Data Abstraction Techniques for Multilevel Algorithms

U. Rüde
Institut für Informatik
Technische Universität München
Arcisstr. 21, D-8000 München 2
Germany
e-mail: ruede@informatik.tu-muenchen.de

Prepared for the proceedings of the
GAMM Seminar on Multigrid Methods
21.9.92 – 25.9.92 in Gosen

November 16, 1992

Abstract

Multilevel methods are fast and efficient solvers for a wide range of technical and scientific applications. Their structural complexity makes the construction of powerful multilevel based software difficult. Conventional software engineering concepts do not provide a sufficient basis for the implementation of general, fast, and robust multilevel applications. In particular, there is a severe tradeoff between the generality of such software and its efficiency. These problems can be alleviated on the basis of a consequent data abstraction. To equally satisfy the demands for generality and efficiency it is necessary to introduce a two level software model based on a generation and an execution phase. Suitable implementation techniques are discussed.

1 The current status of multilevel software

Multilevel algorithms have been the objective of intensive research for the past two decades. Though their superiority for solving partial differential equations
has been demonstrated mathematically and in practical situations, they have not yet become popular among non-specialists. Partly, this can be explained by the non-trivial mathematical properties of the multilevel concept. However, the mathematics of many numerical methods is demanding, and still they are used by practitioners in the form of high quality numerical libraries. Good numerical libraries may be used with only a minimal understanding of the methods they implement. Several such libraries are available as commercial or public domain programs. They encapsulate the expertise of specialists and make it usable by non-specialists.

For multilevel methods this is different. Presently, no sufficiently general, robust and efficient multigrid codes exist. Though several multigrid programs are available publicly, they have not yet reached the level of popularity as other numerical codes, like linear algebra routines or packages for the solution of ordinary differential equations.

Most of the multilevel software falls in two classes. They either focus on simple model problems, like Poisson's equation in a rectangle, or they are integrated packages for a special application field, like flow simulations in two dimensions. The first kind is usually characterized by high efficiency, but a restricted application scope. Typical representatives are the MG00 and MG01 programs (see Stüben [19]), that implement a second order elliptic solver on rectangles and a Helmholtz solver on general domains, respectively.

The second type of multigrid algorithm is applicable to a wider, but still limited problem class, and is usually much less efficient, because of a large organizational overhead. Representatives of this type are PLTMG (see Bank [3]) or KASKADE (see Deuflhard, Leinen, and Yserentant [4] and Roitzsch [11]). These codes solve scalar elliptic equations in general domains including error estimates and automatic adaptive mesh refinement. The package LiSS (see Stüben [19]) solves nonlinear elliptic systems, in particular the Navier Stokes Equations, using piecewise quasi-uniform (so called block structured) grids.

Both types of multigrid codes have in common that they are difficult to extend. The available codes are of little help if they do not directly support the problem to be solved. At this point, any researcher trying to use multigrid on a non-standard problem is forced to develop the algorithm from scratch. Here, the difficulty to understand and program multilevel methods becomes a bottleneck in the advancement of practical multilevel applications. The initial investment to replace a conventional method with a new multigrid method is prohibitive in many projects, where research into multilevel methods is not an objective by itself. In practice, suboptimal, conventional methods are regularly preferred over a multilevel approach, because the conventional methods are more effective in terms of development cost and time.

A closer analysis reveals a structural disadvantage of multilevel methods. Multilevel applications cannot be modularized "vertically" along the classical scheme.
The situation outlined in Section 1 is unsatisfactory. These difficulties are realized by many developers. Some problems can be alleviated by classical modularization techniques, e.g., by using parallel base libraries to make a code portable to various parallel machines, see Hempel and Ritzdorf [6]. However, the general problems caused by the inherent complexity of multigrid in combination with insufficient software design techniques remains. In the following we will try to analyze some of the difficulties.

2 Design aspects of multilevel software

In current programming practice the storage layout of the mesh data structures is fixed globally, and is explicitly used throughout the program. In a FORTRAN based programming style, even the memory management, using global work arrays must be explicitly addressed in all program modules. This alone makes the implementation of multi-dimensional adaptive codes difficult.
But even programs implemented with languages supporting dynamic memory management, like KASKADE in its Pascal version (see Leinen [9]) or C version (see Roitzsch [11]) share the problem that a specific implementation decision is fixed globally and is being used all over the program. Though this is natural according to a conventional programming method, it is a severe violation of modular design from a more abstract viewpoint. In Rüde [12] it is argued that the multilevel mesh structure can be seen as an abstract data type with only a narrow interface to the algorithmic structures built on it. These abstract operations include "access all nodes of a mesh" or "access all neighbors of a node"). Among others, these operations must be provided by all mesh implementations, and provide functionality that is independent of how the mesh is implemented.

However, in current programming practice, multilevel applications are always programmed around a fixed mesh implementation. In many simple codes, the mesh is represented by a two-dimensional array. Here the access to all nodes must be implemented as a double nested loop. Further examples are the tree-like representation of a mesh hierarchy as used in KASKADE, and the piecewise quasi-uniform grids that are used in LiSS. At the development state the packages have reached it is prohibitively difficult to modify or extend the basic mesh structure they are built on.

Independent of the mesh structures, however, the components of a multilevel algorithm typically have the abstract form

\[
\begin{align*}
\text{For all nodes of the mesh} & \\
\text{For all neighbors of the node} & \\
\text{compute some local operation} &
\end{align*}
\]

In many current implementations the abstract operations like "for all nodes” are not presented as an interface of the mesh module, but the implementation forces the program to use the concrete mesh structure throughout the program.

Of course, program extensions will also affect the mathematical foundation of the algorithm. The extension of a linear code to a nonlinear one can not be understood as only an adaptation of the data structure. However, such an extension will be easier to implement, if the base program is well modularized so that the necessary extensions will be confined to few well-defined modules. This, however, is not the case in present multigrid codes. A change in the mesh organization does have global impact on all the program.

Consider the transition form a two dimensional to a three dimensional algorithm as an example. In three dimensions the mesh generation will have to be different, and therefore a modification of the mesh management module will be necessary. On the other hand, the number of dimensions does not affect the
abstract formulation of a multilevel algorithm in any way. Still, current implementations do depend on the dimensionality of the problem in such an essential way, that a modification amounts to a complete redesign of the solver. This is in no way caused by mathematical difficulties but only by how the algorithms are being programmed.

From the point of software architecture these difficulties are caused by programming techniques that violate basic software engineering concepts, like modularity and the locality of code. The decision, whether the data is stored in two- or three-dimensional arrays or in a linked list should in no way be visible throughout the solver, but be isolated in a mesh management module. The mesh management module in turn must not depend on the solver. Ideally, then a change in the mesh structure, in particular, the transition from two to three dimensions, should be performed by only extending the mesh management module, while the solver remains unchanged.

Then the access to the mesh is not programmed directly, but through abstract operations, whose implementation changes when the mesh structure is modified. Practical suggestions, how this can be accomplished are being discussed in Section 4.2.

2.2 Numerical data

Similarly, the type and number of independent variables should not be visible throughout the program. Though of course the numerical manipulations must access this information, it should not in any way be visible in the mesh manipulation and mesh access routines. Changes, like an additional dependent variable should not effect the overall program organization. Consequently, an extension must be traced throughout the program requiring not only the (necessary) understanding of how the mathematics change but also of how the current program is implemented.

This can be accomplished by a simple concept of higher programming languages. Records (structures) can be used to encapsulate the independent variables. Instead of directly using arrays of reals to represent the solution, we define a data type NodeVar (as a record) and use a collection of NodeVars to represent an approximation to the solution. In the mesh manipulation it may now be necessary to manipulate the nodal variables, but with the abstraction of the NodeVar data type the operations can be made independent of this data type.

2.3 Software design concepts

The program design suggested above has been motivated by modularity and program locality considerations. A main criteria for a good software architecture is that pieces of the program should be grouped according to their functional
correlation as opposing the *dynamical* correlation dictated by the sequence how the operations are executed.\(^1\)

In a well designed program, the *multiplicity* of code fragments is avoided. Identical program fragments must not occur multiply throughout the program, but must be isolated in subroutines or modules. Besides the effect of reducing the code length, this simplifies later program extensions and modifications. If a program modification is required, just one instance of the program fragment must be found and modified.

In Section 2.1 we have discussed these aspects for the special cases of multilevel mesh structures and their access interface. Doubly nested loops are the appropriate access structure for two-dimensional arrays, but their explicit use leads to a poor program design, because it makes many algorithmic components *artificially* dependent on a particular design decision. If the solver must be converted to three dimensions, or to non-rectangular meshes, it means that it must be completely rewritten, though the logic of the solver remains unchanged. This applies to all partial differential equation software, but the negative effect is particularly dramatic for multilevel applications that cannot use the "vertical modularity". We will therefore look for programming techniques that allow to modularize such structures. This will be discussed in detail in Section 4.2.

### 3 Generality versus efficiency

The discussion in Section 2 has lead to requirements on the software architecture. Modules must not be tailored too specifically for a special problem, but must be kept general, independent of details that are specified in other modules. The basic idea is to make the program system flexible, general, and extensible by defining the software components so that they can be understood and modified in isolation, without studying the details of other components.

Many of the current developments of multilevel software have a different and opposing objective. Codes like PLTMG, LiSS, FEMGP (see Jung [8]), and KASKADE attempt to provide *integrated environments*. This means that a set of components are bundled together. These systems contain a discretization mechanism, solver, visualization tools, and an interactive user interface. This is convenient for users whose problem falls within the class directly supported by the system, however, it is no advantage if the software can be only used as a basic framework that must extended to fit the requirements of a specific application.

To support a wide class of problems, integrated software must be written to

\(^1\)In some sense the purpose of higher programming languages can be interpreted as providing means to accomplish this criteria. The most popular mechanism to liberate the static program code from the dynamic execution sequence is the *subprogram technique*, that allows to group code in logically related portions and to provide a well defined mechanism to relate this to the dynamic execution sequence.
be generally applicable. For example, the codes PLTMG or KASKADE provide full adaptivity on unstructured meshes. The resulting complexity in the data structures imposes a severe performance penalty in storage and CPU time. Our experience with such software suggests that the overhead compared to uniform rectangular meshes is about a factor of 10. Additionally, the closed software architecture makes it impossible to restructure one of these complex codes to be better suitable for simple situations.

Though there probably are (though this is not sure) problems requiring the flexibility of unstructured meshes, an unstructured mesh solver will clearly lead to a waste of computing resources if applied to problems either not requiring adaptivity, or requiring only limited adaptivity that could be accommodated by simpler data structures. Here the complicated data structures are inappropriate. The large penalty in the adaptive codes is unfortunate, but is a direct consequence of the attempt to construct a single program, powerful enough to accommodate a wide range of problems. The operations to maintain and manipulate the adaptive structure must always be performed, and cannot exploit situations when the resulting mesh is more regular. This is particularly unfortunate in applications, where information about the meshes is known a-priori. The overhead of the advanced codes is caused, because all the operations and decisions are made dynamically, at program run time, not statically when the program is compiled.

As an alternative, we envision a software architecture where besides providing the flexibility required for difficult situations, a-priori information can be reasonably exploited to recover the efficiency of simpler structures. Thus the computing system cannot be based on a single run-time system, but must have the capability to \textit{compile a-priori information into the system}. This a-priori information must be used to generate a solver with maximal efficiency at execution time. From an abstract perspective this corresponds to a \textit{partial evaluation} of problem data at a preprocessing (compilation) phase.

In contrast to the integrated and closed systems this also leads to an \textit{extensible and open} software architecture. It will consist of components and modules that may be combined and extended as required by the problem under study.

In this spirit, the overall solution process will be organized in two phases. To set up the system, the problem class must be specified so that a corresponding application program can be generated from the existing building blocks. New problems may additionally require the extension of modules or implementation of new modules. Using existing and possibly specially designed new modules the actual application can be compiled. It will implement the necessary functionality but will avoid unnecessary overhead.

This \textit{toolbox} approach has already been used and implemented in the Munich Multigrid Workbench [17], see also its recent evolution to an educational tool [see Arbesmeier and Rüde [2, 1]]. In contrast to the philosophy guiding the development of integrated systems, we envision a modular software architecture that presents itself as a collection of tools that can be combined freely.
to specialized applications, in particular also to build integrated systems.

4 Data abstraction and object oriented programming techniques

The above discussion has criticized current programming practice without yet providing concrete suggestions, how to implement the desired features. It is the purpose of this section to make suggestions and provide guidance how to realize them in practice. The discussion here is motivated by considerations how to design the FAMe package (see Rüde [13, 15, 14, 16]). In the FAMe package we attempt to provide a software framework supporting multilevel algorithms with adaptive mesh refinement, error estimation, and efficient solution. Though the discussion in this section is independent of a particular language, we will use a notation similar to the language C++ (see Stroustrup [18]).

In the following we will also analyze the performance effects of various programming techniques. Numerical programs do have special performance demands and the too naive usage of advanced programming methods may completely spoil the overall performance of a program.

4.1 Node data types

For the sake of simplicity we will for a moment assume that a mesh is represented by a logically rectangular grid. Classical multigrid programs would now represent a grid solution by one or more two-dimensional arrays of reals, depending on the number of node variables. However, additional arrays would be required to store the equation's right hand side, coefficients of the operator, and the location of the node, if required. Depending on the type of algorithms further arrays might be needed to store intermediate results.

Clearly this organization is not modular, because now simple operations, like copying the data associated with a node to an intermediate variable, or checking two nodes for equality become dependent on the particular problem and even on internal algorithmic details.

Based on the record concept (struct) in Pascal- and C-like languages a better structure is obvious. The variables representing a node are grouped in a struct, say called NodeVar. The data declaration for a grid would now be something like
typedef struct{
    Real x, y;
} Loc;

typedef struct{
    Real u, v, f, ...;
    Loc loc;
} NodeVar;

int n; read(n);
NodeVar* grid = new NodeVar [n][n];

The advantage is obvious: the data representing a node can now be copied, compared for equality, etc. without taking reference to its internal structure. This also makes the storage management of dynamically allocated grids independent of the data representing a node, see the dynamic allocation of the grid in the last line of the program fragment. Of course, operations that do depend on the number of node variables, like relaxation, will depend on the internal representation of the NodeVar data type, however, many modules will operate on NodeVar objects without looking at its internals.

Most likely, the NodeVar data type should itself be defined using user-defined types. If e.g. the location of a node is explicitly stored, it should be done by using another data type Loc that in turn consists of two Reals. Even, if an extension to three dimensions is never planned, it is an advantage to set a node a to the same location as node b with a single statement a.loc = b.loc and not to have to worry about the details.

It might be considered to represent a node instead of by a struct by an array of suitable length, depending on the components necessary. The grid itself could then be presented by an array with three dimensions, like

```
const int uval = 1, vval = 2, xloc = 3, yloc = 4, ...;
const int maxcomp = ...;
Real g[n][n][maxcomp]; // Declaration of mesh

... g[i][j][uval] ... // example access to u-component
```

Copying a single node will now be performed by a loop from 1 to maxcomp.\(^2\) This technique has the advantage that it can be used in FORTRAN, but it has

\(^2\)This may present significant overhead. For the NodeVar data type the compiler can easily
the disadvantage that it cannot be used if a node needs non-real components, like pointers to its neighbors.

Possible performance differences in the schemes will result from different internal memory layouts. The array of NodeVars will usually have each NodeVar allocated in a contiguous block of memory. Depending on how the data in a grid is accessed, this may have advantages compared to a storage layout where all the like components are stored in contiguous memory. A good design must consider cache locality and the effects of cache read-aheads and the effect of large strides that may lead to memory bank conflicts in highly interleaved memory. Unfortunately these aspects depend much on the particular machine, so that no general guidelines can be given.

4.2 Abstract mesh structures

In the previous section we have presented our discussion of node data types based on the notion of the grid is simply an array of nodes. In this section we will consider a grid as a abstract container\(^3\) of nodes. In this abstraction we will now discuss operations that a mesh module must provide, and how they can be implemented, so that other software components may stay independent of the concrete realization.

Clearly, once we have left the simple idea of a mesh just being a rectangular scheme of nodes, the construction of the mesh itself becomes non-trivial, and the abstract mesh data type must provide operations how it can be generated. We will not discuss general geometric questions of mesh generation here, but will restrict our attention to the operations that are performed when the mesh has already been set up. The most obvious operation a mesh must provide is the access to the nodes it contains.

4.2.1 Iterator subroutines

One solution is to use subroutines to encapsulate the mesh access functionality. Sweeps over the grids are programmed as iterator routines. An iterator is a subroutine taking a mesh and a procedure as a parameter. The abstract function for-all-nodes is thus parametrized by the operation that must be performed for the nodes. When called, the iterator routine loops through the mesh and applies the operation to all components of the mesh. This concept is being used in the BASIS data structure, see Hemker, Maarel and Everaars [5].

determine effective code for for the operations, because it statically knows how much memory a NodeVar needs. In the case a loop is programmed explicitly, optimization is more difficult for the compiler.

\(^3\)The abstraction of array, sets, stacks, queues, etc. into containers is common in object oriented programming.
With this programming technique we are forced to implement all operations as separate procedures. This also leads to a performance penalty, because each operation costs an additional function call. Here optimization techniques, like function inlining, are not possible. This is only tolerable in complicated situations, where the operations are expensive such that a function call is comparatively cheap. However, we must expect a severe performance penalty if the elementary operations are only moderately expensive.

This technique does not permit generalizations. For example we cannot form a double loop over the nodes without writing a special access routine. Furthermore, the operation must have a fixed functionality so that it is impossible to pass the doIt routine an additional parameter. Similarly it is awkward to use this mechanism to collect global data, like a norm of a grid function. This can only be accomplished by side effects in the doIt routine.

4.2.2 Macro expansion

The most direct approach to modularizing the access structures is to view the program constructs for looping as just a textual scheme. Macro preprocessors can be used to generate the necessary source code. As discussed in Jänsch, Schnepper, and Rüde [7] macro processors are suitable tools for moderately complex program generation tasks.

This has been used with a prototype version of the FAMe package. Based on the standard macro processor $m4$ macros like \texttt{FOR\_ALL\_NODES}(g, u) and \texttt{END\_ALL\_NODES} were implemented to provide an abstract access mechanism to

```c
void for_all_nodes(Mesh g, void (*doIt)( NodeVar* )){
    :
}

void relax(NodeVar* x){
    :
}

Mesh g( ... )

main(){
    :
    for_all_nodes(g, relax)
    :
}
```
the mesh. For the two-dimensional arrays these macros were defined to expand to doubly nested loops. Later in the program development process they were changed to represent the appropriate iteration through an unstructured mesh.

The macro technique is simple and leads to efficient code, if the macros are designed carefully. It may be seen as a disadvantage that the resulting source code looks unconventional and may be difficult to understand. Furthermore, the technique has the problem that errors may be difficult to trace to its sources. The macro processor provides no syntax checking mechanisms. Therefore syntactic errors are only detected by the compiler after the preprocessor. These problems can be alleviated by using appropriate tools.\footnote{This has been the main critique, when this programming technique was discussed in the usenet.}

4.2.3 Abstract iterator classes

Abstract iterator classes have been suggested in \textcite{Rude1995} as an effective tool to provide modular access to multilevel mesh structures. The idea is that an abstract mesh data type is accompanied by an associated \textit{iterator} data type. The iterator is an abstraction of the index to an array or the number of a node in a finite element mesh. For multilevel processing, an iterator must primarily provide the following operations.

**Initialization.** The iterator is set to the \textit{first} node in the mesh.

**Increment.** The iterator is set to the \textit{next} node in the mesh.

**Access.** The iterator is used to access the node in the mesh.

**Termination.** The iterator is checked whether the final node in the mesh has been reached.

This functionality could be provided by defining a record (struct) to hold the data for an iterator and a set of subroutines corresponding to the above operations. Besides the overhead this leads to an awkward syntax. Advanced languages like C++ allow to make the interface syntactically nicer and provide means to specify optimization by inline expansion of the operations. Thus the full efficiency of a direct implementation can be approached.

For the following example we assume that a mesh class \texttt{Mesh} and an iterator class \texttt{Iterator} has been implemented in C++. Independent of whether we deal with a two-dimensional array of nodes or a grid structure defined through a recursive tree, an operation traversing all nodes can now look like

\footnote{Good macro preprocessors generate \#line directives that can be used by the compiler to refer to the unexpanded source. Even source level debugging on the original source is possible.}

\begin{verbatim}
...
Mesh \texttt{g}\((\cdots)\); \quad // Declaration and initialization of a mesh

;  

// Now a loop with declaration, initialization,
// test, and advancement of an iterator:
\texttt{for( Iterator i(g); i<=last; i++)}  
\texttt{doIt( g[i] );}  
\texttt{// perform some operation,}  
\texttt{// note that g[i] is access}

Here the C++ facilities for operator overloading have been used to give the constructs \texttt{i++} and \texttt{g[i]} the meaning that is naturally expected if \texttt{i} is a simple index to an array. In general, however, \texttt{g[i]} will be converted to a more complicated operation. A good implementation of the package (and the compiler) will use \textit{inline expansion} for all operations so that no unnecessary overhead results.

The test \texttt{i<=last} is implemented by introducing a so-called \textit{enumeration} data type consisting of just three objects \texttt{first}, \texttt{cont}, and \texttt{last}. The test \texttt{i<=last} will require a conversion from an \texttt{Iterator} object to this type. C++ provides means for specifying the semantics of such type conversions.

Clearly, the iterator concept in the presented form is just a core idea that needs to be further detailed. The iterators should support additional operations, like reverse iteration, iteration through subsets of the nodes (like red-black), and others. Typically, however, this would not be accomplished by providing additional operations for a single iterator class, but by providing additional iterator classes. In C++ this is simplified by the so called \textit{sub-classing} or \textit{inheritance} mechanism. For example, we may introduce an iterator subclass \texttt{BoundaryIterator} that can be used to access all boundary nodes. A loop through all boundary nodes could then look just like the above loop, except that the iterator would be generated to be of type \texttt{BoundaryIterator}.

We wish to point out that the abstract iterator concept can be used to simplify parallel and distributed programming. To accomplish parallel execution we introduce \textit{multi-threaded} iterators. In a data-parallel program such a parallel iterator corresponds to a vector of regular iterators each of which is the private loop counter local to one of the processors. Loop initialization, and the counter increment must be implemented to have the appropriate semantics. This is an elegant way to implement the parallelization of loops.

4.2.4 Performance comparison

One of our main concerns is the performance effect of the programming constructs that we suggest. In Section 3 we have discussed the performance penal-
ties associated with trying to make a program general, if the generality is accomplished at run-time.

The abstract mesh concept must be carefully analyzed what kind of overhead it produces. Clearly, the final overhead will depend on the particular mesh structure. The abstract mesh and iterator concept will not automatically make the operations on fully unstructured meshes more efficient. The idea is to provide a mechanism so that such complicated structures can be exchanged for simpler ones to obtain better efficiency. This, however, does only make sense if the abstraction mechanism itself does not generate too much overhead. In a concrete application the abstract operation $g(i)$ may be represented by the concrete operation $g[i]$, if the iterator is implemented as a simple integer index and the mesh is no more than a one-dimensional array. Here the conversion of the abstract object types to the concrete ones should not cost significant overhead. This is the issue addressed in this section.

In the previous subsections we have suggested three different techniques to implement abstract mesh structures. Clearly, the least performance penalty can be expected from the macro technique. The macros are converted to a conventional program before the program is compiled. The macros can even be used to provide the compiler with e.g. vectorization directives, if desired. Thus we may expect the same performance as for a hand-coded algorithm.

The second technique has used iterator subroutines. Here, dramatic performance losses may result. The essence of this technique is the encapsulation of all elementary operations in subroutines, costing substantial overhead by itself. Additionally, many loop optimization techniques depend on loops that do not have side effects. Compilers therefore cannot fully optimize loops with a function call as they result with this programming technique.

The concept of abstract iterators depends strongly on how well it is supported by the language. If the operations must be implemented with regular functions we can again expect severe performance problems. For iterator class operations, inline expansion is possible, and thus, depending on how sophisticated the system is, we can expect little or no performance loss.

An experiment has shown that iterator classes in C++ produce about 15% overhead compared to a direct coding of the loops, while the iterator subroutines have a performance penalty of more than 100%. These figures strongly depend on the compiler and on the machine used.

5 Conclusions

We have tried to analyze some of the common difficulties in the design and implementation of multilevel software. A more fundamental modularization can help to design better programs in the future. General and efficient codes will require advanced programming techniques, as they are available with the concepts of abstract data types and object oriented programming. The potential
of this new software design method has been discussed for modularizing the mesh data structure as an example. This is just one of many steps, though an especially instructive one. These concepts require advanced programming languages like C++.

Software that over-uses dynamic evaluation leads to severe performance problems. Adaptive codes use complicated run-time structures that produce unacceptable overhead in applications where these structures are not required. We have outlined how the abstract mesh data structure can be used to avoid this problem by a two-phase software architecture. In a generation phase the program is compiled from a specification. In the second phase the resulting application program is executed. This model allows to design a program system that provides maximal flexibility and can obtain full efficiency.

The abstraction of the mesh structure is just one aspect of a good design of multilevel methods. For example, the access to the neighbors of a node, as needed for relaxation, must be similarly formulated in an abstract form. One possibility is to introduce generalized difference stars. The NodeVar data structure itself can benefit from a still more fundamental abstraction. In a multilevel structure various different subclasses of the nodes are needed, that will store different information. For more information see Rüde [12], where it is shown, how the inheritance mechanism and so-called virtual functions can help to implement the necessary functionality.

We wish to point out that none of the software engineering techniques presented above is new per se. Such modularization techniques are consequences of an abstraction that is more or less natural if the software development is liberated from the restrictions of the presently used languages.

References


