AMDiS - Adaptive multidimensional simulations: object oriented software concepts for scientific computing

SIMON VEY, AXEL VOIGT
Crystal Growth Group
Research Center caesar
Ludwig-Erhard-Allee 2, 53175 Bonn
GERMANY

Abstract: We describe the basic ideas and ingredients of adaptive FEM and their implementation in our software toolbox AMDiS. This software is primarily developed to solve realistic computations in materials science. The design of AMDiS is based on a natural hierarchy of locally refined meshes and an abstract concept of general finite element spaces which is combined with an object oriented data structure. In this way dimension independent programming is possible and complex applications can be implemented on an abstract level, keeping the numerical issues away from the user.

Key-Words: Finite element method, adaptivity, scientific software concepts.

1 Introduction

The finite element method is one of the main tools for numerical simulations of partial differential equations. Over the past decades, some very efficient techniques for finite elements have been developed which today provide more and more tools for the efficient solution of large scale applications. Adaptivity is one of these techniques and can be used to increase efficiency by local mesh refinement and/or using locally higher order elements, where applicable. Combining this with unstructured grids and fast solvers for the systems of linear equations make it today feasible to solve multi-scale problems in various fields of materials science, not only including the classical fields of elasticity and fluid dynamics but also microstructure evolution and thin film growth.

An adaptive finite element method adjusts the triangulation and/or the polynomial degree to the solution of the problem. Thereby adaptivity is based on information extracted from a posteriori error estimators. These error estimates are computable estimates for the error between the true solution and the finite element approximation and they are built up from local error indicators. An adaptive method is driven by such estimators and tries to optimize the mesh and/or the order of the elements by equidistributing the local indicator value over all elements [1], under the constraint that the estimate is below a given tolerance, which results in meshes and/or basis functions, which are highly refined and/or have high polynomial degrees only where really needed. The core part of every finite element program is the problem dependent assembly and solution of the discrete problem. Because this is also the most time consuming part, a general finite element software has to handle this part efficiently but on the other hand must provide enough flexibility in problems and finite element spaces to solve a wide class of partial differential equations. Therefore the combination of different spaces, different meshes and even different dimensions might be necessary but should be kept away from the user and handled by optimized library functions.

In this paper we describe the design of our adaptive finite element software AMDiS. The used data structures allow an implementation of the problem dependent part, that does not depend on the actual chosen local function space and the dimension. The concepts used are similar to the software in [2] but extend the ideas by adding an object oriented software design, an efficient memory management system, a problem oriented data structure and an abstract definition of the partial differential equation based on its variational formulation.

2 Software design

In this section we describe the overall software design of AMDiS and some of its components in more

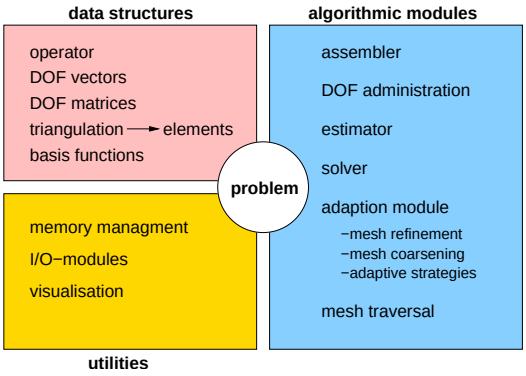

Fig.1: Main components of AMDiS

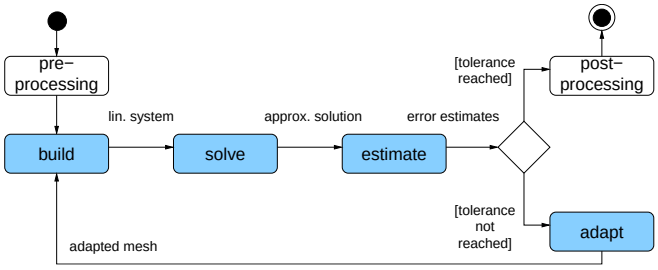

Fig.2: Adaption loop of a stationary problem

detail. A widely spread and quasi-standard way to describe object oriented design is the use of the *Unified Modeling Language* (UML) specified by the *Object Management Group* (OMG). For that reason, we use UML as well to illustrate the design of AMDiS components. In Figure 3 the used UML concepts are shown. For a closer look to UML see [3].

AMDiS is written in C++ (see [4]) in an object oriented manner. The main objectives in the development of AMDiS are

- **high level of abstraction** that allows a fast and dimension independent problem formulation keeping the numerical issues away from the user,
- **generality and extensibility** to enable the software to solve a large class of problems which might be coupled even over different dimensions,
- **efficient implementation** which permits to do the needed calculations in an minimal effort of time.

These are controversial goals in some aspects so only an useful tradeoff can be found. We tried to find this tradeoff by using well proven algorithms from computational science embedded in object oriented design patterns, which lead to a flexible and reusable software. A catalog of the most established design patterns is given in [5].

Figure 1 shows the main components of AMDiS divided in data structures holding needed and calculated probem data, algorithmic modules responsible for all computations, and utilities like memory management or pre- and postprocessing. To be solved the problem takes the center stage of the software, so to say it contains all needed information to solve itself.

In Figure 2 the main steps of solving a stationary problem are shown. In the preprocessing step the problem formulation and the initial triangulation are created and the problem parameters are choosen. Depending on these informations the system matrix and system vectors are assembled by assemblers optimized for this problem. The created system is solved by a specified solver and than local error indicators on each element are calculated, using a-posteriori error estimators based on residual techniques. If the sum of all local error indicators exceeds a given er-

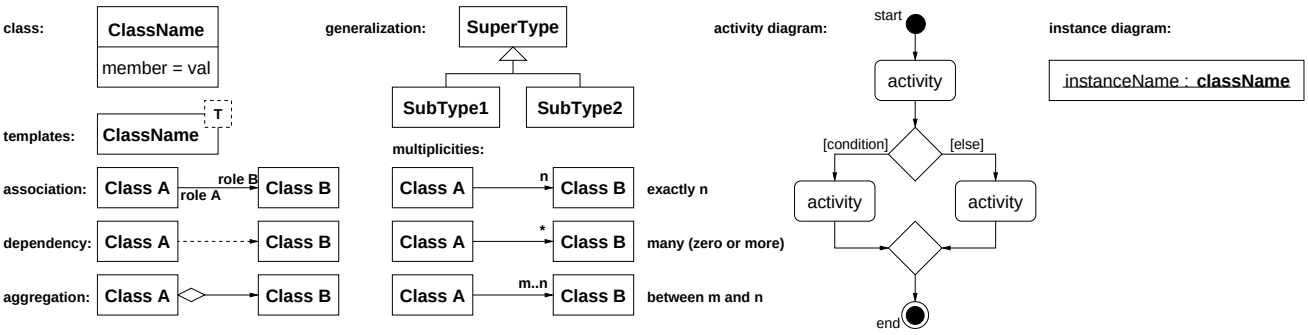

Fig.3: Used UML concepts

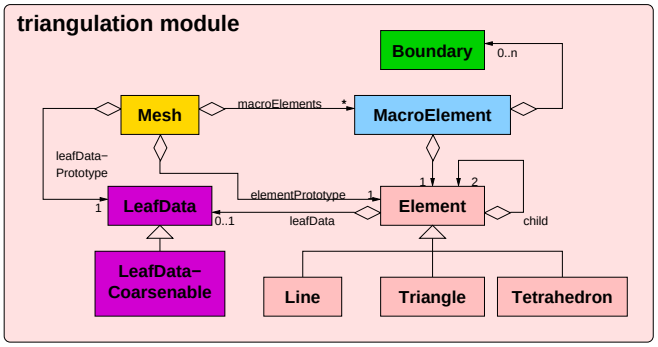

Fig.4: Class diagram of the triangulation module

ror tolerance the mesh is refined and/or local function spaces are adapted on those elements with the largest local error estimates. This build-solve-estimate-adapt loop is repeated until the error tolerance is reached. In the postprocessing step the calculated solution can be visualized.

2.1 Problem classes

As mentioned above the problem classes are the central buildings blocks of the AMDiS library and the main interface to the user. AMDiS contains two different problem base classes. One for stationary and one for instationary problems. To illustrate the design principles of the software, we focus on the description of the stationary problem in this paper. To reach a high level of abstraction as well as a high degree of generality the stationary problem is initially described by an interface at the highest abstraction level used for the simulation: the adaption loop. This means that it has the ability to assemble its system matrices and vectors, to solve the resulting equation system, to estimate the local errors, and to adapt the used discretisation. On this level nothing is known about the data structures or the algorithms used for the single steps. This design leads to a high flexibility and minimizes dependencies between different parts of the software which in turn leads to a high reusability and extensability. To create a concrete problem one has to derive a sub class from **ProblemStatBase** and override its pure virtual methods. To fill them with functionality, many tools are provided by the library. Some of this tools are described in the next subsections.

2.2 Operator classes

The **Operator** class is used for the problem formulation in AMDiS. You can give one or several operators to a **DOFVector** or a **DOFMatrix** (which describe the equation system) and give them the information how to assemble itself. To do this assemblage, for every operator a specific **Assembler** will be created, which is optimized for the special *needs* and *properties* of this operator. One operator can contain several **OperatorTerm**s, which can be of second, first, or zero order. An example for the use of operators can be seen in section 3.

2.3 Triangulation

The domain, the problem is defined on, is discretized by simplices. So a one dimensional mesh consists of lines, a two dimensional mesh consists of triangles, and a three dimensional mesh consists of tetrahedrons.

Even if it is in principle possible to approximate complex domains with simplicial meshes, it is the most time consuming part in many applications. Therefore the concept of composite finite elements described in [6] is implemented in AMDiS. This allows the approximation of complex domains without explicitly discretizing it. In AMDiS the mesh is stored in a hierarchical way. Starting from a coarse macro triangulation, the mesh can be refined locally by element bisection. The two new created elements are stored as *children* of the bisected element. So a binary tree arises. To store this hierarchical structure, instead of storing just a linked list of elements, allows to store many element information only on the macro elements. Furthermore it can be utilized by the neighbourhood search and the coarsening algorithm, and it is the natural structure for multigrid solvers.

In Figure 4 you see the class diagram of the triangulation module. A **Mesh** contains prototypes of **Element** and **LeafData** objects, which can be cloned while refinement, to create new elements and leaf data when needed. Furthermore it contains a list of

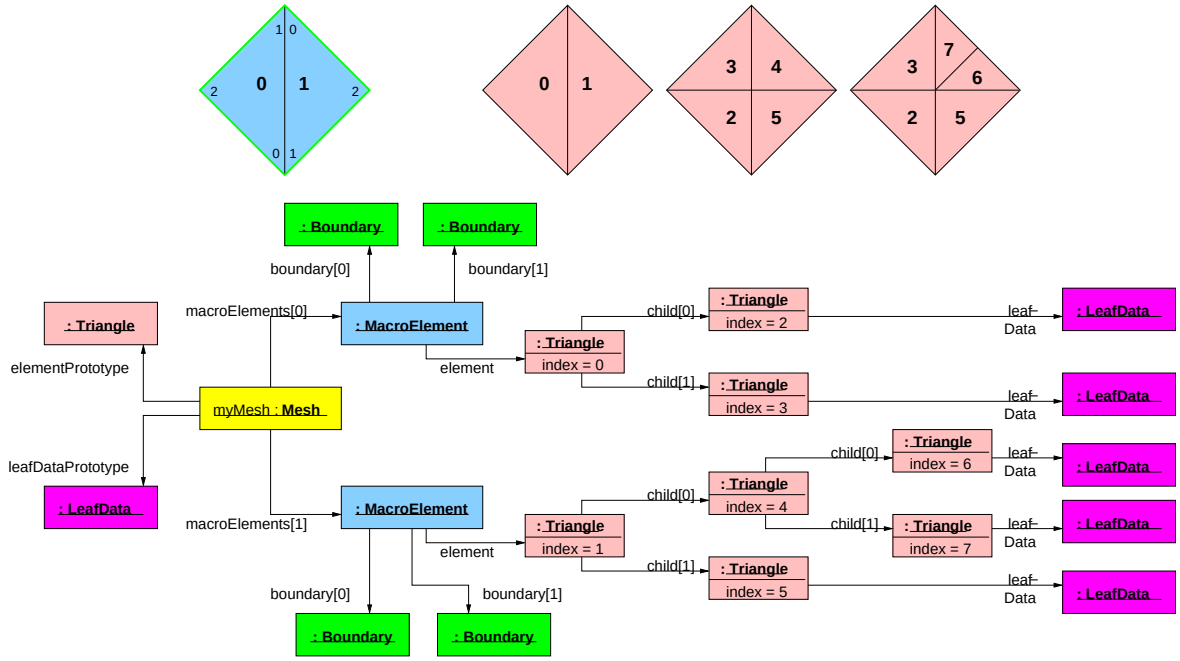

Fig.5: Instance diagram of a refined two dimensional mesh

MacroElements which represents the macro triangulation, and each macro element stores the needed element information, including a pointer to the coresponding element and a **Boundary** object for each side of the macro element that lies at a boundary.

To illustrate these concepts, Figure 5 shows an instance diagram of a simple two dimensional mesh with some local refinement.

2.4 Adaption module

Adaptivity optimizes the discretisation of the problem by heuristic arguments. The discretization can be adapted in different ways. One possibility is to refine elements on which a high error is estimated (*h*-method), another method is to increase the dimension of local function spaces used for the approximation (*p*-method), or a combination of both methods (*h*-*p*-method). In AMDiS adaption is realized only by element refinement at the moment, but implementations of *p*- and *h*-*p*-methods are likely to be included in future versions.

In AMDiS the refinement and coarsening algorithms are realized by the *visitor pattern*. This pattern encapsulates an operation that has to be applied to many objects of an data structure in an own object: the visitor. This enables a clean seperation between data and operations and leads to a clearer and more flexibile software design. In our case the data structure is the mesh containing the elements as its objects and the visitors are the **RefinementManager** and the **CoarseningManager**.

To solve a stationary problem, usually it is not enough to adapt the mesh once to reach the error tolerance. The class **AdaptStationary** contains the whole adaption loop for stationary problems, including the assemblage of the linear system, solving this system, estimating the local errors, and finally marking and refining the elements.

Refinement algorithm: In AMDiS the mesh refinement is done by bisection of its simplicial elements, by cutting the element at the midpoint of a given refinement edge in two child elements. Every simplex can only be refined at its specified refinement edge, to avoid irregular elements. A local vertex numbering enables the determination of the refinement edge: its always the edge that connects the local vertices 0 and 1. In three dimensions there exist three element types which differ by the vertex numbering of their children. The element type of a child element can be determined by the formula (parent's type + 1) modulo 3.

In order to avoid hanging nodes, neighbour elements that share the refinement edge with the bisected element have to be refined, too. So, an edge can only be refined, if it is the refinement edge for all neighbours sharing this edge. If elements in the refinement patch don't satisfy this condition, they are refined recursively, until the original refinement can be done.

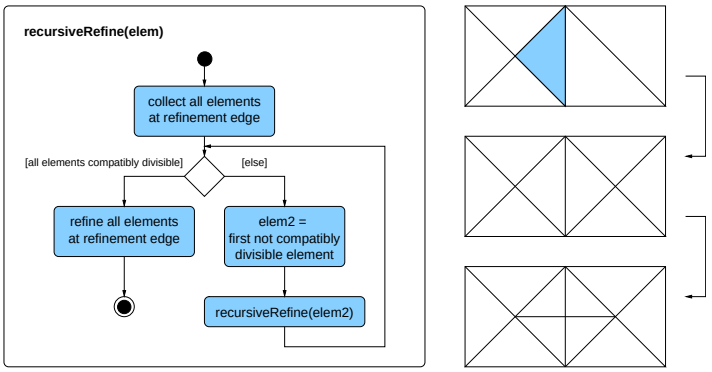

Fig.6: Recursive refinement algorithm and an example in 2d

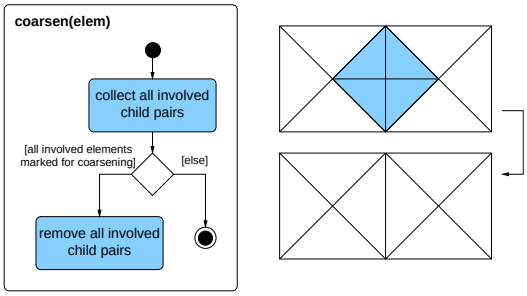

Fig.7: The coarsening algorithm and an example in 2d

In Figure 6 the recursive refinement algorithm and an example in 2d is given. In this example we assume, that all triangles have the longest edge as their refinement edge. Concerning the termination of this algorithm we refer to [7]. The refinement algorithms for the different dimensions are implemented in the dimension specific sub classes of **RefinementManager**.

Coarsening algorithm: An element marked for coarsening, only is coarsened, if all other involved elements are marked for coarsening, too, because otherwise coarsening could lead to hanging nodes in the triangulation. So again all elements laying at the coarsening edge must be collected to the coarsening patch, and only if all these elements are marked for coarsening, the coarsening operation will be performed.

In contrast to the refinement algorithm, here no recursive propagation can appear, because the neighboured parent elements must have the same refinement edge. In Figure 7 the coarsening algorithm and an example in two dimensions are illustrated. As you see in the example, the coarsened mesh doesn't conform to the original mesh before the refinement operation in Figure 6. Because coarsening is done by removing child elements, a macro element never can be coarsened. The coarsening algorithm is implemented in the **CoarseningManager** and its dimension specific sub classes.

3 Application

For real applications in materials science implemented in AMDiS we refer to [8]. Here to illustrate the functionality of AMDiS we solve a simple Poisson equation with a singularity:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= g \quad \text{on } \partial\Omega \end{aligned}$$

with

$$\begin{aligned} \Omega &= [0,1] \times [0,1], \\ f &= -(400|x|^2 - 20d)\exp(-10|x|^2), \\ g &= \exp(-10|x|^2) \end{aligned}$$

and dimension $d = 1$, 2 or 3. For this problem an analytical solution exists $u(x) = \exp(-10|x|^2)$. The implementation of this problem looks as:

```
Problem<double> ellipt("ellipt");
ellipt.initialize();
ellipt.setRHSFunction(NEW F);
ellipt.setBoundaryFunction(NEW G);
ellipt.setEstimator(NEW ElliptEst);
```

adaptive	degree 1	degree 2	degree 3
2d	3,074 dofs	557 dofs	397 dofs
3d	3,855 dofs	3,326 dofs	1,915 dofs

global	degree 1	degree 2	degree 3
2d	8,321 dofs	2,113 dofs	313 dofs
3d	274,625 dofs	35,937 dofs	15,625 dofs

Table 1: Number of needed degrees of freedom for the different dimensions and Lagrange degrees (adaptive and global refinement)

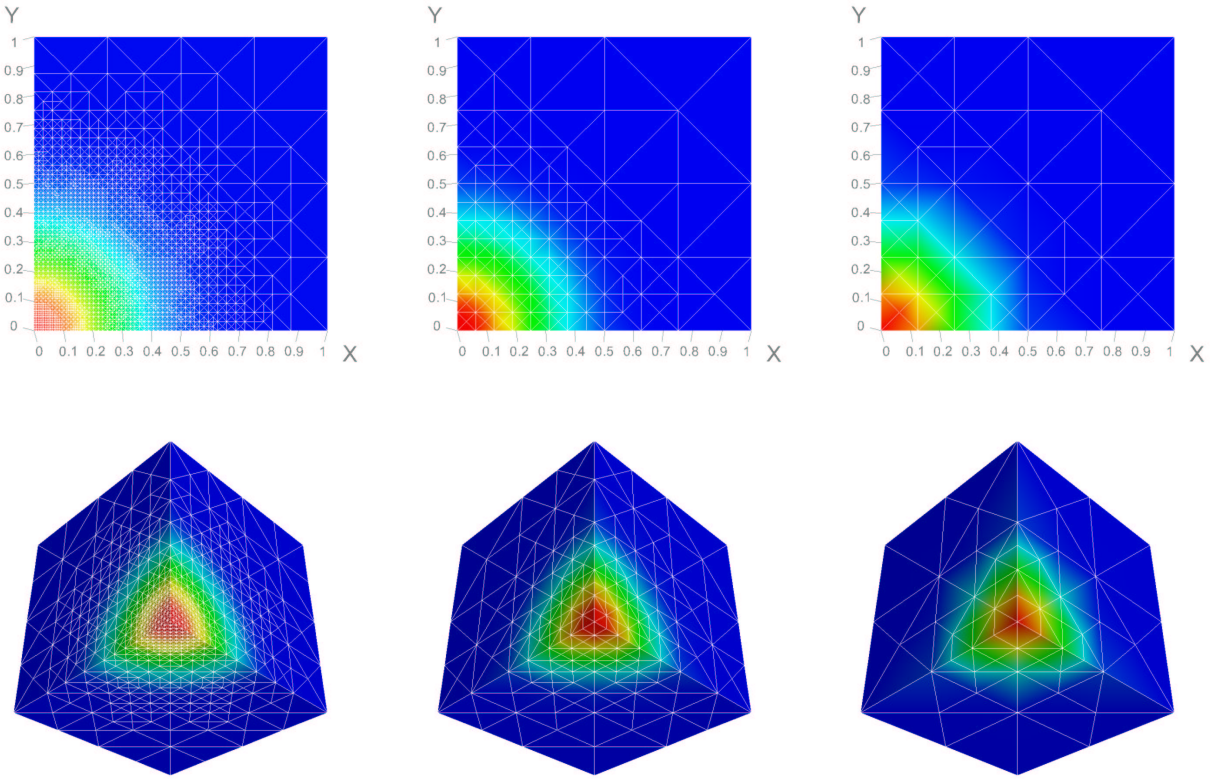

Fig.8: Solutions and resulting grids for Lagrange basis functions of degree 1-3 (from left to right).

```
Operator matrixOperator(
  Operator::MATRIX_OPERATOR, ellipt.getFESpace()
);
matrixOperator.addSecondOrderTerm(new Laplace);
ellipt.addMatrixOperator(&matrixOperator);
Operator rhsOperator(
  Operator::VECTOR_OPERATOR, ellipt.getFESpace()
);
rhsOperator.addZeroOrderTerm(
  new CoordsAtQP(ellipt.getRHSFunction(), degree)
);
ellipt.addVectorOperator(&rhsOperator);
ellipt.adaptMethodStat();
```

For a specified tolerance on the error between the true and the finite element solution $\|u - u_h\|_{H^1}$ Lagrange elements of degree 1, 2 and 3 are used for the simulation. This as well as the dimension is specified in a parameter file. We compare the degrees of freedom needed by global and adaptive refinement of the grid starting from a common macro triangulation. The solution and the corresponding grids are shown in Figure 8 for 2 and 3 dimensions, whereas the number of degrees of freedom are compared in Table 1. The tolerance in the 3 dimensional simulations is reduced by a factor of 10 in order to be able to solve the problem on a standard workstation if global refinement is used.

References:

[1] I. Babuška and W. Rheinboldt, Error estimates for adaptive finite element computations, *SIAM J. Numer. Anal.*, Vol.15, 1978, pp. 736–754.

[2] A. Schmidt and K. G. Siebert, ALBERT - Software for Scientific Computations and Applications, *Acta Math. Univ. Comenianae*, Vol.70, 2001, pp. 105–122.

[3] M. Fowler and K. Scott, *UML distilled - A brief guide to the standard object modeling language*, Addison-Wesley, 1999, second edn.

[4] B. Stroustrup, *The C++ programming language*, Addison-Wesley, 1991, second edn.

[5] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns*, Addison-Wesley, 1995

[6] W. Hackbusch and S. Sauter, Adaptive Composite Finite Elements for the Solution of PDEs Containing non-uniformly distributed Micro-Scales, *Mat. Model.*, Vol.8(9), 1996, pp. 31–43.

[7] I. Kossaczký, A recursive approach to local mesh refinement in two and three dimensions, *J. Comput. Appl. Math.*, Vol.55, 1994, pp. 275–288.

[8] `http://www.caesar.de/cg/AMDiS`