VISTA Status Report
December 1994

W. Bohmayr, F. Fasching, G. Rieger,
S. Selberherr, T. Simlinger

Institute for Microelectronics
Technical University Vienna
Gusshausstrasse 27-29
A-1040 Vienna, Austria

Contents

1 VOOPS — The Vienna Object-Oriented Programming System

1.1 Why Object-Oriented Programming?

The term *object-oriented* has been a catchword gaining more and more importance in recent years [Betz89, Boni91, Budd91, Dani91, Elli90, Flor91, Rumb91, Tell89, Wens91, Wien88]. Rarely, however, there has been a more misinterpreted term in computer science as well. This term has even gained a foothold in areas where object-orientedness in its genuine meaning makes no sense at all.

Looking at the term *object* detached from its computer-science meaning, we see that it is a very common word used in the real world for any-*thing* we can see and touch, be it a house, a table or a stone. Mapping this meaning back to the abstract computer-science world, any-*thing* (i.e. any opaque data set seen as a whole) could be termed an *object*. But this would be just another term for the programming paradigm of *data encapsulation*, i.e. grouping data items which belong together into a single aggregate which we then call *object*.

This is, however, *not* what *object-oriented programming* is supposed to be. The *object-oriented programming paradigm* encompasses the following programming paradigms:

data encapsulation Like stated above, this is the grouping of semantically interdependent data items (*members*) into an aggregate (*class*) seen as a single data item. This is termed *class definition* in object-oriented languages. Although this feature is available in all procedural third-generation programming languages (like a `record` in PASCAL or `struct` in C), an object-oriented language has to extend this paradigm with a

taggedblock storage concept Each of the different aggregate data types (blocks) can be identified at run-time through a unique identifier (tag) given to and stored in that aggregate type. This enables the dynamic identification of encapsulated data at run-time, which is a big advantage over statically typed languages like C or PASCAL. Although not object-oriented *per se*, Common LISP's (see [Wins89, Stee90, Fran88]) `DEFSTRUCT` command implements this concept (there exists a separate object system extension for Common LISP, [Bobr88]).

function encapsulation Every self-contained idea is consistently coded into a sequence of statements grouped together forming a function (*method*). While this feature is available in any third generation language, traditional procedural languages do not intrinsically associate a function with an encapsulated data aggregate. In object-oriented languages however, there *must* be a

data-function association This association of a method with a class together with the method code makes up the *method definition*. We can say, the method is *defined on* or *belongs to* a specific class.

The association of data sets (i.e. classes) with the corresponding functions working on a particular data set (i.e. methods) is an intrinsic advantage of object-oriented programming languages over traditional procedural languages.

inheritance The most distinct features of object-oriented programming languages are *member inheritance* and *method inheritance*. Member inheritance means that a class inherits all members of its *parent class*. This vaguely resembles the inclusion of an already declared `structure` in a new `structure` declaration in C, although in this case the "inherited" members are not opaquely accessible because they are hidden in the included `structure` declaration, whereas in an object-oriented programming language inherited members are in general indistinguishable from intrinsic class members. Method inheritance is implicitly achieved through declaring the parent of a class,

since all (public) methods of the parent class become available to the inheriting class. Again, inherited methods are indistinguishable from intrinsic methods defined on the class. This could be emulated in C through macro-aliasing a new function name to an existing function and applying the function to the included `structure` as mentioned above.

As we can see, all object-oriented programming paradigms can be implemented *in principle* with third generation procedural programming languages. However, considerable expense has to be made to emulate those OOP paradigms. Therefore it is a tedious and error-prone task to implement object-oriented features directly in those languages without proper support.

1.2 Drawbacks

Due to the recent enthusiasm about object-orientation in general and object-oriented languages in particular, some non-obvious but intrinsic drawbacks are often overlooked. First of all, it has to be noted that the problem to which object-oriented approaches should be applied, has itself to be object-oriented. Since an object is an encapsulated entity that has precisely known relations to other entities, an object-oriented approach to a problem which cannot be expressed as such an entity-relationship model will likely fail or at least exhibit severe performance drawbacks.

Another drawback is that arithmetic expressions involving method invocations cannot be simplified by a compiler, if the method to be invoked is not known at compile time. For example, an arithmetic expression calculating the volume of a hollow sphere will typically involve a subtraction of the inner sphere volume from the outer sphere volume. A FORTRAN compiler will simplify and optimize the resulting arithmetic expression resulting in a significant speedup compared to a C++ compiler which will have to invoke the method for computing a sphere volume two times and then perform the subtraction without having an opportunity to optimize the resulting arithmetic expression. Keeping this instrinsic drawback in mind, an object-oriented approach to solving numerical problems at least seems to be problematic and is definitely a non-optimal solution for this class of problems.

Memory management is another issue of concern when object-oriented approaches are considered. Objects are usually allocated separately, and when many (especially small) objects are allocated, the main memory becomes cluttered with those small allocated structures. The drawback is, that on traversion of, e.g., a tree of those small objects many different memory locations have to be addressed and accessed by the program, and performance will probably suffer severly if memory pages then have to be swapped in and out of main memory by the operating system. A traversion of, e.g., a FORTRAN array which is hold in one contiguous block in main memory just requires incrementation of an index variable and does not induce any operating system overhead at all.

Determining the function which has to be called at run time occurs frequently during execution of a program written in an object-oriented language, when the appropriate method to be applied to a given object has to be found by the run time inheritance mechanism. The performance of the resulting program will heavily depend on the efficiency of the inheritance algorithm which has to search the class hierarchy for the appropriate method, but will definitely be inferior compared to procedural languages like FORTRAN or C, where the functions called are already known at compile time and therefore are invoked directly.

1.3 Comparison to Object-Oriented Languages

Although all object-oriented languages implement the basic programming paradigms stated in Section 1.1, they differ in extensions of and the semantics implied by those paradigms. VOOPS does not implement any extensions to the basic programming paradigms, since it was designed to be simple to use, easy to comprehend and compact in its design.

For example, it is not possible in VOOPS to declare some members of a class as private to this class, meaning that they can not be inherited by other classes derived from this class. C++ provides the `private` keyword to declare private members in class definitions, allowing control of the scope of member declarations.

Concerning method definitions, methods defined on a VOOPS class always override a parent method having the same name, if such a method exists in the parent class. In C++ methods which can be overridden must be explicitely prefixed with a `virtual` keyword which provides tight control of method overriding in class definitions.

Multiple inheritance means that a class may have more than one parent. VOOPS does not provide this feature, since it is complicated to implement within the simple concept employed by VOOPS. Mostly the need for multiple inheritance does not arise, when the class hierarchy is designed carefully. However, most modern object-oriented languages like Eiffel [Meye92] and recent C++ versions provide multiple inheritance.

In contrast to Eiffel and Smalltalk [Gold83, Pins88], VOOPS does not define a single root class, from which all other classes would automatically inherit. A single root class would allow the definition of generic methods applicable to all other classes. VOOPS rather considers each class which does not include a reference to a parent class as a root class.

1.4 Architecture of VOOPS

VOOPS' main purpose is to relieve a programmer from the work of coding object-oriented features manually, and to explicitly support them through specialized meta-language constructs in LISP via UNFUG, which implement those features using native host language C constructs. This approach is also used by some C++ compilers which behave like a sophisticated preprocessor and translate C++ code into C source code [Hewl90]. Although the intermediate C code can be output optionally, it is discarded by default and directly compiled to an object file with a native C compiler, thus effectively pretending the behavior of a C++ compiler.

To reflect the concept of classes and methods in C, some preconsiderations have to be made concerning the representation of classes, methods, and objects in C.

First of all, it has to be decided whether the class hierarchy should be *static* (i.e. defined and "cast in concrete" at compile time), or *dynamic* (i.e. extensible through new class and method definitions at run time). Since the PIF object model is defined in the PIF syntax and it is impossible that new PIF object classes could be necessary at run time, there is no need to strive for a dynamic class model, although there exist even public domain solutions for dynamic object-oriented programming extensions for C (e.g. the COOL library by L. NORSKOG). As we will see below, the static class model has the advantage that it can be implemented efficiently, thus allowing for a fast messaging concept.

When using a static class hierarchy, classes can be represented as C structures and methods as C functions in C header or source files, respectively. VOOPS uses UNFUG to evaluate special LISP expressions and insert the resulting string in place of the VOOPS statement into the C source code. Additionally, classes (and methods defined on them) are kept track of internally and stored in a save file at the end of a VOOPS run.

Classes and methods which belong together are grouped into a specific *module* which is referenced by a short mnemonic. All data structure, method, and macro names generated by VOOPS are prefixed with this module mnemonic to avoid possible naming conflicts within the VISTA system. This also enables the use of multiple class hierarchies in parallel without the danger of conflicting class and method definitions.

1.5 The VOOPS Class Model

VOOPS maps classes and member definitions onto C types, structures and unions. A class is translated into a structure definition, with the class members being structure members. *Accessor macros* are defined for each class member which allow access to class members through verbose names instead of complicated and not obviously comprehensible C structure member access and pointer dereference operators. All classes have a default object identifier member which is used to store a unique ID of the object, when it is newly created with the default `New` method.

Inheritance is effected through defining the members of the specified parent class in the currently defined class too. Since all these parent class members are also included as C structure members, one limitation becomes obvious: No two members can have the same name, if they appear in the same inheritance branch of the class tree, since this would produce a name conflict in the resulting C code.

On definition, a unique class name (a LISP symbol) has to be given to a class, which is used as an identifier on subsequent accesses to the class. An optional documentation string given in the class definition is translated into a documentation comment conforming to the VISTA documentation guidelines. Besides the documentation string, members can also have alias names which are aliased via a C macro to the original member accessor macro.

1.6 VOOPS Operation

VOOPS transforms template header files `*_h.tpl` into C header files `*.h`, and template source files `*_c.tpl` into C source files `*.c`. Additionally, the definition of a *master* template header and source file is necessary, where the class table declarations and definitions go into, which are explained in detail below. The relation of the various template header and source files is shown in Figure 1.

Class definitions are made in template header files, and method definitions go into template source files. The *master* header file additionally holds the C type declaration of the generated class table after running VOOPS, and the actual definition of this class table is written into the *master* source file. At the end of the master header file, the C `union` of all classes currently known to VOOPS is defined, which is **the** genuine object for a particular module. A pointer to this union is the universal object pointer to objects of the current module and is available in VOOPS statements through the `SELF` data type.

1.7 The Class Table

The C representation of the VOOPS save file is the class table which holds information about each class known to VOOPS. Besides the class ID, the parent class ID, the class name and structure size, all methods are stored as C function pointers in the class table. But since the number of methods is not known in advance, this table grows with each new unique method. This has the implication that when a new method is added in a `*_c.tpl` file, the master header file of the module has to be rebuilt, and therefore all other files depending on it.

A special feature is the implementation of *class masks*, which are used to check whether a specific method can be applied to a given object, or, in object-oriented terminology, if a specific message can be sent to the object in question. This is done through comparing the object's class mask (accessed through the class ID stored in the object) to the class mask of the class the method is defined on with a logical `AND` operation. If there is a match, the method can be applied, otherwise the object is rejected. Since this feature should be only necessary on debugging, the corresponding code is included in an `#if CHECK` – `#endif` pair.

The class table declaration is generated at the end of the master header file, whereas the class is actually defined at the beginning of the master source file. Many class-independent access macros are defined to

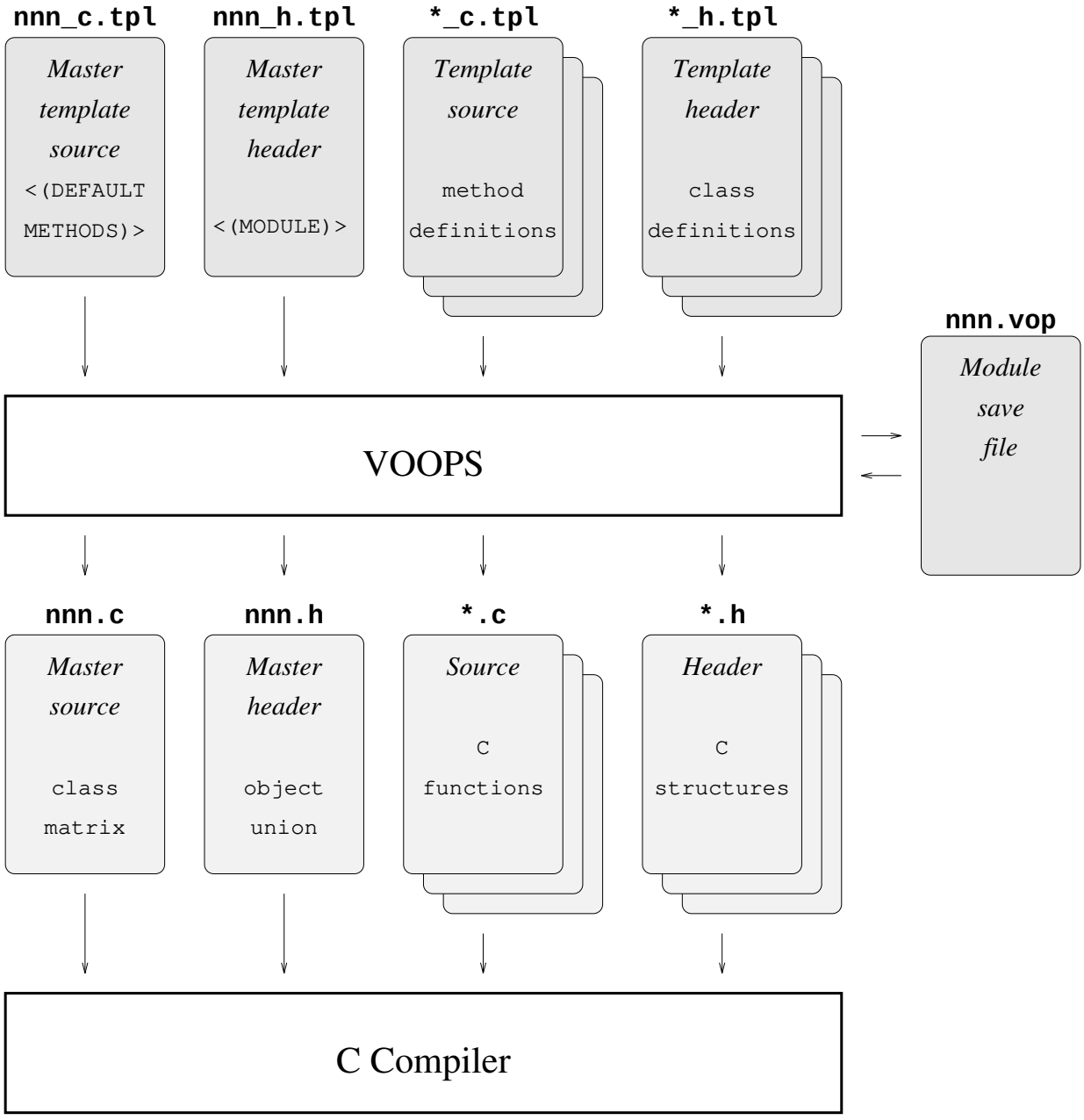

Figure 1: Relation of VOOPS' template header and source files

aid retrieving information from the class table, to obtain and set the object ID of object instances, and to test, set, and clear certain *object flags*.

Object flags are stored in the upper bits of the 32-bit object ID of an object, which are not used to encode the object ID. Due to the 32-bit integer used to store the class mask in the class table, the maximum number of supported classes is currently 32. This means, that at most $ld(32) = 5$ bits are actually used in the object ID entry to encode the object ID. The remaining 27 bits can be used for binary flags which are often useful for tagging objects during sweep- or scan-like algorithms.

1.8 Methods and the Messaging Concept

Method inheritance is directly implemented in the class table construction step, where the function pointers of inherited methods, if not overridden through genuine methods defined on a certain class, are stored in place of the genuine function pointer. If there is neither a genuine nor an inherited method in a certain class for a certain method name, a `NULL` pointer is stored. On invocation of a method, the corresponding method pointer of the class is checked if its non-`NULL`. In this case, the pointer is dereferenced, hence the function is called, otherwise an appropriate error is issued stating that there exists no corresponding method for this class. As we can see, there is only one `NULL`-pointer check and one pointer dereference which is the main reason for the efficiency of the static class model implemented in VOOPS, whereas an OOP system with a dynamic class model would have to search through its more or less complicated run-time class and method structures to find the appropriate applicable method. The above behavior is implemented in the VOOPS `SEND` and `SENDID` statements.

The above algorithm is only to be applied when the class of the object a specific message is to be sent to is not known in advance (i.e. prior to compilation), since then the class has to be determined from the object ID. In case the class is already known, meaning that the class of the object the message is to be sent to is known to be equal to or above in the inheritance branch of the specific class, the method can be applied directly using the VOOPS `SENDCLASS` statement which is just translated into a C function call. However, all methods in an inheritance branch have to be aliased appropriately with C macro definitions. This is done in the master header file following the respective class definition.

2 GRS — The Grid Support Library

The GRS *(Grid Support)* library has been designed to provide read, write, and manipulation functions for grids and attributes stored on PIF. As a basis for both, generic point list handling services are also provided that deal with `orthoProduct` and unstructured point lists as well. Both unstructured grids and tensor product grids can be read, written and manipulated, and interpolation of attributes defined on those grids can be performed with this library. New element types for unstructured grids can be introduced easily and are readily available in the generic interpolation routine.

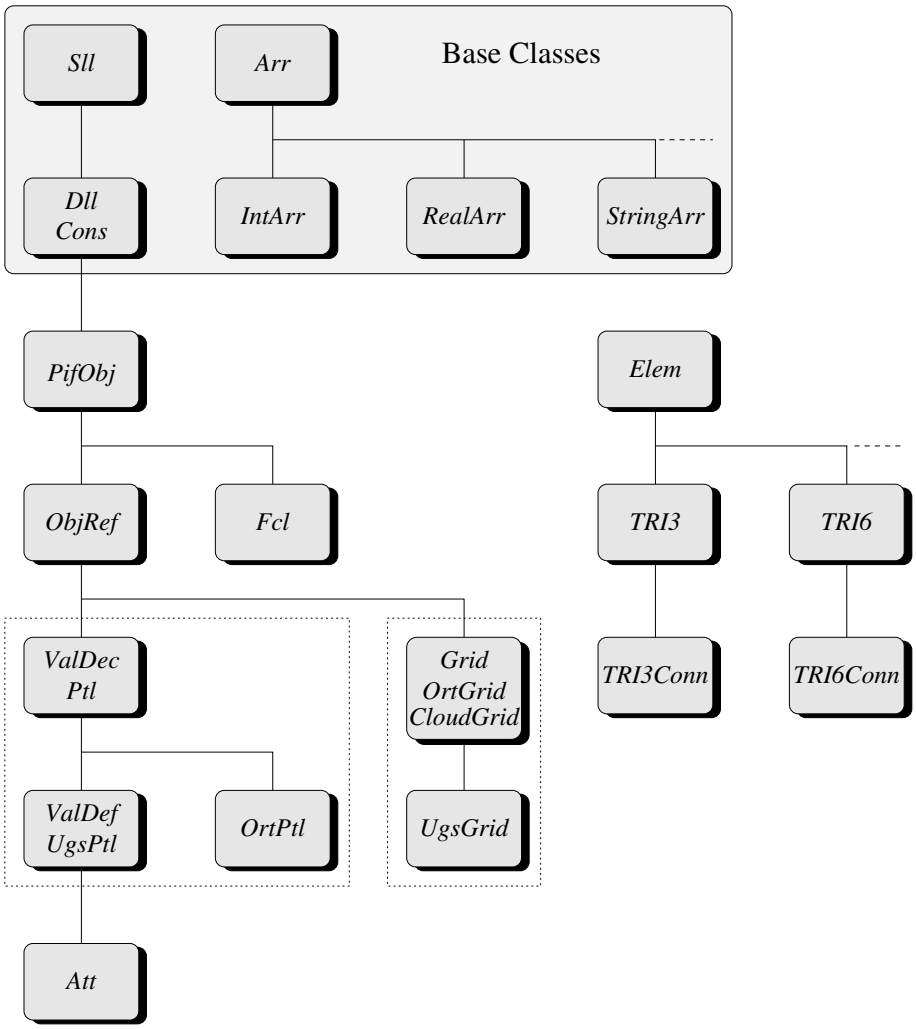

Figure 2: The GRS class tree.

In the following, the architecture of GRS is described first, where the various classes which GRS implements to accomplish its tasks are discussed. The memory-resident structures of GRS objects and their relationships are presented with the help of various figures. Two sections describing the two main purposes GRS was designed for — namely element-wise unstructured grid reading and writing, and generic interpolation of attributes — follow, where many issues of the class designs which looked obscure at the first sight will become clear. Finally, the KIRKPATRICK point location method which is a cornerstone of the unstructured grid interpolation is discussed in more detail.

2.1 Architecture

GRS is implemented with the help of VOOPS, where the various objects represented by GRS are modeled

through VOOPS class definitions. The instances of those classes store the information read from PIF, and the application uses an opaque object pointer to such an instance for all operations to perform.

The GRS library was designed bottom-up in that its classes are closely related to the general syntax of the PIF objects they describe. Therefore GRS is not limited to the two main purposes mentioned above, but rather provides an easily extensible and flexible object-oriented presentation of PIF objects to TCAD applications. Although the methods implemented on those classes are currently limited to fulfill the main design goals, they represent generic building blocks which can be used and extended to create new functionality inside GRS. Due to the intrinsic property of object-oriented designs, that changes in the implementation of a certain class do seldomly affect other classes[1], extensions to GRS can easily be effected. This property of object-oriented designs makes them well suited for academic environments, when students need to accomplish a well-defined and self-contained goal inside a team effort.

The whole GRS library is comprised of 32 template files making up a total of more than 5000 lines of VOOPS code, which get expanded to more than 15000 lines of C code after running VOOPS on them. The library defines 229 methods on 24 classes.

The various class definitions result in a *class tree* which is shown in Figure 2. The solid lines denote an inheritance branch, whereas the dashed lines denote a continuation by further similar classes on demand. While the array and element classes seem to be unrelated to the other classes in this diagram since they define their own inheritance trees, they are actually used in many places throughout GRS for point list and attribute representations on the one hand, and unstructured grid element representation on the other hand.

2.2 Unstructured Grid Element Representation

Not all applications can get along with sequential access to the individual grid elements of an unstructured grid. Therefore GRS maintains the grid elements in a dynamic array of object pointers, where each pointer points to a grid element object. Applications can therefore address a grid element directly by obtaining the element pointer from this array.

This direct access is needed by GRS itself also, when it traverses all elements containing a particular point of the grid. A corresponding array holds for each point of the grid an index array containing the indices of all elements adjacent to that point. The grid elements itself contain indices to the points (nodes) which make up their shape. The number of those point indices depends on the grid type, and is defined in the individual class of this grid element. Storing the element index has been preferred over storing a pointer to the element, because PIF also deals with indices, and a pointer would have prohibited or at least impeded the writing unstructured grids to PIF.

Each element is characterized by its name, topological dimension, number of nodes, and methods for decomposition, interpolation, point-in-element-test and point restriction. The name is usually a short mnemonic and used as the class name of this grid element, for example `TRI3` (a triangle with three nodes) or `QA4` (a quadrangle with four nodes). The topological dimension is two for "flat" elements stored in a `faceList`, and three for solid elements stored in a `solidList`.

Note that the topological dimension of a grid element may be different from its geometrical dimension, for example when describing the triangulation of a surface of a three-dimensional geometric object. Let N be the number of element nodes and D the topological dimension. Then a valid grid element has to have $N >= D + 1$ element nodes. The decomposition method decomposes a given element into *primitive* elements of the respective topological dimension, which always have exactly $N = D + 1$ element nodes. For example, a `QA4` element may be decomposed into two `TRI3` elements. This decomposition is used

[1] provided the class hierarchy is designed carefully!

in various applications throughout VISTA, the most prominent of which is the `xpif2d` visualizer, which always operates on two-dimensional primitives (i.e. triangles).

On interpolation, the requested point is located inside a particular element of the grid and the corresponding interpolation method of the respective grid element class is invoked. For the point location, the point-in-element-test method is needed, which returns a boolean value indicating if a given point lies inside or outside the element. The interpolation method returns an interpolated value given a point lying inside the element, the coordinates of its element nodes and the respective attribute values on the element nodes. This is usually achieved through transforming the node and point coordinates to a unity element and evaluating the *form functions* of this unity element.

The point restriction method is used in hierarchical grids to determine if a point location is possible on the grid. This is the case if all children of an inner node inside a hierarchical element tree cover the area of its parent totally. But if — for example after a grid adaption step — the points of the lowest level move slightly and the grid hierarchy is not rebuilt, there may be elements whose children do not fully cover the parent area. Then a point location on such an element may produce incorrect results, as would the interpolation using this malformed hierarchical grid. Figure 3 shows the cases of a correct triangulation allowing point location, and an incorrect triangulation where point location may produce wrong results.

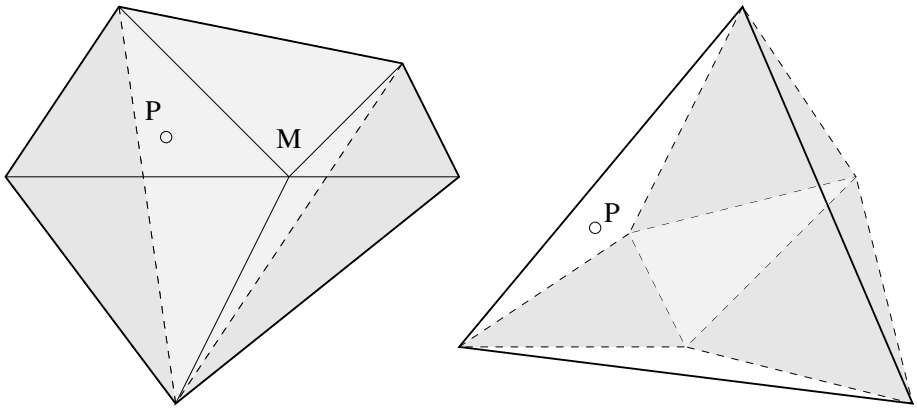

Figure 3: Correct and incorrect grid element hierarchies for point location. The left triangulation shows the result of a KIRKPATRICK grid hierarchy construction step by removing the center point M and retriangulating the resulting polygon. The newly generated triangles are shaded, and locating point P will successfully return the light shaded middle triangle. The right triangulation (as it may result from an oxidation simulation using a hierarchical triangular grid) however fails to locate point P, because despite the fact it is included in the thickly outlined parent triangle, its shaded children do not include point P.

2.3 Unstructured Grid Reading and Writing

Simple unstructured grid reading and writing is one of the two main purposes GRS is intended for. This functionality is both used in GRS internally for the unstructured grid interpolation, where a grid must be read into a memory-pertinent representation, as well as provided to other applications that need to read or write unstructured grids. For example, the capacitance simulator VLSICAP uses this functionality to write its output attributes onto triangular grids.

Full memory-pertinent structures are not necessarily during element-wise reading and writing of unstructured grids, therefore GRS does not store the point list and the list of grid elements internally. Rather it allocates just the structures describing the current grid and the current status of grid reading and writing. The actual data pointers in the face list structure and in the attribute structure are left unused, only the index of the current grid element in the face list and in the attribute is remembered.

Applications wanting to read or write an unstructured grid have to call an initialization function which allocates and fills this structures appropriately. The point list of the grid can be written with the usual PAL calls, since in this case GRS needs nothing more than just the point list handle for referencing it in the face list defining the grid elements. After initialization, the application has to call the GRS element read or write function, which read or write exactly one name list entry of the face list, and hence read or write exactly one element from or to the grid. A cleanup function has to be called finally, that frees the GRS-internal structures.

2.4 Generic Interpolation

As stated in [Hala94], interpolation of attributes between different grids is one of the most important tasks inside a TCAD framework. This task has to be as accurate, reliable, and as fast as possible. Depending on the grid the attribute is defined on, GRS makes provisions for three possible cases:

- tensor product interpolation,
- point cloud interpolation, and
- unstructured grid interpolation.

For tensor product grids, linear and higher-order (using the AKIMA method [Akim70, Hyma83]) interpolation is implemented on the tensor product grid class. In the case of higher-order interpolation, the spatial derivatives of the attribute under consideration have to be calculated prior to interpolation. This is done in the corresponding grid preparation method of this class. Additionally, the grid axis ticks have to be sorted into ascending order in order to allow an $O(log(n))$ time point location. The respective attribute values have to be shuffled accordingly, which is also done in the grid preparation method.

Searching in point cloud grids is possible through using a search structure like an *N-ary tree* [Same90], which allows an $O(log(n))$ point location on the point cloud. However, the interpolation function for point cloud grids is currently not implemented in GRS, but rather exists as a stand-alone prototype using a modified SHEPARD's method [Agis91, Alfe89].

When interpolation on an unstructured grid is to be performed, a search structure has to be used also, which enables locating the element in which a given point lies. Besides the N-ary tree, several other geometric search structures can be applied which allow an $O(log(n))$ effort point location [Prep85], like the *slab method*, the *chain method*, the *triangulation refinement method* and the *trapezoid method*.

Among those methods, the triangulation refinement method (in the following referred to as the KIRKPATRICK point location method [Kirk83]) was chosen for the point location in unstructured grids, since it uses triangles (in two-dimensional space) and relations between them as data structures, and a triangle represents already the primitive geometric grid element in two dimensions. Furthermore, this method uses a hierarchical tree of those triangles, and this tree structure is equally well suited to represent hierarchical unstructured grids too. Therefore two goals (point location and hierarchical grid representation) can be met with a single data structure, which is an important conceptual advantage in the application of this method to unstructured grids.

This tree structure is called a *search-directed acyclic graph* (SDAG), and has to be built prior to interpolation on unstructured grids. With this search structure, the grid element in which a given point lies and therefore is to be used for the interpolation can be located in $O(log(n))$ time.

3 LISP-PED

A new version of the PIF Editor has been developed. It was designed as a highly flexible and user extensible interactive tool. It consists - beside a couple of more general libraries - of an X11 interface, a PIF interface, and the XLISP interpreter with some LISP programs. Most configurations and all user interaction modes can be controlled with LISP files or commands.

3.1 Event Structure Principle

When designing the LISP interface for PED it was desired to create a programming-language like extension for configuration and extension purposes.

A simple procedural language is not sufficient due to the callback philosophy that triggers for each X11 event a callback that has to perform the required actions. But the information about previous actions is required for correct handling of events. This problem has been solved with an infinite state machine (automaton).

This automaton has a stack that allows nesting of functionality. Its input alphabet is a subset of LISP expressions. The rules may be defined at runtime.

3.2 PED Grammar Examples

The following is a small part of the grammar implemented in the PED. Keywords are in uppercase with leading colon (an XLISP feature lets symbols starting with `':'` evaluate to themselves). Non-terminals are in lowercase, and non-keyword terminals in `<...>`.

There are two simple ways to parse a point:

```
point       ::= :POINT coords
point       ::= <cls-point>
```

The first is by giving the coordinates of the point; the second matches an already existing point.

To allow the user to enter many points sequentially we have a point-mode:

```
point-mode ::= :POINT-MODE {:REPEAT point}
```

The coordinates - with the simplification of 2 dimensional geometry - are matched by 2 real numbers or by a list of 2 real numbers:

```
coords      ::= <float> <float>
coords      ::= <list of 2 numbers>
```

Three exemplary rules for a line are: creation of a line by two points, a handle of an existing line, and snapping an existing line by coordinates.

```
line        ::= :LINE point point
line        ::= <cls-line>
line        ::= :SNAP-LINE coords
```

All simple geometric objects are packed in a non-terminal for objects:

```
object      ::= point
object      ::= line
object      ::= face
```

This "object" non-terminal may be used in the "delete" rule:

```
delete      ::= :DELETE object
```

3.3 Applying Event Data to the State Machine

Data is passed to the state machine by functions that take a LISPexpression, and pass it, eventually after evaluation, as input token.

The binding of low level event data and the state machine is implemented in several stages. In principle it always works via a callback function that passes client or callback data that may depend on event details to the state machine.

- *mouse*: The mouse callbacks are subject to the `mode` configuration (see below). Button press and release events with respect to modifier keys cause interpretation of strings. Motion events, that are not supposed to work with the state machine, invoke user callback functions.
- *keyboard*: Key presses are caught with X11 translations and accelerators and are passed to the PED command-line (text) widget as editing sequences. But pressing of `Enter/Return` key causes processing of the command line contents.
- *menus*: Selection of menu items invokes generic callback functions that are in the PED typically bound to a string processing state machine entry.
- *dialog windows*: In some situations a dialog shell is popped up that allows data input of some kind. The confirm-callback is gets the new data and passed it to the automaton.

To get actual information, especially about the cursor position, into strings some formatting information may be put in configuration strings that is replaced with the mouse coordinates in case of callback invokation.

3.4 Configuring the Interactive Mode

The *interactive mode* of the PED is the special setting of informations for the user, of the current meaning of the mouse buttons, of the mouse movement indications, and of popped up dialogs or panels.

It is controlled by actions built into the rules (in some cases the mode is saved by the state machine). There are two different expert levels for pure users and for experienced programmers.

In detail the following elements may be configured:

- The `title` which is a string that tells the user what mode is currently valid. In the default configuration it is displayed in the lower left corner of PED.

- The three `mouse buttons`, each represented by a list of two strings, the first of which is a short text telling the user the current meaning (configuration) of the button, and the second a string to be processed with the function `ped::logical-command` after replacing some special text sequences when the button is pressed.
- Two *callback functions with some client data* one of which is invoked when the PED enters the wait-for-event loop and might draw something to the Canvas widget that should be related to the previous user actions and the current cursor position, and the other to exactly undraw it. Typically the two arguments are the same function and invert some part of the drawing area reversibly. The appropriate client-data is in most cases data produced by previous actions, terminals, or non-terminals and might e.g. be a coordinate pair or a point handle.
- A *popup shell* that is activated when waiting for user input.

  Note: client-data for popup shell callbacks is usually specified on creation of the shell

An example for the rule creating a simple line looks like this:

```
((newrule
 "line"
 NIL
 '(:LINE
   (mode "Enter first point of line"    ; title
         ("first point" "(point %G)")   ; left mouse button
         ("first point" "(point %G)")   ; middle
         ("" "")
         #'ped::elastic-point           ; draw soft cursor with respect to
                                        ; magnet
         #'ped::elastic-point           ; undraw by inverting
         NIL                            ; no client-data
         NIL)                           ; no popup shell
   "point"
   (mode "Enter second point of line"   ; title
         ("second point" "(point %G)")  ; left mouse button
         ("second point" "(point %G)")  ; middle
         ("cancel" ":CANCEL")           ; right - cancel current states
         #'ped::elastic-line            ; draw line from first point to cursor
         #'ped::elastic-line            ; undraw it
         ($ 3)                          ; first point is client-data for
                                        ; elastic functions
         NIL)
   "point"
   (line ($ 3) ($ 5))                   ; finally create line and return it
))
```

These new features make it possible to invoke extern executables like grid generators or volume modellers from PED. Figure 4 shows the output of a TRIGEN run that has been performed after specifying the device structure from scratch without leaving PED.

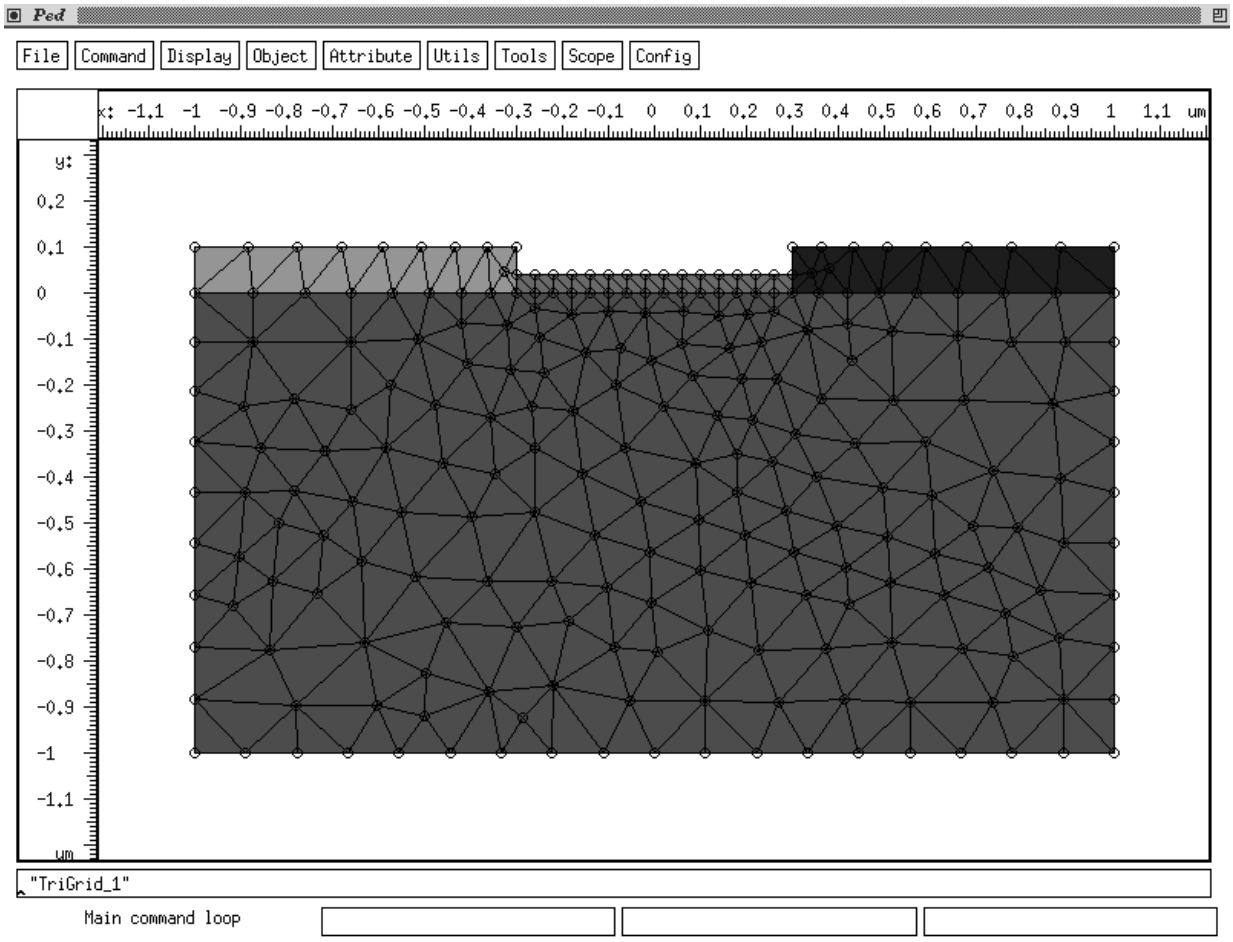

Figure 4: PIF Editor after TRIGEN invokation

4 Trajectory Split Method - a New Strategy for the Crystalline Mode of the VISTA Monte Carlo Ion Implantation Module

4.1 Introduction

As a part of the VISTA process simulation tools a program for Monte Carlo simulation of ion implantation has been developed. The program can handle arbitrary two-dimensional geometries (the three-dimensional module is under development and it already exists as alpha version) containing regions of amorphous and crystalline materials. A well-known disadvantage of the Monte Carlo approach is its considerable demand for computer resources to obtain results with satisfying statistical accuracy.

4.2 The Conventional Strategy for the Crystalline Mode

The conventional Monte Carlo approach for crystalline targets [Hobl91] is based on the calculation of a large number of distinct ion trajectories. Each trajectory is usually followed from the ion starting point at the surface of the target up to the stopping point of the ion. Since the majority of ion trajectories end at the most probable penetration depth inside the structure the statistical representation of this target region is good. Peripheral areas of the dopant concentration are normally represented by a much smaller number of ions (typically 10^4 times less than the maximum). This results in an insufficient number of events there and leads to a statistical noise that cannot be tolerated.

4.3 The Trajectory Split Algorithm

Recently, a "rare event" approach for one-dimensional structures has been suggested to significantly improve the calculation time performance of the UT-MARLOW code [Yang94]. Based on this algorithm the new "trajectory split" method [Bohm95] results in a much better statistical accuracy at the periphery of the dopant distribution, it is easy to implement into existing codes, and it requires considerably less computation time to calculate implantation profiles with a given precision.

The fundamental idea of our simulation approach is the partioning of the simulation run into two stages and the utilization of the information we can derive from the flight-path of the ion up to a certain depth inside the target. First a portion of ion trajectories is calculated conventionally to obtain roughly the shape of the ion distribution and to determine the maximum concentration of this primary implantation (C_{max}). For the further trajectories, the local dopant concentration C_{loc} is checked at certain points of the flight-path. In case the ratio C_{loc} / C_{max} is below given levels (we defined ten split levels at 0.3, 0.09, ..., 0.3^{10}), a trajectory split point is defined at this checkpoint. We store the position of the ion, its energy and the vector of velocity and use this data for virtual branches of ion trajectories starting at this split point.

In our implementation of this method we have defined simple splits of one ion trajectory into two virtual branches at each split point. To obtain the correct concentration a weight was assigned to each branch. In this manner a binary tree of virtual trajectories is formed for each regular ion (Fig. 5). Such a virtual trajectory branch is calculated with the *same models and parameters* as a regular trajectory, but it starts *at the split point* with *initial conditions obtained from the regular ion*.

4.4 The Merits and the Applicability of the New Method

The trajectory split approach has been implemented in the two-dimensional Monte Carlo implantation module of the VISTA framework. These modules use advanced physical models for calculation of ion implantation into crystalline silicon, therefore they are capable of predicting the channeling effects and the transient amorphization using the modified Kinchin-Pease model [Norg75].

To demonstrate the merits and the applicability of the new method we performed several ion implantation simulations into a trench derived from the Etching and Deposition Module using the amorphous mode (Fig. 6), the conventional mode (Fig. 7), and the new trajectory split mode (Fig. 8). For the simulations we used a boron implant of 10^{14}cm^{-2} at 20keV. The beam tilt angle was 7° calculated from the [001] axis for dechanneling.

4.5 Conclusion

Comparison of the two crystalline modes with respect to the CPU time (we used a HP 735/100 workstation) needed for the same statistical significancy shows a clear advantage of the trajectory split method compared to the conventional one. The CPU time reduction due to the new simulation strategy is more than five times in this particular application. It depends on the statistical accuracy requirements, the dimensionality of the device structure, and the ion energy.

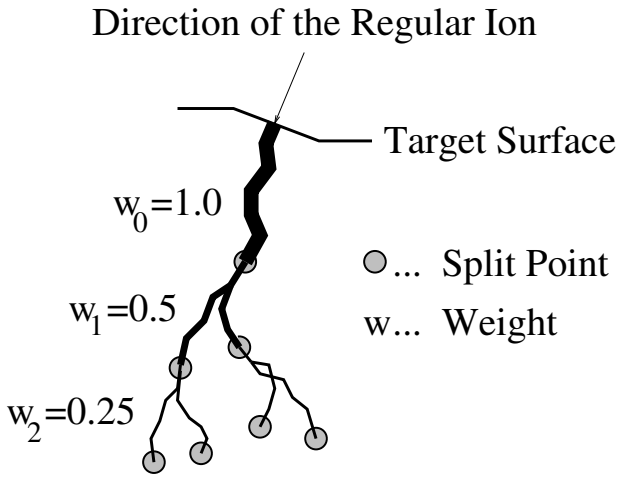

Figure 5: Topological structure of the virtual trajectory branches and their weights

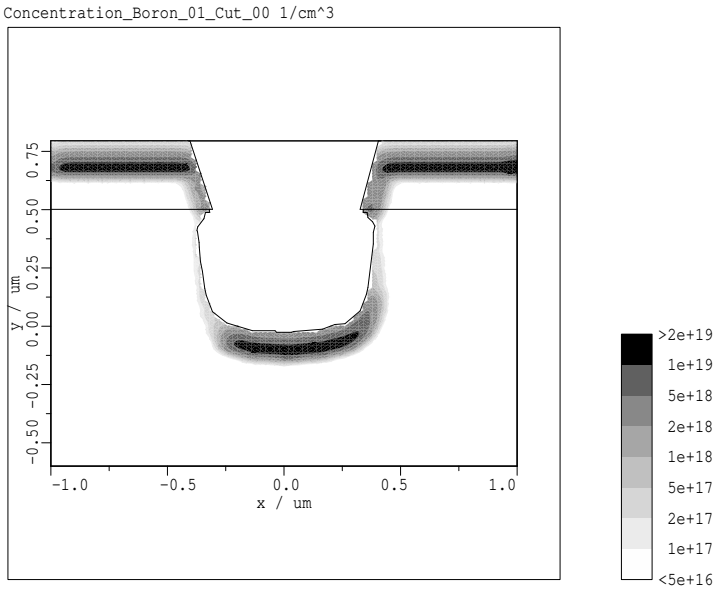

Figure 6: Dopant Concentration into (100) silicon (amorphous mode, t_{CPU}: 4' 29")

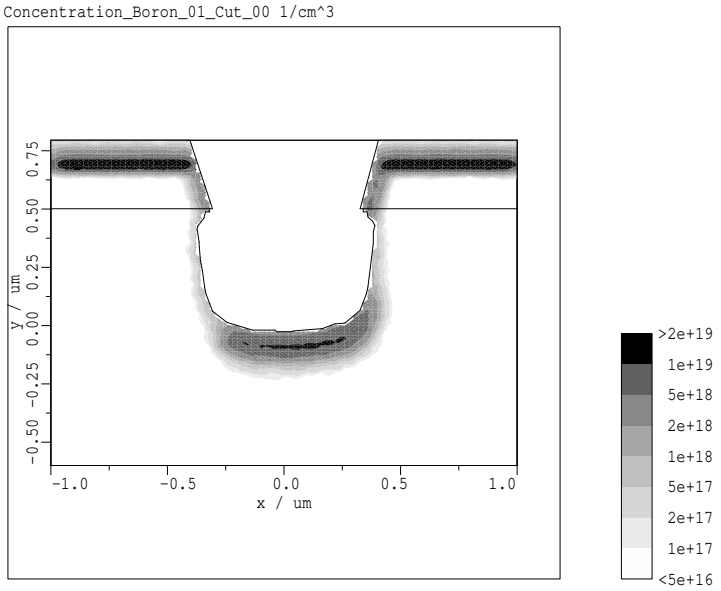

Figure 7: Dopant Concentration into (100) silicon (conventional crystalline mode, t_{CPU}: 1h 48')

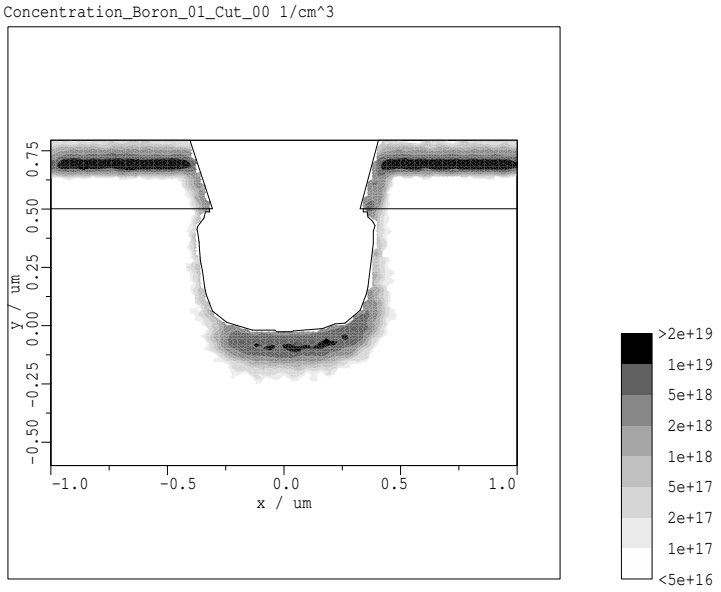

Figure 8: Dopant Concentration into (100) silicon (trajectory split mode, t_{CPU}: 21' 00")

5 MINIMOS NT – Two-Dimensional Device Simulation

MINIMOS NT, version 0.91β, is a new two-dimensional device simulator entirely written in C. Basing on the features of the standard MINIMOS program a fully modularized structure for MINIMOS NT has been developed (see Fig. 9) providing maximum flexibility in modifying and extending the features and capabilities of MINIMOS NT. Also, the concept tries to avoid essential limitations, i.e. special device geometries or fixed number of contacts. In the following sections an overview is presented what MINIMOS NT can do and what it cannot do, also a short introduction is given how the input and control data in the PIF input file must be specified.

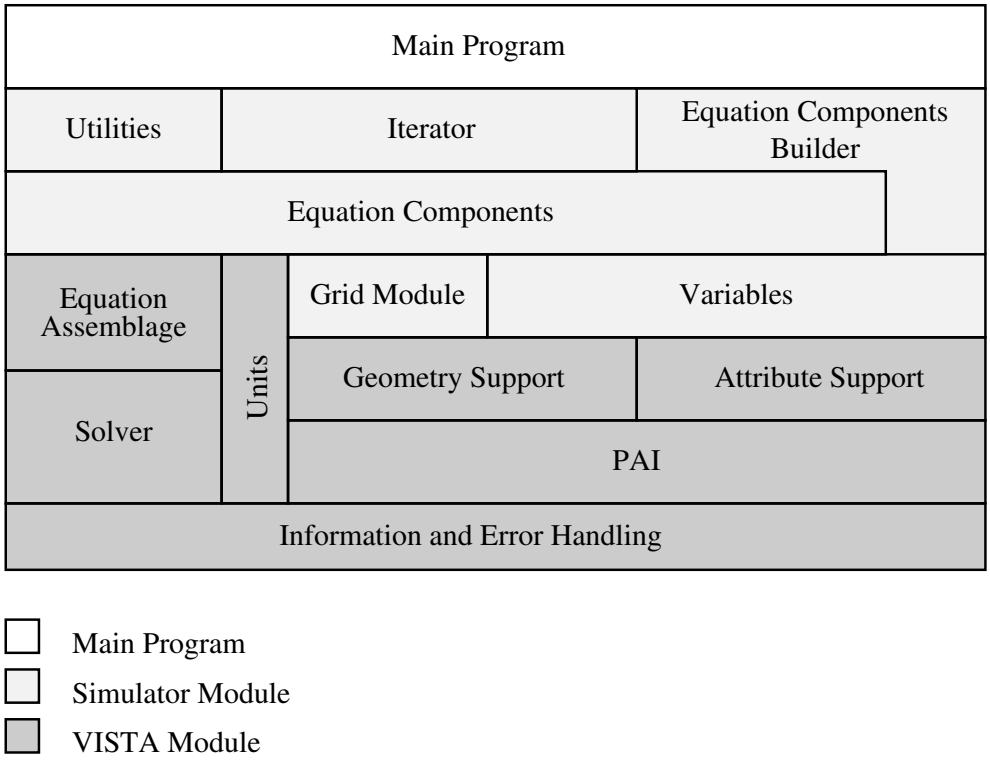

Figure 9: Structure of MINIMOS NT

5.1 Features of MINIMOS NT

- Arbitrary rectangular geometries, i.e. geometries built of horizontal and vertical lines, with no limitations on the number and the location of contacts, insulators and semiconductor regions.
- The device region can be arbitrarily partitioned in segments. For each segment different materials and physical properties (i.e. permittivity, effective density-of-states, band edge energies or mobilities) can be specified. Also, a spatial variation of these parameters inside of segments is allowed.
- Abrupt segment interface modeling. As mentioned above, for each segment different properties can be given. Thus, abrupt changes of properties across segment interfaces can occur and are handled by appropriate interface modules (see section below).
- For submicron devices the standard drift-diffusion models have been extended by five-equation physics which include the effects of energy balance, i.e. considering the carrier temperatures. The command line option `-hydro` enables hydrodynamic simulation.
- The command line option `-transient` enables transient simulation.

- Mobility function module which provides mobilities for Silicon as well as for Gallium-Arsenide. For Silicon, the well known MINIMOS mobility functions are used.
- Several contact conditions such as voltage controlled, current controlled, charge controlled or a generic condition which is a combination of all the previous ones.
- VISTA framework integration. Even though MINIMOS NT is a *stand alone* program it is fully compatible with the VISTA framework in using the PIF data format for specifying the control and input data. Furthermore, the calculated results as potentials, carrier densities, carrier temperatures and related attributes as electric field, current density and so on are stored onto PIF. Information on iteration progress and final integral results as contact voltages, currents and charges are written to standard error.
- VISTA command line handling. Command line arguments are used for controlling the simulator's behavior. If the program is called with a help option a comprehensive list of command line arguments with a short description is printed. It is also possible to specify a file from which the command line arguments should be read.

5.2 Anti-Features of MINIMOS NT

The following list shows the features which are not yet implemented in the current version 0.91β of MINIMOS NT:

- Other than rectangular grids. A triangular grid support module is under development and will be implemented in a later release.
- Automatically generated initial grid. The initial grid must be specified by the user or elsewhere. Only a simple grid which consists of a horizontal and a vertical grid line for each point of the geometry is built which can be refined by a golden ratio criterion (see section below).
- Grid refinement. No grid refinement is done during simulation process. This means that the initial grid or the refined initial grid must be suitable to obtain final results with the desired accuracy.
- No generation/recombination models are implemented. This includes also impact ionization.
- AC analysis.
- Even though transient analysis is implemented, there is no easy way for specifying transient input data. With the actual version only step functions can be applied easily.

5.3 The Device Geometry

The devices geometry is defined as geometry object on the input PIF (see the PED manual and the description of the PIF language syntax for further details). Since the actual version of MINIMOS NT supports ortho product grids only, only *Manhattan*-geometries can be used, which are geometries built of horizontal and vertical lines, because the grid and the geometry have to be conform, i.e. a geometry line must also be part of a grid line. Apart from that the complexity of device geometries is only limited by computer and time resources (see Fig. 10).

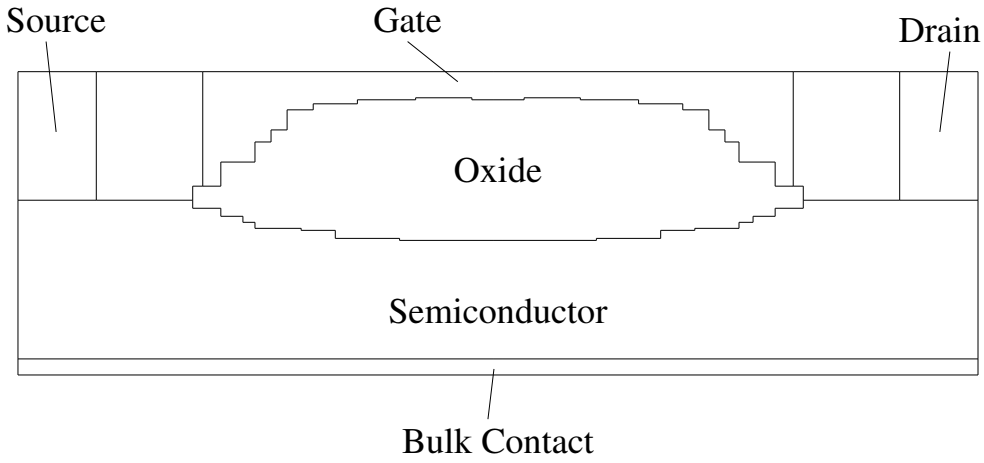

Figure 10: Parasitic field-oxide MOS transistor as an example for a device geometry specification for MINIMOS NT

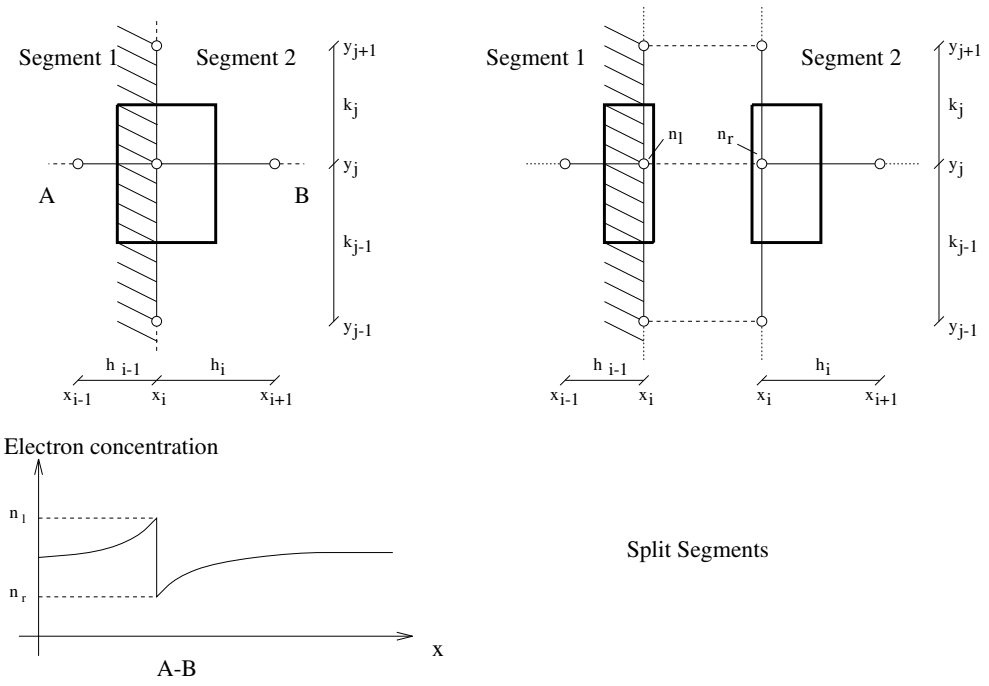

Figure 11: Modeling abrupt junctions at segment interfaces.

5.4 Abrupt Segment Interface Modeling

As a result of an idealization process, on segment interfaces an abrupt change of certain values can occur (see Fig. 11). For example high electron mobility transistors where a narrow band gap InGaAs layer is sandwiched between two wide band gap AlGaAs layers show an abrupt change in electron concentration across the hetero junction interface. Mathematically, this behavior is described by an left limit value and right limit value when approaching the interface from the left and from the right, respectively. Furthermore, a third value directly at the interface can be specified (e.g. to consider segregation effects). If no interface value is necessary, it can be thought identical to one of the left or right limits.

Since there are at least two values at the interface now, an additional formula is needed to connect the segments across the interface. For the electron concentration this is the module considering the thermionic

emission current

$$J_{\text{interface}} = q\left[v_{nl}(T_l)n_l - \frac{m_l}{m_r}v_{nr}(T_r)n_r \exp\left(-\frac{\Delta E_C}{kT_r}\right)\right] \quad (1)$$

where the indices l and r identify the left and the right limit, respectively and $v_{ni}(T_i) = \sqrt{\frac{2kT_i}{\pi m_i}}$ are the "emission velocities".

5.5 The Initial Grid

Since there is no initial grid generation included in MINIMOS NT a suitable grid must be specified onto the PIF input file. Depending on the geometry for each geometry point a horizontal and a vertical grid line is added to the initial grid if it does not already exist. A simple method to obtain an initial grid is to define grid lines very close to geometry lines in critical regions, i.e. the channel region of a MOS transistor, and then to automatically refine the grid using the "`-sectioAurea` *<ratio>*" command line option, where *<ratio>* is the upper limit of the ratio of two successive grid line spacings. The ratio is built with the larger spacing as numerator such that *<ratio>* >= 1.0. A value of `-sectioAurea <=` 1.618 means 1.618 (the so called *golden ratio*) is used producing the densest possible grid. The initial grid itself is selected by the command line option "`-inputGRID` *<name>*".

5.6 The Segment Description

The segment description defines all input data related to a certain segment:

```
(attribute ChannelSegmentDescription
  (attributeType "SegmentDescription")
  (nameList (ref segments_1 (valueList 8)))
  (attribute ChannelSegmentMaterialType
    (attributeType "MaterialType")
    (valueType asciiString)
    (valueList "InGaAs"))
  (attribute ChannelSegmentPermittivity
    (attributeType "Permittivity")
    (valueType real)
    (units "VacuumPermittivity")
    (valueList +13.2300)))
```

The example above shows the segment description for the channel region of a high electron mobility transistor. The channel region is specified by segment 8 of the device geometry (`(nameList (ref segments_1 (valueList 8)))`) and the attributes meaning is:

`ChannelSegmentMaterialType` defines the material type of the segment (`"InGaAs"`).

`ChannelSegmentPermittivity` defines a constant permittivity for the segment. If this attribute is not defined the permittivity provided by the material server for the material `"InGaAs"` is used. If the material server does not find the permittivity in the material database an error is issued.

5.7 The Carrier Description

The carrier description defines all input data related to a specific carrier type in a segment. The following paragraph shows an example for carrier description on a PIF input file:

```
(attribute ChannelElectronDescription
  (attributeType "CarrierDescription")
  (nameList (ref segments_1 (valueList 8)))
  (attribute ChannelElectronType
    (attributeType "CarrierType")
    (valueType asciiString)
    (valueList "Electron"))
  (attribute ChannelMobilityElectronsHydro
    (attributeType "HydroMobility")
    (valueType asciiString)
    (valueList "HaenschInGaAs"))
  (attribute ChannelMobilityElectronsDrift
    (attributeType "DriftDiffusionMobility")
    (valueType asciiString)
    (valueList "StandardInGaAs"))
  (attribute ChannelElectronBandEdgeEnergy
    (attributeType "BandEdgeEnergy")
    (valueType real)
    (units "eV")
    (valueList +1.2480))
  (attribute ChannelElectronEffectiveDensityOfStates
    (attributeType "EffectiveDensityOfStates")
    (valueType real)
    (units "1/cm^3")
    (valueList +3.5100e+17)))
```

This carrier description specifies the input data for electrons in the channel of a high electron mobility transistor. The channel region is defined by segment 8 of the device geometry (`(nameList (ref segments_1 (valueList 8)))`).

The meaning of the attributes shown above is as follows:

`ChannelElectronType` defines the carrier type which is `"Electron"` here.

`ChannelMobilityElectronsDrift` defines the name of the mobility function used for drift-diffusion simulation. For a list of mobility names see the section below.

`ChannelMobilityElectronsHydro` defines the name of the mobility function used for hydrodynamic simulation. For a list of mobility names see the section below.

`ChannelElectronBandEdgeEnergy` is the value of the related band edge energy in eV, i.e. the conduction band edge energy for electrons and the valence band edge energy for holes where an arbitrary reference energy can be chosen.

`ChannelElectronEffectiveDensityOfStates` is the effective density-of-states in cm^{-3}.

5.8 The Mobility Functions

The mobility function module consists of a function table where the mobility names (also called *Identifier*) are defined. Using the mobility function name and depending on the material type (actually only *GaAs* and *Si* are supported), the carrier type (*Electron* or *Hole*) and the PDE-set (*Drift-Diffusion* or *Hydrodynamic*) the appropriate mobility function is chosen (this implies that different mobility function names can define the same mobility function). Furthermore, the mobility function name *Default* means the mobility function used if no mobility function attribute in the carrier description of the PIF input file is given.

The following table shows the actually implemented mobility functions (this list is also printed with the `-mobilityList` command line option specified):

List of predefined mobility functions:	
Identifier	Description
Default	Minimos 6 Si Electrons
Default	Minimos 6 Si Holes
Minimos6	Minimos 6 Si Electrons
Minimos6	Minimos 6 Si Holes
Constant	Constant Mobility Si Electrons
Constant	Constant Mobility Si Holes
Default	Standard GaAs Electrons
Default	Standard GaAs Holes
Default	Haensch Hydro GaAs Electrons
Default	Haensch Hydro GaAs Electrons
StandardGaAs	Standard GaAs Electrons
StandardInGaAs	Standard InGaAs Electrons
OvershootGaAs	Overshoot GaAs Electrons
OvershootInGaAs	Overshoot InGaAs Electrons
StandardGaAs	Standard GaAs Holes
2Valley	2-Valley Hydro GaAs Electrons
2ValleyFixed	2-Valley Fixed Hydro GaAs Electrons
Haensch	Haensch Hydro GaAs Electrons
HaenschInGaAs	Haensch Hydro InGaAs Electrons

The mobility function module is found in the files `xexmobfn.c` and `xemobfn.h`.

5.9 Contact Specification and Stepping

Contact conditions are also given by the PIF input file:

```
(attribute GateSegmentDescription ; GATE
  (attributeType "SegmentDescription")
  (nameList (ref segments_1 (valueList 3)))
  (attribute GateMaterial
    (attributeType "MaterialType")
    (valueType asciiString)
    (valueList "Al"))
  (attribute GateContactType
    (attributeType "ContactType")
    (valueType asciiString)
    (valueList "Schottky"))
  (attribute GateWorkFunctionEnergy
    (attributeType "WorkFunctionEnergy")
    (units "eV")
    (valueType real)
    (valueList -0.678))
  (attribute GateSchottkyBarrierHeight
    (attributeType "BarrierHeight")
    (units "eV")
    (valueType real)
    (valueList 0.9))
```

```
(attribute GateSchottkyRecombinationVelocity
  (attributeType "RecombinationVelocity")
  (units "cm/s")
  (valueType real)
  (valueList 1.0e+4))
(attribute GateContactVoltage
  (attributeType "ContactVoltage")
  (units "V")
  (valueType real)
  (valueList -0.75 1.0 0.25)))
```

Segment 3 is used as gate contact `(nameList (ref segments_1 (valueList 3)))` with the attributes:

`GateMaterial` defines the contact material (Aluminum `"Al"` here).

`GateContactType` must only be specified for Schottky contacts.

`GateWorkFunctionEnergy` is the negative Fermi energy level of the gate contact metal related to the reference energy level (see the description of the band edge energies above). For ohmic contacts the work function energy is always zero.

`GateSchottkyBarrierHeight` If the gate is a Schottky contact the barrier height must be also specified which is nearly the difference of the conduction band edge energy of the successive semiconductor and the metal Fermi energy level (note that this value does not depend on the reference energy level).

`GateSchottkyRecombinationVelocity` If the gate is a Schottky contact the thermionic emission velocity must be specified.

`GateContactVoltage` defines the contact as voltage controlled. If the *attributeType* is `"ContactCurrent"` the contact is current controlled. If the *attributeType* is `"ContactVoltage"` one value defines a fixed contact voltage, whereas three values mean to step the contact voltage. The first value is the start voltage, the second value the final voltage and the third value defines the voltage increment. The example shows a start voltage of -0.75V, a finale voltage of 1.0V and an increment value of 0.25V. One contact can be specified for stepping only.

References

[Agis91] M.E. AGISHTEIN AND A.A. MIGDAL. Smooth Surface Reconstruction from Scattered Data Points. *Comput. & Graphics,* Vol. 15, No. 1, 1991, pp. 29–39.

[Akim70] H. AKIMA. A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures. *Journal of the ACM,* Vol. 17, No. 4, 1970, pp. 589–602.

[Alfe89] P. ALFELD. Scattered Data Interpolation in Three or More Variables. In: *Mathematical Methods in Computer Aided Geometric Design*, Ed. by T. LYCHE AND L. SCHUMAKER. Academic Press, Boston, 1989, pp. 1–33.

[Betz89] D.M. BETZ. *XLISP: An Object-Oriented Lisp, Version 2.1*. Peterborough, New Hampshire, 1989.

[Bobr88] D.G. BOBROW ET AL. Common Lisp Object System Specification, X3JI3 Document 88-002R. *ACM SigPLAN Notices*, No. 23, 1988.

[Bohm95] W. BOHMAYR AND S. SELBERHERR. Trajectory Split Method for Monte Carlo Simulation of Ion Implantation Demonstrated by Three-Dimensional Poly-Buffered LOCOS Field Oxide Corners. In *submitted to: VLSI-TSA 95,* 1995.

[Boni91] D.S. BONING, M.L. HEYTENS, AND A.S. WONG. The Intertool Profile Interchange Format: An Object-Oriented Approach. *IEEE Transactions on CAD,* Vol. 10, No. 9, 1991, pp. 1150–1156.

[Budd91] T. BUDD. *Object-Oriented Programming*. Addison-Wesley, 1991.

[Dani91] J. DANIELL AND S.W. DIRECTOR. An Object Oriented Approach to CAD Tool Control. *IEEE Transactions on CAD,* Vol. 10, No. 6, 1991, pp. 698–713.

[Elli90] M.A. ELLIS AND B. STROUSTRUP. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Flor91] J.J. FLORENTIN. *Object-Oriented Programming Systems*. Chapman & Hall, 1991.

[Fran88] FRANZ INC. *Common LISP — The Reference*. Addison-Wesley, 1988.

[Gold83] A. GOLDBERG. *Smalltalk-80 : The Language and its Implementation*. Addison-Wesley, 1983.

[Hala94] S. HALAMA. *The Viennese Integrated System for Technology CAD Applications — Architecture and Critical Software Components*. PhD thesis, Technische Universität Wien, 1994.

[Hewl90] HEWLETT PACKARD COMPANY. *HP C++ Programmer's Guide*, 1990.

[Hobl91] G. HOBLER, H. PÖTZL, L. GONG, AND H. RYSSEL. Two-Dimensional Monte Carlo Simulation of Boron Implantation in Crystalline Silicon. In *SISDEP 91,* 1991, pp. 389–398.

[Hyma83] J.M. HYMAN. Accurate Monotonicity Preserving Cubic Interpolation. *SIAM Journal on Scientific and Statistical Computing,* Vol. 4, No. 4, 1983, pp. 645–654.

[Kirk83] D. KIRKPATRICK. Optimal Search in Planar Subdivisions. *SIAM J.Computing,* Vol. 12, No. 1, 1983.

[Meye92] B. MEYER. *Eiffel : The Language*. Prentice-Hall, 1992.

[Norg75] M.J. NORGETT, M.T. ROBINSON, AND I.M. TORRENS. A Proposed Method of Calculating Displacement Dose Rates. *Nucl. Eng. Des.,* Vol. 33, 1975, pp. 50–54.

[Pins88] L.J. PINSON. *An introduction to object-oriented programming and Smalltalk*. Addison-Wesley, 1988.

[Prep85] F.P. PREPARATA AND M.I. SHAMOS. *Computational Geometry*. Springer, 1985.

[Rumb91] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, AND W. LORENSEN. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Same90] H. SAMET. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[Stee90] G.L. STEELE. *Common LISP — The Language*, 2nd edition. Digital Press, 1990.

[Tell89] E.R. TELLO. *Object - Oriented Programming for Artificial Intelligence*. Addison-Wesley, 1989.

[Wens91] J.S. WENSTRAND. An Object-Oriented Model for Specification, Simulation, and Design of Semiconductor Fabrication Processes. Technical Report ICL91-003, Integrated Circuits Laboratory, Stanford University, 1991.

[Wien88] R.S. WIENER AND L.J. PINSON. *An Introduction to Object-Oriented Programming and C++*. Addison-Wesley, 1988.

[Wins89] P.H. WINSTON AND B.K.P. HORN. *LISP*, 3rd edition. Addison-Wesley, 1989. Reading, Mass., USA.

[Yang94] S.-H. YANG, D. LIM, S. MORRIS, AND A.F. TASCH. A More Efficient Approach for Monte Carlo Simulation of Deeply-Channeled Implanted Profiles in Single-Crystal Silicon. In *NUPAD 94,* 1994, pp. 97–100.