VISTA—The Data Level

Franz Fasching, Walter Tuppa, and Siegfried Selberherr, *Fellow, IEEE*

***Abstract*— In order to meet the requirements of advanced process and device design, a new generation of *technology CAD* (TCAD) simulation 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 supported by tools and a procedural interface with multi-language bindings for data storage and retrieval by application programs. In this work, the data level of the *Viennese integrated system for TCAD applications* (VISTA) [1], which includes the *profile interchange format* (PIF), the *PIF binary file manager* (PBFM) and the *PIF application interface* (PAI), is described from a *framework* point of view.**

I. Introduction

THE DEVELOPMENT of integrated simulation frameworks has received considerable attention by the semiconductor industry, as well as universities and other organisations. Amongst the most well-known products are an integrated system for statistical VLSI design [2], the MECCA system from AT&T [3], the PROSE environment from UC Berkeley [4] or the SATURN system from SIEMENS [5].

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 [6], 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 [7], and the BPIF implementation from UC Berkeley [8]. Another data level built on PIF featuring object-orientedness is the PIF/Gestalt system from MIT [9]. A recent approach is the SWR 1.0 specification of the CFI/TCAD TSC [10], defining an object-oriented application interface for TCAD data access.

Manuscript received April 7, 1993. This work was supported by ADEQUAT (JESSI project BT1B) as ESPIRIT project 7236; the "Forschungsförderungsfonds für die gewerbliche Wirtschaft," project 2/285; and the research laboratories of: Austrian Industries—AMS, Unterpremstätten, Austria; Digital Equipment, Hudson, NY; Siemens, Munich, Germany; and Sony Corp., Atsugi, Japan. This paper was recommended by Associate Editor D. Scharfetter.

The authors are with the Institute for Microelectronics, Technical University Vienna, A-1040 Vienna, Austria.

IEEE Log Number 9212370.

II. Requirements and Design

Seen from the applications's point of view, there are numerous requirements which the data level of a TCAD system must satisfy. Firstly, there has to be a simulation database where simulation problem descriptions, histories and results are stored. A clear, syntactic 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 operating system- and machine-independent, to achieve easy portability to different platforms. Compatibility with external TCAD tools must be ensured by facilitating their migration into the framework through the use of appropriate concepts.

The procedural interface must be characterized by its ease of use, and an orthogonal architecture, to minimize the effort involved in the creation of new tools. The interface should automatically perform conversions of coordinate systems, physical units, simulation grids, etc. so that the application can concentrate on its actual task. Fast random access to simulation data and compact database sizes are crucial for 3-D simulators, so these issues cannot be neglected when designing a procedural interface. 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. Flexibility is needed to support all possible framework architectures (client–server, master–slave, parity, ...). It is questionable, if a true client-server architecture using the network will exhibit the required performance in a TCAD environment, since large data amounts like grids, attributes or solver stiffness matrices (especially in 3-D) have to be communicated between client and server. Although mapping client memory into the server substantially improves performance, this approach is neither portable, nor works over the network.

In fact, a confinement to a specific architecture would result in an inflexible data level and thus lead to a framework that cannot be adapted to the environmental needs of a simulation site. A precondition for this 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

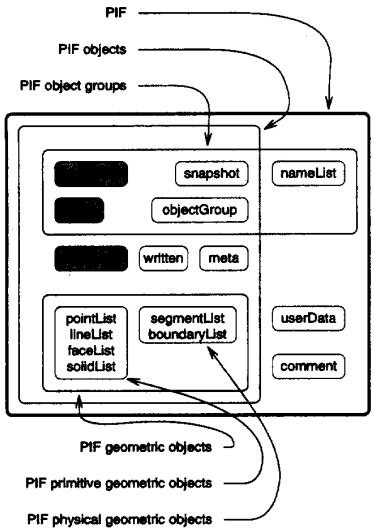


Fig. 1. The logical PIF structure.

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

Fig. 2. 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. 3. Attribute defined on a grid.

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. Since many different tools have to interact in a TCAD framework, the data level has to apply semantic rules to ensure the "understandability" of common simulation data for all tools integrated in the environment.

Thus, 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 storage format (*PIF logical binaries*, PLB's)
- Parallel access to PLB's
- The ability to build databases of PLB's into *PIF binary files* (PBF's)
- Database utilities to manage PBF's
- Networking capabilities.

The procedural interface to the database services is called *PIF application interface* (PAI, [11]) and makes extensive use of automatic code generation to achieve platform independence and generate the individual language interfaces. It is described in detail in the next chapter.

The PIF plays a very important role in the VISTA data level. Its ASCII format, first proposed by Duvall [12], was modified to meet the requirements of a TCAD framework. It is used as an intersite data exchange format and is, in its its binary form [11] the database storage format of the data level. Fig. 1 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. 2 shows a `materialType` attribute defined over a segment and Fig. 3 shows a 3-D `electricField` attribute defined over a 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 PLB's, 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 PLB's, 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

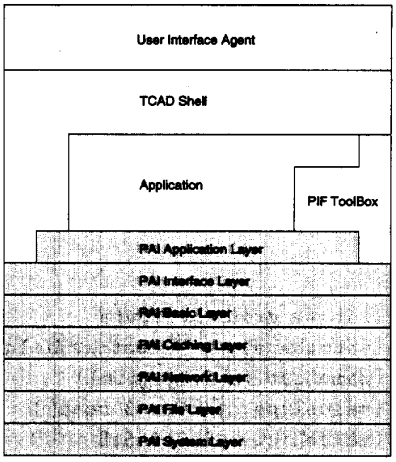


Fig. 4. Layout of the PIF application interface.

size, supposed the operating systems file size limitation is higher. Since the PAI is capable of opening up to 16 PLB's, an application has a maximum of 16 gigabyte of data available. Typically, a single PBF holds one or two PLB's containing a geometry, attributes and grids of a single tool run. Through the special `link` construct objects in other PLB's or even other PBF's 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 XLISP as the VISTA task-level extension language: PIF uses a LISP-like syntax and XLISP as the task-level language gives way to a 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.

III. Implementation of the Application Interface

The PAI is split into seven layers with strict interfaces between each other. The different layers are shown in Fig. 4.

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 PBF's (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 3.10), 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 common file-based systems.

3.1. 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 PBF's through the error system if a memory fault or address violation occurs.

3.2. 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).

3.3. 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 PBF's. 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

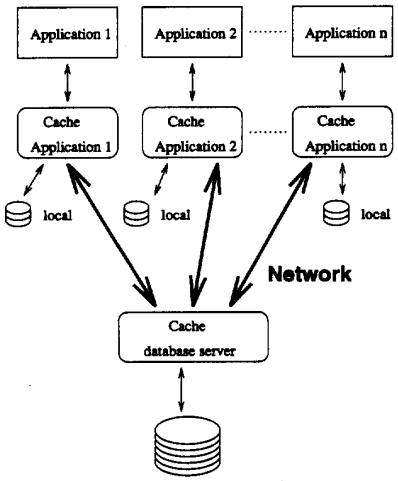


Fig. 5. Local and network storage.

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 PBF's for intermediate storage of simulation data. Temporary files are stored in PBF's without a physical name and deleted automatically upon closing.

3.4. Network Layer

The functional interface exhibited by this optional layer is equivalent to the one of the file layer for access to PBF's, but allows instead accesses to PBF's over the network. In order to minimize network traffic, the functions of the file layer are used for local and temporary PBF's. The network databases are accessed through a database server as shown in Fig. 5, which opens, reads, writes and closes PBF's.

The client uses the database server for all file I/O functions on the network PBF's, 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. 6 shows an example network configuration with tools and database servers interacting over the network.

3.5. 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 [13]. 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.

3.6. Basic Layer

This layer is the lowest to implement structured data nodes. Fig. 7 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.

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

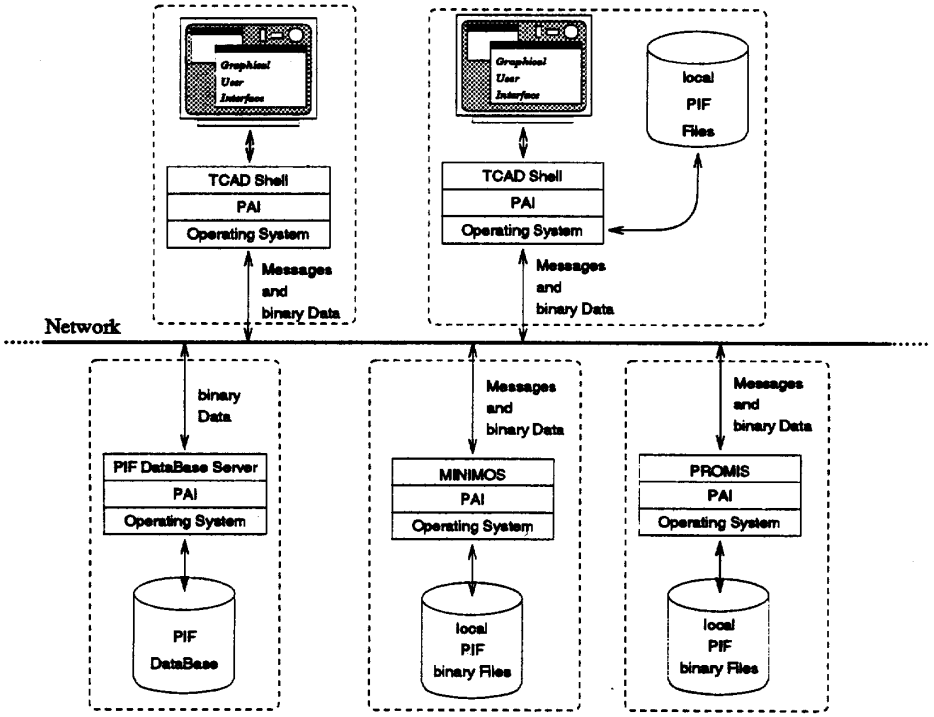


Fig. 6. Examples of PAI networking capabilities.

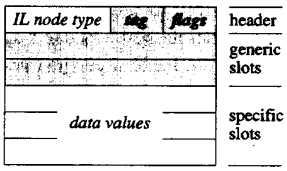


Fig. 7. Implementation of a basic layer mode.

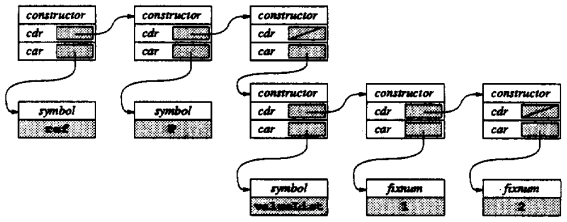


Fig. 8. Example of a LISP internal data structure.

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 [14]. 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. 8 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. 9. 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 PLB's, shown in Fig. 10. Since this list is only searched when the file is opened, this is no performance drawback. The data area is shared by all PLB's in the PBF, but on write operations it is checked that no crosspointers into disjoint PLB's occur. To allow fast access to all symbols, these are stored in a hash table

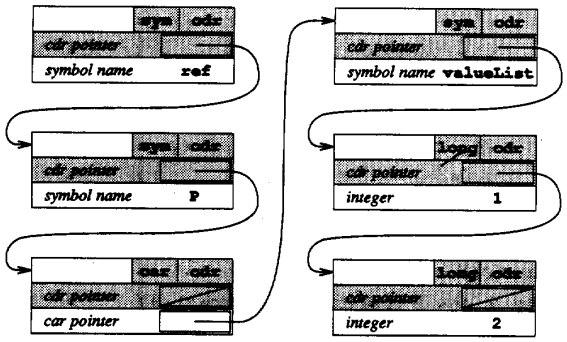


Fig. 9. Example of a PIF binary file and basic layer data structure.

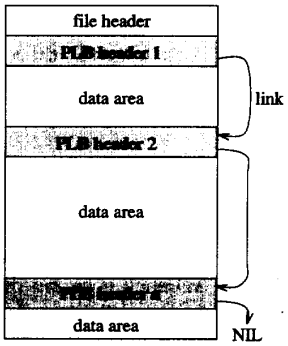


Fig. 10. Layout of a PIF physical file with multiple logical files.

which is unique for each PLB so that there are no conflicts between different PLB's.

3.7. 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. 7 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. 11.

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

Fig. 11. Abstract syntax description of the PIF `snapshot` construct.

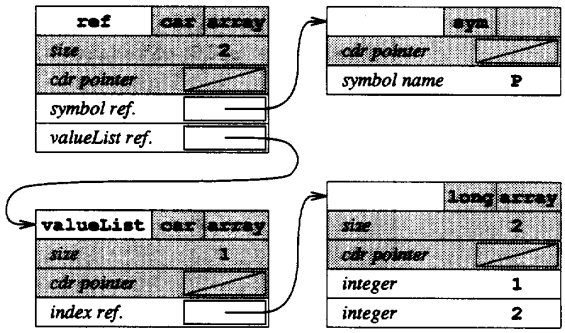


Fig. 12. Example of an interface layer data structure.

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 `reference` construct represented with interface layer nodes is shown in Fig, 12. 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. 9, 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 [15]) which defines the semantic meaning of the PIF.

3.8. 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

```
/* 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. 13. Interface layer code example.

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.

3.8.1. 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.

3.8.2. 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 PLB's 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 PBF's. 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.

3.9. Procedural Interfaces

The short code example in Fig. 13 shows the C calls to generate an example of a PIF data structure. `namelist` is the handle to the parent `nameList` object [15]. The two element array `points` holds the indices of the points on which the line is created. In the Application Layer code example Fig. 14 this part of information is generated by the function `pal-WriteLineList1`. In addition to the reference construct this function generates the whole `lineList` construct, as can be seen in Fig. 15.

3.10. 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 PLB's human-readable, there is the ASCII PIF representation holding the same information. The PBFM 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 PLB's 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 PLB's of a PBF, delete any PLB within

```
{
    /* 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. 14. Application layer code example.

a PBF, repair a not cleanly closed PBF and check PIF ASCII files for lexical correctness (Fig. 16).

IV. Semantic Issues

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 PLB's. 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 [15] have to be obeyed. The PAI takes care of different

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

Fig. 15. PIF construct produced by example code.

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 PLB's 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 [15]. However, these tools are controlled by the task level and belong to the tool rather than to the data level.

A particularly 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

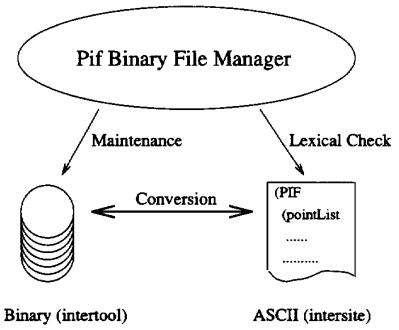


Fig. 16. Using the PIF binary file manager.

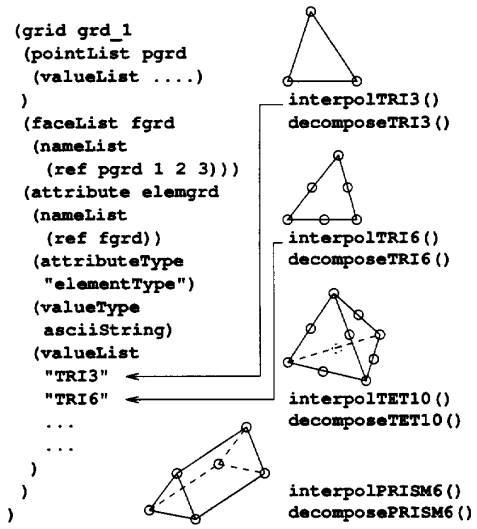


Fig. 17. Support for unstructured grids.

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. 17 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.

```
(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))
)
```

Fig. 18. Tensor product grid example.

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. 18 shows a tensor product grid of topological dimension 2 lying in 3-D space. This capability is needed e.g. to describe distributed boundary conditions of a 3-D 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 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.

V. Performance Evaluation

Besides the goals of classical intertool PIF implementations featuring object-orientedness (PIF/Gestalt, [9]) or suitability for TCAD environments (BPIF, [8]) our implementation stresses efficiency in terms of run-time performance and database compactness. Thus, writing and reading 10 000 3-D points of a PIF `pointList` takes 0.51 and 0.66 s (real time) respectively on a DECstation 3100; the database written is 250 kBytes 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.

VI. Conclusion and Future Aspects

Using PIF as the interchange format of the data level of a TCAD system is an effective choice. However, the crucial part of the data level is an efficient application interface like the VISTA PAI. The source code of the VISTA PAI is available free of charge from the Institute for Microelectronics [15]. 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 powerful TCAD framework.

6.1. Graphic PIF

On top of the PAI's basic layer, almost any imaginable procedural interface for LISP-based languages can be built using automatic code generation tools. This is used to create a special Graphic PIF (GPIF) for a comprehensive device editor which is currently under development, which supports backward referencing and strips unnecessary PIF data.

6.2. 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.

References

[1] S. Selberherr, F. Fasching, C. Fischer, S. Halama, H. Pimingstorfer, H. Read, H. Stippel, P. Verhas, and K. Wimmer, "The Viennese TCAD system," in *Proc. VPAD 1991*, Oiso, Japan, pp. 32–35, 1992.

[2] H. Matsuo, H. Masuda, S. Yamamoto, and T. Toyabe, "A supervised process and device simulation for statistical VLSI design," in *Proc. NUPAD III*, Honolulu, HI, 1990, pp. 59–60.

[3] P. Lloyd, H. K. Dirks, E. J. Prendergast, and K. Singhal, "Technology CAD for competitive products," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 1206–1216, 1990.

[4] A. S. Wong, Technology CAD Frameworks and the PROSE Implementation Ph.D. dissertation, Univ. Califonia, Berkeley, CA, 1992.

[5] H. Jacobs, W. Hänech, 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*, Honolulu, HI, June 1990, pp. 55–56.

[6] 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*, vol. CAD-7, pp. 489–500, Apr. 1988.

[7] 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*, vol. 12, pp. 283–295, Feb. 1993.

[8] A. S. Wong and A. R. Neureuther, "The intertool profile interchange format: A technology CAD environment approach," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 1157–1162, Sept. 1991.

[9] D. S. Boning, M. L. Heytens, and A. S. Wong, "The intertool profile interchange format: An object-oriented approach," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 1150–1156, Sept. 1991.

[10] SWR Working Group of the CFI/TCAD TSC, *Semiconductor Wafer Representation Architecture*, 1.0 edition. Austin, TX: CAD Framework Initiative, July 1992.

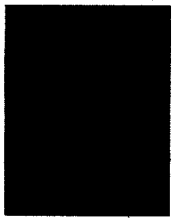
[11] F. Fasching, C. Fischer, H. Read, S. Selberherr, H. Stippel, and W. Tuppa, "A PIF implementation for TCAD purposes," in *Proc. SISDEP 1991*, Zürich, Switzerland, 1991, vol. 4, pp. 477–482.

[12] S. G. Duvall, "An interchange format for process and device simulation," *IEEE Trans. Computer-Aided Design*, vol. CAD-7, pp. 741–754, 1988.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. New York: Morgan Kaufmann, 1990.

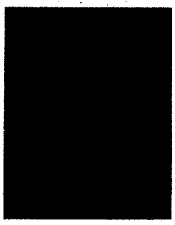
[14] P. H. Winston and B. K. P. Horn, *LISP*, 3rd ed. Reading, MA: Addison-Wesley, 1989.

[15] Institute for Microelectronics, *PAI Release*, 1.0 edition. Vienna, Austria: Technical Univ. Vienna, March 1992.

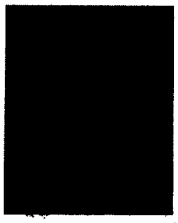


Franz Fasching received the degree of "Diplomingenieur" in control theory and industrial electronics from the Technical University Vienna in 1989. In February 1990, he joined the "Institut für Mikroelektronik," where he is currently working towards his doctoral degree.

His work is focused on the VISTA Technology CAD environment and improvement and extension of the capacitance simulator VLSICAP.



Walter Tuppa received the degree of "Diplomingenieur" in computer science from the Technical University Vienna in 1991 and in electrical engineering in 1992. He joined the Institut für Mikroelektronik in July 1991, where he currently works on the VISTA project towards his doctoral degree in electrical engineering.



Siegfried Selberherr (M'79–SM'84–F'93) received the degree of "Diplomingenieur" in control theory and industrial electronics from the Technical University of Vienna in 1978. He finished his thesis on "Two Dimensional MOS-Transistor Modeling" in 1981, and received the "venia docendi" on "Computer-Aided Design" from the Institut für Allgemeine Elektrotechnik und Elektronik—previously called the Institut für Physikalische Elektronik—at the Technical University of Vienna, in 1984.

He has been the head of the Institut für Mikroelektronik since 1988. His current topics are modeling and simulation of problems for microelectronics engineering. He has authored and coauthored more than 200 publications in journals and conference proceedings, as well as the book *Analysis and Simulation of Semiconductor Devices*.

In 1983 Dr. Selberherr received the Dr. Ernst Fehrer award; in 1985 he received the award of the "Nachrichtentechnische Gesellschaft;" in 1986 he was honored with the "Dr. Herta Firnberg Staatspreis;" and in 1987 he received the "Heinz Zemanek" award. He is a member of the Association for Computing Machinery (1979), the Society of Industrial and Applied Mathematics (1980) and the Verband deutscher Elektrotechniker. He is editor of *The Transactions of the Society for Computer Simulation, Electrosoft* (since 1992), *Mikroelektronik* and the book series *Computational Microelectronics* (Springer-Verlag).