{\mathrm{n}} \operatorname{grad}(\Psi) + \mathrm{q}\,D_{\mathrm{n}} \operatorname{grad}(n) \end{aligned} \tag{4}$$

Here, R represents the generation-recombination rate, and D_{n} the diffusion coefficient.

Another important part in TCAD is the modeling of waves, which arise, e.g., in mask exposure simulation or in simulating the skin effect in interconnect lines found in todays microchips. The Maxwell equations can be separated into coupled scalar equations for the vector components of the electric as well as magnetic field strength. As an example, we show two of these equations, where B, H represents the magnetic part, and E, D the electric part of the electromagnetic field:

$$\begin{aligned} -\frac{\partial B_x}{\partial t} &= \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial z} \\ \frac{\partial D_x}{\partial t} &= \frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - J_x \end{aligned} \tag{5}$$

Ideally we would like to solve a physical problem by just stating the problem in abstract mathematical notation and leave the implementation details to automatisms in the computer. However, this is not yet possible and we need to consider discretization schemes for appropriate numerical modeling.

2.2 Numerical Modeling

Different grid types and dimensions of topological elements, linear and nonlinear solvers with their associated numerical issues have to be considered during application development and demand great care to ensure high software quality while also addressing performance issues. The development of several different discretization schemes has been necessary in order to best model the underlying physics and to accommodate the mathematical peculiarities of each of these equations while transferring them to the discrete world of digital computing.

A simple example is given by a generic Poisson equation:

$$\mathcal{L}_{\text{ell\_fv}}(\Psi) \equiv \sum_{v \to e} (\Delta_{e \to v} \Psi) \frac{A}{d} \varepsilon = V \varrho \,, \tag{6}$$

where Ψ denotes the solution quantity and Δ denotes the difference. The geometrical factors A and d denote the cross section for the flux and the distance between the two edge points, respectively. The integration is performed using finite volumes [35] and leads to a multiplication with the volume V.

For the special case of a TM mode, the following updating formulation can be derived from the Maxwell equations (5) by the Yee discretization scheme [60]. We only present Yees discretization scheme as an example:

$$\begin{aligned} & E_z^{n+1}(i,j) = E_z^n(i,j) \\ + \; & \frac{\Delta r}{\Delta x}[H_y^{n+1/2}(i+\frac{1}{2},j) - H_y^{n+1/2}(i-\frac{1}{2},j) \\ - \; & \frac{\Delta r}{\Delta y}[H_x^{n+1/2}(i,j+\frac{1}{2}) - H_x^{n+1/2}(i,j-\frac{1}{2}) \end{aligned} \tag{7}$$

We present implementation examples of these types of equations in the application section.We also demonstrate the development of simple applications with generic components.

2.3 Mathematical Model

Data that are of intrinsically geometric structure are described well by the mathematical theories of topology and differential geometry. This is the category of data that scientific visualization deals with, in contrast to more abstract data structures such as trees or graphs which are used e.g. in information visualization. These mathematical theories thus form a canonical framework for this category of data (as exploited in Butler's vector bundle model [17]) which we compactly review here:

Let E, B be topological spaces and $f : E \to B$ 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 $U_b \times F$ (that maps each element of this product space to the element of the first space) yields U_b again:

$$\begin{aligned} & (E, B, f : E \to B) \text{ bundle} \quad \leftrightarrow \\ & \exists F : \forall b \in B : \exists U_b : f^{-1}(U_b) \overset{\text{hom}}{\simeq} U_b \times F \\ & \qquad\qquad\qquad\qquad \text{and } pr_1(U_b \times F) = U_b \end{aligned}$$

E is called the *total space* E, B is called the *base space* and F is called the fiber space. In words, this definition requires that a total space E (in our case a scientific data set) can be written *locally* as the product of a base space B and a fiber space F. If this can be done globally, then the fiber bundle is called *trivial*. Such cases will cover most scientific data; the Möbius strip is a famous example of a non-trivial fiber bundle with $B = \mathbb{S}^1$ and $F = \mathbb{R}^1$. Its trivial counterpart is the infinite cylinder $\mathbb{S}^1 \times \mathbb{R}^1$.

Organizing data, which represent a total space, by the concept of a fiber bundle means to separate properties of the base space (which usually will be a manifold, but more general topological spaces are possible as well) from the properties of the fiber space (e.g. the tangential space containing vector fields). Both properties can be implemented independently: for instance, the same mechanisms can be used to access a vector field given on a tetrahedral mesh, a structured grid or on a particle set because the fiber space is identical. On the other hand, the base space is identical for different scalars, vectors, tensors etc. data given on the same tetrahedral grid.

The decomposition of the usually discretized *base space* is well described in topology theory as a ("closure-finite, weak-topology") **CW-complex** by exploiting incidence and adjacency relationships among n-dimensional cells (open subsets of the base space that are homeomorph to an n-dimensional ball) , with two n-cells a, b being adjacent, iff $a \cap b \neq \emptyset$. Then, a CW-complex $\mathcal{C}$ is a hierarchical system of spaces $X^{(-1)} \subseteq X^{(0)} \subseteq X^{(1)} \subseteq \cdots \subseteq X^{(n)}$, constructed by pairwise disjoint open cells such that $X^{(n)}$ is obtained from $X^{(n-1)}$ by attaching adjacent n-cells to each $(n-1)$-cell and $X^{(-1)} = \emptyset$. The respective subspaces $X^{(n)}$ are called the n-skeletons of the base space. Supporting this hierarchical structure of subspaces, each of them possibly carrying a fiber space as well, allows a systematic handling of scientific data while sharing common characteristics.

The *fiber space*, usually continuous, is well described by concepts of differential geometry such as tensor algebra. In the simpler case of a vector bundle, operations from linear alge-

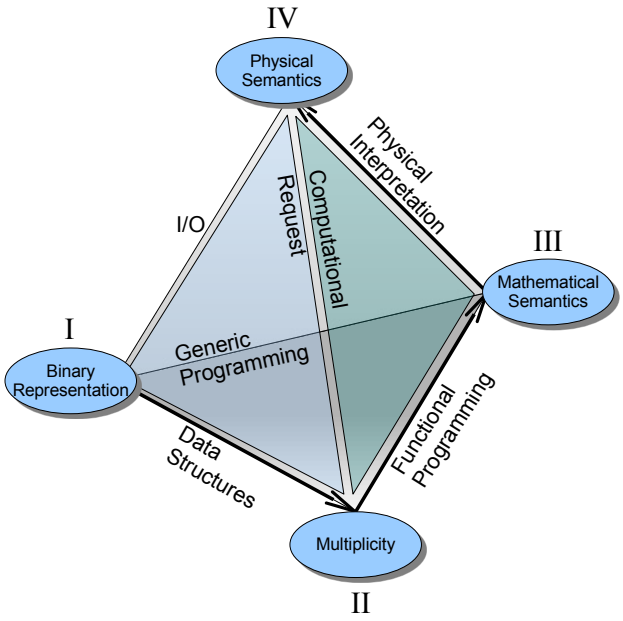

Figure 1: Tetrahedron span by the four abstraction levels of a data model and their interfaces, constituting a certain application scenario all together.

bra can be utilized and the fiber space is fully described by its dimensionality, also called the multiplicity as it defines the number of quantities per point. However, this information is insufficient to represent the more advanced entities of differential geometry, which distinguishes e.g. between tangential (positional) vectors ∂_x and co-vectors dx (normal vectors). This distinction is apparent through their inverse coordinate transformation rule. Moreover, the partial derivative of a tensor field, i.e. the derivation of a tensor field by a coordinate function, is no longer a tensor field. Instead, the partial derivative needs to be replaced by the covariant derivative to achieve a coordinate-free formulation. This involves the so-called Christoffel symbols, which have the same multiplicity as a tensor field of 3rd order, but require a non-tensorial coordinate transformation rule. Obviously the number of components (multiplicity) needs to be accompanied by some meta-information to be completely described, i.e. the transition rule.

2.4 Data Model Design

Data exchange and data access, whether it may be internally within an application via independent software components, or externally among applications, requires interfaces which explicitly communicate all properties of a dataset. The more explicit a description of a certain dataset is through a certain interface, the wider is its coverage as it avoids implicit assumptions that are only known within a small set of applications or application components.

2.4.1 Abstraction Levels

We identify four abstraction levels required to describe scientific data (similar to the scheme in the Sophus library [26]):

I.) *binary representation:* What are the atomic types and their binary representation? Usually we do not want to care about these low levels issues but low-level I/O libraries need to deal with their implementation details. Examples are IEEE float numbers such as `float` or `double` and their byte ordering.

II.) *multiplicity and parallelism:* How many of the atomic types do we need to assemble? For instance, how many atomic types (e.g. floats) do we need to numerically describe coordinates given on a two-dimensional domain? This multiplicity information allows to build the "carrier" of elementary (possibly parallelized) numerical operations; examples are tuples and arrays of atomic types, such as `float[3], double[3]`.

III.) *mathematical semantics:* What is the mathematical purpose of a certain object? For instance, a tangential vector, a normal vector or a coordinate location are all represented numerically by same number of atomic types; however, their mathematical properties - as defined from their chart transition rules - are different; examples are objects implementing the semantics of differential geometry, such as ∂_x vs. dx.

IV.) *physical semantics:* What is the physical meaning of a certain data set? Given a vector field, we need to associate a certain physical meaning to it in order to determine its purpose, e.g. its role in a differential equation, or which visualization methods are appropriate for it. For instance, the information that a data file contains multiple scalar fields is insufficient if it is not determinable which scalar field refers to temperature or pressure.

While the mathematical semantics (level III) specifies which operations are possible on a certain field, the specification of which operations are meaningful on it (level IV) are mostly user-driven and on the application level. Currently no ontological scheme for physical quantities exists and a textual representation which is subject to human interpretation is used instead. A possible automatized approach where to utilize physical quantities or to refer to the governing equations that were used to create the respective data sets.

2.4.2 Abstraction Interfaces

In a single application with proprietary I/O, the abstraction levels are usually intertwined. However, identifying each level within a complex context eases the development of reusable software components because each level can be implemented independently. We can analyze the different requirements for the interfaces between these levels through their relationships as depicted in the tetrahedron of Fig. 1:

I.)→ II.) *data structure:* The grouping of atomic types to tuples is implemented via assembling into structured data complexes; this step builds the multiplicity abstraction level.
II.)→ III.) *functional programming:* Adding mathematical operations to a data structure, such as interpreting a tuple of floats as elements of a vector space, constitutes the mathematical abstraction layer.
I.)↔ III.) *generic programming:* Implementing (mathematical) operations which can operate on an equivalence class of data structures is the domain of generic programming.

III.)→ IV.) *physical interpretation:* The specification of the purpose of a certain mathematical object for a specific usage scenario completes the data model. It defines which quantities play what role within the governing equations of an application.
I.)→ IV.) *I/O:* Selective I/O (in contrast to an all-memory core dump) is only possible once the physical semantics of the involved variables are known; not everything in memory needs to be saved to disk. This knowledge is only available on the application level and is imperative for *inter-application* communication.
II.)→ IV.) *computational request:* High-performance numerical operations in a computer are implemented via optimized bulk operations on tuples of atomic types. These numerical operations are independent from their mathematical semantics, but driven by the governing equations or human interaction based on their physical meaning.

The expressiveness of a data model can be measured by its coverage with respect to these abstraction levels. A model of higher abstraction level can always be imposed on a model of lower abstraction level. The respective couplings among each of these abstraction layers have their specific needs for various programming paradigm.

2.4.3 Case Study

We discuss the abstraction interfaces within the software components FiberLib2, GSSE, the HDF5 I/O library [39] and the Cactus Code [23].

I.) → II.) *data structure:* An implementation of this abstraction interface is provided by FiberLib2 for a volatile storage and HDF5 for persistent storage. In contrast, GSSE as a purely generic software component does not provide any storage mechanisms at all but utilizes iterators that implement an data model of abstraction level II.
II.)→ III.) *functional programming:* This abstraction layer is implemented in FiberLib2 through meta-data that allow to identify vectors and co-vectors, and is complemented by GSSE to extend the set of of data model operations.
I.)↔ III.) *generic programming:* GSSE implements one such set of (mathematical) operations independently of a specific data structure, with FiberLib2 as one possible carrier.
III.)→ IV.) *physical interpretation:* FiberLib2 allows this abstraction layer through textual human-interpretable identifiers for so-called grid and field objects, but does not yet provide support beyond this (such as an ontology for physical quantities).
I.)→ IV.) *I/O:* This interface is implemented via F5 [9], the HDF5 I/O layer of FiberLib2. It may be utilized as a standalone C library that directly speaks to an application without requiring the application to know about the FiberLib's data model.
II.) → IV.) *computational request:* This relationship is represented by the Cactus Code [23], which provides an abstraction layer in its kernel (the "flesh") for operations on grid functions and takes care of parallelization issues. The concrete implementation of data structures and operations is deferred to plugins (the "thorns"). As such, the Cactus Code is orthogonal in its functionality to GSSE (implementing generic operations) and FiberLib2 (implementing generic data structures). Integration of FiberLib2 and GSSE within Cactus will be subject of future work.

3. PROGRAMMING PARADIGMS

In many areas of scientific computing there is a manifold of software applications and tools available which provide methods and libraries for the solution of very specific problem classes [55, 47, 14, 11]. They are mostly specialized on a certain type of underlying mathematical model, resulting in a solution process which is highly predictable. However, it is important to note that such applications impose restrictions on possible solution methods which cannot be foreseen at the beginning of modern program development. But, as it can be observed from the last section, the development of highly reusable software components for this highly complex area demands different programming paradigms for an efficient realization. A software component is reusalbe if it can be used beyond its initial use within a single application or group of applications without modification.

Initially only one- and two-dimensional data structures were used due to the limitations of computer resources. The imperative programming paradigm was sufficient for this type of task. With the improvement of computer hardware and the rise of object-oriented programming paradigm, the shift to more complex data models was possible. Finally, the generic programming paradigm has emerged and has eased the development of independent algorithms. Therewith more operations on each data structure were possible and the distance to the mathematical semantic was steadily reduced.

Another important part in developing components for scientific computing are performance aspects, which should be handled orthogonally to the development of applications. Optimizations can thereby be treated separately. Related to Section 2.4.2, the performance aspects are part of the generic programming edge. With the multi-language approach, performance aspects cannot be considered orthogonally because of the use of compiled modules which require an interface layer in order to build applications.

3.1 OO vs. Generic vs. Imperative

Implications to application development can be observed clearly by studying the evolution of the object-oriented paradigm from imperative programming. The object-oriented programming paradigm [58] has significantly eased the software development of complex tasks, due to the decomposition of problems into modular entities. It allowed the specification of class hierarchies with its virtual class polymorphism (subtyping polymorphism), which was a major enhancement for many different types of applications. But another important goal in the field of scientific computing, *orthogonal libraries*, cannot be achieved easily by this paradigm. A simple example for an orthogonal library is a software component, which is completely exchangeable, e.g. a sorting algorithm for different data structures [47]. An inherent property of this paradigm is the divergence of generality and specialization [57, 3, 5]. Hereby the object-oriented programming paradigm is pushed to its limit with the requirements in the field of scientific computing, due to interface specifications, performance issues, and orthogonality. Even though the trend of combining algorithms and data structures is able to provide generalized access to the data structures through objects, it is observable that the interfaces of these objects become more complex as more

functionality is added. The intended generality often results in inefficiency of the programs, due to virtual function calls which have to be evaluated at run-time. Compiler optimizations such as inlining or loop-unrolling cannot be used efficiently, if at all. A lot of research was carried out to circumvent these issues [6], but major problems arise in the details [21].

Modern paradigms, such as the generic programming paradigm (GP [38, 30]), have the same major goals as object oriented programming, like re-usability and orthogonality. However, the problem is tackled from a different point of view [22]. Together with meta-programming (MP) [2], generic programming accomplishes both a general solution for most application scenarios and highly specialized code parts for minor scenarios without sacrificing performance [31, 4, 24] due to partial specialization. The C++ language supports this paradigm with another type of polymorphism which is realized with template programming [50], the parametric polymorphism. Combining this type of polymorphism with meta-programming, the compiler can generate highly specialized code without adversely affecting orthogonality. This allows the programmer to focus on libraries which provide concise interfaces with an emphasis on orthogonality, as can be found in the STL[34] or the BGL[47].

The generic programming paradigm establishes homogeneous interfaces between algorithms and data structures without sub-typing polymorphism by an abstract access mechanism, the iterator concept. This concept specifies an abstract mechanism for accessing data structures in a concise way.

Functional programming (FP) [36] eases the specification of equations and offers extendable expressions while retaining the functional dependence of formulae by higher order functions. The features of meta-programming offer the embedding of domain-specific terms and mechanisms directly into the language, as well as compile-time algorithms to obtain optimal run-time. To analyze the shift of paradigms and the advantages of using different paradigms, we review the development of applications at the Institute for Microelectronics at the TU Vienna. The programming paradigms used are stated as well, to highlight the development and achievements enabled by the different paradigms:

Name	Year	Paradigm	Information
`MINIMOS`	1980	imperative	[46]
`S*AP`	1989	imperative	[44]
`WSS`	2000	OO	[13]
`VGML`	2005	OO, GP	[27]
`GSSE`	2006	OO, GP, FP, MP	[28]

The shift from imperative programming to a large variety of different programming paradigms, such as generic programming, functional programming, and meta-programming, results in an enormous simplification of code. This development can be estimated using the total lines of code which is necessary to write applications [24].

year	lines of code	paradigm	language
1980	100.000	imperative	Fortran
1990	300.000	imperative	C, Fortran
2000	600.000	imperativ, OO	C, C++
2006	20.000	OO, GP, FP, MP	C++

By using generic components, the application design is reduced to specify the important parts only. All other source code is hidden in the components. The drastic reduction of source lines arises from the fact, that all complex data structure relevant code is now concentrated in one library, a generic topology library (GTL, [29]). The mathematical formalism is covered by a generic discretization library (GDL, [51]).

The C and Fortran languages do not offer techniques for a variable degree of optimization, such as controlled loop unrolling [49, 33]. Such tasks are left to the compiler. Therefore, libraries have to use special techniques such as practiced by ATLAS [59] or have to rely on manually tuned code elements which have been assembled by domain experts or the vendors of the microprocessor architecture used. Thereby, a strong dependence on the vendor of the microprocessor is incurred. In short, these methods hugely complicate the development process of high performance libraries.

The basic parts of how to achieve high performance in C++ can be summarized as follows:

- Parametric polymorphism: with the compiler's data-type-based function selection at compile time, a global optimization with inlined function blocks and inter-procedure/inter-library optimization is possible.
- Lightweight object optimization [48]: allocation to registers is possible with the reduction of structures to their basic parts.

The unique way parametric polymorphism is realized in C++ [50, 40] makes it possible to write compile-time libraries that enable an optimization across the boundaries of the libraries, thereby reaching new performance optima at the expense of increased compile time, but also reduction of implementation task due to more abstraction and thus reusability. This has already been demonstrated in the field of numerical analysis, yielding figures comparable to Fortran [55, 54, 56], the previously undisputed candidate for this kind of calculation.

3.2 Property-based vs. Content-type

The question of what paradigms are most appropriate to implement certain parts is tightly coupled to the kind of queries that need to be dealt with. Just as programming languages distinguish different data types, so mathematics and physics discuss several different entities such as scalar or vector fields. The properties of these entities are unambiguously described in the fiber bundle model. In the FiberLib data model these properties are mapped to specific entries in a five-level hierarchical system, which is reviewed in section 4.2. The inverse problem is to find out the entity from a given set of properties. This query may be addressed by adding a content-type meta-tag that answers exactly this

question. However, while such a content-type solution might well aid for a debugging purpose, it defeats the purpose of the fiber bundle model, as it is an alias to the total space instead of allowing separate views to the base and fiber space. This separation is the strength of the fiber bundle model and ensures the high re-usablity of operations for similar data types. In particular, a certain operation will hardly be required to make use of all properties of the total space at the same time; instead, it is more appropriate for an operator to query whether the current data entity of interest will provide the properties that are required to perform the desired operation. This way the "what" inquiry is transformed into a "how" inquiry, which is more appropriate for re-usability of software components as well as for interoperability among applications.

4. COUPLING GENERIC ENVIRONMENTS

4.1 GSSE

Generic library design deals with the conceptual categorization of computational domains, the reduction of algorithms to their minimal conceptual requirements, and strict performance guarantees. The benefits of this approach are the re-usability and the orthogonality of the resulting software. Generic libraries were pioneered by the Standard Template Library (STL) in C++, but both software and language technology have long gone beyond STL. GSSE is based on two generic libraries, GTL and GDL.

We compiled libraries which contain functionality meeting four main criteria. First, the library should be complete so that all applications can be written exclusively using this library (as well as standard libraries). Indeed, completeness increases the usability enormously, because no components have to be added while existing components can be adapted. Second, the parts of the library should be usable for a broad range of different applications. Each of the software components is not only written for a very specific purpose, but for a manifold of problems. Third, the interoperability of the library must not be affected by its completeness. Even if the complete library can be used by itself, it has to provide standardized interfaces which guarantee compatibility for data structures which have not been foreseen in the initial design.

The GTL was developed to interpret data structures as cell complexes of a certain dimension. Because of the combinatorial properties of complexes, a higher-dimensional data structure can also be considered as the projection onto any lower-dimensional one, thereby traversed in multiple ways. Related to the base space property of the fiber model, the GTL operates on the n-skeletons. Because of the distinction between local and global properties, data type designers can at the same time control which incidence relations, thus which kinds of efficient traversal, are available. Local properties can be depicted by local or explicit neighborhood information, whereas global properties are modelled by implicit neighborhood information.

The GDL was developed to describe a set of common operations for solving partial differential equations, namely those concerned with assembling equations. By examining several libraries a common framework was developed that encompassed finite element, finite volumes, and finite difference techniques. The library uses C++ techniques to very succinctly express how to assemble the discretized form of the partial differential equations.

4.2 FiberLib2

Inspired by the Butler's vector bundle data model [17] and OpenDX, "FiberLib2" is a re-implementation of a data model which was originally conceived [8] for visualizing numerical data originating from of general relativity (GR). Since the mathematics of GR requires explicit treatment of otherwise implicitly assumed properties of space and time, designing a data model to cover GR improves its genericity. The model is represented by an acyclic graph of five levels with actual data sets only at the end nodes of the graph. The location of a data set throughout the path in the graph defines the semantics of the specific data set, allowing to naturally group data sets [into "index spaces"] which share common properties. These levels are

1. **slice** (grouping all data which belong to a certain point in the parameter space)
2. **grid** (grouping data which refer to a certain data source)
3. **topology** (grouping data which describe a - mostly topological - property of a grid instance)
4. **representation** (grouping all data which describe a topological property with respect to "something else", e.g. a coordinate system or cell - vertex relationships)
5. **field** (actual data sets storing numerical information).

This model eases database-like queries such as which data exist for a certain timestep or which vertices exist per cell of a certain mesh. Internal reverse representations also allow inverse queries, such as asking for which time steps or on which grids a certain field exists, or for the cells per vertex.

From these five levels, only the grid and field levels are exposed to the application. Their identifiers carry semantic information beyond the pure mathematical description. The topology level corresponds to the skeleton spaces of a CW-complex or multiple refinement levels and is used to identify the topological properties of a grid. The representation level allows multiple coordinates of the same field to co-exist, such that a numerical scheme may choose the best one suitable for a certain problem. Not all data sets need to be stored explicitly, e.g. functional representations such as the vertex coordinates of a uniform grid may be built from few parameters.

Beyond the pure in-memory representation of the data model, its data structures map more or less directly to various file formats. The hierarchical structure inherent to HDF5 provides a natural representation. We made this HDF5 representation of the data model usable an independent C library in order to equip external application with a compatible I/O layer. This C library, called "F5", provides a simple interface for most application scenarios which occur in external applications.

4.3 Coupling FiberLib2 and GSSE

FiberLib2 and GSSE have both been developed independently, but share the same mathematical background: topology and differential geometry. FiberLib2 provides a data model consisting of base space and fiber space that GSSE may operate on: The GTL part of the GSSE offers all different types of topological traversal through the base space, whereas with the GDL of the GSSE, algorithms and equations of the fiber space can be described easily.

Both environments are complementary. Interoperability possibilities between applications are provided by FiberLib2 and added as feature to GSSE, which primarily aims at application design. The GTL provides a complete and comprehensive iterator hierarchy, yielding a unique data access mechanism for all different dimensions and topological structures. The GDL eases the specification of equations as well as complete algorithms. By close coupling of these generic environments, it is possible to implement well scaling applications rapidly. Several advantages can be derived directly from the functional nature of the GDL. On the one hand, side-effects which complicate the actual software development are reduced to a minimum. On the other hand, the functional character of the equations can be retained and equations can therewith be build step by step. The result is a high-performance multi-paradigm environment with a comprehensive and complete underlying data model.

5. APPLICATION DEVELOPMENT

5.1 Traversal and Data Access with the GTL

The GTL allows the specification of algorithms independently from the actual dimension or topological structure of the grid. In order to find a concise definition, however, we have to identify parts of the code which are commonly used and re-formulate the used code parts in a manner which fulfills the requirements of

- re-usability and
- formulation close to mathematics.

The following code snippet shows the conventional implementation using GTL methods only. It is used to specify the $\mathrm{div}(u \otimes u)$ term of the Navier Stokes equations using finite volume schemes. We use `u` on vertices to access the value of the fluid velocity. On the edges we have `A, d` as Voronoi geometry factors of the dual graph as well as the normal flux `u_n`. The iterators `vertex_edge` as well as `edge_vertex` are used to retrieve all incident edges of a vertex and all incident vertices of an edge.

```
array<3> div_u_u;

vertex_edge eit(v);
for(; eit.valid(); ++eit)
{
array<3> inter = 0;

  edge_vertex vit(*eit);
  for(; vit.valid(); ++vit)
  {
    inter += u(*vit);
  }
 inter *= A(*eit) / d(*eit) * u_n(*eit);
 div_u_u += inter;
}
```

Both, the mechanisms for quantity access as well as the mechanisms for combined iteration and accumulation can be implemented as separate code elements. It can also be seen easily that only the iterator of the innermost loop is used when accessing quantities.

5.2 Discrete Formulation with the GDL

The GDL implements function objects which encapsulate the functionality of loops over incident elements combined with accumulation. A possible implementation of such a loop can be found in the following code snippet which is written within the context of the Phoenix 2 library [15].

```
eval(Env & env, Initial & init, Summand & summand)
{
  base_elem(at<0>(env.args));
  Iterator iter(base_elem);
  result = init.eval(env);

  while(iter.valid())
  {
    result += summand.eval(newenv(env, *iter));
    ++iter;
  }
}
```

The iterator `iter` is constructed via the base element which is passed to the function object. Then the result is initialized using another function object called `init`. This is required to abstract the initial value from the underlying numerical data type as well as the accumulation operation (e.g. multiplication).

After the initialization, all incident elements are traversed with the iterator `iter`. The `valid()` method states if the dereferentiation of the iterator would yield a valid result. The `summand` function object is evaluated on the newly traversed elements `*iter`. The Phoenix2 library provides convenient construction of functional data structures such as higher order functions, named and unnamed variables, binders and operators, however, it imposes the use of environments which slightly reduce the transparency of the code. For the discrete formulation, the following code can be used.

Using object generators [2], the formulation of mathematical expressions can be simplified enormously, mostly because types can be derived automatically. The expression can be written in the following form.

```
sum<base_traversed>(Initial)[Summand]
```

Here, `base` denotes the type of the base element such as `vertex`, while `traversed` stands for the type of the traversed element (e.g. `edge`). In this case the expression (`vertex_edge`) traverses all edges which are incident to one vertex.

Functional programming allows us to formulate a discretization scheme concise while it still provides the dimensional

and topological independence. Therefore we can formulate the Navier Stokes equations using finite volumes in the following manner:

```
sum<vertex_edge>()
[
  sum<edge_vertex>()[u(_1)]
  * A(_1) / d(_1) * u_n(_1)
]
```

The unnamed function _1 is used to evaluate the quantities in the innermost loop. At this level, different formulations in finite element schemes or finite volume schemes lead to different formulae.

5.3 Implementation of Equations with GSSE

A generic Laplace (6) can be implemented by the following source code. First, the right hand side is set to zero:

```
eq = sum<vertex_edge>
[
  sum<edge_vertex>(0.0, _e)
  [ psi * orient(_1, _e)
  ] * A / d * eps
]
```

With small changes the Laplace equation can be extended to a Poisson equation. With the next code snippet the great extensibility of the functional programming approach can be clearly observed. Most of the source code remains unchanged; only minor parts have to be added.

```
eq = sum<vertex_edge>
[
  sum<edge_vertex>(0.0, _e)
  [ psi * orient(_1, _e)
  ] * A / d * eps
] - V * rho
```

The data accessor implementation takes care of accessing data sets with different data locality, e.g., data on vertices, edges, facets, or cells. In this case, V and rho are located on different objects.

The source code of the discretized drift-diffusion equations (4) is shown in the following code snippet:

```
// Poisson equation
equation_poiss=
sum<vertex_edge>
[
  sum<edge_vertex>(0.0, _e)
  [ psi * orient(_1, _e)
  ] * A / d * eps
] - ( n-p+nA-nD) * ( V*q/(eps0 * epsr)))

// Continuity equation for electrons
equation_n = sum<vertex_edge>
[
 sum<edge_vertex>(0.0,_e)
 [
  orient(_e, _1) * n(_1) *
  Bern(locate(_e)
   [sum<edge_vertex>[psi]/U_th])*A/d ]
  )
 ]
]
```

The Bernoulli function, given in (3), is mapped to `Bern`, while `psi` is a functional object providing access to a quantity. Due to the functional specification, a special mechanism has to be introduced `locate(_e)` to obtain the edge information in the innermost part.

While the Yee formulation of equation (7) makes use of staggered grids, the application on structured topologies causes an enormous simplification. Instead of special grids, we employ higher dimensional elements such as edges and faces for the representation of electrical field strength and magnetic induction. It turns out that the tensorial character of the quantities fits into the dimensionality concept of the topological elements. The final source code is presented in the following code snippet. The minimal requirement to specify such complex equations can be seen clearly.

```
E += dt * d / eps * sum<edge_facet>(0.0, _e)
[
 H / A * orient(_e, _1)
]
```

5.4 Continuous Layer

Even though these formulae can be specified for several different discretization schemes, the formulation still requires much in-depth knowledge of each single discretization scheme. The use of different namespaces offers the possibility to use different discretization schemes as well as different mechanisms of operators. For this reason it is possible to provide different discretization schemes using the same formulation. The discretization schemes can be easily exchanged by using various namespaces.

The major advantage of this method is that the actual mathematical problem is specified continuously rather than discretely. Therefore, the in-depth knowledge of the final library user can be reduced enormously while still providing the flexibility of exchanging discretization schemes.

5.5 Final Implementation

The following code shows the final implementation of the introduced concepts for the discretization of the Navier Stokes equations.

The function objects, or actors, are combined with mechanisms of the GDL to specify the complete equation at compile time. The run-time part consists of the evaluation of the function objects on the vertices using incidence information only.

```
...
scalar_quantity psi("psi");
scalar_quantity rho("rho");
AUTO(equ_poiss, div(grad((psi)) - rho );
...
for (vertex_iterator vit = vertex_begin();
     vit.valid(); ++vit)
{
  linear_equation<numeric_type>  poisson
        = equ_poiss(*vit);
  //  ... assemble the equations ...
}
```

The expression `AUTO`[1] is only used to a missing feature in the current C++ language. With the upcoming new C++0x standard the keyword `auto` is part of the language and we can write the following instead:

```
auto equ_poiss = div(grad((psi)) - rho ;
```

With GDL and GTL we have a framework which provides a very high language flexible generic implementation for a wide variety of different problems within the range of scientific computing, especially the solution of partial differential equations. Due to the generic implementation, the performance of software can be improved orthogonally to the actual formulation of the problem.

5.6 Data Output

The file resulting from the numerical simulation has to encompasses the time-dependent output of dynamic eqn. (4) as Ψ, n, p and eqn. (5) as E, H, A, D. These fields live on different skeletons of the grid:

- Vertex fields: Ψ, n, p
- Edge fields: A, D
- Face fields: E, H

Using the HDF5 I/O layer of FiberLib2, this output information will appear in a file listing (using the HDF5 standard tool `h5ls`) as:

```
/T=1.0/GSSE/Points Group
/T=1.0/GSSE/Points/Cartesian Group
/T=1.0/GSSE/Points/Cartesian/Positions Dataset {98317}
/T=1.0/GSSE/Points/Cartesian/psi Dataset {98317}
/T=1.0/GSSE/Points/Cartesian/n Dataset {98317}
/T=1.0/GSSE/Points/Cartesian/p Dataset {98317}

/T=1.0/GSSE/Edges Group
/T=1.0/GSSE/Edges/Points Group
/T=1.0/GSSE/Edges/Points/Positions Dataset {81317}
/T=1.0/GSSE/Edges/Cartesian/A Dataset {81317}
/T=1.0/GSSE/Edges/Cartesian/D Dataset {81317}

/T=1.0/GSSE/Faces Group
/T=1.0/GSSE/Faces/Points Group
/T=1.0/GSSE/Faces/Points/Positions Dataset {53965}
/T=1.0/GSSE/Faces/Cartesian/E Dataset {53965}
/T=1.0/GSSE/Faces/Cartesian/H Dataset {53965}

/T=1.0/GSSE/Connectivity Group
/T=1.0/GSSE/Connectivity/Points Group
/T=1.0/GSSE/Connectivity/Points/Positions Dataset {70965}
```

The file structure provides a grouping of the simulation fields according to their topological relationship within the mesh. From the mathematical point of view, this file structure is self-describing, yielding an abstraction level III. What is left for future work is the replacement of textual descriptions such as `psi, A, D` etc. by some ontology-based scheme for describing physical quantities independently of their naming conventions, which even among physicist lead to confusion. The planned introduction of metric units into HDF5 will be useful for this purpose.

[1]Using the GNU C++ compiler or the BOOST library, it can be implemented using the preprocessor expression `#define AUTO(var, value) typeof(value) var = value`

6. CONCLUSIONS

We have discussed the coupling of two a-priori independently developed software components, FiberLib2 and GSSE. We analyzed their abstraction capabilities and found them to share the same mathematical semantics. The first component, aiming at inter-application aspects, and the second one, aiming at intra-application design, complement each other perfectly. Their coupling results in a high performance framework which allows to specify equations close to their mathematical notation and to create output data files that are able to fully communicate the mathematical content in a self-describing way.

7. ACKNOWLEDGMENTS

The authors want to thank the Institute for Microelectronics. Special thanks go to Yaakoub El Khamra, Mayank Tyagi and Hartmut Kaiser from the Center for Computation & Technology at LSU for lively and inspiring discussions.

8. ADDITIONAL AUTHORS

Additional authors: Philipp Schwaha (Institute for Microelectronics), email: `schwaha@iue.tuwien.ac.at`; Michael Spevak (Institute for Microelectronics), email: `spevak@iue.tuwien.ac.at`.

9. REFERENCES

[1] A. Fabri. CGAL- The Computational Geometry Algorithm Library, 2001. `citeseer.ist.psu.edu/fabri01cgal.html`.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[3] R. Affeldt, H. Masuhara, E. Sumii, and A. Yonezawa. Supporting Objects in Run-Time Bytecode Specialization. In *Proc. of the Symp. on Part. Eval. and Semantics-Based Prog. Manip.*, pages 50–60, New York, NY, USA, 2002. ACM Press.

[4] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[5] H. M. Andersen and U. P. Schultz. Declarative Specialization for Object-Oriented-Program Specialization. In *PEPM '04: Proc. of the 2004 ACM SIGPLAN Symp. on Part. Eval. and Semantics-Based Prog. Manip.*, pages 27–38, New York, NY, USA, 2004. ACM Press.

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. In *Proc. of the OOPSLA Conf.*, volume 26, pages 345–420, 1996.

[7] O. S. Bagge. CodeBoost: A framework for transforming C++ programs. Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, March 2003.

[8] W. Benger. *Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model*. PhD thesis, Free University Berlin, 2004.

[9] W. Benger. Fiberbundle hdf5. `http://www.fiberbundle.net/`, 2005.

[10] G. Berti. *Generic Software Components for Scientific Computing*. PhD thesis, University of Cottbus, 2000. `http://www.math.tu-cottbus.de/~berti/diss/`.

[11] G. Berti. GrAL - The Grid Algorithms Library. In *ICCS '02: Proc. of the Conf. on Comp. Sci.*, volume 2331, pages 745–754, London, UK, 2002. Springer-Verlag.

[12] E. Bethel. Interoperability of visualization software and data models is not an achievable goal. In *IEEE VIS 2003*, 2003.

[13] T. Binder, A. Hössinger, and S. Selberherr. Rigorous Integration of Semiconductor Process and Device Simulators. *IEEE Trans.Comp.-Aided Design of Int. Circ. and Systems*, 22(9):1204–1214, 2003.

[14] C. E. Board. *CGAL-3.2 User and Reference Manual*, 2006.

[15] Boost. *Boost Phoenix2*, 2006. http://spirit.sourceforge.net/.

[16] D. M. Butler. Sheaf data model, July 2005. US Patent 6,917,943.

[17] D. M. Butler and S. Bryson. Vector bundle classes from powerful tool for scientific visualization. *Computers in Physics*, 6:576–584, nov/dec 1992.

[18] D. M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45–51, sep/oct 1989.

[19] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):pp. 377—387, 1970.

[20] L. Cook and C. Matarazzo. The tri-lab data models and format (dmf) project. `http://www.ca.sandia.gov/asci-sdm/cgi-bin/sdmframedisplay.cgi/asci-sdm/DMFNecdc/DMFNecdcv4.html`.

[21] D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *CC '00: Proc. of the 9th Conf. on Compiler Constr.*, pages 82–93, London, UK, 2000. Springer-Verlag.

[22] T. Geraud and A. Duret-Lutz. Generic Programming Redesign Pattern. In *Proc. of the 5th Conf. on Pattern Lang. of Progr. (EuroPLoP '2000)*, Irsee, Germany, 2000.

[23] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf. The cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing - VECPAR '2002, 5th International Conference*. Springer, 2003.

[24] D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser, and S. Schupp. Generic Programming and High-Performance Libraries. *Int. J. of Parallel Prog.*, 33(2), June 2005.

[25] R. B. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provisions for regular and irregular grids. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 298–305, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[26] M. Haveraaen, H. A. Friis, and T. A. Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 3(6):241–270, 1999.

[27] R. Heinzl and T. Grasser. Generalized Comprehensive Approach for Robust Three-Dimensional Mesh Generation for TCAD. In *Proc. Conf. in Sim. of Semiconductor Processes and Devices*, pages 211–214, Tokio, September 2005.

[28] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. A High Performance Generic Scientific Simulation Environment. In *Proc. of the PARA Conf.*, page 61, Umea, Sweden, June 2006.

[29] R. Heinzl, M. Spevak, P. Schwaha, and S. Selberherr. A Generic Topology Library. In *Library Centric Sofware Design, OOPSLA*, Portland, OR, USA, October 2006.

[30] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. G. Siek. Algorithm Specialization in Generic Programming - Challenges of Constrained Generics in C++. In *PLDI '06: Proceedings of the ACM SIGPLAN 2006 conference on Programming language design and implementation*, New York, NY, USA, June 2006. ACM Press.

[31] J. Järvi, J. Willcock, and A. Lumsdaine. Concept-Controlled Polymorphism. In *GPCE '03: Proc. of the 2nd Conf. on Generative Prog. and Comp. Eng.*, pages 228–244, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[32] P. Kochevar. Database management for data visualization. In *Workshop on Database Issues for Data Visualization*, pages 109–117, 1993.

[33] L. Lee and A. Lumsdaine. Generic Programming for High Performance Scientific Applications. In *JGI '02: Proc. of the 2002 joint ACM-ISCOPE Conf. on Java Grande*, pages 112–121, New York, NY, USA, 2002. ACM Press.

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

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

[36] B. McNamara and Y. Smaragdakis. Functional Programming in C++ using the FC++ Library. *SIGPLAN*, 36(4):25–30, Apr. 2001.

[37] P. Moran. Field model: An object-oriented data model for fields. Technical report, NASA Ames Research Center, 2001.

[38] D. R. Musser and A. A. Stepanov. Generic Programming. In *Proc. of the ISSAC'88 on Symb. and Alg. Comp.*, pages 13–25, London, UK, 1988. Springer-Verlag.

[39] NCSA. Hierarchical data format version 5. `http://hdf5.ncsa.uiuc.edu/hdf5/`, 2003. National Center for Supercomputing Applications, Illinois.

[40] C. E. Oancea and S. M. Watt. Parametric Polymorphism for Software Component Architectures. In *Proc. of the OOPSLA Conf.*, pages 147–166, New York, NY, USA, 2005. ACM Press.

[41] P. Rhodes, R. Bergeron, and T. Sparr. A data model for multiresolution scientific data environments. In *NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometrical Methods for Scientific Visualization, Tahoe City, California, October 15–17*, 2000.

[42] P. J. Rhodes, R. D. Bergeron, and T. M. Sparr. A data model for adaptive multiresolution scientific data. *Data Visualization: The State of the Art*, pages 257–272, 2003.

[43] M. Rumpf, A. Schmidt, and K. G. Siebert. Functions defining arbitrary meshes - a flexible interface between numerical data and visualization. *Computer Graphics Forum*, 15(2):129–142, 1996.

[44] R. Sabelka and S. Selberherr. A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures. *Microelectronics Journal*, 32(2):163–171, 2001.

[45] D. Scharfetter and H. Gummel. Large-Signal Analysis of a Silicon Read Diode Oscillator. *IEEE Trans. Electron Dev.*, 16(1):64–77, 1969.

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

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

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

[49] J. Siek and A. Lumsdaine. The Matrix Template Library: Generic Components for High-performance Scientific Computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.

[50] J. G. Siek and A. Lumsdaine. Concept Checking: Binding Parametric Polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, 2000.

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

[52] D. Stalling, M. Westerhoff, and H.-C. Hege. Amira - an object oriented system for visual data analysis. In C. R. Johnson and C. D. Hansen, editors, *Visualization Handbook*. Academic Press, 2005.

[53] L. A. Treinish. Data explorer data model. `http://www.research.ibm.com/people/l/lloydt/dm/dx/dx_dm.htm`, Mar. 1997.

[54] T. L. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[55] T. L. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[56] T. L. Veldhuizen and 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-Verlag, 1998.

[57] E. N. Volanschi, C. Counsel, G. Muller, and C. Cowan. Declarative Specialization of Object-Oriented Programs. In *Proc. of the OOPSLA Conf.*, pages 286–300, New York, NY, USA, 1997. ACM Press.

[58] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, 1990.

[59] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *9th SIAM Conf. on Parallel Proc. for Sci. Comp.*, 1999. CD-ROM Proceedings.

[60] K. S. Yee. Numerical Solution of Initial Boundary Value Problems involving Maxwell's Equations in Isotropic Media. *IEEE Trans. Antennas and Propagation*, 14(1):302–307, 1966.