Microelectronics Journal, 26 (1995) 137–158

Feature article

The Viennese integrated system for technology CAD applications

S. Halama[1], F. Fasching[1], C. Fischer[1], H. Kosina[1], E. Leitner[1], P. Lindorfer[2], Ch. Pichler[1], H. Pimingstorfer[1], H. Puchner[1], G. Rieger[1], G. Schrom[1], T. Simlinger[1], M. Stiftinger[1], H. Stippel[1], E. Strasser[1], W. Tuppa[1], K. Wimmer[1] and S. Selberherr[1]

[1]*Institute for Microelectronics, TU Vienna, Gusshausstrasse 27-29, A-1040 Vienna, Austria*

[2]*National Semiconductor, M/S D3-677, PO Box 58090, Santa Clara, CA 95052-8090, USA*

Modern technology CAD systems consist of several simulation tools and a so-called *technology CAD framework* for tool integration and tool development. We discuss the general requirements for such a TCAD framework. From a review of existing TCAD systems with an emphasis on software aspects, we deduce general architectural guidelines. Bearing these in mind, we motivate our ideas, concepts and choices for the *data level*, the *user interface* and the *task level* of VISTA, the *Viennese Integrated System for Technology-CAD Applications*. The resulting structures and implementations of these three major parts of VISTA are then described, and a brief overview of integrated tools and future activities is given.

1. Introduction

1.1 The TCAD scenario

The introduction of the framework concept into the TCAD field has had a considerable impact on the way technology CAD is applied. The essence of this concept is to distinguish between specialized simulation tools which are able to perform specific simulations, and the integrating framework which ties together these tools to solve actual TCAD design tasks. The very common, unlucky approach of point tools, equipped with pre- and postprocessors, eventually coupled by converters, is fortunately becoming obsolete. Simulation tools are increasingly integrated into TCAD systems which serve a vast variety of users. Depending on the users' technical background and needs, different demands on such systems arise:

The *process* or *device engineer* uses simulation tools, or, more generally, TCAD applications and task-level procedures, to simulate a given process or

0026–2692/95/$7.00

device. For this purpose the engineer edits *task level* information (like manufacturing parameters, the process flow or optimization goals) and uses applications (point tools) which are provided and edited by *application engineers*. The application engineers in turn make use of the framework's global libraries and similar facilities to add new process and device simulation functionality and to maintain existing simulators (Fig. 1).

In contrast to application engineers, process and device engineers do not need to know about programming aspects and technical details of the framework. Ideally, they should perceive the framework (including the tools) as a 'virtual wafer fab'.

Finally, the framework (including all *generic*, i.e. framework-related, applications and services) are maintained by *framework engineers*.

1.2 The engineers' requirements

All three categories of engineers interact in some way with the user interface and the task level part of the framework, either in a 'user role' or in a 'programmer role'. Hence, the major difficulty for user interface and task level implementation (in contrast to the data level or other internal parts which are 'just' visible from the programmer's point of view) is the vast variety of interests and perspectives which must be considered.

- For casual users who seldom need to use simulation tools (for instance, to track down bugs in manufacturing), ease of use, robustness and continuity of the user interface properties are the most important features. Familiar visual elements should be employed where they are available.

- For device or process engineers who use TCAD tools more often, flexibility at the task level is the most crucial issue. It should be easy to define new, complex simulation tasks without having to bother with the internal workings of the TCAD system. Within the task level environment, the details of simulation sequences should not be simulator-dependent and should not, as is the case in shell-based solutions, depend upon the operating system.

- For specialists in physical modelling or numerical techniques, the most important features are openness of the system and a firm base for customization, calibration and extension. Full access to the simulation tools and models must be supported. Furthermore, it should be possible to integrate existing tools in a homogeneous user interface without having to redesign the tools.

- From the framework and application engineers' point of view, the use of a high level of abstraction is desirable, as it usually reduces the effort for using and maintaining the system.

- For software support groups, besides the points listed above, maintainability and portability are very important. This includes the use of open portable subsystems, since the entire system will be less portable than its least portable subsystem.

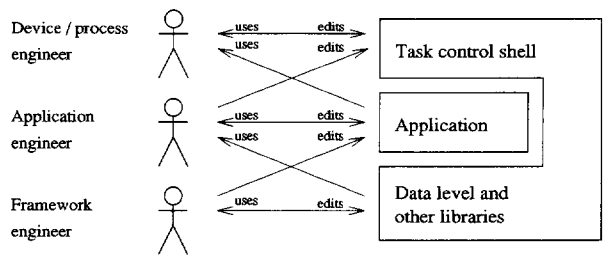

Fig. 1. Process, application and framework engineers interact with the TCAD system by both using (arrows pointing left) and modifying (arrows pointing right) the components of the system.

1.3 The applications' requirements

Seen from the applications' point of view, there are numerous requirements which the data level of a TCAD system must satisfy. A persistent simulation database is needed where simulation

problem descriptions, histories and results are stored. A clear, procedural interface must provide access to the simulation data, and should convey all physical and nonphysical information used by the applications. The interface must contain homogeneous language bindings for all programming languages which are commonly used to develop TCAD tools. Moreover, the interface must be sufficiently operating system- and machine-independent to achieve easy portability to different platforms.

The procedural interface must be easy to use, and must contain all foreseeable functions which arise in simulator coupling (like transformation of coordinate systems, conversion of physical units; it may provide high-level functions like interpolation or grid generation). It is desirable that these functions may be used in an opaque manner, so that the application engineer can concentrate on the actual task of the application. Fast random access to simulation data and compact database sizes are crucial for three-dimensional simulation, so these issues cannot be neglected when designing a procedural interface and data representation.

1.4 Maintenance and comprehensibility

The size of a typical classical single process or device simulation program lies in the range of one to two megabytes of source code. VISTA (as an example of a TCAD framework) currently requires more than ten megabytes of code in different implementation languages (predominantly C), not including simulators (Fig. 2).

The remarkable size of the code alone indicates that special care is required to ensure the comprehensibility and maintainability of the system, and that pure software issues become much more relevant than in the case of single simulation tools (regardless of their sophistication). It is indispensable that the basic structure and concepts of the system can be learned and understood with only a moderate expenditure of time and effort. Therefore, a major demand is

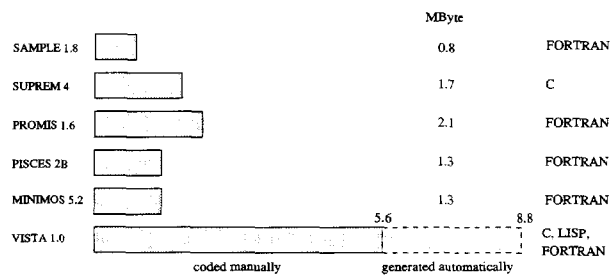

Fig. 2. Comparison of source code sizes.

that the design and implementation of all components adhere to a few simple and consistent basic concepts. We will see that these demands often provide the decisive argument in critical architectural choices.

2. Architecture

2.1 Review of existing approaches

Several workstation-based systems can be found which address the issue of multi-tool integration into a unified user interface, mostly on top of the X Window system. PRIDE [1], based on Sunview, exhibits a user interface and task level architecture which is strongly influenced by the preprocessing–computation–postprocessing task model of TCAD. SIMPL-IPX [2] directly uses Xlib, and features a central interactive graphical editor which has menu-oriented facilities for running simulators. In all the above cases, implicit or explicit assumptions about the design process have an impact on the (top-down) *design* of the software, and restrict design tasks which can be performed or implemented. A more flexible and extension-oriented user interface architecture has been accomplished in PROSE [3] from UC Berkeley and in the IBM's WIZARD user interface [4], which is mainly due to the use of the generic Tcl interpreter [5] and Tk toolkit [6]. The need for continuous extensions and for a concept for tool integration into a common framework has also been widely recognized [7, 8], but has hardly been addressed in existing TCAD systems.

Other well-known TCAD systems are an integrated system for statistical VLSI design from Hitachi [9], the MECCA system from AT&T [10], or the SATURN system from SIEMENS [11].

However, only few of these frameworks feature a data level for simulation data access. Most of the existing TCAD environments use data converters to couple simulators using different data formats. Doing this not only causes the number of converters needed to rise quadratically with the number of simulators present, but it also prevents the user from taking advantage of the services provided by a TCAD-oriented data level. Using a data level, simulators can be split up into separate tools of well-defined functionality, allowing tool developers to concentrate on their particular task.

Early implementations of data levels, like the DAMSEL system from CNS/CNET [12], feature two-dimensional geometries and simple data structures for easy usage by existing simulators. Among data levels designed for TCAD environments are the CDB/HCDB from CMU [13, 14] (with emphasis on horizontal design), the BPIF implementation from UC Berkeley [15], or the keyword based ASCII data level of IDDE [8]. Another data level built on PIF featuring object-orientedness is the PIF/Gestalt system from MIT [16].

A recent approach is the SWR 1.0 specification [17, 18] (issued by the *Semiconductor Wafer Representation* technical subcommittee of the *CAD Framework Initiative* (CFI), an international standardization committee for electronic CAD) which defines an object-oriented application interface for TCAD data access and suggests the use of a client-server framework architecture. The intriguing idea of this standard definition is to separate the physical modelling completely from tedious tasks such as grid generation, interpolation or geometry handling by providing these functions as a black-box server which is accessed by the simulation clients via a procedural interface. This method is very well-suited for, for instance, the simulation of topography formation; however, it can be detrimental to applications with high data throughput or applications which exhibit performance advantages thanks to a tight coupling between physical models and numerical techniques. Furthermore, the definition of a rather high-level interface – which implicitly requires a high degree of functionality – is, in combination with the opacity of a server concept, an impediment to gradual implementation and maintenance (there is no layering that would provide milestones for implementation and verification).

2.2 General guidelines

For the data level, architectural transparency is essential for gradual implementation and maintenance, and to accommodate possible framework architectures, like client-server, master-slave, parity, shared library, shared memory, or plain monolithic applications with file-based coupling. A well-balanced and thorough layering of the functionality and semantics of the data level implementation is most important, as this will allow considerable re-use, e.g. when an RPC-based client-server interface is introduced between two layers, or an object-oriented system is imposed on the top layer.

The user interface is challenged by the inherent semantic complexity and diversity of information flow between the user and the TCAD system, which is related to the broad physical background of process and device simulation. Unfortunately, there is no publicly available monolithic user interface toolkit which is flexible enough to meet the changing requirements, while simultaneously providing the specialized functionality to support TCAD information flow efficiently.

It must be admitted that the requirements described in the introduction do not readily suggest a specific design or architecture for the

TCAD framework. However, taking together the requirements and the lessons learned from the prominent examples above, some important general guidelines for data level, user interface, and task level can be derived:

- *Bottom-up* — as the very top TCAD problem and application is hard to narrow down (there is no 'generic design task' in TCAD), a *bottom-up* approach is favourable. This implies that not only implementation, but also definition and design of higher-level functionality and behaviour are shifted towards the end of the development phase.

- *Layering* — where possible, implementation should be done in distinct layers of increasing functionality and abstraction.

- *Separation and orthogonality* — the (firm) framework code should be clearly distinguished from (volatile) TCAD applications. Duplication of functionality should be avoided.

- *Consistency* — where possible, the generalization of existing concepts within the framework should be favoured over the introduction of new ones.

- *Interpret design tasks* — the need for defining and using design task macros in a flexible (non-taxonomical) manner suggests a requirement for full programming capabilities, the task level programs being executed by a task level *interpreter*.

2.3 The VISTA architecture

The *Viennese Integrated System for Technology-CAD Applications* [19] is an *integration* and *development* framework for process and device simulation tools. Its development was lead by the guidelines stated above. VISTA consists of a *data level* part which provides a common library for accessing and manipulating simulation data [20], a set of utilities for visualization and high-level data manipulation, a user interface, and an interactive shell [21] which integrate all services (including the simulation tools) on the *task level*.

The structure and implementation details of data level, user interface, and task level will be described in detail in Sections 3, 4 and 5, respectively. We have found that comprehensibility and (mutual) consistency of these three parts are extremely crucial. This is why they must not be considered as isolated problems and implementations, but as parts of a whole problem and implementation.

3. Data level implementation

For the data level, we have decided to start from the *Profile Interchange Format* (PIF), as initially proposed by Duvall [22], and to extend and modify it to meet the requirements stated earlier. A binary implementation and a procedural interface with different levels of functionality is the optimal solution with respect to the previous considerations. Since there is no public and efficient implementation available, we had the opportunity to implement the application interface from scratch.

The procedural interface to the database services is called *PIF Application Interface* (PAI) [23]. It makes extensive use of automatic code generation to achieve platform independence and to generate the individual language interfaces.

3.1 The VISTA PIF implementation

The ASCII version of the PIF is used as an intersite data exchange format. The binary form [23] is used as the database storage format of the data level. Figure 3 shows the logical PIF structure with corresponding object relationships. Note that the majority of the simulation information is carried in the grey shaded *geometry*, *grid* and *attribute* constructs, while the *objectGroup* and *meta* objects are important extensions for TCAD-related data. Both the *geometry* and the *grid* constructs are built out of primitive

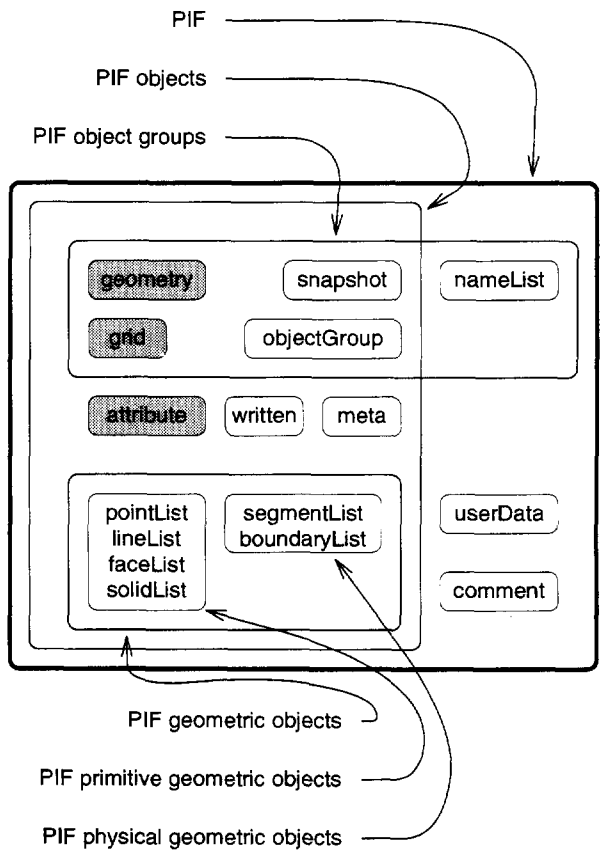

Fig. 3. The logical PIF structure.

geometric objects (points, lines, faces and solids). The *geometry* construct additionally holds a simulator's view of a simulation geometry through *segmentList* and *boundaryList* constructs.

The rather generic *attribute* construct is used for attaching *any* kind of information to an object in an orthogonal way. The *attributeType* subconstruct describes the meaning (semantics) of an attribute. The *valueType* subconstruct declares the type of the attribute values, which can either be a scalar type (string, character, integer, floating point) or nested vectors or arrays of scalar types (note that arbitrary type complexity with clearly defined semantics can be implemented without requiring any syntax extensions).

There is no artificial conceptual distinction between lumped attributes (like simple descriptive strings defining the material of a geometric segment) or fields (like vector quantities defined over multidimensional grids), which is another milestone towards a clearly structured architecture and a simple implementation. Figure 4 shows a *materialType* attribute defined over a segment, and Fig. 5 shows an *electricField* attribute defined over a three-dimensional grid.

The binary format of the PIF uses LISP-conform *constructor nodes* (*CONS* nodes) to map all ASCII PIF expressions to binary form. To improve performance and data compactness, several additional features have been added, such as a symbol hash table for fast object access by name and a compressed array storage format for large arrays which typically occur in TCAD applications for attributes on grids.

It is important to note that the PAI defines the interface for applications, thereby hiding the PIF syntax from the applications. Although the low-level PAI routines are strongly related to the PIF

```
(attribute geometry_attribute
   (attributeType "MaterialType")
   (nameList (ref my_segments (valueList 1)))
   (valueType asciiString)
   (valueList "Silicon")
)
```

Fig. 4. Attribute defined on a segment.

```
(attribute grid_attribute
   (attributeType "ElectricField")
   (nameList (ref my_grid))
   (valueType (vector 3 real))
   (valueList        1.2 3.4 6.5
       4.4 3.5 4.7
       .....)
)
```

Fig. 5. Attribute defined on a grid.

syntax, as the TCAD application sees just the surface of the PAI procedural interface, they have to know only little about the PIF.

3.2 Implementation of PIF application interface

The PAI is split into seven distinct layers with strict interfaces between each other. The different layers are shown in Fig. 6.

Each layer calls only functions of the next lower layer. This leads to modules with properly separated functionality. Each layer is responsible for a unique storage concept of the whole *PIF Binary File* (PBF), with increasing functionality and abstraction towards the upper layers. The application interface works on PBFs (intertool format); for data exchange with other hosts there is the PIF ASCII form (intersite format). To convert PIF files between these two formats there is the *PIF binary file manager* (see Section 3.4), implemented as a separate PIF tool on top of the PAI.

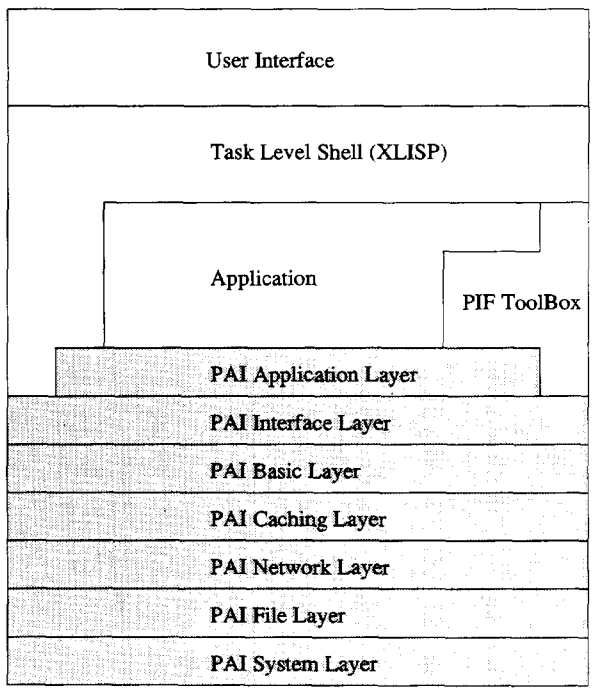

Fig. 6. Layout of the PIF application interface.

The PAI is able to handle simulation data in three geometric and infinite nongeometric dimensions. Thus it is possible to read and write distributed attributes ranging from scalar to N-order tensor values on one- to three-dimensional grids. All PIF objects can be selectively and directly accessed with the PAI, either by handle or by name. The PAI will read only the necessary parts of a PBF into a cache avoiding performance drawbacks of most file-based systems.

3.2.1 System layer

This lowest layer of the PAI is the link to the operating system, and defines simple access routines to the file input and output services. In ANSI C only the buffered file I/O is defined and standardized, but buffering is not needed by the PAI since this is done in the caching layer above. This is the only layer which has system dependent functions, and implements also basic functions for network access (TCP/IP and DECnet).

3.2.2 File layer

The standardized file I/O functions of the system layer are used by the file layer to handle the physical I/O of PBFs. It guarantees that a PBF is only opened by one application at a time for writing (file locking). It inhibits multiple write accesses to one PBF, but allows multiple read-only accesses.

3.2.3 Network layer

This optional layer is functionally equivalent to the file layer but allows instead accesses to PBFs over the network. To minimize network traffic, the functions of the file layer are used directly for all local and temporary PBFs. The network databases are accessed through a database server as shown in Fig. 7, which opens, reads, writes and closes PBFs.

3.2.4 Caching layer

This layer buffers I/O data to minimize disk and network accesses. Depending on the application,

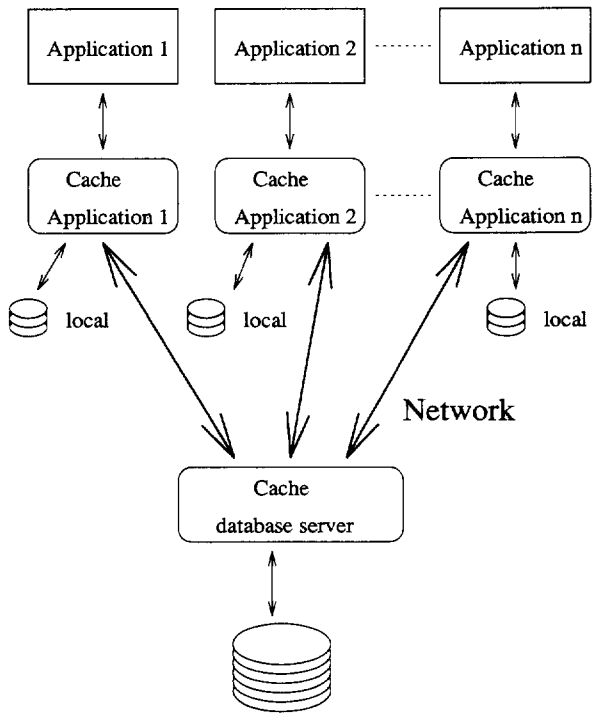

Fig. 7. Local and network storage.

the size of this buffer can vary from a few hundred kilobytes to several megabytes. The advantage of the cache is that data requested by read operations frequently can be found in the cache, while write operations can be delayed until closing of the file, depending on the page size and total cache sizes and on the page replacement algorithm. The runtime option of unbuffered write operations ensures consistency of the PBF during update operations, and allows one to examine a PBF while a tool is running and to write to it, which is an invaluable help in debugging simulators.

3.2.5 Basic layer

This layer is the lowest to implement structured data nodes. It provides a functional interface for binary storage of full LISP syntax expressions by presenting the notion of atoms (primitive data items like a number, character or string value) and constructor nodes (*CONS* nodes for list creation) to the upper layers, as described in [24]. In contrast to LISP memory storage concepts, all nodes of the basic layer (and hence the PIF Application Interface) are originally kept on file and are just cached through the cachine layer.

3.2.6 Interface layer

This layer is the implementation of the PIF syntax, providing administrative, access and inquiry functions. Through this layer, only *syntactically valid* (binary) PIF files can be read and written (simply speaking, the proper nesting of PIF constructs is ensured), and all named PIF objects (like points, lines, faces, attributes) can be inquired by their name or by handle. As the procedural interface is directly related to the PIF syntax, the whole interface layer can be generated automatically from an abstract syntax description during the build process of VISTA.

3.2.7 Application layer and language bindings

The application layer is the most important, primary interface for applications to read and write simulation data in binary PIF. Many *semantic* rules are implemented in the application layer (in contrast to the interface layer which only checks the *syntax*), ensuring interoperability of applications in the VISTA framework. High-level functionality and data-manipulation services are provided to relieve TCAD tools from tedious 'everyday' work.

FORTRAN interface

As the application layer is coded in C and a considerable number of proven simulators are coded in FORTRAN, language bindings for most application layer functions and all inquiry functions to FORTRAN are provided. This binding is strongly dependent on the respective compilers, since there is no broad standard for passing parameters from FORTRAN to C. Fortunately, all required language binding code is deducible from a formal description, and is hence generated automatically during the build

phase of VISTA. Adding a new binding for another platform requires only few additions in the configuration files.

LISP interface

The LISP interface of the PAI is built directly on top of the interface layer (bypassing the application layer) to exploit the advantages of PIF/LISP conformity. Syntactically, PIF is a subset of LISP and a direct binding to the interface layer enables the use of all the built-in symbolic manipulation and list manipulation features of the LISP interpreter to access and modify PIF data.

3.3 Use of the PAI

The short code example in Fig. 8 shows the C calls to generate an example of a PIF data structure. *namelist* is the handle to the parent *nameList* object [25]. The two element array *points* holds the indices of the points on which the line is created. In the Application Layer code example, Fig. 9, this part of information is generated by the function *palWriteLineList1*. In addition to the reference construct this function generates the whole *lineList* construct, as can be seen in Fig. 10.

3.4 PIF binary file manager

As mentioned above, the whole PAI works on the binary representation of the data for fast access. For data exchange via eMail or FTP, or for making binary PIF files human-readable, there is the ASCII PIF representation holding the same information. The *PIF Binary File Manager* (PBFM) is able to convert the binary to ASCII PIF, and *vice versa*. Thus data exchange between different hardware platforms is possible by converting PLBs to ASCII PIF on one platform, transferring the ASCII PIF file, and converting back to the binary format on another platform.

```
{
    /* local variables */
    paiObject valuelist, ref;
    paiLong points[2];

    points[0] = 1;
    points[1] = 2;
    ref = pilCreateRef(
          namelist,              /* parent nameList construct */
          pointlist,             /* referenced pointList P */
          pilCREATE_NESTED);     /* create a new reference construct */
    valuelist = pilCreateValueList(
          ref,                   /* parent ref construct */
          pilDATA_INTEGER,       /* data type is integer */
          points,                /* data points indices */
          0, 2,                  /* which values to write */
          pilCREATE_NESTED);     /* create a new valueList construct */
}
```

Fig. 8. Interface layer code example.

3.5 Semantic issues and high-level functionality

The design of the PIF syntax facilitates future extensions without syntax changes, but unavoidably introduces ambiguity. The PIF itself, be it its ASCII or binary form, only defines a general syntax for expressing physical situations. Ambiguities arise from the possible multilateral descriptions of the same physical problem in terms of PIF syntax. Most of these ambiguities are, however, physical modelling. To unambiguously interpret PIF data, there are additional semantic constraints which applications have to adhere to (at the cost of some PIF flexibility), and ambiguity resolution mechanisms built into the application interface.

```
{
    /* local variables */
    paiObject linelist;
    paiLong endindices[1];
    paiLong objdx[2];

    endindices[0] = 2;
    objdx[0] = 1;
    objdx[1] = 2;
    linelist = palWriteLineList1(
          parent,          /* handle to PIF file */
          "myLine",        /* name of the lineList */
          1,               /* number of lines */
          endindices,      /* endindices of the lines */
          2,               /* number of used points */
          pointlist,       /* handle to referenced pointlist P */
          objdx,           /* point indices */
          palCREATE_NEW)   /* create a new lineList */
}
```

Fig. 9. Application layer code example.

```
(lineList "myLine"
      (nameList (ref P (valueList 1 2))))
```

Fig. 10. PIF construct produced by example code.

There are sometimes severe data conflicts and gaps to fill between simulators (e.g. a simulator working on an unstructured grid coupled to a simulator using a tensor product grid) which are dealt with in the PIF ToolBox, comprised of generic PIF tools such as grid generators, interpolators and attribute and geometry manipulators. These services are available as high-level libraries to be used in applications as well as stand-alone tools on the task level.

3.6 Grid support

A particularly challenging problem is the support of the innumerable different grid types in use today. A distinction between tensor product and unstructured grids has been made, because we did not want to lose an orthogonal grid's unique features by decomposing it into rectangles/cuboids.

For general (unstructured) element-based grids an object-oriented approach has been taken for dynamically adding new element and grid types by just providing a unique name, an interpolation and a decomposition function. Using automatic code generation, these object methods are threaded into generic PIF ToolBox functions, thus adding support for the new grid element to the whole framework. Figure 11 shows some example elements and how they are referenced in a PIF grid. After recompiling the PAL, the new element type is known to applications through a unique constant identifier.

A tremendous advantage is that applications need not take care of new element types: reading attributes defined over a grid can be done without knowledge of the grid, since there is a *generic interpolation facility*, based on the element definitions, which is automatically invoked when requesting an attribute value at a location (x, y).

3.7 Performance evaluation

Besides the goals of classical intertool PIF implementations featuring object-orientedness (PIF/Gestalt, [16]) or suitability for TCAD

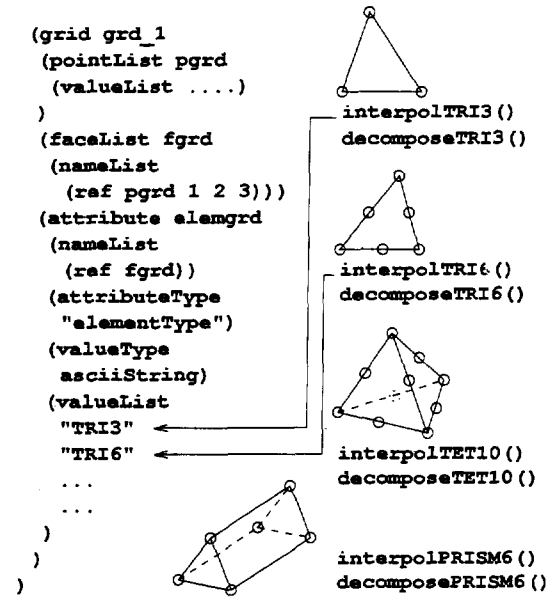

Fig. 11. Support for unstructured grids.

environments (BPIF, [15]) our implementation stresses efficiency in terms of runtime performance and database compactness. Thus, writing and reading 10,000 points (in three-dimensional space) of a PIF *pointList* takes 0·51 and 0·66 seconds (real time), respectively, on a DECstation 3100; the database written is 250 kB in size. Therefore, linking a TCAD application to the VISTA environment is not a performance issue. In contrast to a client-server approach, the administrative and communication overhead is negligible for any application consuming a few seconds of CPU time – the commonly used argument, that PIF is not practical because of its low runtime performance, no longer holds true.

4. Implementation of VISTA user interface

4.1 Structure

The X Toolkit [26] is ideally suited for a very flexible and highly systematic user interface architecture. From a set of building blocks (so-called *widgets*), all higher-level functions and applications can be built. The required speciali-

zation for TCAD can be achieved by both implementing TCAD-specific widgets, or by assembling multiple widgets into TCAD-specific user interface components. The object-orientedness of the X Toolkit coincides with the desired bottom-up approach, and is very well suited for future extensions. Nevertheless, a widget set has to be chosen, from which the required specific widgets for TCAD purposes can be subclassed.

The basic structure of the VISTA user interface is shown in Fig. 12. Basic, generic functionality is provided by the generic *Athena widgets* which are part of the original MIT X11 distribution [26]. We have decided to use this widget set rather than any other open standard for several reasons, but mainly because a migration from these generic widgets to another widget set (like OSF/Motif [27] or Open Look) is significantly easier than *vice versa*. The *Athena* widget set was not intended to be sufficient for all purposes, and thus does not fulfill all the needs of a TCAD user interface, but it does provide the required generic functionality, it is highly portable, it is available on virtually every modern workstation platform, and it is easy to comprehend. In addition specialized VISTA widgets have been developed for supporting TCAD-related information flow. The VISTA widgets are also created and accessed via specific functions, so that they can quite easily be replaced by other widgets, should the need arise.

For the sake of widget-set independence, a wrapping layer has been put on top of the widget sets. All widgets are created and modified via specific functions rather than via the generic interface of the X Toolkit, which should facilitate future migrations.

The top layer, the VUI (*VISTA User Interface*) library, provides some often needed higher-level operations, and contains most of the user interface *policy* which is shared among VISTA applications. This library takes care that different parts

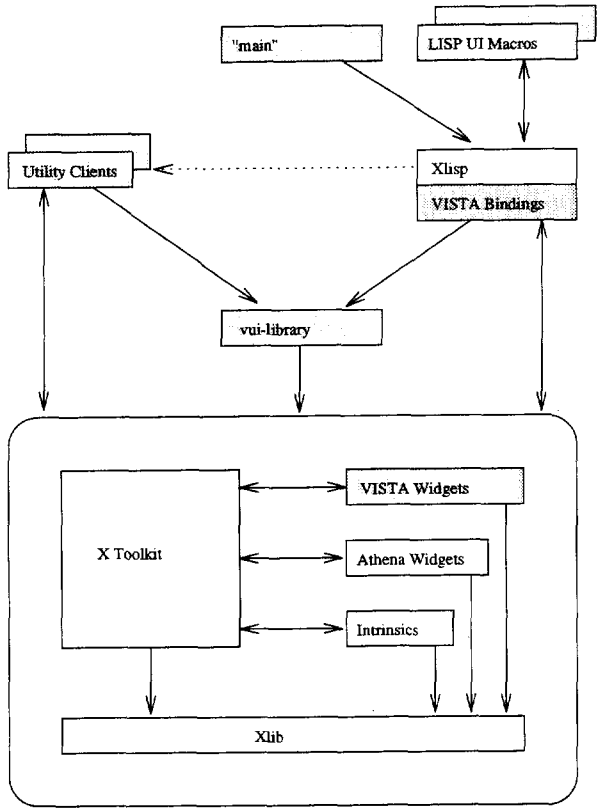

Fig. 12. Structure of the VISTA user interface. Shaded boxes represent extensions to the public domain products XLISP and the MIT X Window system. The arrows indicate the sequence of function calls between different parts of the user interface.

of VISTA look alike and behave similarly. Interactive applications (like visualization clients or the device editor) have their own VUI-based user interface, whereas applications requiring no user interaction (batch-mode tools like simulators or data converters) are provided with a front-end user interface which is executed by the XLISP interpreter.

4.2 The VISTA widget set

The VISTA widgets are subclassed from either X Toolkit Intrinsics or Athena widgets (Fig. 13). The *Canvas*, *PED*, *PedGraph* and *Ruler* are parts of the interactive PIF Editor (PED); the *IntValue*,

RealValue, *TextLine* and *FileSel* are widgets for the specification of integer, real and string values, and files, respectively; and the *SVGraph* widget is a widget for displaying simple vector graphics plots.

4.2.1 The PIF editor

A data level implementation would be incomplete without an interactive graphical editor for manipulating the geometrical data (device geometries) stored in the binary PIF. The *PIF editor* (PED) is the front-end user interface for the interactive creation and modification of geometrical data in one, two and three spatial dimensions and of all attributes (like the material type) which define the device structures (see Fig. 14).

The PED makes use of the *Canvas*, *Ruler* and *PedGraph* widgets, and is itself implemented as a widget (see Fig. 14). This allows the use of multiple subwindows for editing one and the same device geometry, editing of several logical PIF files in one PED process, and even using the PED as a component in 'surrounding' applications. Thus, arbitrary additional menus or other widgets can be added without interfering with the PED itself.

The PED can work on all PIF files independent of the specific semantical contents. It is a generic tool for building a simulator input PIF file from scratch, for modifying existing device structures, and for visualizing geometric PIF information.

The hierarchical PIF geometry structure is supported by 'snapping' on existing lower-level geometrical objects, and by automatically creating missing intermediate-level objects during input. Common techniques like background grid or colouring according to different physical or logical criteria are used to facilitate the comprehension and assimilation of the spatial information.

4.2.2 Vector graphics widget

The XToolkit and Athena widget set do not provide 'classical' two-dimensional vector graphics capabilities, which are a firm requirement for any CAD discipline. To support platform-independent vector graphics output we have implemented a minimum-functionality vector graphics widget (Fig. 15) which is built directly on the generic Xlib and XToolkit.

The widget remembers all drawing commands and provides zoom and pan functions for the user, which henceforth the programmer does not need to bother with. Callbacks can be utilized for example to digitize data points. This widget is used as interactive back-end of VISTA's visualization library.

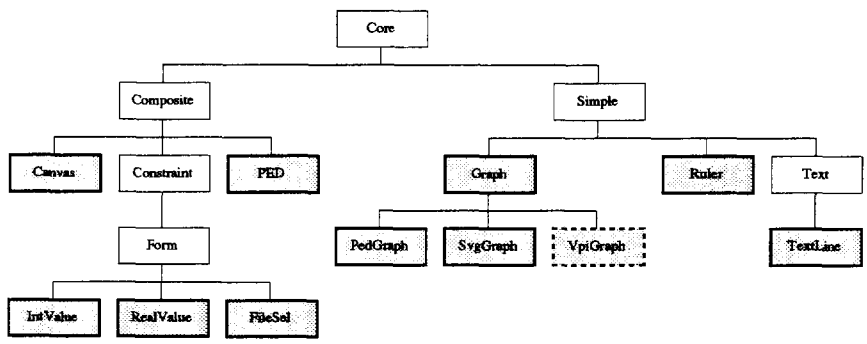

Fig. 13. Widget set used by VISTA. The VISTA extensions are shaded, the Intrinsics and Athena widgets used for subclassing are blank.

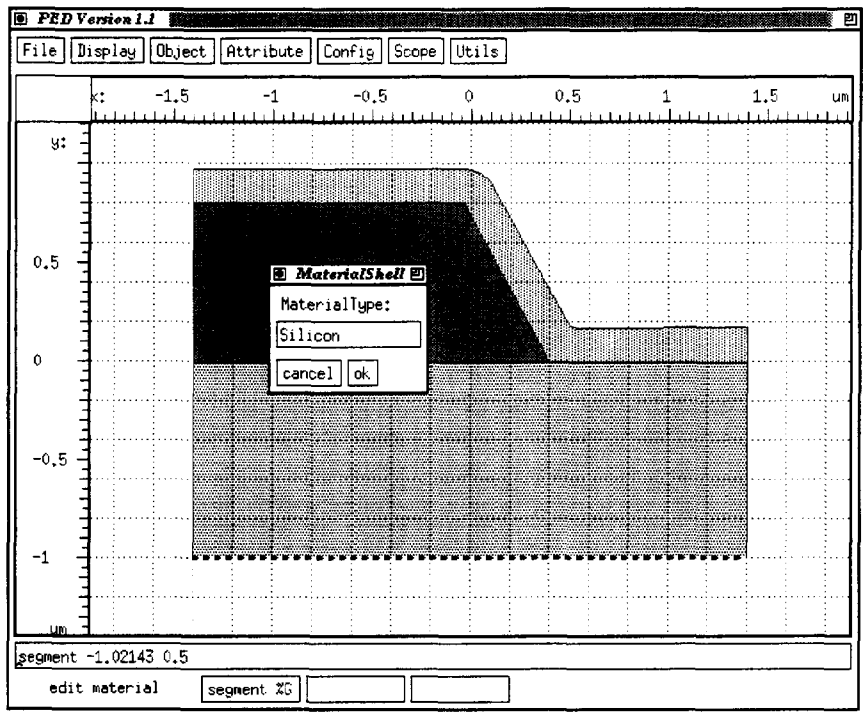

Fig. 14. The PIF editor widget.

The contents of the widget (i.e. the 'plot') can be converted to PostScript format; other converters can be added easily due to the compact set of drawing commands which is used.

4.2.3 File selection

As it is not provided with the Athena widget set, we have implemented an advanced file selection widget (see Fig. 16), which allows operating system transparent specification of files (including a *GNU Emacs* [28] like filename completion) using a string subwidget, and operating system independent traversal of the directory tree and selection of existing files using list subwidgets.

The selection of logical PIF files (one physical file can contain multiple logical PIF files) is implemented as a so-called *widget macro*.

4.3 The VUI library

4.3.1 Widget macros

The VUI library contains functions which create often-used combinations of several widgets in one step. The widgets are positioned and all required connections and callback functions are defined. These widget macros behave as if they were single widgets (composed of several sub-widgets). This approach is similar to the OSF/Motif 'Convenience Function' concept [27], and helps to maintain a unified appearance for different VISTA applications.

The following examples show some TCAD-specific widget macros.

The periodic table shown in Fig. 17 is implemented in C and is used by applications to let the user select 'pure element' materials from a material database which is shared by all

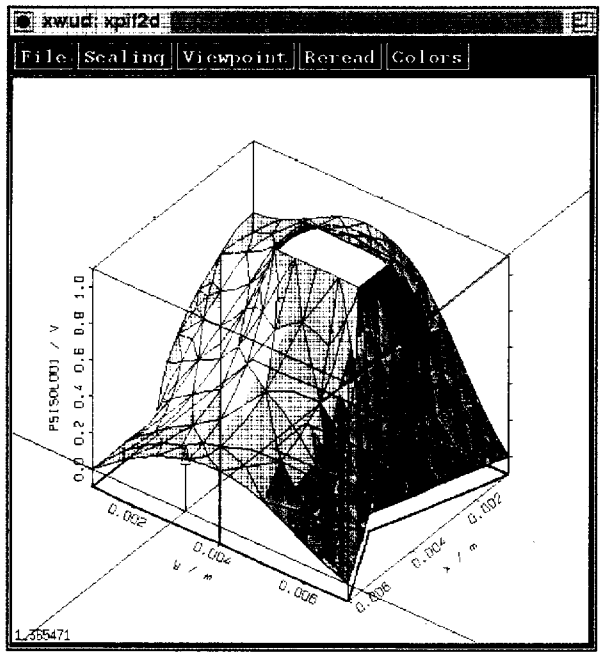

Fig. 15. Vector graphics display widget is used for displaying the output of PIF-based visualization tools.

simulation tools. It provides a familiar method for the identification and specification of single chemical elements. This widget macro is used, for example, to ask the user for the bombarding ion species for the Monte Carlo simulation of ion implantation.

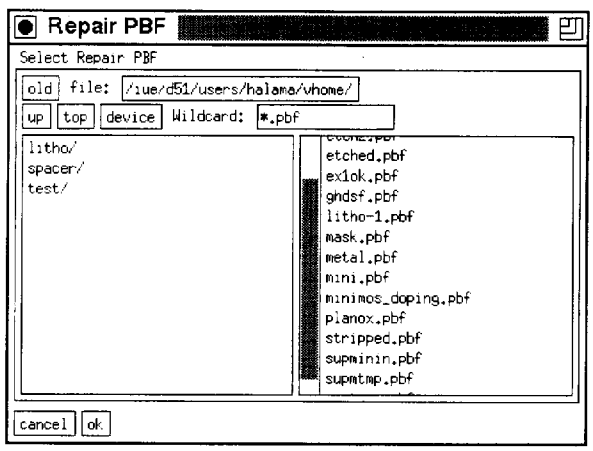

Fig. 16. VISTA file selection widget.

The symbolic PIF browser widget macro (see Fig. 18) is generated by a prototype LISP program. It is a generic intuitive facility on the task level for the selection of PIF objects and represents the hierarchical structure of the PIF file in iconic form. It can be used in any step which requires the specification of one or more PIF objects, like for example visualization (choice of attribute to be visualized), inquiry or post-processing operations.

4.3.2 Tool control panels

Applications which don't require user interaction (*batch mode* programs like most classical simulators) can very easily be provided with a 'supply parameters and run' user interface. As an example of such a widget macro, the tool control panel for the Monte Carlo simulation of ion implantation is shown in Fig. 19. This panel is created from a formal specification of the tool and its parameters by an automatic interface generator, which is implemented in LISP. This high-level user interface tool, in most cases, relieves the application engineer from the need to use XToolkit programming to make new tool control panels.

5. Implementation of the task level shell

5.1 XLISP

At a first glance, a proper choice for the task level environment seems to be non-trivial (there are many options to choose from), but it becomes almost obvious when the proposed architectural guidelines and requirements are considered. A UNIX- (or any other operating system) shell-based solution does not fulfil the portability requirement, whereas the use of an integrating master application (like an interactive device editor) alone does not offer the desired extension (programming-) language features. Although a Tcl/Tk based solution is a very common and promising option [4, 14, 29, 30], it does not exhibit the important features (especially with respect to the architectural guidelines) that our choice has.

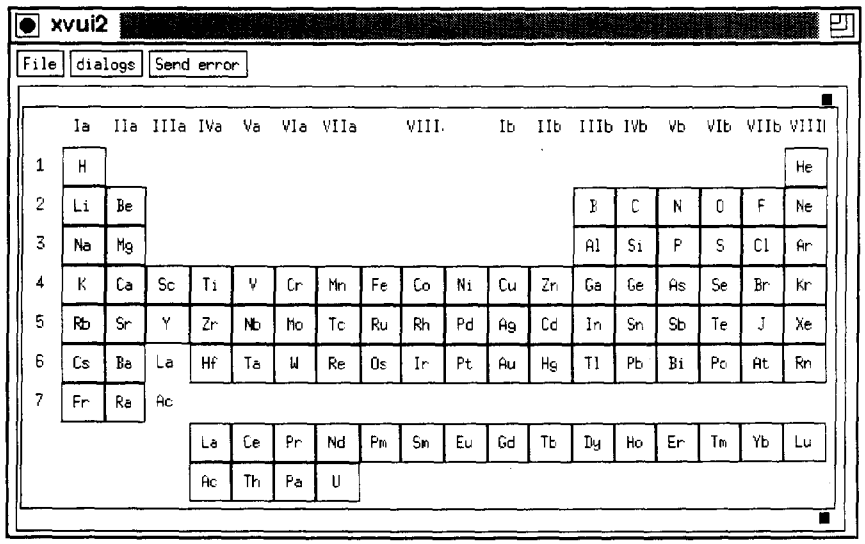

Fig. 17. Periodic table widget macro lets the user select chemical elements as bulk or implantation material.

The VISTA task level is built on XLISP [31], a public domain LISP interpreter, which is available in source code and meets all demands. It is coded in highly portable and comprehensible C code, fulfils all software requirements and provides full programming capabilities. It can be extended and customized for TCAD purposes by both adding C-coded primitives or by loading LISP code at runtime.

Most importantly, LISP is a superset of PIF, so that architectural homogeneity of the TCAD system is strengthened by this choice. The conceptual common denominator between LISP, the PIF (both binary and ASCII), and the PAI is so significant that the net effort for the programmer to comprehend the entire system is considerably decreased. Additional synergetic features are the ability to store task-level LISP expressions on the data level (by using the PAI basic layer) and to manipulate PIF information directly on the task level.

There are several other remarkable LISP-based implementations of task level environments in related fields, which lets us believe that XLISP is a sound choice, like the well-known *GNU Emacs* [28] text editor, the generic CAD system *AutoCAD* [32], or *Winterp* [33] ('Widget *Interp*reter'), which comes with the MIT X11 distribution.

The applicability of XLISP for the execution of typical TCAD optimization tasks has already been demonstrated [34, 35]. The programming language features offered by LISP are an extremely powerful (LISP is, for example, among the most often used implementation languages for expert systems) and an efficient basis for carrying out complex task flows, like nested optimization loops. The concept of our task level environment is to present whole simulation tools as single LISP functions, so that the XLISP interpreter can be used for programming with simulator primitives on the task level.

5.1.1 Interaction of framework components

Remembering the casual user, we recognize that a good link between the TCAD extension language interpreter, which integrates all system components on the task level and represents the *main program* of the TCAD system, and the user interface is required in a way that the existing

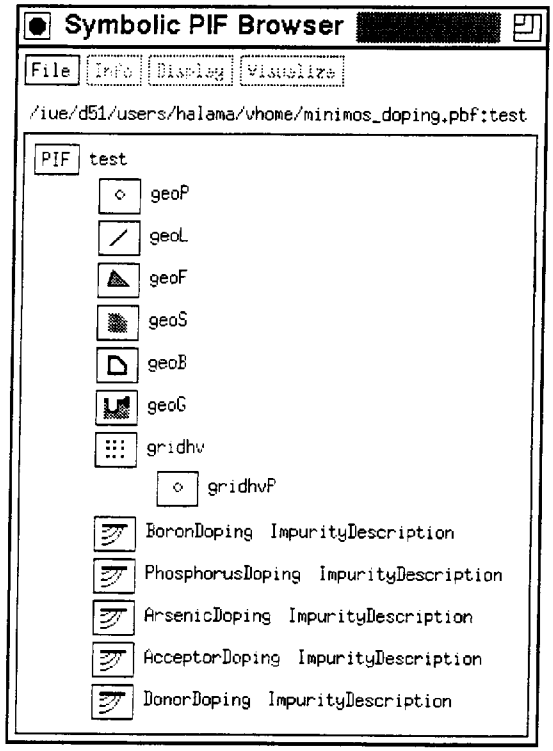

Fig. 18. The symbolic PIF browser widget macro is created according to the data contained in a (binary) logical PIF file and reflects its hierarchical structure.

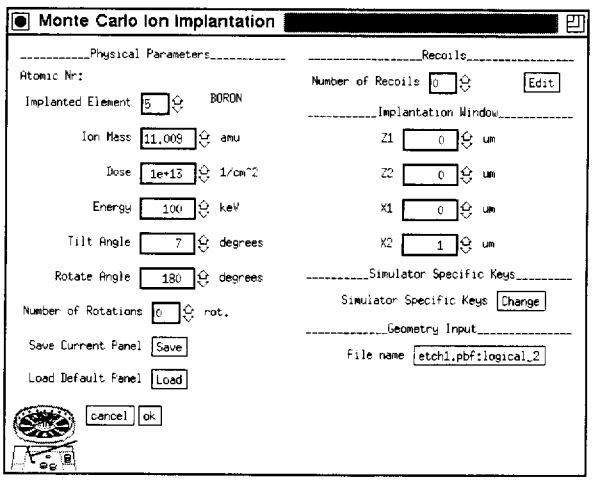

Fig. 19. This widget macro is the user interface for the PROMIS Monte Carlo ion implantation module.

interpreter is simultaneously used for all interpreted user interface parts.

But there are also other software components which need to be accessible from within the extension language environment, so that a *generic method* for linking C-coded functions to XLISP is highly desirable. External simulator executables need to be started, provided with appropriate input and their termination needs to be recognized to trigger subsequent simulation steps.

To preserve the consistency and simplicity of XLISP and to provide a homogeneous procedural interface and programming environment, we had to implement the X Window interface (*VISTA UI Bindings* in Fig. 12) for XLISP from scratch rather than using the *Winterp* [33] implementation. As there are other C-coded parts of the framework which need to be accessible on the extension language level, a generic, automatic method for linking given functions with the XLISP interpreter has been implemented.

The code required to implement LISP interfaces for framework modules which are relevant for TCAD purposes is currently 335 kB (see Table 1), but is steadily increasing. There are only a few functions (indicated by parentheses in Table 1) which are manually bound to the interpreter for reasons of consistency and efficiency (see also Section 3.2.7). The vast majority of the interface code is generated automatically during the build phase of VISTA.

5.2 The callback concept

The object-oriented callback concept of the XToolkit may be generalized in a very straightforward manner, and successfully applied to those parts of the TCAD framework where a proper decoupling and high flexibility of the control flow is desirable. It is obvious that this is of special value for a flexible task level implementation.

TABLE 1 Number of functions (N_f), number of constants (N_c), size of code for the XLISP interface, and module size of every module that is linked with the XLISP interpreter. Numbers in parentheses indicate manual binding; all other code is generated automatically

Module	Description	N_f	N_c	Binding	Code size
xvw	Extended Widget Set	105	19	115 kB	531 kB
vui	User Interface Library	83	3	96 kB	254 kB
ve	Global Error System	14	37	20 kB	165 kB
svg	Graphics Library	22	10	21 kB	73 kB
ptb	PIF Toolbox	8	29	26 kB	190 kB
ver	Version Control	0	7	1 kB	0 kB
vos	OS Interface	(63)	12	(2) kB	214 kB
pai	PIF Application Interface	(20)	(117)	(54) kB	2256 kB
	Total	315	234	335 kB	3683 kB

Events coming from the X Window system are passed to the XLISP interpreter. If a LISP expression was associated with the activated widget at creation time, this expression is evaluated by the interpreter and can be used to change parameter values, trigger other events like the execution of a simulator, or start the evaluation of a LISP program or any other tool.

The same callback concept is also used for the control of simulator execution. If a simulation tool terminates, it signals the termination to the parent process, which again causes an associated callback expression to be evaluated. Callbacks can be triggered by the user interface, error handler, network layer, timer, or by termination of child processes.

Using the standard prototype

void callBackFunc(object_identifier, client_data, call_data);

for callback functions (which is already specified by the X Toolkit) it is possible to use one unified consistent method for various purposes throughout the framework, thereby gaining simplicity and flexibility.

5.3 Simulation flow control

Using LISP as prototyping environment, a *Simulation Flow Control* (SFC) module [36] has been developed which is able to simulate process step sequences (process flows) automatically. It is responsible for the definition and editing of simulation sequences, for the automatic execution of these sequences, for all aspects of tool invocation, and for the storage and retrieval of computed results. As the interfaces between all tools are well defined on the data level as well as on the tool control level, it is possible to replace any simulator call in a simulation sequence in a plug-and-play fashion. A prototype interactive graphical flow editor (see Fig. 20) has been implemented (also using LISP) which provides the user with visual programming capabilities for the definition and modification of the simulation flow. To simplify the *a posteriori* analysis of simulation runs, all intermediate results may remain available as PIF files.

6. The VISTA tool set

6.1 Simulators

The simulators MINIMOS 6 [37] (for two-dimensional simulation of MOS transistors) and PROMIS 2.0 [38] (a two-dimensional process

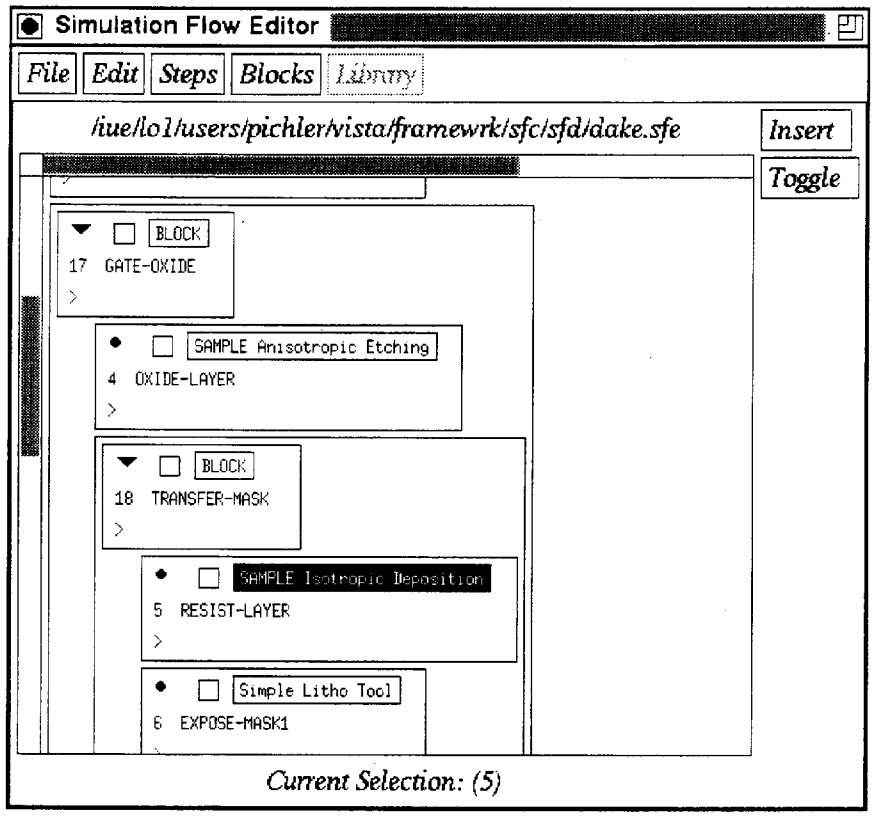

Fig. 20. The simulation flow editor – a prototype visual programming interface for the simulation flow control (SFC) module.

simulator for ion implantation and diffusion) have been integrated into VISTA. Both were already existing in-house applications, coded in FORTRAN. MINIMOS is integrated using a wrapper approach, whereas PROMIS has been modularized and makes direct use of the PAI. Similarly, VLSICAP, a two-dimensional finite element program (in FORTRAN) for capacitance computation and interconnect characterization, has been integrated into VISTA, again making use of the PAI.

SAMPLE (coded in FORTRAN, developed at UC Berkeley) has been integrated for the two-dimensional simulation of etching and deposition. In a wrapper approach (which is always to be used when the source code of the tool is not available), we had to make use of separate pre- and postprocessing modules to overcome some limitations of SAMPLE.

Using the VISTA framework as a *development platform*, a new rigorous two- and three-dimensional simulation tool for etching and deposition [39], and a three-dimensional Monte Carlo implantation simulator [40] have been implemented at TU Vienna.

It should be noted that many present *developments* of simulation tools are in C, but a major part of the integrated simulators *in use* is coded in FORTRAN, which verifies the importance of a dedicated FORTRAN data level interface.

6.2 Framework tools

VISTA presents some of its services as distinct tools (separate executables that read or write PIF files) which are invoked from the task level shell. These tools are either interactive (like the PIF editor, the visualization tools, or the material database browser), or batch-mode tools (like a general-purpose two-dimensional grid generator, or all converters required for wrapping approaches). All of these tools make full use of the PAI or other high-level libraries and, in the case of interactive tools, of the user interface library. As a considerable part of all new tools is comprised by VISTA's libraries, code re-use and features like shared libraries are fully exploited.

Recently, a bidirectional data link to TMA's Technology CAD system [41] has been developed at National Semiconductor. It converts binary PIF to TMA's 'Technology Interchange Format' and *vice versa*, making use of VISTA's high-level data manipulation services. This data link allows the use of popular TCAD tools like SUPREM-3, SUPREM-4 and PISCES within the VISTA framework.

7. Conclusion and future aspects

It is the scope of methods and unpredictable requirements that makes technology computer-aided design a challenging discipline. It is this property that dictates the rigorousness of a future-oriented TCAD framework. The need for comprehensibility, on the other hand, prohibits the (otherwise feasible) creation of a framework by combining existing solutions.

In VISTA, we have achieved a homogeneous and comprehensible architecture by favouring the generalization of existing concepts over the introduction of new (maybe even better suited) partial solutions. A bottom-up *design* has been used, wherever applicable, to be prepared for unforeseeable future requirements.

Using PIF as the interchange format of the data level was the initial choice, which was motivated by its intrinsic flexibility and open-endedness. However, the *crucial* part of the data level is an efficient application interface like the VISTA PAI.

One of the major reasons for using the XToolkit to implement the specialized user interface functionality as widgets is that it provides a *clear concept* for re-use and for future extensions. Finally, the well-known advantages [5, 6, 33] of an interpretive language for composing a user interface from widget-level building blocks and the consistency with the data level implementation have verified that XLISP is the right choice for VISTA's extension language interpreter. The generalized use as 'main' program for the task level, as essential part of the user interface, and as central facility for CASE-related tasks, contributes significantly to the consistency, maintainability and simplicity of the system. Automatic code generation (also using XLISP) helps to raise the level of abstraction on which problems like language bindings are solved.

7.1 Tool abstraction

The use of high-level tool abstraction methods for CAD tool management has been demonstrated through the Cadwell design framework [42]. We believe that a generalized and unified concept for the abstract characterization of tools (even on a low level, down to single functions) is highly desirable, as it can be employed for many different language bindings. Right now, from an abstract tool description in LISP syntax, several pieces of interface code (generating a main program which takes care of argument-passing for a given function, thereby forming a stand-alone executable, for instance) can be generated automatically. The same tool abstraction could in the future also be used to describe a tool for optimizers or other sequencing or analysis tools.

7.2 Visual programming interfaces

Visual programming capabilities are very valuable for the efficient support of any user. Almost

every non-trivial task in TCAD is to a considerable extent data-flow oriented. The whole task is defined by the arrangement of modules and the flow of data between them, which also implies the sequence of tool execution. A prototype of a simulation flow editor (see Fig. 20) has been implemented, which will yield feedback for further work in this field.

Again, both the callback concept (for module activation) and the tool abstraction concept (for module description) can be used for the implementation of a final visual programming *widget*, thereby preserving system simplicity from the programmer's point of view. A generalized visual programming facility is especially desirable for building specific applications through intuitive assembly of several generic functions or modules. We have tried to implement all new parts of VISTA (especially the visualization and the PIF ToolBox functions) using *generic* and *orthogonal* approaches. Visual programming will then equip the process/device engineer with programming power without the need to do actual programming work. It can be expected that a visual programming interface will significantly contribute to the ease of use of the TCAD framework.

7.3 Object-oriented design representation

In a TCAD environment, it would be convenient to represent devices to be simulated as objects belonging to a device class hierarchy and with methods attached to them. Thus a device would 'know' how to simulate itself, i.e. its class would have methods attached which call the appropriate simulator. To achieve this, the design representation of the data level has to be fully object-oriented, and the procedural interface has to provide means to build class hierarchies and attach methods to classes. Since PIF provides a LISP-like syntax it is ideally suited to extend it with such object-oriented features. A C++ language interface would present those features to applications. Methods attached to PIF objects would be coded in C++ and made available to the extension language through the Tool Abstraction Concept. However, since only a minority of today's TCAD applications are written in C++, there is presently no strong need for such an interface.

Acknowledgements

The VISTA project has been sponsored by the research laboratories of Austrian Industries-AMS at Unterpremstätten, Austria; Digital Equipment Corp. at Hudson, USA; Siemens at Munich, FRG; and Sony at Atsugi, Japan, and by the 'Forschungsförderungsfonds für die gewerbliche Wirtschaft, project 2/285 and project 2/299, as part of ADEQUAT (JESSI project BT1B), ESPRIT project 7236 and project 8002.

We are very grateful to A. Gabara (University of California, Berkeley, California), N. Khalil (Digital Equipment Corporation, Hudson, Massachusetts), E. Masahiko, M. Mukai and P. Oldiges (Sony Corp., Atsugi, Japan), H. Masuda (Hitachi Device Development Center, Tokyo, Japan), L. Milanovic, G. Nanz, C. Schiebl, R. Strasser and M. Thurner (Campusbased Engineering Center, Digital Equipment Corporation G.m.b.H, Vienna, Austria), M. Noell (Motorola APRDL, Austin, USA), H. Read (Carnegie Mellon University, Pittsburgh, Pennsylvania) and K. Traar and G. Punz (Siemens AG, Vienna, Austria) for their patience and support, for their efforts in installing and testing VISTA, and for their contributions and criticisms.

References

[1] M.R. Simpson, PRIDE: An integrated design environment for semiconductor device simulation, *IEEE Trans. Computer-Aided Design*, 10(9) (1991) 1163–1174.

[2] E.W. Scheckler, A.S. Wong, R.H. Wang, G. Chin, J.R. Camanga, A.R. Neureuther and R.W. Dutton, A utility-based integrated system for process simulation, *IEEE Trans. Computer-Aided Design*, 11(7) (1992) 911–920.

[3] A.S. Wong, *Technology computer-aided design frameworks and the PROSE implementation.* PhD thesis, University of California, Berkeley, 1992.
[4] R.W. Knepper, J.B. Johnson, S. Furkay, J. Slinkman, X. Tian, E.M. Buturla, R. Young, G. Fiorenza, R. Logan, Y.S. Huang, R.R. O'Brien, C.S. Murthy, P.C. Murley, J. Peng, H.H.K. Tang, G.R. Srinivasan, M.M. Pelella, D.A. Sunderland, J. Mandelman, D. Lieber, E. Farrell and M. Kurasic, Technology CAD at IBM. In Fasching *et al.* [43], pp. 25–62.
[5] J.K. Ousterhout, Tcl: An embeddable command language, *1990 Winter USENIX Conference Proceedings,* 1990, pp. 133–146.
[6] J.K. Ousterhout, An X11 toolkit based on the Tcl language, *1991 Winter USENIX Conference Proceedings,* 1991, pp. 105–115.
[7] J. Mar, Technology CAD at Intel, in Fasching *et al.* [43], pp. 63–74.
[8] P.A. Gough, An integrated design environment for semiconductors, in Fasching *et al.* [43], pp. 131–146.
[9] H. Matsuo, H. Masuda, S. Yamamoto and T. Toyabe, A supervised process and device simulation for statistical VLSI design, *Proc. NUPAD III,* 1990, pp. 59–60.
[10] P. Lloyd, H.K. Dirks, E.J. Prendergast and K. Singhal, Technology CAD for competitive products, *IEEE Trans. Computer-Aided Design,* 9(11) (1990) 1209–1216.
[11] H. Jacobs, W. Hänsch, F. Hofmann, W. Jacobs, M. Paffrath, E. Rank, K. Steger and U. Weinert, SATURN – A device engineer's tool for optimizing MOSFET performance and lifetime, *Proc. NUPAD III,* 1990, pp. 55–56.
[12] C.H. Corbex, A.F. Gerodolle, S.P. Martin and A.R. Poncet, Data structuring for process and device simulations, *IEEE Trans. Computer-Aided Design,* CAD-7 (1988): 489–500.
[13] D.M.H. Walker, Ch.S. Kellen, D.M. Svoboda and A.J. Strojwas, The CDB/HCDB semiconductor wafer representation server, *IEEE Trans. Computer-Aided Design,* 12(2) (1993) 283–295.
[14] D.M.H. Walker, J.K. Kibarian, Ch.S. Kellen and A.J. Strojwas, A TCAD framework for development and manufacturing, in Fasching *et al.* [43], pp. 83–112.
[15] A.S. Wong and A.R. Neureuther, The Intertool Profile Interchange Format: a technology CAD environment approach, *IEEE Trans. Computer-Aided Design,* 10(9) (1991) 1157–1162.
[16] D.S. Boning, M.L. Heytens and A.S. Wong, The Intertool Profile Interchange Format: an object-oriented approach, *IEEE Trans. Computer-Aided Design,* 10(9) (1991) 1150–1156.
[17] SWR Working Group of the CFI/TCAD TSC, *Semiconductor Wafer Representation Architecture,* CAD Framework Initiative, Austin, Texas, USA, 1.0 edition, 1992.
[18] D. Boning, G. Chin, R. Cottle, W. Dietrich, S. Duvall, M. Giles, R. Harris, M. Karasick, N. Khalil, M. Law, M.J. McLennan, P.K. Mozumder, L. Nackman, S. Nassif, V.T. Rajan, D. Schröder, R. Tremain, D.M.H. Walker, R. Wang and A. Wong, Developing and integrating TCAD applications with the semiconductor wafer representation, *Proc. NUPAD IV,* 1992, pp. 199–204.
[19] S. Halama, F. Fasching, C. Fischer, H. Kosina, E. Leitner, Ch. Pichler, H. Pimingstorfer, H. Puchner, G. Rieger, G. Schrom, T. Simlinger, M. Stiftinger, H. Stippel, E. Strasser, W. Tuppa, K. Wimmer and S. Selberherr, The Viennese Integrated system for Technology CAD Applications. In Fasching *et al.* [43], pp. 197–236.
[20] F. Fasching, W. Tuppa and S. Selberherr, VISTA – The Data Level. *IEEE Trans. Computer-Aided Design,* 13(1) (1994).
[21] S. Halama, F. Fasching, H. Pimingstorfer, W. Tuppa and S. Selberherr, Consistent user interface and task level architecture of a TCAD system, *Proc. NUPAD IV,* 1992, pp. 237–242.
[22] S.G. Duvall, An interchange format for process and device simulation, *IEEE Trans. Computer-Aided Design,* CAD-7(7) (1988) 741–754.
[23] F. Fasching, C. Fischer, S. Selberherr, H. Stippel, W. Tuppa and H. Read, A PIF implementation for TCAD purposes, in Fichtner and Aemmer [44], pp. 477–482.
[24] P.H. Winston and B.K.P. Horn, *Lisp,* Addison-Wesley, 1989.
[25] *PAI Release 1.0,* Institute for Microelectronics, Technical University Vienna, Austria, 1992.
[26] P.J. Asente and R.R. Swick, *X Window System Toolkit, The Complete Programmer's Guide and Specification,* Digital Press, 1990.
[27] *OSF/Motif Programmer's Guide, Release 1.1,* 1991.
[28] R. Stallman, *GNU Emacs Manual,* 1986.
[29] A. Neureuther, R. Wang and J. Helmsen, Perpective on TCAD integration at Berkeley, in Fasching *et al.* [43], pp. 75–82.
[30] P. Lloyd, C.C. McAndrew, M.J. McLennan, S. Nassif, K. Singhal, Ku. Singhal, P.M. Zeitzoff, M.N. Darwish, K. Haruta, J.L. Lentz, H. Vuong, M.R. Pinto, C.S. Rafferty and I.C. Kizilyalli, Technology CAD at AT&T, in Fasching *et al.* [43], pp. 1–24.
[31] D.M. Betz, *XLISP: An Object-Oriented Lisp, Version 2.1,* 1989.
[32] *AUTOCAD Release 11 Reference Manual,* publication AC11RM.E1, 1990.
[33] N. Mayer, WINTERP: An object-oriented rapid prototyping, development and delivery environment for building user-customizable applications with the OSF/Motif UI Toolkit, Technical report, Hewlett-Packard Laboratories, Palo Alto, 1991.

[34] H. Pimingstorfer, S. Halama, S. Selberherr, K. Wimmer and P. Verhas, A technology CAD shell, in Fichtner and Aemmer [44], pp. 409–416.
[35] H. Masuda, H. Pimingstorfer, H. Sato, K. Tsuneno, K. Ichikawa, H. Tobe, H. Miyazawa, M. Nakamura, K. Kajigaya, O. Tsuchiya and T. Matsumoto, Applied TCAD in mega-bits memory design, in Selberherr *et al.* [45], pp. 21–24.
[36] Ch. Pichler and S. Selberherr, Process flow representation within the VISTA framework, in Selberherr *et al.* [45], pp. 25–28.
[37] H. Kosina and S. Selberherr, A hybrid device simulator that combines Monte Carlo and drift-diffusion analysis, *IEEE Trans. Computer-Aided Design,* 13(1) (1994).
[38] K. Wimmer, R. Bauer, S. Halama, G. Hobler and S. Selberherr, Transformation methods for nonplanar process simulation, in Fichtner and Aemmer [44], pp. 251–256.
[39] E. Strasser and S. Selberherr, A general simulation method for etching and deposition processes, in Selberherr *et al.* [45], pp. 357–360.
[40] H. Stippel, S. Halama, G. Hobler, K. Wimmer and S. Selberherr, Adaptive grid for Monte Carlo simulation of ion implantation, *Proc. NUPAD IV,* 1992, pp. 231–236.
[41] V. Axelrad, Y. Granik and R. Jewell, CAESAR: The virtual IC factory as an integrated TCAD user environment, in Fasching *et al.* [43], pp. 293–307.
[42] J. Daniell and S.W. Director, An object oriented approach to CAD tool control. *IEEE Trans. Computer-Aided Design,* 10(6) (1991) 698–713.
[43] F. Fasching, S. Halama and S. Selberherr (eds.), *Technology CAD Systems,* Springer-Verlag, 1993.
[44] W. Fichtner and D. Aemmer (eds.), *Simulation of Semiconductor Devices and Processes, Vol. 4,* Hartung-Gorre, 1991.
[45] S. Selberherr, H. Stippel and E. Strasser (eds.), *Simulation of Semiconductor Devices and Processes, Vol. 5,* Springer-Verlag, 1993.