The Viennese Integrated System for Technology CAD Applications

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

Institute for Microelectronics, TU Vienna,
Gußhausstraße 27–29, A–1040 Wien, AUSTRIA

Abstract

In order to meet the requirements of advanced process and device design, a new generation of TCAD frameworks is emerging. These are based on a data level providing a common data interchange format. Such a format must be suitable for building simulation databases, and needs to be accompanied by supporting tools and by a procedural interface with multi-language bindings for data storage and retrieval by application programs. The complexity and scope of a rigorous TCAD framework requires special efforts to create a system which is both transparent to the user and comprehensible to the programmer. A consistent architecture and strict adherence to general software engineering guidelines can contribute significantly to the solution of this problem. We discuss general requirements and architectural issues of the data level, the user interface and the task level environment, and present their implementation in VISTA, the *Viennese Integrated System for Technology CAD Applications.*

1. Introduction

The *Viennese Integrated System for Technology CAD Applications* is an integration and development framework for process and device simulation tools. VISTA consists of a data level part which provides a common library for accessing and manipulating simulation data[1], a set of utilities for visualization and high-level data manipulation, and user interface and interactive shell[2] which integrate all services (including the simulation tools) on the "task level".

In the following sections, we will investigate the role of the data level, the user interface and the task level environment within a TCAD scenario first from a general perspective, and then present the VISTA implementation.

General requirements for the data representation, user interface, task level environment, and related services will be discussed in Section 2. Then, after a review of existing implementations and general software architecture guidelines, we will motivate the major ideas and choices for data level, user interface, and task level of VISTA. The structure and implementation details of data level, user interface, and task level will be described in Sections 4, 5, and 6, respectively.

2. Requirements

2.1 The TCAD Scenario

The term "user interface" suggests to start with a closer look at the human aspects of engineering. With the introduction of frameworks into the TCAD field, the simple "programmer creates application for users" model of the TCAD software situation is no longer sufficient. The modern TCAD scenario looks more like the model depicted in Fig. 1, which will be used in the remainder of this paper (Fig. 1 already implies the most often used framework architecture, where applications are controlled by some sort of integrating task level shell and share common libraries).

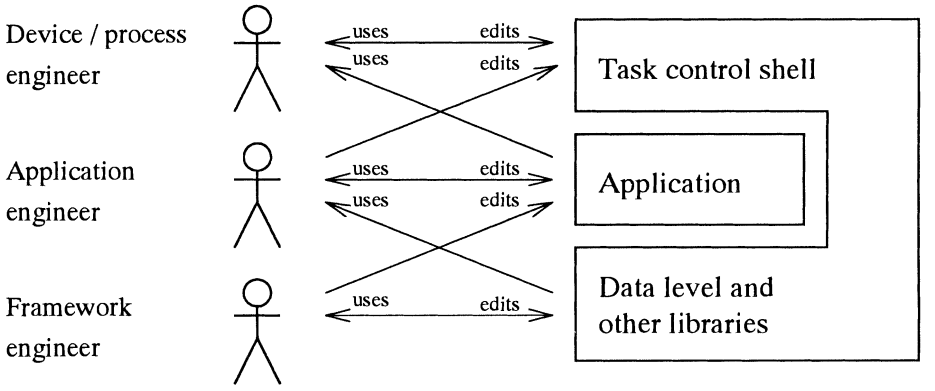

Figure 1: Process, application, and framework engineers interact with the TCAD system by both using (arrows pointing left) and modifying (arrows pointing right) the components of the system.

The *process* or *device engineer* uses simulation tools, or, more generally speaking, TCAD applications, to simulate a given process or device. For this purpose the engineer edits task level information (like manufacturing parameters, the process flow, or optimization goals) and explicitely or implicitely uses applications (simulators) 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. The process/device engineers make use of framework services (like data level access or the task level environment), too, but in contrast to the application egineers, they do not know about the libraries and details of "programming" with the framework.

The framework is edited and extended by *framework engineers* who also take care of generic (framework-related) applications.

2.2 The Engineers' Perspective

All three categories of engineers interact in some way with the user interface and task level part of the framework, either in a "user role" or in a "programmer role". Hence, one of the major difficulties in choosing a strategy for a user interface and task level implementation (in contrast to the data level or other internal libraries which are "just" visible from the programmer's point of view) is the vast variety of interests and perspectives which must be considered. Depending on the "engineering role" and the user's technical background and needs, different demands may arise:

- For casual users who seldomly 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. The user interface should employ familiar visual elements where they are available.
- For device or process engineers who use TCAD tools more often, flexibility on 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 too simulator-dependent and should not, as is often the case in UNIX-shell based solutions, depend on the operating system at all.
- For specialists in physical modeling or numerical techniques, the most important features are openness of the system as well as good aid for making modifications and extensions. The user interface should support full access to the simulation tools and models. Furthermore, it should be possible to integrate existing tools into a homogeneous user interface without having to redesign the tools.
- From the framework and application engineer's 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.

2.3 The Applications' Perspective

Seen from the applications' point of view, there are numerous requirements which the data level of a TCAD system must satisfy. Firstly, there has to be a persistent simulation database where simulation problem descriptions, histories and results are stored. A clear, procedural interface provides access to the simulation data and conveys all physical and nonphysical information used by the application. The interface must contain language bindings for those 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. Integrability with external TCAD tools must be ensured by providing access to simulation data on different levels of functionality and abstraction (this includes well-defined low-level interfaces).

The procedural interface must be characterized by its ease of use, and an orthogonality to minimize the effort involved in the creation of new tools. The interface should be able to automatically perform conversions of coordinate systems, physical units, simulation grids, etc., 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. Since some TCAD applications may want to use their own internal data structures, the interface has to adapt easily to application-specific data structures.

Once these demands are satisfied, simulators may be run as standalone applications coupled by a common data format. However, a full TCAD integration imposes further requirements upon the data level, under the assumption, that simply "wrapping" the simulator is not a desirable integration method.

Since many different tools with a highly complex sematical background have to interact in a TCAD framework, it is indispensable that the data level has to express semantic rules to ensure the "understandability" of common simulation data for all tools integrated in the environment. Whether all of this semantics can and should be implemented in software and hence be reflected by framework services is a tough question, both technically and economically.

2.4 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. Fig. 2 compares a few prominent examples of single simulation tools with VISTA (as an example of a TCAD framework) which currently requires nine megabytes of code in different implementation languages (predominantly C), not including simulators.

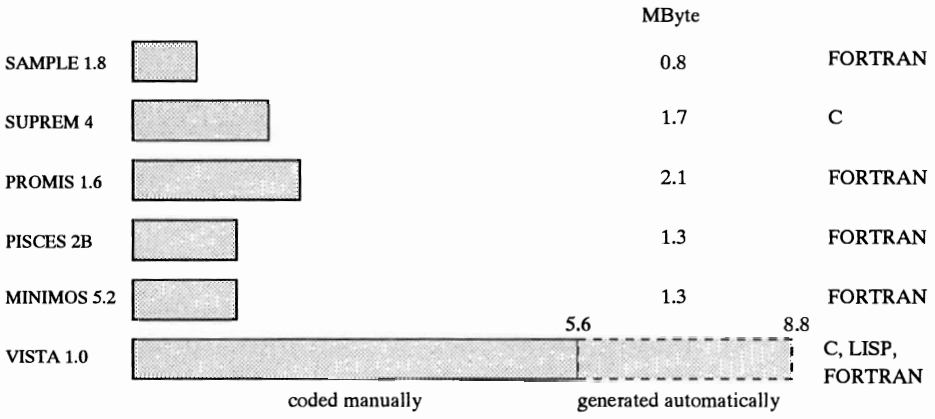

Figure 2: Comparison of source code sizes

Because of the remarkable size of the code, it is obvious that special care is required to ensure the consistency and maintainability of the framework, and that pure software issues become much more relevant than in the case of single simulation tools, regardless of their sophistication. In fact, in the case of a single simulation tool, underestimating or neglecting software issues might lead to an improper and unflexible implementation, although the tool will probably still work. In the case of a whole TCAD framework however, the resulting implementation (if ever achieved) would be completely unusable.

Additionally, it is indispensable that the basic structure and the details of the system can be learned and understood with only a moderate expenditure of time and effort. Therefore, a major demand is that the design and implementation of all components adhere to a few, simple, and mutually consistent basic concepts. We will see that this demand has a severe impact on the potential use of existing solutions from which the framework can be built.

3. Architecture

3.1 Existing Approaches

Several workstation-based systems can be found which address the issue of multi-tool integration into a unified user interface, mostly based on the X Window system. PRIDE[3], 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[4], based directly on Xlib, features a central interactive graphical editor which has menu-oriented facilities for running simulators. In both cases, implicit or explicit assumptions about the design cycle 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[5], which is mainly due to the use of the generic Tcl interpreter[6] and Tk toolkit[7].

Other well-known simulation frameworks are an integrated system for statistical VLSI design from Hitachi[8], the MECCA system from AT&T[9], or the SATURN system from SIEMENS[10].

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, 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 [11], feature two-dimensional geometries and simple data structures for easy usage by existing simulators. Among data levels designed for TCAD environments there are the CDB/HCDB from CMU [12], and the BPIF implementation from UC Berkeley [13]. Another data level built on PIF featuring object-orientedness is the PIF/Gestalt system from MIT [14].

A recent approach is the SWR 1.0 specification[15][16] (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 modeling 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, e.g., 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 sole definition of a rather high-level interface — which implicitely requires a very large amount of functionality — makes it difficult to implement this standard in a rigorous way. This is due to the absence of intermediate-functionality definitions and goals, there is no layering that would provide natural milestones for implementation and verification.

3.2 General Guidelines

It seems that the inherent semantic complexity and diversity of information flow between user and TCAD system, which is presumably related to the rich physical

background inherent to process and device simulation, can only be represented by corresponding specialized user interface elements. Furthermore, the rapid development of advanced process and device simulation tools calls for a data level and user interface which can easily accommodate new demands without necessarily having to change the underlying concepts.

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.

The requirements described in section 2 are not easy to fulfill within today's software environments. However, some general rules 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 design* is favorable. This implies that the definition of higher-level functionality and behavior is 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* — All of the framework code should be kept independent from the (rapidly changing) TCAD tools themselves.
- *Consistency* — Where possible, the generalization of existing concepts within the framework should be favoured over the introduction of new ones.
- *Interprete design tasks* — The need for defining and using design task macros in a flexible (non-taxative) manner suggests to ask for full programming capabilities, the task level programs being executed by an *interpreter*.

On the data level, a certain architectural transparency is desirable to be able to accomodate and implement possible framework architectures, like client-server, master-slave, parity, ... Although for some, especially geometry-oriented applications a client-server architecture might be advantageous, it is highly questionable, if a true client-server architecture using the network will exhibit the required performance for more general simulation requirements, since large data amounts like grids, attributes or solver stiffness matrices (especially in three-dimensional applications) have to be communicated between client and server. Although mapping client memory into the server substantially improves performance, this approach is neither portable, nor does it work over the network.

A well-balanced and consequent layering of the functionality and semantics of the data level implementation is most important as this is indispensable for re-using the implementation, when, e.g. an RPC-based client-server interface is introduced between two layers, or an object-oriented system is imposed (presumably on the top layer).

An early confinement to a specific architecture would result in an inflexible data level and thus lead to a framework that cannot be adapted to (unpredictable) environmental needs of a simulation site. A firm precondition, however, is the multiprocessing ability of the application interface, to enable parallel simulator runs using the same data set as well as ensuring clusterwide access to the data. The data level of a TCAD environment must be able to manage and archive simulation sequences in order to

ensure the reproducability of the results and easy backtracking through the simulation history. The environment has to provide facilities for intersite data exchange, message passing between applications, and error reporting, handling and recovery, which have to be consistent with the data-level implementation.

3.3 The VISTA Design

For the data level, we have decided to start with the *Profile Interchange Format* (PIF), as initially proposed by S. Duvall[17] and to extend and modify it to meet the requirements stated earlier. From the above considerations, it is clear that a binary implementation and a procedural interface with different levels of functionality (described in Section 4) are required. Since there is no public and efficient implementation available, we had the opportunity to implement the application interface from scratch.

For the user interface, the X Toolkit[18] already offers an ideal method for achieving a very flexible and consistent architecture when the required specialized parts of the user interface are implemented as so-called *widgets*. From this set of building blocks, all higher-level functions and applications can be built. This coincides with the proposed bottom-up concept and, due to the object-orientedness of the X Toolkit and widget set, is very well suited for future extensions. A widget set has to be choosen from which the required specific widgets for TCAD purposes can be subclassed.

For the task level environment, a proper choice is in general non-trivial, but becomes almost obvious, when the proposed architectural guidelines and requirements are considered. A UNIX- (or any other operating system) shell based solution does not fulfill the portability requirement, whereas the use of an integrating master application (like an interactive device editor) alone does not offer programming language features.

A portable interpreter appears to be the only appropriate solution that meets all demands. Hence we have chosen to build on XLISP[19], a public domain LISP interpreter, which is available in source code. It is coded in highly portable and comprehensible C code, fulfills all software-oriented requirements and provides full programming capabilities. It can be extended for TCAD purposes by adding C-coded primitives or by loading LISP code at run time.

There are some remarkable implementations of task level environments in related fields, which confirm the feasibility of using LISP as extension language:

The well-known *GNU Emacs*[20] text editor features a LISP extension language which is used to implement special editing modes and to provide text browser style interfaces to a number of unix applications. The computationally expensive parts are still coded in C, so that LISP mainly ties together high-level primitives.

The generic CAD system *AutoCAD* uses the Scheme-like extension language interpreter *AutoLISP*[21] which allows direct access to the data level. It is an invaluable tool for customization and for implementation of a multitude of specific applications.

Winterp[22] ("*W*idget *Interp*reter") is an application development environment, based on the public-domain XLISP interpreter. It provides interfaces to the X11 Toolkit Intrinsics and to the OSF/Motif widget set and is distributed with the public-domain MIT X11 distribution. Unfortunately, this potential candidate for building a TCAD task level environment and user interface upon it lacks two requirements: It does not readily accomodate additional C-code layers between the X Toolkit (OSF/Motif widgets) and XLISP interpreter, which inhibits the introduction of higher-level user interface layers which need to be shared among C applications, and the object-oriented

interface in LISP can not easily be extended to be used by C applications in a *homogeneous* way.

3.3.1 Interaction of Framework Components

The user interface has to allow easy access to *all* services provided with the framework, such as the data level implementation (including high-level data manipulation and interactive editing of device structures), visualization, or the error system.

Additionally, 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 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 and provided with appropriate input and their termination needs to be recognized to trigger subsequent simulation steps.

4. Data Level Implementation

The data level is the *backbone* of the whole framework. The data level of the VISTA system was designed to meet most of the above requirements. It features:

- A layered procedural interface for applications to store and retrieve all TCAD relevant data,
- Language bindings to FORTRAN, C and LISP,
- A common ASCII interchange format (*Profile Interchange Format*, PIF),
- A compact binary inter-tool and storage format (*PIF Logical Binaries*, PLBs),
- Parallel access to PLBs,
- The ability to build databases of PLBs into *PIF Binary Files* (PBFs),
- Database utilities to manage PBFs,
- Networking capabilities.

The procedural interface to the database services is called *PIF Application Interface* (PAI, [23]) and makes extensive use of automatic code generation to achieve platform independence and generate the individual language interfaces. It is described in subsection 4.2.

4.1 VISTA's PIF Implementation

The ASCII version of the PIF is used as an intersite data exchange format. The binary form[23] is used as database storage format of the data level. Fig. 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 conveying TCAD-related data. Both the `geometry` and the `grid` constructs are built out of primitive geometric objects (points, lines, faces and solids). The `geometry` construct additionally holds a simulator's point of view of a simulation geometry through `segmentList` and `boundaryList` constructs.

The `attribute` construct is used to attach *any* kind of information to an object. The `attributeType` subconstruct describes the meaning of an attribute. Thus the PIF attribution mechanism is the most flexible means in attaching information to geometries and grids, since they can express anything ranging from a simple descriptive string to a vector field defined on a tensor product grid. With this unified concept there is no separation between fields and attributes necessary, which is another milestone to a clearly structured architecture allowing a simple implementation. Fig. 4 shows a `materialType` attribute defined over a segment and Fig. 5 shows an `electricField` attribute defined over a three-dimensional grid.

In contrast to other approaches, attributes types are semantically standardized to prevent incompatibilities in tool communication (e.g. one tool writing a "`Potential`" attribute, and a second tool trying to read an "`ElectricPotential`" attribute), although each tool is free to define its own local attribute types.

Due to its generality and flexibility, a PBF may hold an unlimited number of PLBs, and one PLB (conforming to one ASCII PIF) in turn may hold an infinite number of objects. So, a PBF may contain anything from just one PLB with a few comments, to hundreds of PLBs, each holding several geometries, attributes, grids and process flow descriptions. The maximum size of a PBF is limited by the adressing capability of the PAI, which in turn is affected by the machine word length. On a 32-bit machine the PAI can address one gigabyte (some bits are reserved) which is therefore the maximum PBF size, supposed the operating systems file size limitation is higher. Since the PAI is capable of opening up to 16 PLBs, an application has a maximum of 16 gigabyte of data available. Typically, a single PBF holds one or two PLBs containing a geometry, attributes and grids of a single tool run. Through the special `link` construct objects in other PLBs or even other PBFs may be referenced.

The binary format is closely related to the ASCII format inasmuch as the hierarchical structure of the ASCII PIF is preserved in the binary form through the use of LISP-like constructor nodes. However, to improve performance and data compactness, several additional features have been implemented, 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, although the structure of the PAI is derived from the PIF syntax, the PAI itself is independent from the underlying database, and thus could be interfaced (probably with losses in performance and compactness) to other databases, since the TCAD application sees just the PAI procedural interface and has to know little about PIF. Thus, multiple different implementations of the low-level application interface routines are possible, because the applications have just to rely on the specification of the PIF application interface services.

The decision to use PIF was made in conjunction with the decision to adopt LISP as the VISTA task level extension language: PIF uses a LISP-like syntax and LISP as the

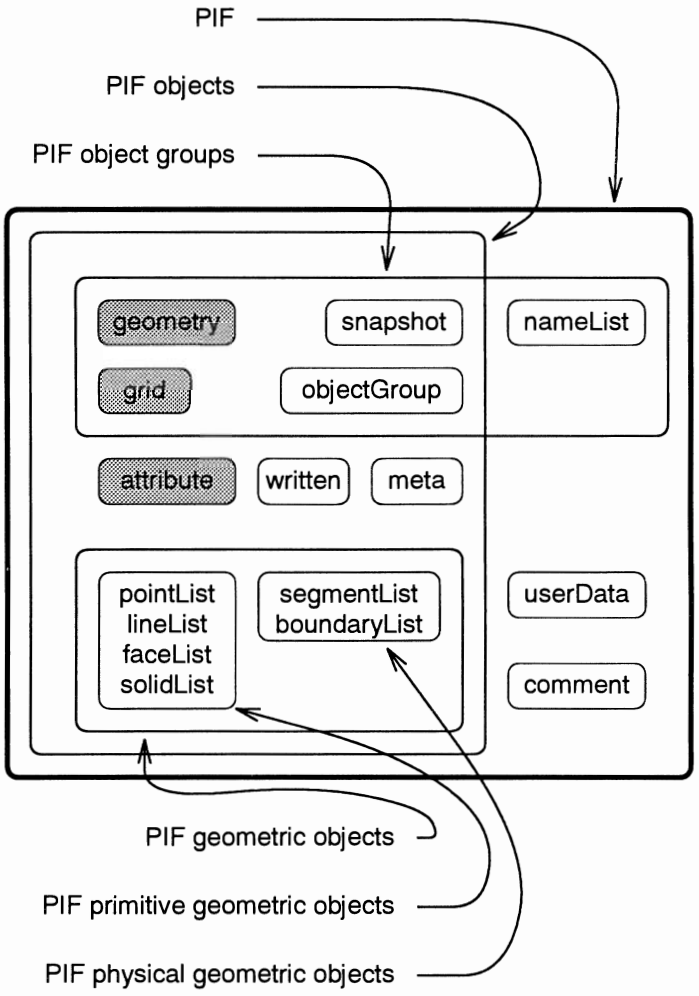

Figure 3: The logical PIF structure.

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

Figure 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
        .....)
)
```

Figure 5: Attribute defined on a grid.

task level language provides seamless and homogeneous fusion of data and task level concepts. With this unique combination it is equally possible to modify simulation data in the database directly as LISP data as well as store LISP expressions (e.g. task level programs) in the PBF. Thus a process flow representation can be directly embedded in the TCAD data level; there is no artificial separation, and homogeneous data storage, retrieval and maintenance services are available for both semiconductor wafer and process flow representations.

4.2 Implementation of the PIF Application Interface

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

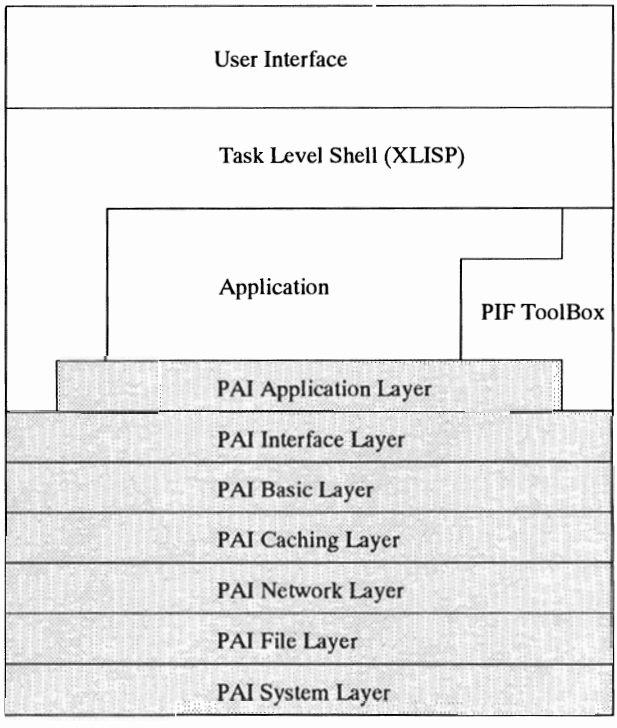

Figure 6: Layout of the PIF application interface.

Each layer calls only functions in the underlying layer. This mechanism leads to separate modules with distinct functionality as used by individual tools. Each layer is responsible for a unique storage concept of the whole PBF, with increasing complexity 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 4.4), implemented as a separate PIF tool on top of the PAI.

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.

4.2.1 Error Handling

Errors detected in the PAI are signaled to the global VISTA error handling system, which allows the user to specify different error handlers for each type of error. In addition to program-signaled errors, the error system handles system faults and program exit too. The default error handler prints out the function, the line number and source file name of the function, where the error occured.

New error handlers can be registered by each application to handle error conditions in a program specific way. For example, the caching layer installs its own exit handler on initialization to panic-close all open PBFs through the error system if a memory fault or address violation occurs.

4.2.2 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 standarized, but buffering is not needed by the PAI since this is done in the caching layer above. If the unbuffered UNIX style file I/O exists in a specific operating system, this is used instead. This is the only layer which has system dependent functions and implements also basic functions for network access (TCP/IP and DECnet).

4.2.3 File Layer

The standarized 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). Avoiding multiple write accesses to one PBF allows an easier implementation of the data base, since the physical file cannot change during access (unless it is closed); multiple read-only accesses are allowed. The locking of a PBF is not implemented through system functions. It works through a mark in the header of the PBF and a special lock status, where multiple accesses of the same file at the same time are detected. The file layer also allows the creation of temporary PBFs for intermediate storage of simulation data. Temporary files are stored in PBFs without a physical name and deleted automatically upon closing.

4.2.4 Network Layer

The functional interface exhibited by this optional layer is equivalent to the one of the file layer for access to PBFs, but allows instead accesses to PBFs over the network. In order to minimize network traffic, the functions of the file layer are used for 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.

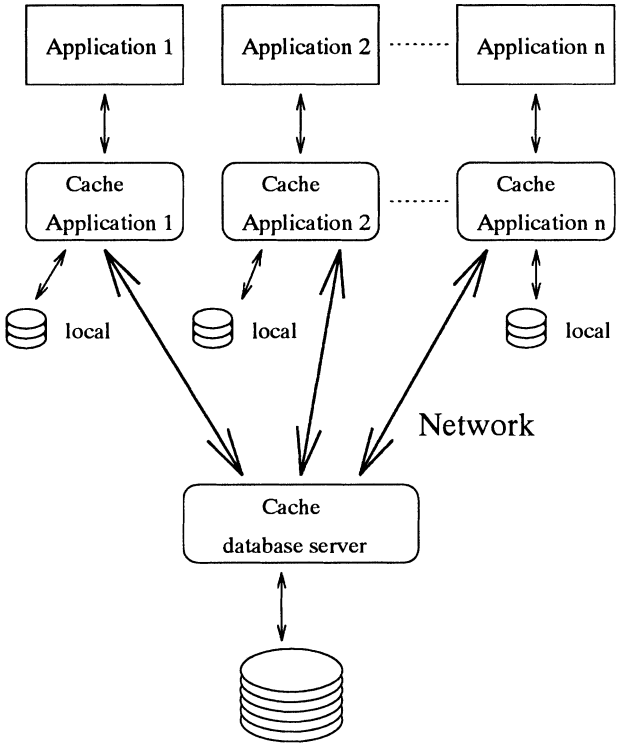

Figure 7: Local and network storage.

The client uses the database server for all file I/O functions on the network PBFs, but all database operations are done locally with the help of the basic and interface layers. For fast access to the data, the server holds some data blocks of the files in a local cache similar to the caching layer. This cache is shared by all clients and is not cleared upon closing a PBF, so that a following reopen and usage of the same file, even by a different application, is fast due to its remaining in the server cache. All write operations are delayed and buffered through a cache to maximize performance. The runtime option of unbuffered write operations ensures consistency of the PBF during update operations, and allows to examine a PBF while a tool is running and writing to it, which is an invaluable help in debugging simulators.

Another aspect of the network layer is the capability of message passing. It allows the application to contact other programs (e.g. the XLISP interpreter on the task level) over the network. Fig. 8 shows an example network configuration with tools and database servers interacting over the network.

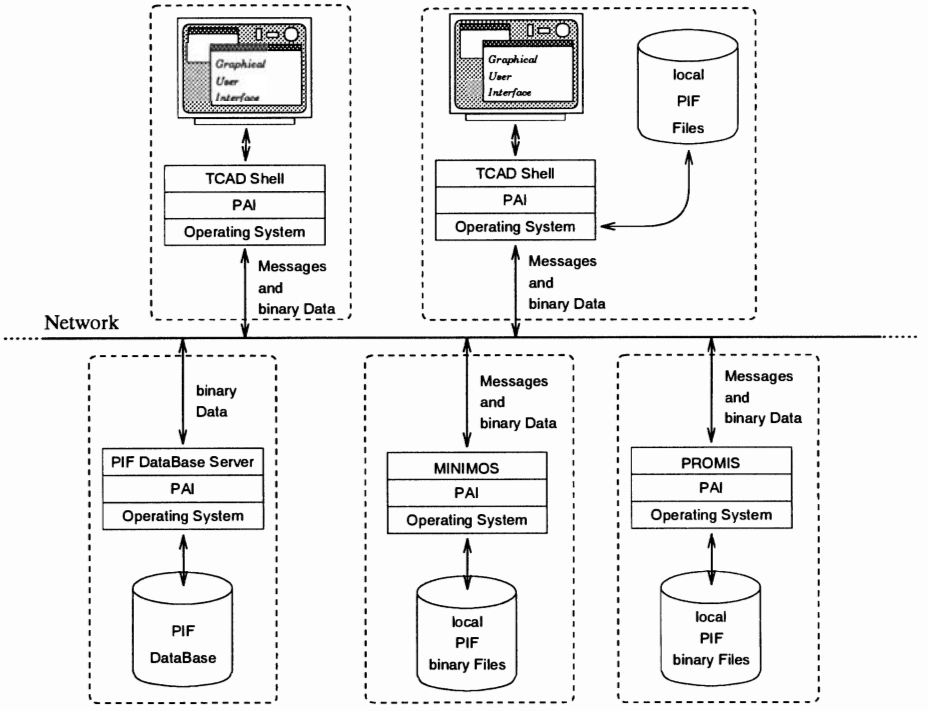

Figure 8: Examples of PAI networking capabilities.

4.2.5 Caching Layer

This layer buffers I/O data to minimize disc and network accesses. Depending on the application, 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 caching layer is designed in such a way that the page-replacement algorithm can be substituted with a different one like LRU or random replacement of memory pages [24]. Currently, an algorithm implementing a combination of these two methods is used. The memory pages are usually as big as or — for better performance — even bigger than operating system cache pages. This layer also allocates file space for all types of objects. To optimize cache hits, all small objects with a few allocation units in size are contiguously stored in one big chunk whereas large data pieces are always appended at the current end of the file. Since the above layers need the functionality to update data items, a `free()` operation is also implemented so that no space on the physical file is permanently wasted. From the above layers, the caching layer can be seen as a big `malloc()/free()` library with access functions that perform cached file access.

4.2.6 Basic Layer

This layer is the lowest to implement structured data nodes. Fig. 9 shows the general structure of such a basic layer node. The header word of the node determines its type and structure, i.e. the type and number of the generic and specific data slots. The former are common to each node type whereas the latter carry the actual data visible to upper layers. Thus the shaded fields in this figure are maintained and used by the basic layer. For the unshaded fields the basic layer just reserves space and provides access functions.

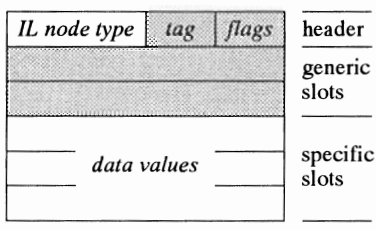

Figure 9: Implementation of a basic layer node.

The possible data types of the *specific* slots as determined by the *tag* field of the header word are:

Car	pointer to another node
Symbol	unique symbol name in the logical PIF file
Symref	reference to a symbol node
Char	character node
Byte	unsigned byte value (8 bit)

Short short data value

Word unsigned short data value (16 bit)

Long long data value

LongWord unsigned long data value (32 bit)

Float real data value

Double double presision data value

LongDouble quad precision data value

All these types correspond to the C language types of the same size. To connect nodes together into a list or to implement arrays like strings (consequently stored as character arrays), the `flags` in the header word are used. The possible values are any combination of the following definitions, responsible for determining the *generic* slots of the node:

Cdr the node has an implicit pointer to a successor node

Array the node is an array (its size is stored as a separate entry)

Compressed the data of the node is compressed

With the basic layer a functional interface to a LISP-like information storage concept is implemented. The interface presents 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 [25]. One significant difference to a LISP interpreter's memory structure is that every basic layer node is implicitly a *CONS* node providing a *CDR* pointer as one slot in the data slots and carrying an atomic data value, i.e. the *CAR* pointer is redundant and therefore removed. The actual *CONS* node is implemented as a basic layer node with the atomic data value being a *CAR* pointer.

It should be noted that in contrast to LISP storage concepts, all nodes of the basic layer (and hence the PIF Application Interface) are originally kept on a file and are just cached through the caching layer. This implies, that all reference pointers stored in a basic layer node are file offset pointers and thus do *not* point to memory locations. It is the caching layer's duty to resolve those references correctly.

To illustrate the different storage concepts, let us consider the simple PIF expression `(ref P (valueList 1 2))`.

This construct represents a reference to the first two points of a `pointList` P. Fig. 10 shows how this construct would be stored in the XLISP interpreter with separate *CONS* nodes. The corresponding PBF structure, as it is handled in the basic layer, is shown in Fig. 14. The *CONS* nodes are fused with the *CAR* data values, to compact the data structures and minimize data access time, which is crucial on slower external storage media. Furthermore, this concept retains the principle extension language storage structures on the data level.

Using this concept of LISP-like information nodes the PLB is stored, whereas the PBF is built as a linked list of PLBs, shown in Fig. 11. Since this list is only searched when the file is opened, this is no performance drawback. The data area is shared by all PLBs in the PBF, but on write operations it is checked that no crosspointers into disjoint PLBs occur. To allow fast access to all symbols, these are stored in a hash table which is unique for each PLB so that there are no conflicts between different PLBs.

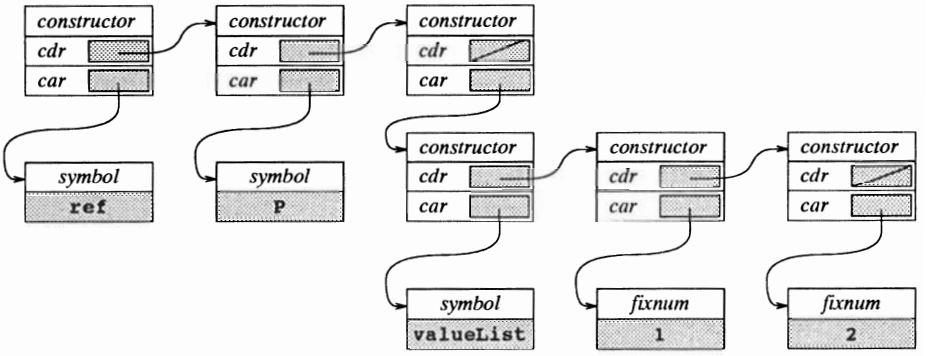

Figure 10: Example of a LISP internal data structure.

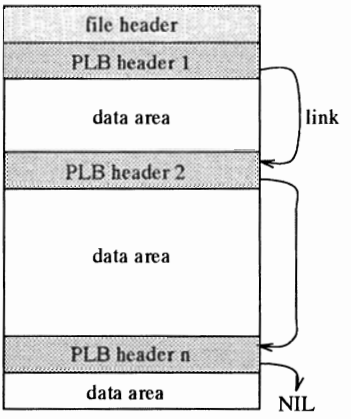

Figure 11: Layout of a PIF physical file with multiple logical files.

4.2.7 Interface Layer

This layer is the implementation of the PIF syntax, providing administration, access and inquire functions. To improve performance, it uses a structure of the basic layer array node to implement the interface layer nodes. A reserved field of the header word of a basic layer node Fig. 9 is used to store the type information of the interface layer node (labelled *IL node type*) as an integer value.

The `tag` field of the basic layer node contains the *CAR* type identifier, stating that this node contains *CAR* pointers to other nodes as data values. The `flags` field of the basic layer node has the *Array* and *Cdr* bits set, indicating that the interface layer node may have successors (pointed to by the *CDR* pointer) and multiple *CAR* data values (the number of which is stored in the *Size* field of the node).

The corresponding name of the object and all related information is stored in an automatically generated syntax table. This reduces the file size of a PLB significantly.

The *IL node type* identifies the PIF object type like *pointList*, *geometry* or *valueType* which are defined by the syntax. This field is automatically checked upon creation of a node through the syntax table. All access functions and the syntax table are generated automatically from a syntax description an example of which can be seen in Fig. 12.

```
(rule 'snapshot
      '(deriv
      LPAR SNAPSHOT OBJNAME
      (opt comment)
      (llist nameList)
      (llist attribute)
      RPAR))
```

Figure 12: Abstract syntax description of the PIF `snapshot` construct.

The *snapshot* construct is defined as a named PIF object, whose name can be used to search for the object. It has an optional comment, an optional list of references through the *nameList* construct and an optional list of *attributes*. Therefore, the node will have four specific slots. The first slot will hold the unique name of the *snapshot*. The second will hold the comment, the third the references and the last the *attribute* definition. This information is also used to limit the search depth when traversing the tree (in case not all slots have to be searched) on PLB inquiries, and to check the correctness of the PLB upon node creation.

The previously mentioned example of a `ref`erence construct represented with interface layer nodes is shown in Fig. 13. The `ref` and `valueList` constructs have specific interface layer node representations, whereas the symbol name and the `valueList` indices are genuine basic layer nodes, since they just represent primitive data values. Compared with the basic layer-only representation of the same reference construct in Fig. 14, a significant reduction in the number of required nodes and total storage size can be seen, resulting in faster data access.

Since the syntax defines many fields optional to allow a wide range of possible constructs, additional "language rules" are needed to define a well constructed PLB which can be understood by different simulators. Many of these rules are implemented by the application layer, others are described in the PIF CookBook (see [26]) which defines the semantic meaning of the PIF.

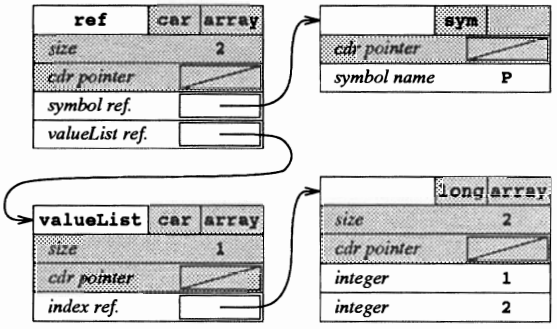

Figure 13: Example of an interface layer data structure.

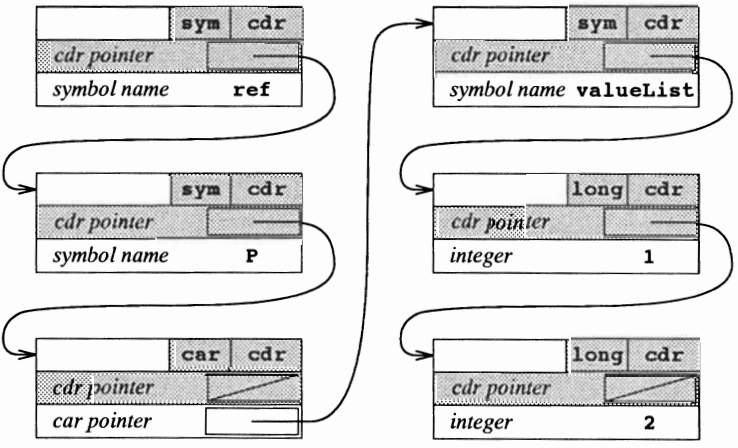

Figure 14: Example of a PIF binary file and basic layer data structure.

4.2.8 Application Layer and Language Binding

This layer implements some functionality common to simulators and utility programs. Its design is intended to be extendable in order to adapt the interface to new simulators or special demands. Many semantic rules and checks are implemented in the application layer, making the adaption of existing simulators to PIF easier, and ensuring interoperability in the VISTA framework. High-level functionality and automatically invoked data-manipulation services are provided to relief TCAD tools from tedious "everyday" work. The routines of the application layer implement geometry-manipulating as well as attribute-manipulating functions, because we think that both aspects are closely related in a TCAD environment. This fact is expressed in the uniform data representation of geometries and attributes on geometrical objects in PIF.

FORTRAN Interface

As the application layer is written in **C** and most simulators are written in **FORTRAN**, we have developed language bindings for most application layer functions and all inquiry functions to **FORTRAN**. This binding is strongly dependent on the two compilers, since there is no standard in parameter passing of strings in **FORTRAN** and the implementation of logical values (`.TRUE.` may be represented as the cardinal number 1 or -1). So we generate all binding functions automatically out of a formal description and additional information about the specific **FORTRAN** compiler. All string and logical variable conversions to C types are done automatically before the user-supplied **C** code is called. Adding a new binding or another compiler requires only few additions in the configuration files.

LISP Interface

The LISP interface of the PAI is not built on top of the application layer, since it makes no sense to use LISP for computationally intensive calculations on PIF data. The extension language of the task level is primarily used to generate input PLBs and control information for TCAD tools or to read output values of simulation results for further investigations. Thus the extension language interpreter connects to the interface layer, allowing full access to PBFs. For convenience there is an additional LISP library to support the creation of whole PIF constructs (like generated with the application layer).

High level functions of the PIF ToolBox are automatically bound to LISP by the Tool Abstraction Concept (TAC) and so available to the TCAD shell. The big difference between Application Layer functions and ToolBox functions is that the second get their input from the PLB and write their output back to the same or another PLB. No data manipulation is done in LISP.

4.3 Use of the PAI

The short code example in Fig. 15 shows the C calls to generate an example of a PIF data structure. `namelist` is the handle to the parent `nameList` object [26]. The two element array `points` holds the indices of the points on which the line is created. In the Application Layer code example Fig. 16 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. 17.

```
{
    /* 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
*/
}
```

Figure 15: Interface layer code example.

```
{
    /* 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 */
}
```

Figure 16: Application layer code example.

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

Figure 17: PIF construct produced by example code.

4.4 PIF Binary File Manager

As mentioned above, the whole PAI works on a binary representation of the data for fast access. This type of data storage is optimized for architecture-dependent coupling of simulators in non human-readable form. For data exchange via eMail or FTP, or making PLBs human-readable, there is the ASCII PIF representation holding the same information. The PBFM (Fig. 18) is able to convert the binary to ASCII PIF and vice versa. Thus data exchange between machines with different byte ordering (little and big endian) and floating point formats (e.g. IEEE, VAX and IBM) is possible by converting PLBs to ASCII PIF and back to binary format on a machine with different architecture. The maintenance functions of the PBFM allow the user to list all PLBs of a PBF, delete any PLB within a PBF, repair a not cleanly closed PBF and check PIF ASCII files for lexical correctness.

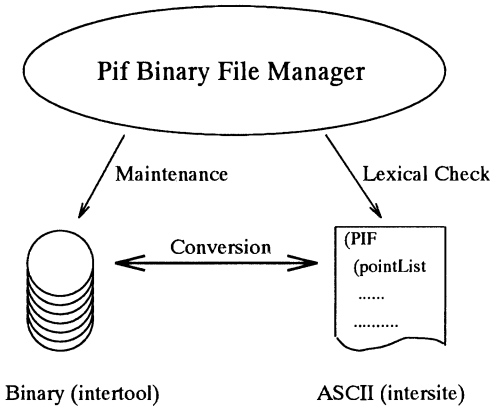

Figure 18: Using the PIF Binary File Manager.

4.5 Semantic Issues and High-Level Functionality

The PIF itself, be it its ASCII or binary form, only defines a syntax. It does not prescribe any interpretation of the stored TCAD data. This is one issue which accounts for the flexibility and general-purposeness of the PIF. On the other hand, many ambiguities arise from the possible multilateral description of the same physical problem in terms of PIF syntax. There are many ways to describe a geometry, ambiguities in recognizing a grid or an attribute, and general PIF semantics. These ambiguities arise from different coordinate systems, hierarchical or non-hierarchical geometry specifications and using one or many lists of primitive geometric objects, with or without references to other PLBs. Grids can be of unstructured or tensor product type, defined on a segment or the whole geometry and attributes can be defined on the grid, its points, lines, faces or solids. Moreover, the application interface has to know what to do with different units of measure, when to write and what to reference in a `snapshot`, `geometry` or `written` construct, where to define attributes, what attribute types to use, ...).

However, ambiguities and multilateral descriptions are a general problem, because the more general a syntax is and the more functionality a procedural interface has, the more semantic standardizations are needed to make applications work properly in a common environment.

In order to unambiguously interpret PIF data, there have to be both semantic constraints which applications have to adhere to (losing PIF flexibility), and ambiguity resolution mechanisms built into the application interface. However, only few additional semantic rules specified in the PIF CookBook [26] have to be obeyed. The PAI takes care of different coordinate systems through a transformation matrix applied to geometrical data, and accounts for different units of measure through a unit conversion system (e.g. point coordinates can be written in micrometers and read in inches, different spatial axes can have different units). It automatically resolves links to other PLBs and provides a multitude of inquiry functions for locating a certain PIF construct wherever it appears in the PLB.

The more severe semantic differences between simulators (e.g. a simulator working on an unstructured grid coupled to a simulator using a tensor product grid) are dealt with in the PIF ToolBox, comprised of generic PIF tools such as grid generators, interpolators, attribute and geometry manipulators using the PAI and preparing a PLB according to the semantic standards of the PIF CookBook [26]. However, these tools are controlled by the task level and belong to the tool rather than to the data level.

A particularily difficult problem is the support mechanism for the innumerable different grid types used today. A distinction between tensor product and unstructured grids has been made, because we didn't want to lose an orthogonal grid's unique features by decomposing it into rectangles/cuboids. Therefore the special `orthoProduct` construct was introduced, which significantly enhances the efficiency of storing tensor product grids while preserving its advantageous structure. However, since the number of different unstructured grid types increases steadily, a specification mechanism for dynamically adding new grid types just by providing a unique name, an interpolation and a decomposition function has been implemented. Using automatic code generation tools, these routines are linked into generic PIF ToolBox functions, thus adding support for the new grid element to the whole framework. Fig. 19 shows some example elements and how they are referenced in a PIF grid. Through using an `attribute` defined over the `faceList` it is possible to specify different element types in one and the same grid.

New element types are introduced by specifying the name, dimensionality, number of nodes and a decomposition and interpolation function in a element definition table. After recompiling the PAL, the new element type is known to applications through a unique constant identifier. But most applications need not explicitly 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 function, which is automatically invoked when requesting an attribute value at a location `(x,y)`. The generic interpolation routine knows the grid type the attribute is defined on, and correspondingly invokes the **orthoProduct** interpolation or determines the element in which the requested location lies, then invoking the element interpolation function defined for that element type.

Tensor product grids are supported through the `orthoProduct` construct. The grid has an origin point, may have different topological and topographical dimensions and each dimension may have a different base vector. Conforming to the PIF syntax, the number of supported dimensions is infinite. The example Fig. 20 shows a tensor product grid of topological dimension 2 lying in three-dimensional space. This capability is needed e.g. to describe distributed boundary conditions of a three-dimensional device.

The assembly of solver matrices is not supported by the PAI, since we believe that this task is very problem-specific and current networks don't exhibit the necessary

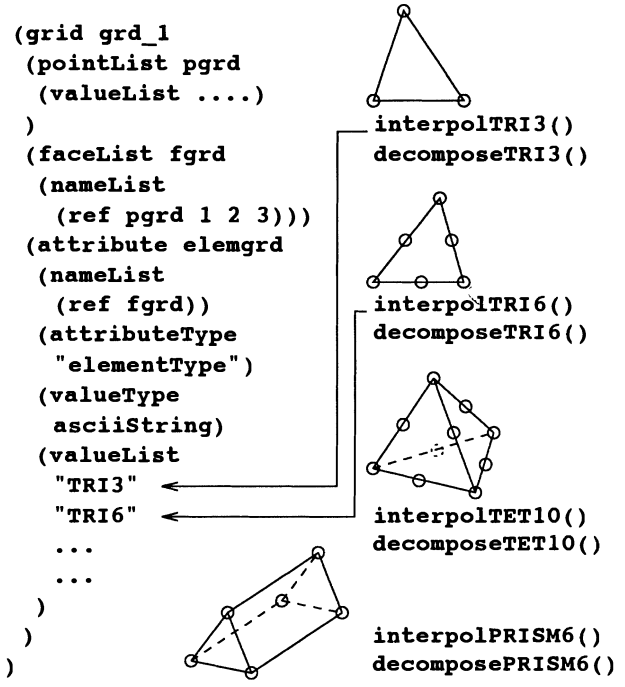

Figure 19: Support for unstructured grids.

```
(orthoProduct my_tensor_grid
   ; the 3D base point
   (origin
     (units "um")
     (valueType (point 3 real))
     (valueList 1.0 1.0 0.0))
   ; the two axis vectors
   (base
     (valueType (vector 3 real))
     (valueList 1.0 0.0 0.0
          0.0 1.0 0.0))
   ; the two axis specifications
   (axes
     (valueType real)
     (valueRange 0.0 5.0 0.23)
     (valueList 0.0 0.1 0.22 0.37 0.4))
)
```

Figure 20: Tensor product grid example.

performance to transfer these large amounts of data in an acceptable time frame to a solver server. The "know how" of a simulator is always contained in its physical models, the knowledge of which is essential in matrix assembly. A simulator using a standard matrix assembly method would lose much of it's advantages. This holds true for grid generation and the partial differential equation solver too.

4.6 Performance Evaluation

Besides the goals of classical intertool PIF implementations featuring object-orientedness (PIF/Gestalt, [14]) or suitability for TCAD environments (BPIF, [13]) our implementation stresses efficiency in terms of run-time 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 run-time performance no longer holds true.

5. Implementation of the VISTA User Interface

5.1 Structure

The structure of the VISTA user interface is shown in Fig. 21. The bottom layer is the X Toolkit[18], an object-oriented subroutine library, designed to simplify the development of X Window applications. The X Toolkit defines methods for creating and using widgets, which appear to the user as pop-up windows, scrollbars, text-editing areas, labels, buttons, etc. Basic functionality is provided by the generic *Athena widgets*, which are part of the MIT X11 distribution. We have decided to use this widget set rather than any other open standard, because a migration from these generic widgets to another widget set (like OSF/Motif, or Open Look) is significantly easier than vice versa.

A widget-wrapping layer has been put on top of these widgets in order to achieve some widget-set independence. All widgets are created and modified via specific functions rather than via the generic interface of the X Toolkit. This facilitates the potential migration of the entire user interface onto another X Toolkit-based platform.

In addition specialized VISTA widgets have been developed on top of the widget-wrapping layer for supporting TCAD-related information flow. The VISTA widgets are also created and accessed via specific functions, so that they can more easily be replaced by other widgets, should the need arise.

The top layer, the VUI (*VISTA User Interface*) library serves two purposes. It provides some often needed higher-level operations and it simultanously contains most of the policy which is shared among VISTA applications. In other words, the VUI library takes care that different parts of VISTA look alike and behave similar. Interactive applications (like visualization clients or the device editor) have their own VUI-based user interface, whereas applications requiring no user interaction (like simulators or converters) are provided with a front-end user interface which is executed by the XLISP interpreter.

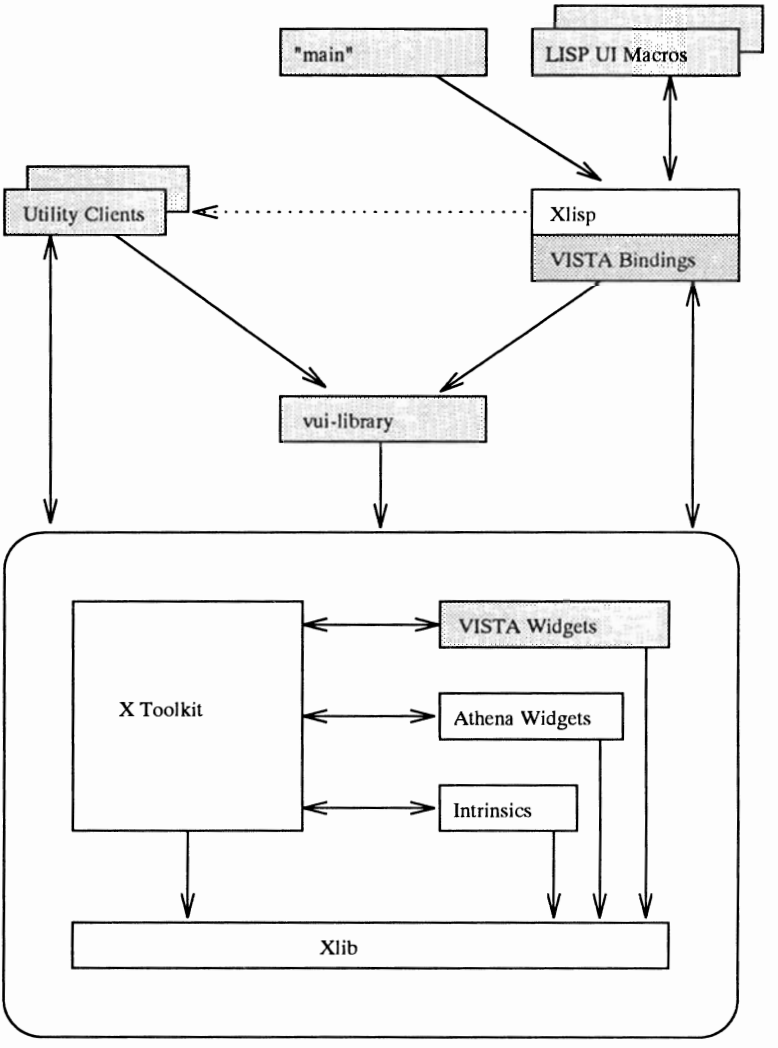

Figure 21: The structure of the VISTA user interface. Shaded boxes represent extensions to the public domain products XLISP and the MIT X Window system. The arrows indicate the sequence of function calls between different parts of the user interface.

5.2 The VISTA Widget Set

The *Athena* widgets were developed by MIT's project Athena (it is part of the X release). This 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.

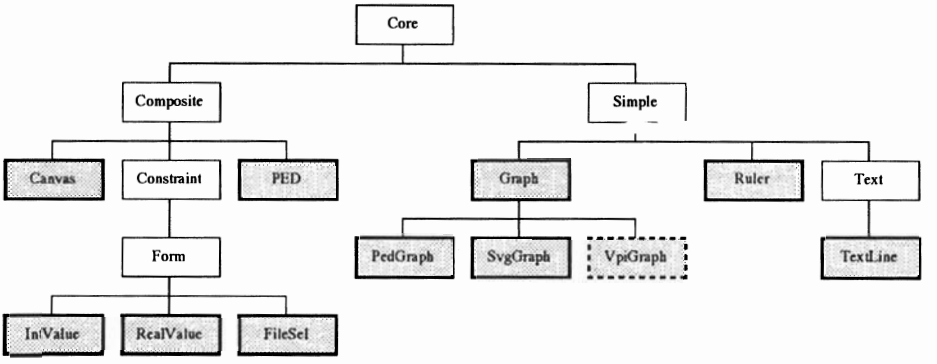

Figure 22: The widget set used by VISTA. The VISTA extensions are shaded, the Intrinsics and Athena Widgets used for subclassing are blank.

The VISTA widgets are subclassed from either X Toolkit Intrinsics or Athena widgets (Fig. 22). 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.

5.2.1 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. 23)

The PED makes use of the *Canvas*, *Ruler*, and *PedGraph* widgets and is implemented as a widget itself (see Fig. 23). 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 PIF data is held in a memory-resident intermediate representation which is slightly extended with respect to the binary PIF to allow efficient processing of interactive manipulations. At the end of the editing session or during intermediate save operations the PIF file is updated.

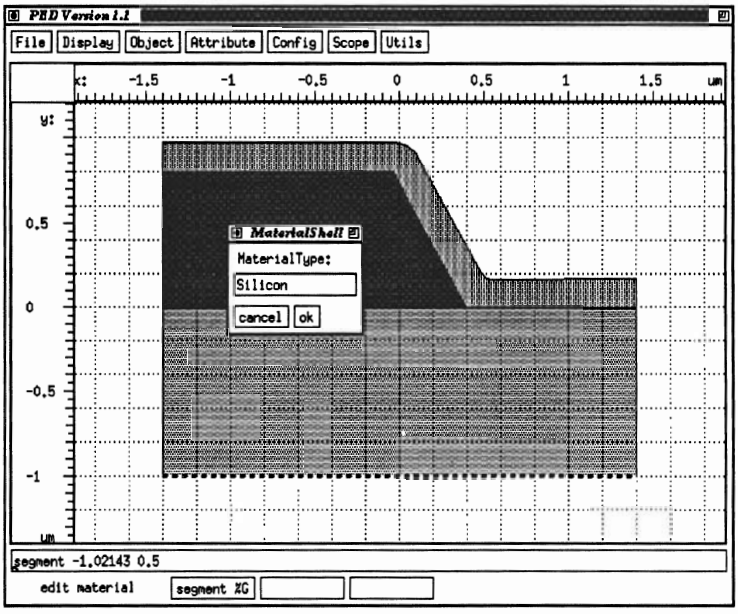

Figure 23: The PIF editor widget.

The hierarchical 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 coloring according to different physical or logical criteria are used to facilitate the comprehension and assimilation of the spatial information.

The top-level execution control of the PED is implemented as an extensible and configurable automaton which filters all user input and triggers appropriate actions. Through the default configuration for this automaton, all graphical functionality is accessible via mouse and keyboard input is used for all non-graphical data.

The implementation as a widget, together with the overall architecture of the user interface and task-level allows LISP-coded editing macros to be added easily. It should also be noted that the use of LISP as an extension language for an interactive geometry editor has already proven to be a very successful strategy for interactive geometrical CAD[21].

5.2.2 Vector Graphics Widget

The X Toolkit 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. 24) which is built directly on the generic Xlib and X Toolkit.

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. Call-backs can be utilized for example to digitize data points. This widget is used as interactive back-end of VISTA's visualization library.

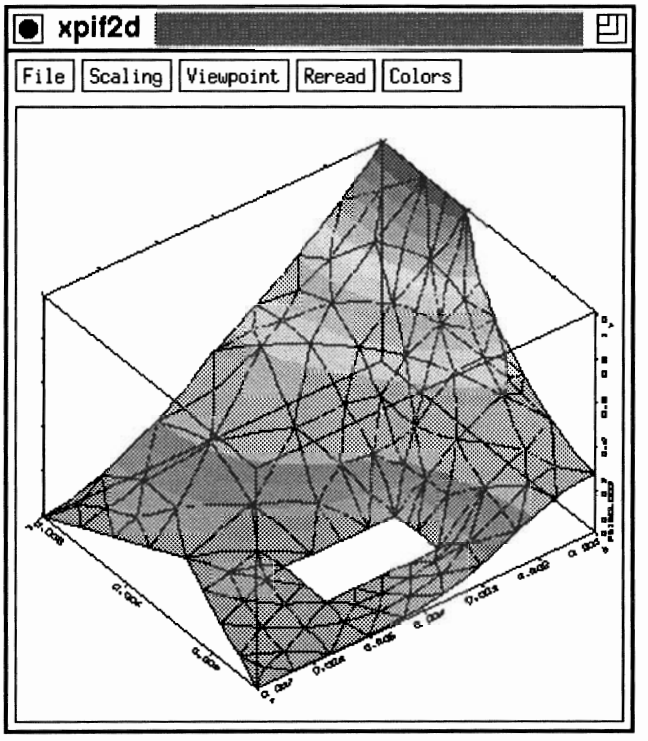

Figure 24: The vector graphics display widget is used for displaying the output of PIF-based visualization tools.

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

5.2.3 File Selection

As it is not provided with the Athena widget set, we have implemented an advanced file selection widget (see Fig. 25), which allows operating system transparent specification of files (including a *GNU Emacs* like filename completion) using a string subwidget and operating system independent traversal of the directory tree and selection of existing files using list subwidgets.

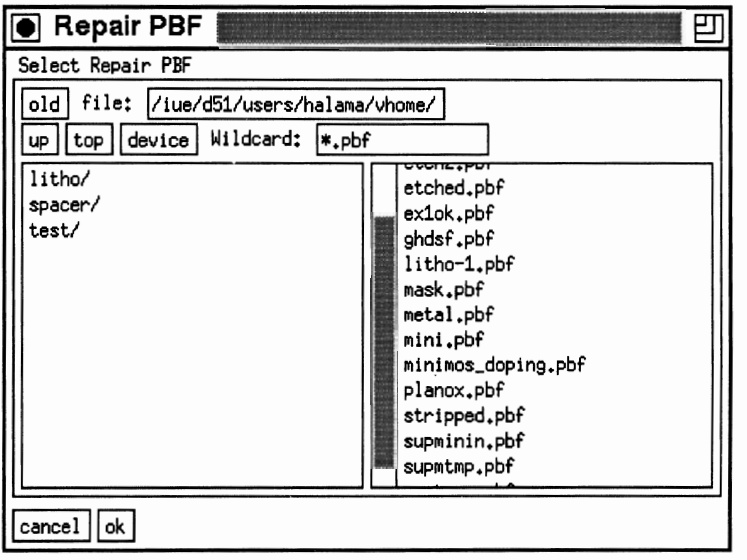

Figure 25: The VISTA file selection widget.

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

5.3 The VUI Library

5.3.1 Bouquets

The VUI library contains functions which create often-used combinations of several widgets in one step, arrange them and set up all required connections and callback functions. These widget *bouquets* behave as if they were single composite widgets and are indistinguishable from the user's point of view. 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 widget bouquets for performing TCAD-related subtasks:

The periodic table shown in Fig. 26 is implemented in C and is used by applications to let the user select a "pure element" material from a material database, which is

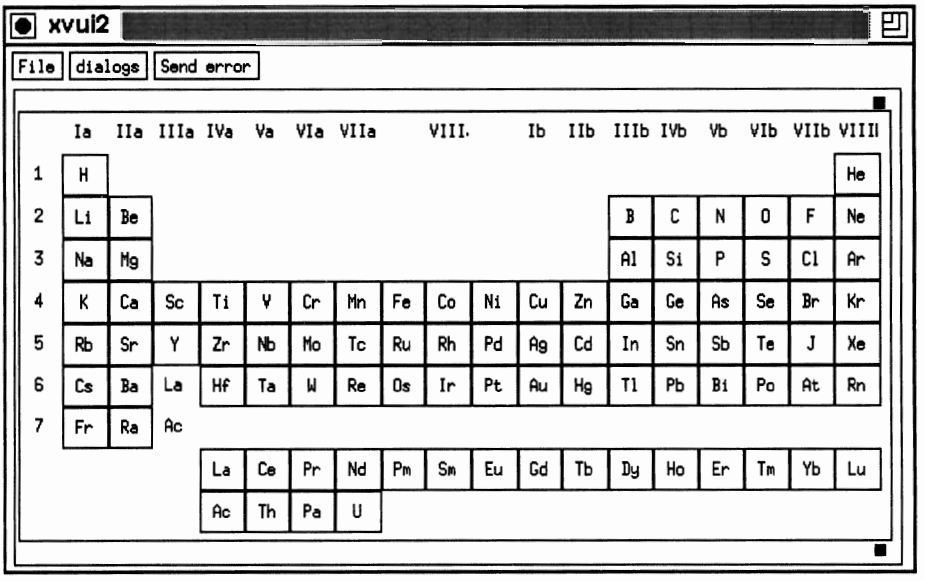

Figure 26: The periodic table bouquet lets the user select chemical elements as bulk or implantation material.

shared by all (fully integrated) simulation tools. It provides a familiar method for the identification and specification of single chemical elements. This bouquet is used, e.g., to ask the user for the bombarding ion species for the Monte Carlo simulation of ion implantation.

The symbolic PIF browser bouquet (see Fig. 27) is generated by a protoype 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.

5.3.2 Tool Control Panels

Applications which don't require user interaction (like all classical simulation programs) can very easily be provided with a "supply parameters and run" user interface. The widget bouquets shown in Fig. 28 and Fig. 30 are tool control panels which are created from a formal specification of the tool and its parameters by an 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 X Toolkit programming to make new tool control panels.

6. Implementation of the Task Level Shell

The applicability of XLISP for the execution of practical TCAD optimization tasks has already been demonstrated[28]. The programming language features offered by LISP are a powerful and efficient basis for carrying out complex task flows, like, e.g., nested optimization loops.

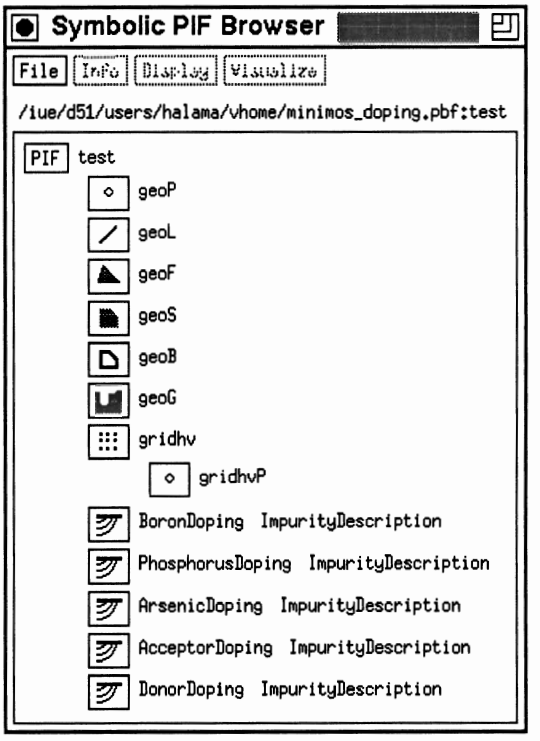

Figure 27: The symbolic PIF browser bouquet is created according to the data contained in a (binary) logical PIF file and reflects its hierachical structure.

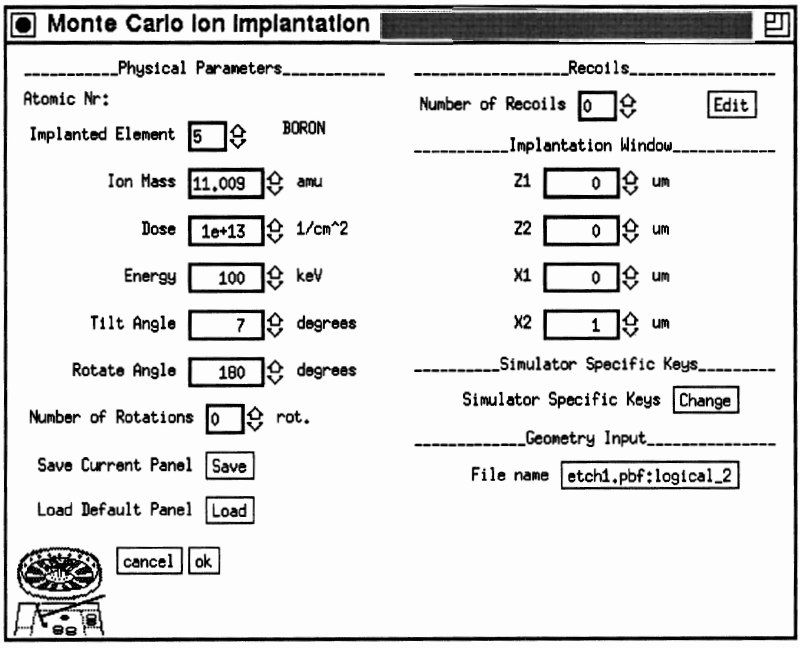

Figure 28: This widget bouquet is the user interface for the PROMIS Monte Carlo ion implantation module.

The *XLISP* interpreter which is used for the task level implementation of VISTA has also bindings to the *VUI-library*, the *X Toolkit*, and the widget sets. This integration of the user interface into the task level interpreter (which can be viewed as the "main program" of the TCAD system) is indispensable for supporting the casual user with a comfortable point-and-click interface and for providing the experienced user and programmer with a high degree of customizability and extendability.

Furthermore, some parts of the user interface have been implemented as *LISP macros* and are loaded at run-time. The XLISP integration has also proven to be a valuable tool for the rapid prototyping of user-interface concepts and for tasks like the automatic generation of widget bouquets, like the tool panels in Fig. 28 or Fig. 30.

The X Window interface is not a feature of the original XLISP interpreter, although OSF/Motif bindings[22] are available, which, unfortunately, besides the disadvantages mentioned earlier, make heavy use of the "non-standard" object system of XLISP.

To preserve the consistency and simplicity of XLISP and in order to provide a homogeneous procedural interface and programming environment, we had to implement the X Window interface (*VISTA UI Bindings* in Fig. 21) for XLISP from scratch. 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 size of the original XLISP interpreter (Version 2.1) is 270kB. Some specific extensions have been made to the interpreter, mainly for event handling and for introducing miscellaneous operating system interfaces, like for process control (for running simulators), and other features which are needed for TCAD purposes. These 200kB of extensions thus increase the size of the task level interpreter to 470kB.

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

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.

The code required to implement LISP interfaces for framework modules which are relevant for the task level is currently 335kb (see 1), but is steadily increasing. There are only a few functions (indicated by parentheses in Table 1) which are manually bound to the interpreter. The vast majority of the interface code is generated automatically during the build phase of VISTA.

In order to integrate the X11 event handling mechanism in a consistent way, the standard read-eval-print loop was extended to handle events coming from the X11 system: Events coming from the user interface (like an expose request for a window) are passed to the interpreter and processed simultaneously with keyboard input or events coming from other (network) streams connected to the XLISP interpreter or from signals which indicate events like child process termination. During the read phase, the interpreter first checks whether the evaluation queue of pending expressions needs to be emptied and optionally evaluates these "callback" expressions. Then it waits for any event coming from streams, terminal input, signals (e.g. from terminating simulator runs), or the X Window system. When any of these events occurs, the interpreter puts the associated callback expression into the abovementioned evaluation queue — or in the case of terminal input — waits for completion of the expression and then evaluates it.

6.1 The Callback Concept

The object-oriented callback concept of the X Toolkit 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.

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.

By agreeing upon a standard prototype for callback functions (which is already specified by the X Toolkit):

```
void callBackFunc(object_identifier, client_data, call_data);
```

it is possible to use a unified consistent method in various places throughout the framework, thereby gaining in simplicity and flexibility.

6.2 Task Level Tool Integration

As already pointed out earlier, there are different methods to create a task level interface for a given tool. Linking the tool with the interpreter is only feasible for small-sized applications, which have library-like properties. For larger tools (like simulators) a LISP interface must be coded to be able to run them from within the TCAD shell.

Fig. 29 shows a simple example of the task level interface of a conversion tool, which converts SUPREM 3 output files to PIF. The corresponding tool control panel, which is automatically created at run-time, is shown in Fig. 30.

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 yet this property that dictates the rigorosity 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 favoring the generalization of existing concepts over the introduction of new (maybe even better suited) partial solutions. A bottom-up *design* has been used, wherever possible, to be prepared for unforseeable future requirements.

Using PIF as the interchange format of the data level was the initial choice, which was motivated by the sole existence of the PIF standard at that time, and 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 X Toolkit 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[6][7][22] of an interpretive language for composing a user interface from widget-level building blocks have verified the choice of XLISP as extension language interpreter. The generalized use as extension language interpreter, "main" program of the user interface, and central facility for CASE-related tasks contributes significantly to the consistency and maintainability of the system. Automatic code generation (also using XLISP) helps to raise the level of abstraction on which problems like language binding are solved. Because of its homogeneity, the combination of LISP as the task level extension language and PIF as the data level interchange format has proven to be a flexible basis for a comprehensible and powerful TCAD framework.

```
; This function actually runs the sup2pif executable
;
(defun sup2pif (asc pbf pif)
  (vos::run
     (vos::os2vospec "sup2pif")
     (format NIL "~A ~A ~A" asc pbf pif)
  )
)

; This function is called when the confirm button is pressed
;
(defun sup2pif-confirm (widget client-data call-data)
  (sup2pif
     (vos::get-vospec  (vuu::get-value 'ASC client-data))
     (vos::get-vospec  (vuu::get-value 'PLB client-data))
     (vos::get-logical (vuu::get-value 'PLB client-data))
  )
)

; This defines the control panel for the suprem 3 wrapper
;
(setq sup2pif-widget
  (vui::generate-tool-panel
    (setq sup2pif-description
      `(
         (NAME     "SUPREM3 to PIF")
         (ICON     "topif")
         (CONFIRM  #'sup2pif-confirm)
         (FORM
           (ASC  VOSFILE  "SUPREM3 .sav file: "  "*.sav" NIL)
           (PLB  VOSPLB   "       output PLB: "  "*.pbf" "sup3prof.pbf")
         )
       )
    )
  )
)

; This creates an entry in the pulldown menu, which will pop up
; the tool control panel
;
(vui::add-to-menu vui::current-menu-button
      "suprem3->PBF"
      #'(lambda (widget client-data call-data)
          (vui::popup-tool-panel sup2pif-widget)
      )
      NIL
)
```

Figure 29: Task level LISP code which is required to integrate the SUPREM 3 output wrapper.

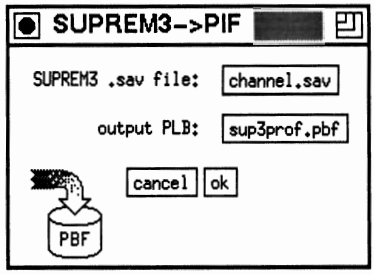

Figure 30: The tool control panel for the SUPREM 3 to PIF converter.

7.1 Tool Abstraction

The use of high-level tool abstraction methods for CAD tool management has been demonstrated through the Cadwell design framework[29]. We believe that a generalized and unified concept for the abstract characterization of tools (even in a low-level fashion, down to single functions) is highly desirable as it can be employed for many different purposes such as the automatic creation of a user interface or the generation of different language bindings. 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 can also be used to describe a tool for optimizers or other sequencing or analysis tools.

Another — already existing — application for a unified tool abstraction concept is the automatic integration of given C functions into the XLISP task level interpreter.

7.2 Visual Programming Interface

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 thus fairly well defined by the arrangement of modules and the flow of data between them, which also implies the sequence of tool execution. Again, both the callback concept (for module activation) and the tool abstraction concept (for module description) can be used for the implementation of a visual programming widget, thereby not increasing the system complexity from the programmer's point of view.

Visual programming is especially well-suited for building specific applications through the 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 approaches. Visual programming puts the process/device engineer in a programmers role without the need to do programming work. It is 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 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.

We are very grateful to

A. Gabara
University of California, Berkeley, California

N. Khalil
Digital Equipment Corporation, Hudson, Massachusetts

P. Lindorfer
National Semiconductors, Santa Clara, California

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, Texas

H. Read
Carnegie Mellon University, Pittsburgh, Pennsylvania

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] F. Fasching, W. Tuppa, and S. Selberherr. VISTA — The Data Level. *IEEE Trans.Computer-Aided Design*, 1993 (in publication).

[2] S. Halama, F. Fasching, H. Pimingstorfer, W. Tuppa, and S. Selberherr. Consistent User Interface and Task Level Architecture of a TCAD System. In *Proc.NUPAD IV*, pp 237–242, 1992.

[3] M.R. Simpson. PRIDE: An Integrated Design Environment for Semiconductor Device Simulation. *IEEE Trans.Computer-Aided Design*, 10(9):1163–1174, 1991.

[4] 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):911–920, 1992.

[5] A.S. Wong. *Technology Computer-Aided Design Frameworks and the PROSE Implementation.* PhD thesis, University of California, Berkeley, 1992.

[6] J.K. Ousterhout. Tcl: an Embeddable Command Language. In *1990 Winter USENIX Conference Proceedings*, pp 133–146, 1990.

[7] J.K. Ousterhout. An X11 Toolkit Based on the Tcl Language. In *1991 Winter USENIX Conference Proceedings*, pp 105–115, 1991.

[8] H. Matsuo, H. Masuda, S. Yamamoto, and T. Toyabe. A Supervised Process and Device Simulation for Statistical VLSI Design. In *Proc.NUPAD III*, pp 59–60, 1990.

[9] P. Lloyd, H.K. Dirks, E.J. Prendergast, and K. Singhal. Technology CAD for Competitive Products. *IEEE Trans.Computer-Aided Design*, 9(11):1209–1216, 1990.

[10] 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. In *Proc.NUPAD III*, pp 55–56, 1990.

[11] 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-7:489–500, 1988.

[12] D.M.H. Walker, C.S. Kellen, D.M. Svoboda, and A.J. Strojwas. The CDB/HCDB semiconductor wafer representation server. *IEEE Trans.Computer-Aided Design*, CAD-12:283–295, 1993.

[13] A.S. Wong and A.R. Neureuther. The Intertool Profile Interchange Format: A Technology CAD Environment Approach. *IEEE Trans.Computer-Aided Design*, 10(9):1157–1162, 1991.

[14] 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):1150–1156, 1991.

[15] SWR Working Group of the CFI/TCAD TSC. *Semiconductor Wafer Representation Architecture.* CAD Framework Initiative, Austin, Texas, USA, 1.0 edition, 1992.

[16] 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. In *Proc.NUPAD IV*, pp 199–204, 1992.

[17] S.G. Duvall. An Interchange Format for Process and Device Simulation. *IEEE Trans.Computer-Aided Design*, CAD-7(7):741–754, 1988.

[18] P.J. Asente and R.R. Swick. *X Window System Toolkit, The Complete Programmer's Guide and Specification.* Digital Press, 1990.

[19] D.M. Betz. *XLISP: An Object-Oriented Lisp, Version 2.1*, 1989.

[20] R. Stallman. *GNU Emacs Manual*, 1986.

[21] *AUTOCAD Release 11 Reference Manual*, 1990. Publication AC11RM.E1.

[22] 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.

[23] F. Fasching, C. Fischer, S. Selberherr, H. Stippel, W. Tuppa, and H. Read. A PIF Implementation for TCAD Purposes. In Fichtner and Aemmer [30], pp 477–482.

[24] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publisher Inc., 1990.

[25] P.H. Winston and B.K.P. Horn. *Lisp.* Addison Wesley, 1989.

[26] Institute for Microelectronics, Technical University Vienna, Gußhausstraße 27–29, 1040 Wien, AUSTRIA. *PAI Release*, 1.0 edition, 1992.

[27] *OSF/Motif Programmer's Guide, Release 1.1*, 1991.

[28] H. Pimingstorfer, S. Halama, S. Selberherr, K. Wimmer, and P. Verhas. A Technology CAD Shell. In Fichtner and Aemmer [30], pp 409–416.

[29] J. Daniell and S.W. Director. An Object Oriented Approach to CAD Tool Control. *IEEE Trans.Computer-Aided Design*, 10(6):698–713, 1991.

[30] W. Fichtner and D. Aemmer, editors. *Simulation of Semiconductor Devices and Processes*, volume 4, Konstanz, 1991. Hartung-Gorre.