Integrating object-oriented and generic programming paradigms in real-world software environments

Experiences with AMDiS and MTL4

Peter Gottschling Thomas Witkowski Axel Voigt

Institut für Wissenschaftliches Rechnen
Technische Universität Dresden
{Peter.Gottschling, Thomas.Witkowski, Axel.Voigt}@tu-dresden.de

Object-oriented software development is a broadly used programming paradigm that is successfully applied to a huge number of large-scale software systems, including many scientific HPC applications. Generic programming on the other hand is able to release unnecessary interface restrictions, thus allowing for lifting applicability to a potentially infinite number of types. At the same time, conceptual specialization enables algorithmic specialization at compile time leading to optimal performance. Both paradigms expose many parallels and are simultaneously orthogonal in many aspects. Although their combination does not create a theoretical contradiction, the integration of OO and generic software expose some technical limitations and is accompanied with several technical difficulties. We demonstrate the problems on the real-world example of integrating the generic linear algebra library MTL4 (Matrix Template Library) into the OO finite element software AMDiS (Adaptive Multi-Dimensional Simulations). We show solutions for these problems and present the benefits in terms of improved generality and increased performance.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented programming; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—object-oriented programming; D.2.2 [*Software Engineering*]: Design Tools and Techniques—software libraries

General Terms Design, Languages, Generic Programming

Keywords Matrix Template Library, AMDiS

1. Introduction

Software libraries as they are used today typically consist of collections of related functions and data types, often as archives of object files in conjunction with header files that define interfaces to components in the library. The Standard C Library [16], the FORTRAN BLAS libraries for linear algebra [3, 4, 13], and the LEDA graph library [12] are prototypical examples. Such libraries are limited by the fact that one can use the functions in the library only with the data types supplied by (or specified by) the library. A user who wishes to compose two independently developed libraries may be unable to do so, simply because the interfaces in software libraries today are over-specified. Instead of requiring only what is necessary for a routine to function correctly, interfaces express additional “administrative” requirements for their inputs.

Object-oriented and generic programming provide more powerful mechanisms of reusability by using different kinds of polymorphism. While OO mechanism are clearly more flexible than classical programming, the common derivation from a certain base class still imposes avoidable constraining. In this regard generic programming is more flexible and we will illustrate this in the next section.

2. Paradigms

2.1 Generic Programming

To introduce the paradigm on an example, consider the strstr routine in the Standard C library, which searches for an occurrence of a sub-sequence in another sequence. Although the same basic algorithm could be used to search sequences made up of many other types, strstr only works for character sequences.

Generic programming is an important paradigm for the development of highly-reusable software libraries [1, 15]. The term "generic programming" is perhaps somewhat misleading because it is about much more than simply programming *per se*. Fundamentally, generic programming is a systematic approach to classifying entities within a problem domain according to their underlying semantics and behaviors. The attention to semantic analysis leads naturally to the *essential* (i.e., minimal) properties of the components and their interactions. Basing component interface definitions on these minimal requirements provides *maximal* opportunities for re-use.

2.1.1 Lifting

Concrete implementations are evolved into generic implementations through a process known as *lifting*, in which unnecessary requirements on types are removed from an implementation (the abstraction level of the implementation is "lifted"). Consider the following two implementations. The first computes the sum of doubles stored in an array; the second computes the sum of elements in a linked list.

```
double sum(double *array, int n)
{
  double s = 0.0;
  for (int i = 0; i < n; ++i )
    s = s + array[i];
  return s;
}

double sum(node *first, node *last)
{
  double s = 0.0;
  while (first != last) {
    s = s + first→data;
    first = first→next;
  }
  return s;
}
```

Abstractly, both implementations are doing the same thing: traversing a collection of elements and summing up the values. However, the implementations also impose additional requirements (and ones that are unnecessary for the purposes of summation). In the first implementation, the elements must be doubles stored in an array. In the second implementation, the elements must be of type node∗ with doubles stored in the data field.

Summing a collection of elements only requires that we are able to visit all of the elements in the collection and extract the corresponding values. A generic algorithm should therefore be able to work correctly with any collection of elements supporting traversal and element access. For instance, one could define a generic implementation of sum as:

```
template <typename InputIterator, typename T>
T sum(InputIterator first, InputIterator last, T s) {
  while (first != last)
    s = s + *first++;
  return s;
}
```

This algorithm is implemented as a function template, parameterized on InputIterator and T. The algorithm can be used with any type substituted for InputIterator, as long as that type supports the ++ operation for moving from one element to another and the ∗ operation for accessing a value. Similarly, the type bound to T must support assignment and addition. Note that although we have specified a particular syntax for these parameterized types (we have to write the algorithm down somehow), we have only specified policy — we have not specified *how* these operations must be carried out. In fact, the sum algorithm can be used with arrays, linked lists, or any other type that meets the requirements for InputIterator and T.

```
double x[10];
double a = sum(x, x+10, 0.0);

vector<int> y(10);
int b = sum(y.begin(), y.end(), 0);

list<complex<double>> z(10);
complex<double> c = sum(z.begin(), z.end(),
                        complex<double>(0.0,0.0));
```

Computing the product of a set of numeric values is essentially the same form of calculation — only the binary operation is replaced within the loop's statement.

The same is true for maxima, minima, logical conjunction, bitwise exclusive-or, i.e. every binary operation. Abstracting this binary operation leads to the generic STL algorithm accumulate [1]. The following listing illustrates its usage to compute a sum, a product and a logical disjunction of different types of containers:

```
double x[10];
double s= accumulate(x, x+10, 0.0, plus<double>());

vector<int> y(10);
int p= accumulate(y.begin(), y.end(), 1,
                  multiplies<int>());

list<bool> z(10);
bool d= accumulate(z.begin(), z.end(), false,
                   logical_or<bool>());
```

Compilers cannot optimize such calculation because their code transformation strategies rely on semantic properties that are only available for intrinsic types and known operations. Future compilers will be able to handle semantic properties on user-defined operations and types. We could recently show that this ability allows for optimizing STL accumulate if the according semantic behavior is declared [6]. The foundation of this are concepts that we will introduce in the next subsection.

2.1.2 Concepts

Concepts are sets of *syntactical* and *semantical* requirements on a type or a tuple of types. A concept that adds requirements to another concept is called *refinement*. A type or tuple of types that fulfill all requirements of a concept is called a *model* of this concept. The requirements consist of

Valid Expressions: compilable C++ expressions that the type must provide,

Associated Types: other types than the modeling related to the concept,

Invariants: semantical properties, which can required to be true for all object of the type(s) — like associativity — or required on function arguments as precondition — like symmetric values of a matrix — and

Complexity Guarantees: maximum limits on compute time and memory needs often depending on complexity requirements on the parameters.

Currently, concepts are only documented and it is the user's responsibility to call generic functions correctly. Errors concerning valid expressions and associated types are syntactic and caught by the compiler. Invariants are semantic properties and undetectable by current compilers. It is very likely that the next standard of C++ will introduce concepts into the language [9] but this is beyond the scope of this paper.

2.2 Genericity in Inheritance-based OO

Similar multi-functionality as in Section 2.1 can be achieved without generic programming by using inheritance. The difficulties and limitations arising from this derivation approach are discussed in this subsection. Nevertheless, this limited form of abstraction is used very often in OO software; maybe more often than generic implementations.

2.2.1 Case study: accumulate

Instead of a concept like Iterator, we need an explicitly implemented base class with virtual functions:

```
struct iterator_base
{
  virtual iterator_base& operator++() = 0;
  virtual int operator*() const = 0;

  virtual bool
  operator==(const iterator_base& x) const = 0;

  virtual bool
  operator!=(const iterator_base& x) const {
    return !(*this == x);
  }
};
```

Classes entirely consisting of pure virtual functions are called "interface classes" with respect to their usage. In our example we have one virtual function that is not pure but is realized in terms of pure virtual functions and we therefore regard the type still as interface class.

The actual iterator type is than derived from this interface class:

```
struct list_iterator : public iterator_base
{
  list_iterator(list_element* me) : me(me) {}

  list_iterator& operator++() {
    me= me→next; return *this;
  }
  int operator*() const { return me→v; }
  bool operator==(const iterator_base& x)
```

```
    const {
      return me ==
        dynamic_cast<const list_iterator&>(x).me;
    }
  private:
    list_element* me;
};
```

In the same fashion, an interface class for binary operations is provided:

```
struct operation_base
{
  virtual int operator()(int, int) const = 0;
};
```

This class is derived in straightforward manner to realize binary operations as addition:

```
struct add_op : public operation_base
{
  int operator()(int x, int y) const { return x + y; }
};
```

The function accumulate[1] can be implemented against the two interfaces:

```
int accumulate(const iterator_base& begin,
               const iterator_base& end,
               int init, const operation_base& op)
{
  for (; begin != end;
       ++const_cast<iterator_base&>(begin))
    init= op(init, *begin);
  return init;
}
```

Using appropriate iterator and operation types, different reduction operations on different containers can be realized:

```
list l; l.push_back(3); l.push_back(5);
int sum= accumulate(l.begin(), l.end(), 0, add_op());
int p= accumulate(l.begin(), l.end(), 1, mult_op());

int a[]= {4, 5};
int asum= accumulate(array_iterator(a),
                     array_iterator(a+2), 0, add_op());
```

2.2.2 Discussion

Although the usage of interface classes allows for a comparable lifting process, the approach has several serious disadvantages but also some advantages. We will start with the latter.

[1] For the sake of simplicity we omitted the abstraction of the operations' argument and result types.

Compile time: With the interface approach, the lifted function is only compiled once with the interface types. The distinction between the different calculations is realized at run-time by means of virtual function tables. The generic implementation requires a new compilation for each combination of types. As a consequence, the sources must reside in header files[2] and cannot be stored to libraries.[3]

Executable size: As mentioned before, generic functions need multiple compilations and as a result of this, the generated executable contains code for each instatiation. A function programmed against an abstract interface exist only once. On the other hand, the virtual functions introduce some additional memory need to store the virtual function tables. Except for some pathological examples, one can expect that this additional space is less than the extra space needed for having separate machine code for every instantiation of a generic function.

Performance: The higher compilation efforts for generic programming has a double performance benefit. Functions/functors within the multi-functional computations do not need to be called indirectly via expensive function pointers but can be called directly. Whenever appropriate they can be even inlined saving the function call overhead entirely. We measured the performance gain in the considered case study of the two accumulate implementations. Adding 1000 **int** in an array takes 42.3 μs with the interface approach and 1.8 μs with STL accumulate (without concept-based optimizations [6]) on a 2.6 GHz AMD Athlon.

Handling of r-values: Considering the code examples before, it is convenient to pass parameters of containers given as r-values [11] — i.e. results of expressions such as l.begin() — directly to a generic function, e.g., accumulate(l.begin(), ..). Types like pointers and iterators have small objects and passing by value to functions is suitable. The interface approach does not allow passing arguments because it would create objects of abstract types. When using pointer arguments or non-const references for arguments r-values like l.begin() must be stored first into variables before they can be passed to function. Const references are more convenient in this regard but disable modifica-

[2] Unless **export** will be commonly supported by compilers.

[3] Libraries in the classical sense as opposed to template libraries.

tions or require error-prone **const_cast**s as we did in the example

Concept refinement: that is adding (syntactic) requirements is feasible with the interface approach. Publicly deriving from an interface class (say Interface1) by adding pure virtual functions leads to another interface class (say Interface2). This is isomorphic to creating Concept2 by adding syntactic requirements to Concept1. Analogously, types that model only Concept1 but not Concept2 can be derived from Interface1 but not from Interface2 in the inheritence-oriented implementation. Although this approach allows for emulating a concept hierarchy, the class hierarchy is unnecessarily inflated sacrificing clarity and finally also contributing to increasing compile time (without improving execution time).

Intrusiveness: The genericity emulation by inheritance induces not only a deep class hierarchy as mentioned before, more critical for the universal applicability is that the technology is intrusive. No iterator type that is not derived from iterator_base can be used with the interface implementation of accumulate from Section 2.2.1. *Not even if the iterator provides the correct interface!* To enable such an iterator for this accumulate, one must change the class definition. Needless to say that such modifications are not always feasible with third-party software or intrinsic types. For instance, pointers cannot be passed to the interface implementation but to the generic function. Generic functions are written in terms of free functions and type traits so that appropriate third party classes and intrinsic types can be provided with the necessary interface without intruding into the definition.

Résumé: It is not our goal to compare object-oriented and generic programming in general. The two approaches complete each other in many respects and this is beyond the scope of this paper. However, when only considering the aspect of maximal applicability with optimal performance the generic approach is undoubtly superior. Especially if functions of a library are used with types defined outside this library, possibly necessary interface adaption is quite easy without modifying the type definition while the addition of extra base classes forces changing the type definition what is not always possible (or desirable). In contexts where functions are used with limited numbers of types and they are defined in the same library, derivation can be an appropriate technique to achieve polymorphism.

3. Realization in Real-world Software Packages

Many program techniques work well as long as applied to simple tasks but reveal severe difficulties once used in large-scale software development. For this reason, we consider state-of-the-art packages of significant code complexity.

3.1 Matrix Template Library 4

MTL4 is a generic library for high-performance numeric operations on matrices and vectors [7, 14]. At the moment it provides a dense vector format, compressed sparse matrices, dense matrices and a large spectrum of recursively lay out dense matrices (like Morton-order in the simplest case). Traditional dense matrices can be stored in row-major order like C/C++ arrays or column-major like Fortran arrays. Sparse matrices are available as compressed row storage (CRS) and compressed column storage (CCS). The elements of the vectors can be intrinsic arithmetic or user-defined types as quaternions, intervals, high-precision numeric types and matrices and vectors themselves.

The library contains multiple new techniques as *implicit enable-if* and *meta-tuning* (short for performance tuned based on meta-programming). The latter allows for performance optimization like changing the block size of an unrolled loop or modifying tile sizes in blocked algorithm that otherwise require code rewriting. Thanks to meta-tuning the user can choose such parameters with template arguments in the function call and the compiler will generate corresponding code, e.g., dot<12>(u, v) computes a dot product with an unrolled loop using a block size of 12. The technique has proved high efficiency, partly out-performing assembler libraries as GotoBLAS [8].

3.2 AMDiS

AMDiS (**A**daptive **M**ulti-**Di**mensional **S**imulations) [20] is a toolbox for multi-dimensional simulation of physical phenomena and processes with a focus on solving problems in material science. It is based on the concepts of ALBERTA [18], an adaptive finite element toolbox for the numerical solution of partial differential equations (PDEs), but extends this concepts in several points and realizes them in a modular object-oriented design. Besides a much more modern software design, which allows for a more flexible use of the software, the main additional feature is the extension of several concepts to systems of PDEs and coupling of problems over different dimensions.

AMDiS solves general systems of second order PDEs. Problem formulations can be done on a high level of abstraction in a dimension independent way. Numerical implementation details are user-transparent, as far as possible. This makes it very easy for the user to implement and to test equations in 1D or 2D, and to solve them afterwards on a 3D geometry without changing the code. To solve time-dependent equations, a stationary problem and a time iteration interface have to be defined. The latter allows for controlling the whole process by implementing functions of this interface.

To solve complex problems on large domains with long timescales, AMDiS supports adaptivity in space and time. The goal of adaptivity is to achieve a solution that satisfies a given maximal error threshold (with as little computational effort as possible). In several case studies [2, 5, 17] we have shown the necessity of adaptivity for complex physical simulations.

A novel parallelization concept, called full domain covering meshes, is implemented in AMDiS [19]. Classical domain decomposition approaches divide the full problem domain into smaller partitions, which are than distributed to all processes. When data changes at partition boundaries, it must be communicated with the corresponding processes. Using full domain covering meshes reduces the parallel communication overhead. Here, each process computes a solution on the whole domain. Outside of its local partition a relatively coarse mesh is used. At the end of computation, the different processes combine their solutions into one global solution by the partition of unity method.

4. Integration

A significant fraction of ongoing development activities in AMDiS is dedicated to raising the performance. A central component with large potential of acceleration is the so-called DOFMatrix that contains the assembled element matrices over the degrees of freedom, therefore the name. It is heavily used in the linear solvers, thus strongly impacting the overall performance of every application.

The compressed sparse matrices in MTL4 provide optimal performance for general purpose. Innovative matrix types particularly suitable for FEM applications are under development. Refactoring the DOFMatrix based on MTL4 matrices will have an immediate performance boost.[4]

There are more advantages of using a generic matrix type as opposed to a hard-wired single type implementation. The DOFMatrix can be changed regarding the element type to complex by modifying only one line of code; or to fixed-size matrices for achieving better locality (i.e. higher performance) by blocking; or to intervals to examine the numerical stability of solvers; or to types with operators overloaded for automatic differentiation (AD) [10]; or to any combination of the above.

Last but not least, MTL4 matrices and the respective operations will be extended in the near future towards parallelism on shared and distributed memory. Furthermore, accelerations for future high-performance processor generations are planned.

In order to achieve maximum genericity, we templatized the DOFMatrix class regarding the used matrix type from MTL4 (or somewhere else):

```
template <typename BaseMatrix
             = mtl::compressed2D<double> >
class DOFMatrix
{
   /* ... */
  private:
    BaseMatrix matrix;
};
```

The first problem with this templatization is that the entire definition of DOFMatrix must be completely visible for all using entities. It must be therefore moved from source to header file(s). As it only concerns the DOFMatrix itself and not the utilizing library segments, this modification is fortunately local and can be limited to the two files DOFMatrix.h and DOFMatrix.cpp.

The second issue is that declarations of DOFMatrix exist in many other files which need to be adapted to the templatization. Such declaration adaption can be reduced to one single code modidification when instead of repeating declarations in all concerned files one dedicated file exist in the library containing only function and class declarations. Since this file is entirely inde-

[4] Unfortunately, the refactoring is not entirely finished by the time of that writing so that we cannot provide performance plots yet. We expect showing them at the workshop.

pend on the remainder of the sources it can be included in all other files. It is a common practice to name such a file like the library with an abbreviated ‘forward’ suffix, i.e. in our case amdis_fwd.h.

The third modification is that free and member functions with DOFMatrix as argument must be templatized as well. Those that are already template functions need an additional template argument for BaseMatrix. Unfortunately, these program alterations appear through a multitude of files while the changes are typically limited to few lines.

The most significant problem is the conflict between template classes and virtual functions. AMDiS uses virtual functions to a large extent, typically in the form:

```
class c1
{
  virtual void f1(DOFMatrix& m) = 0;

  void f2(DOFMatrix& m) {
    f1(m); /* ... */;
  }
};

class c2 : public c1
{
  void f1(DOFMatrix& m) { /* ... */ }
};
```

Virtual functions cannot be templatized in C++,[5] e.g., one cannot write:

```
class c1
{
  template <typename BaseMatrix>
  virtual void f1(DOFMatrix<BaseMatrix>& m) = 0;

  template <typename BaseMatrix>
  void f2(DOFMatrix<BaseMatrix>& m) {
    f1(m); /* ... */;
  }
};
```

It is possible to templatize the entire class:

```
template <typename BaseMatrix>
class c1
{
  virtual void f1(DOFMatrix<BaseMatrix>& m) = 0;

  void f2(DOFMatrix<BaseMatrix>& m) {
    f1(m); /* ... */;
  }
};
```

[5]The effort in compiler technology would be enormous and outweight the benefits.

The consequence of such modification would be an avalanche of more code changes since all classes containing virtual member functions with c1 as argument would then require modifications as well and all classes containing .... More importantly, this extensive class templatization would be conceptually wrong because c1 is not directly parametrized by BaseMatrix but only some of its member functions.

Facing all these technical complications, in particular the last one of templatizing virtual functions, we refrained from general templatization of AMDiS, at least for the near future. The general templatization would cause a complete redesign of AMDiS and also bearing the risk of significant increases of applications’ compile times.

Instead, we opted for a compromise that we call *installation-static genericity* similar to the choice of the scalar type during the installation of PETSc.

In fact, PETSc is not generic, its development was started before the introduction of generic programming in C++. Moreover, it is written in C impeding genericity at large. PETSc contains a macro definition for its scalar type and all vectors and matrices consist unexceptionally of this scalar type. When installing PETSc, the user must choose the scalar type and applications must be written for this type. If applications with different scalar types exist, multiple installations of PETSc are required. Despite the fact that PETSc only works with **float**, **double** and their respective complex types as scalars one can consider this as a limited form of genericity.

In AMDiS we fix the type of DOFMatrix’s base matrix:

```
class DOFMatrix
{
    typedef mtl::compressed2D<double>
      base_matrix_type;

    /* ... */
  private:
    base_matrix_type matrix;
};
```

This does not affect the design of the library, i.e. program sources does not need to moved from source files to header files and polymorphism is still dominantly realized by means of virtual functions. However, all functions in AMDiS that explicitly rely on the internal structure of DOFMatrix — mostly linear solvers and assembly routines — are consequently refactored towards generic implementations. As a consequence, the base_matrix_type can be changed without further need of modifications in the remainder of the library. This allows for easy switching to complex types, intervals, infinite-precision numerics, or to any other appropriate numeric type. Furthermore, the entire structure of the sparse matrix can be changed within one line towards unassembled, sparse banded, optimized symmetric, or distributed matrices, all under development in MTL4.

5. Conclusion

The integration of the generic linear algebra library MTL4 into the OO finite element software AMDiS disclosed several technical difficulties. These difficulties impeded the full usage of MTL4's genericity. Instead, we constraint ourselves to *installation-static genericity*, i.e. data types can be freely choosen during installation but not in the applications. This allows for future extensions as complex matrices or sparse band matrices without the need for fundamental redesign of AMDiS.

References

[1] Matthew H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[2] Eberhard Bänsch, Frank Haußer, Omar Lakkis, Bo Li, and Axel Voigt. Finite element method for epitaxial growth with attachment-detachment kinetics. *Journal of Computational Physics*, 194:409–434, 2004.

[3] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[4] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of basic linear algebra subprograms: Model implementations and test programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[5] Frank Haußer and Axel Voigt. A discrete scheme for parametric anisotropic surface diffusion. *Journal of Scientific Computing*, 30:223–235, 2007.

[6] Peter Gottschling and Walter E. Brown. Fundamental mathematical concepts for the stl in C++0x. Technical Report N2645=08-0155, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2008.

[7] Peter Gottschling, David S. Wise, and Michael D. Adams. Representation-transparent matrix algorithms with scalable performance. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125, New York, NY, USA, 2007. ACM Press.

[8] Peter Gottschling, David S. Wise, and Adwait Joshi. Generic support of algorithmic and structural recursion for scientific computing. In *Parallel Object-Oriented Scientific Computing workshop at ECOOP08*, Cyprus, Greece, 2008.

[9] D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Concepts (revision 5). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2008.

[10] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[11] Howard E. Hinnant, Dave Abrahams, and Peter Dimov. A proposal to add an rvalue reference to the C++ language. Technical Report N1690=04-0130, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming language C++, September 2004. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1690.html`.

[12] C. Uhrig K. Mehlhorn, S. Näher. Leda: Library of efficient datatype and algorithms. http://www.mpi-sb.mpg.de/LEDA/, 1998.

[13] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[14] Andrew Lumsdaine, Jeremy Siek, Lie-Quan Lee, and Peter Gottschling. The Matrix Template Library home page. `http://www.osl.iu.edu/research/mtl`, 2006.

[15] David R. Musser and Alexander A. Stepanov. Generic programming. In P. (Patrizia) Gianni, editor, *Symbolic and algebraic computation: ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Berlin, 1989. Springer Verlag.

[16] P. J. Plauger. *The Standard C Library*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1992.

[17] Andreas Rätz and Axel Voigt. PDE's on surfaces - diffuse interface approach. *Communications in Mathematical Science*, 4:575–590, 2006.

[18] Alfred Schmidt and Kunibert G. Siebert. *Design of Adaptive Finite Element Software: The Finite Element*

Toolbox ALBERTA, volume 42 of *LNCSE*. Springer.

[19] Simon Vey and Axel Voigt. Adaptive full domain covering meshes for parallel finite element computations. *Computing*, 81(1):53–75, 2007.

[20] Simon Vey and Axel Voigt. AMDiS: adaptive multidimensional simulations. *Computational Visualization and Science*, 10:57–67, 2007.