SFB-Bericht 26/88, SFB 124, University of Kaiserslautern, 1988

Supporting Engineering Applications by new DB-Processing Concepts - An Experience Report

Christoph Hübel , Bernd Sutter
University Kaiserslautern
Erwin-Schrödinger-Straße
D - 6750 Kaiserslautern
Federal Republic of Germany

Abstract

Database management systems for non-standard applications, in particular for engineering applications (NDBS), constitute nowadays one of the most important challenges in the area of database research. Some of the major obstacles are concerned with problems of modelling and processing complex engineering objects. Some new kinds of overall system architectures have been proposed, and appropriate concepts for handling the new types of application objects have been developed in the last five years.

Based on PRIMA, an NDBS-kernel prototype implementation, we motivate a workstation-oriented architecture for NDBS application systems. We explain a prototyped application system in the environment of VLSI-chip design, which serves as a practical example in handling complex objects. Analyzing the weaknesses of this initial approach, we derive general concepts for application linkage, discussing in particular key issues for an efficient object processing and language binding.

1. Introduction

Adequate modelling and efficient processing of application objects constitute the key problems of employing database management systems (DBMS) which support so-called non-standard applications. By comparison with conventional applications, the relevant entities of the non-standard 'world' are much more complex, i.e. they are internally structured and consist of heterogeneous components. Describing such entities with means of a conventional data model results in a *modelling problem.* Based only on simple tuples, records, and sets, etc., it is very difficult or nearly impossible to describe and process these application objects as a whole, including all essential semantic aspects. In addition, application-independent operations, such as insertion of new objects, selection of objects with specific (complex) properties, or object deletion, as well as object modification must be tediously implemented by a great number of elementary operations on elementary objects. The molecule-atom data model (MAD) outlined in /1/, is one step in solving these problems (other approaches are described in the literature /2/, /3/, /4/). Atoms are the basic elements of the MAD model used to represent the real world entities. They play a similar role to tuples in the relational model. Each atom is composed of attributes of various types, is uniquely identifiable, and belongs to its corresponding atom type. The attributes' data types can be chose from a richer selection than in conventional models yielding a more powerful structuring capability at the attribute level. Relationships between atoms (entities) are expressed by so-called links that are defined as link types between atom types. Links are used to efficiently map all types of relationships. This direct mapping and the provision of bidirectional yet symmetric links (represented via reference/back-reference pairs) greatly increases the flexibility of the MAD model. Thus, in the database all atoms are connected

by links form meshed structures. Based on this atom network, it is feasible to dynamically construct molecules using atoms as elementary building blocks. Molecules are defined within statements of the so-called molecule query language (MQL) and have to be derived at run time. Each molecule belongs to a molecule type specified in the MQL query. This type description establishes a connected, directed, and mainly acyclic type graph as subgraph of the database schema (this graph becomes cyclic only if recursive types are involved). Thus, each type description specifies a starting point (i.e. root atom type) and all participating atom and link types. MAD's most important objectives can be characterized as follows:

- dynamic definition and derivation of molecules
- molecules are dynamically derived based on the elementary building blocks called atoms
- molecules consist of structured sets of atoms of possibly different types
- all relationships among atoms are represented in a direct and symmetrical way allowing for shared subobjects
- set oriented molecule processing is offered by the descriptive SQL-like query language MQL

Having in mind all these characteristics, we believe that the MAD model constitutes a powerful tool for the description and the representation of engineering world's entities. Until now, we have developed a prototype system that offers MAD as its data model interface. This system is called PRIMA (PRototype Implementation of the Molecule Atom data model) /5/, and serves in the main as a research vehicle gaining experiences in both implementing and utilizing the proposed NDBS concepts.

Similar to the modelling problem, the use of processing schemes, which are associated with conventional data models, results in a *processing problem* for non-standard applications. The traditional schemes are restricted to the simple data structures provided by such data models as well as to the 'one tuple at a time' semantics determined by the concept of 'subschema' application linkage. However, the application programs prefer to handle the molecular objects in an application-oriented manner, since these represent the application entities. Therefore, application-dependent operations have to be integrated into the NDBS. In this paper, we discuss the concepts of an additional system layer (as part of the NDBS), the so-called application layer (AL), giving a solution to the processing problem

AL constitutes the topmost layer of a non-standard database system (NDBS), as described in /6/. Fig. 1 shows the overall NDBS architecture and illustrates the central role of AL as the link between the application and the application-independent NDBS-kernel system, i.e. between the *application model interface* and the *data model interface.* An application model is characterized by a special interface which supports a specific class of applications by offering expressive, application-oriented data structures combined with their corresponding operations. Based on the data model objects (molecules and atoms), the AL achieves this orientation towards the application. The encapsulation of data and algorithms results in

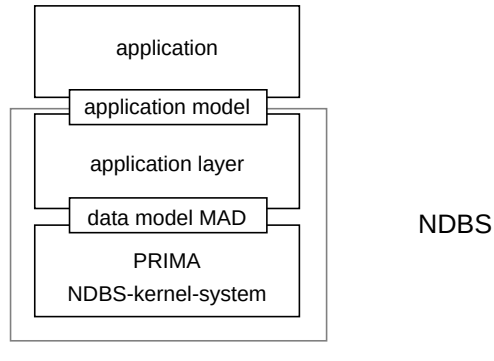


Fig. 1: Our overall NDBS architecture

a kind of abstract data type (ADT) /7/. Some approaches to integrate ADT's into data models are described in literature /8/, /9/. These investigations are founded on the same aim: ADT's should enrich the modelling capability by aspects of procedural description. So it is in our case, the ADT operations constitute the AL programs which have to manipulate the neutral, but complex molecular objects delivered by the data model interface. Since this is done by means of conventional programming languages, we require adequate concepts for language binding.

The remainder of this paper is organized as follows. In the next chapter, we discuss some architectural aspects placing further restrictions on the application linkage. In chapter 3, we introduce a concrete application in the environment of VLSI-chip design. The implementation issues of the prototyped design tool is described in detail, in order to illuminate our first approach in processing molecular objects. These investigations deliver arguments and examples which influence the discussion of general concepts for application linkage. Based on these insights, we explain our refined approach in efficient molecule processing and illustrate the key issues for AL organization.

2. Architectural Aspects of Engineering Applications

New kinds of computer-based applications require new and suitable concepts for data management support. The following account sketches the characteristic properties of such applications, especially in an engineering environment. Engineering applications are in the main:

- Data intensive: one reference at the application level normally produces a great number of references at the representation level (i.e. employing a conventional DBMS this will be the data model level).
- I/O intensive: often, the results of application operations can only be understood by representing them on a graphic screen.
- Dialogue oriented: the engineering user must frequently interact during the development process; this implies that response time is one of the most important optimization subjects.
- Power consuming: the algorithms carrying out the application operations, often consume a great deal of processor power.

In addition, there are further requirements from the application environment, which concern:

- special I/O functions (graphic I/O, acoustic I/O, analogous I/O, ...),
- dynamic extensibility of system architecture (an application system for one or two engineers should be extensible to a system for one or two hundred of engineering users, without any loss of performance and without any change of the system's interface).

The consideration of these application properties leads to a workstation-based hardware architecture. In this case, we have decentralized and autonomous processors. Each processor provides processing power to a fixed number of users (mostly, it will be one user). Coupled via a local area network (LAN),

the workstations have access to a central database server maintaining all the data representing the application environment.

When the number of workstations is increased, the overall application system will be able to serve an increased number of users. However, from a certain number, response time will overrun, because of the bottleneck given by the central database server and the LAN. The closer the coupling between database server and workstation is, the more they will mutually influence one another (due to increasing communication traffic). Therefore, it is a very important aim of system design to realize a loosely coupled system of components, which are more independent and have only a limited need for database access and communication. To reduce the access frequency we introduce a private database, which is managed by the corresponding workstation. The private database can be loaded dynamically from the central DB, with the data representing a specific part of the application environment (this usually depends on current user activities). Normally, data are kept in the private database for a long span of time (hours, days, or weeks) until the engineering user decides to leave his current work in order to finish a single task or to reach the next step of design.

Fig. 2 shows the architecture mentioned above. This proposal implies the usage of a kind of workstation, which offers enough processing power, universal functionalism, and sizable main memory and secondary storage (at least 1 MIPS, 8 Mbytes main memory, and a few hundred Mbytes of secondary storage).

The presented arguments concern not only hardware, but also software aspects. For example, the decentralized components within the distributed system can work autonomously, if and only if the software permits this. So, in the following, we would like to discuss some software requirements. In particular, we answer the question of assigning software and hardware components. To simplify the problem, one must distribute the boxes sketched in Fig. 1 among the boxes shown in Fig. 2. /10/ contains a more general consideration of these architectural aspects. In the context of this paper, we concentrate our explanation on two different approaches. These are illustrated in Fig. 3. /11/ proposed the concept of so-called *tight*

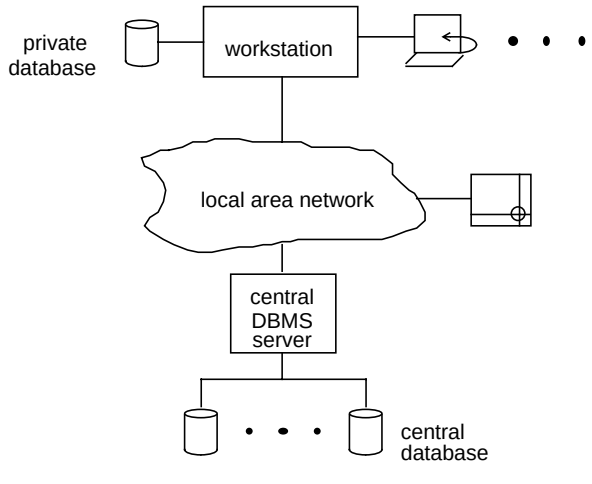


Fig. 2: A detailed hardware architecture for non-standard application systems

database cooperation based on coherent DBMS (resp. NDBS-kernel) for workstation and server site (Fig. 3a). The central idea of this approach consists of the following issues:

- high level for request formulation, and
- low level for data preparation.

This means, the workstation requests are made at the query language level, whereas the *extracted/injected* data are transmitted at page level. This aims to reduce the server processing time, since the database server has only to determine the pages with relevant information (in particular, no projection, only basic qualifications, no join operation, and no type conversion). It seems, however, that there is a serious disadvantage in this proposal: the extracted and transmitted data are not restricted to the required data. Thus, the expected mutual influence is very strong, because multiple database management systems running on multiple workstations have to handle copies of internal data (i.e. description and control data) outside the central database. This has consequences for the validation of access and control structures, too. All these problems are known in the context of multiprocessor database systems /12/. Certainly, there are solutions, but we do not believe that these are very efficient in the context of over one hundred coupled database management systems.

Another approach is shown in Fig. 3b. Here, the NDBS-kernel runs on the central server. The AL and the application themselves are associated with the workstation site. The cooperation between AL and kernel system is also provided at query language level. Contrary to the first approach, the data transmission is made at the data model level. The transmitted data are exactly limited to the required information. Mutual influence is reduced to the case of accessing overlapping information (in particular, when several workstations require the same data). In any case, the dependencies between database server and workstation processing decrease, since no internal structures leave the boundary of the kernel system.

The second approach has the advantage that the transmitted objects already approximate the processing objects. Hence, no further object evaluation becomes necessary on the workstation site. The strongest argument, however, is given by the fact that we use only a single main memory area to buffer subjects of transmission as well as subjects of processing. This so-called *object buffer* is placed on workstation site, and consequently lies near to the programs, which have initially demanded the corresponding objects. Therefore, the access to this data can be implemented in a more efficient way, than in the case of buffering pages in a conventional DBMS buffer /13/, /14/.

With these arguments in mind, the outlined overall system architecture (see Fig. 3b) will constitute our reference architecture for the following considerations.

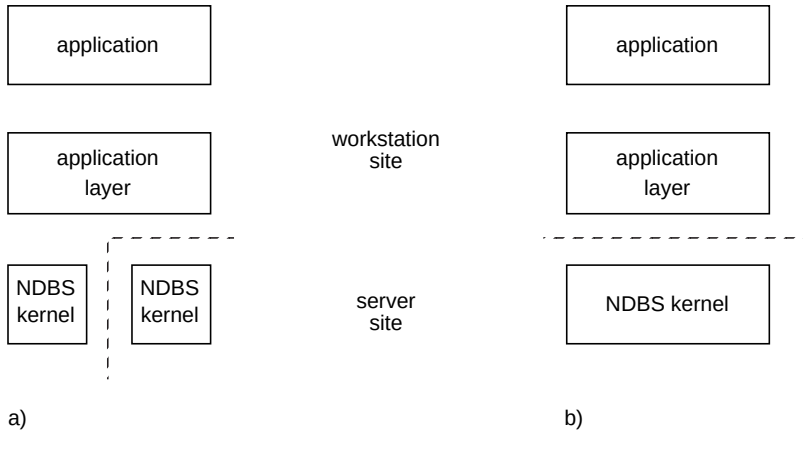


Fig. 3: Distributions of software components

3. A VLSI Application on PRIMA

Now, we present PRIMACHIP, our prototype chip planning tool. PRIMACHIP is used to gain first experiences in processing molecular objects. After an introduction in the VLSI-design process, a brief description of the architecture of PRIMACHIP follows. Then, we discuss the interface to the NDBS kernel, and furthermore, we explain and illustrate our processing model for manipulating molecular objects.

3.1 The VLSI-design process

Todays VLSI technology allows the integration of more than two million elements (e.g. transistors, capacitors or resistors) on a single chip. Therefore, a partitioning of the overall design is necessary. There are two distinguishable division strategies:

- design domains

  /15/ suggests a subdivision of the design process into the behavioral (or functional), structural, and physical part. The behavioral part results in a functional description of the circuits (e.g. algorithms, functions). The structural part describes e.g. functional blocks, register-transfer units, or circuit plans. Finally, the physical part provides details of the physical structures of the corresponding geometric objects.

- hierarchical levels

  Structuring the design process in several hierarchical levels is an orthogonal concept to reduce the design complexity. At the highest level, a chip is divided into cells representing data memory, program memory, I/O-driver, ALU, control unit, etc. At the next level, a cell is partitioned into subunits, e.g. ROM, instruction decoder, or busses. Consequently, these cells can be divided into further subcells (multiplexer, comparator, AND-circuit, etc.). Fig. 4 shows a hierarchical partitioning of a chip into three levels at the example of an floor plan. The cell currently being processed is called 'cell under design' (CUD), and the cells of the next subordinate level constitutes subcells.

In the following, we wish to consider the physical design in more detail. We will orientate ourselves on the PLAYOUT system, a chip planning tool currently under development at the University of Kaiserslautern /16/. A chip planner generates a floor plan for every cell (synonymously called module), i.e. a topo-

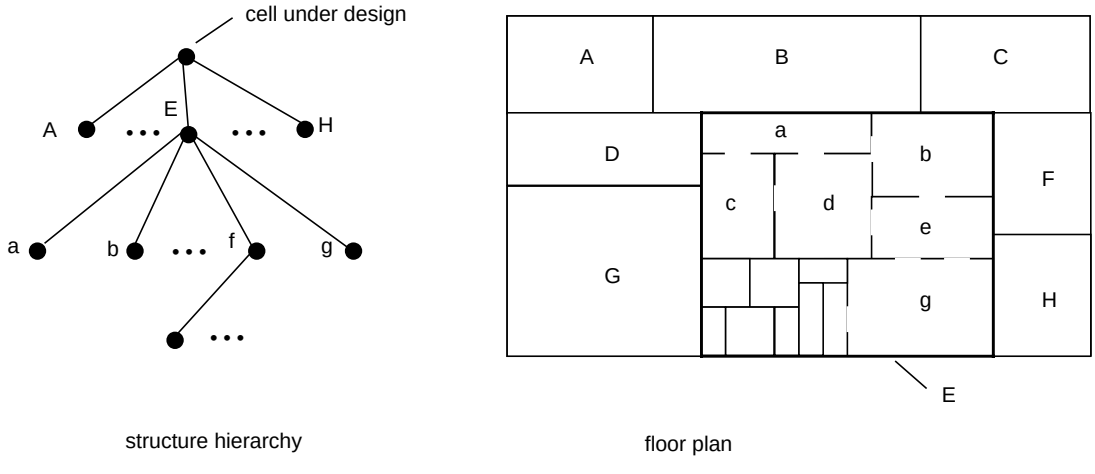
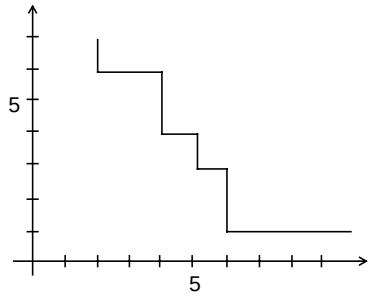


Fig. 4 : Hierarchical partitioning of a chip into three levels

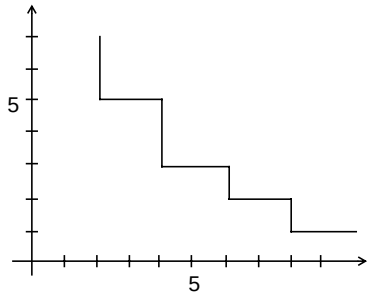
graphical placement of all its submodules and, at the next level, for their submodules. This process is separated in several phases which are illustrated in Tab. 1. As we will see, most phases of the process are double-faced: they consider the actions at one hierarchical level as well as the cooperation between the levels.

In a first phase, the bipartition (cluster) algorithm takes up the task of topologically placing all submodules of the CUD. It receives a module list and a net list as input information from the structural description. The module list contains information about all submodules of a CUD, whereas the net list describes the connections among the modules. Corresponding to a cost function, the algorithm places a topological order on the submodules and organizes them in a so-called slicing tree (Fig. 5). The leaves contain the submodules and the root represents the CUD. The inner nodes represent subclusters for the associated modules and contain topological neighborhood information. In a latter step however, the nodes will be assigned information as to whether the two modules (or subclusters) are connected by a vertical or horizontal cut (h/v orientation); for example, there is a horizontal cut between the cell group A,B,C and the other cells.

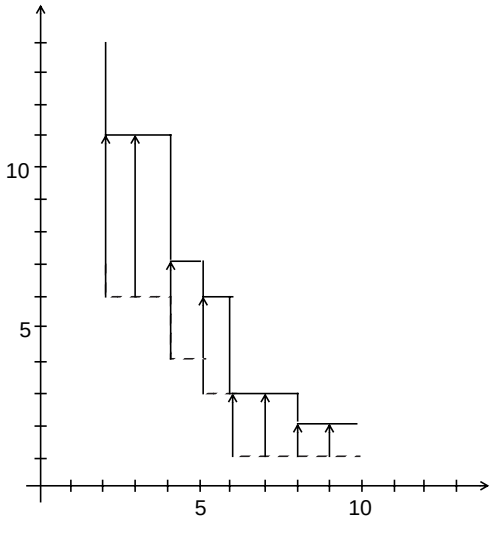
Before describing the bottom-up sizing as the next step, we have to introduce a sizing function. Therefore, we have a look at the underlying sizing model of the chip planner which determines the possible module sizes. In the sizing model, only rectangular modules are allowed. The basic modules are standard cells, called macros. For technological reasons the ratio of the module side lengths is not arbitrarily deformable. The module size is therefore given by a list of point pairs (called sizing function) each specifying the possible breadth and height of a module. The addition of the sizing functions of two modules in



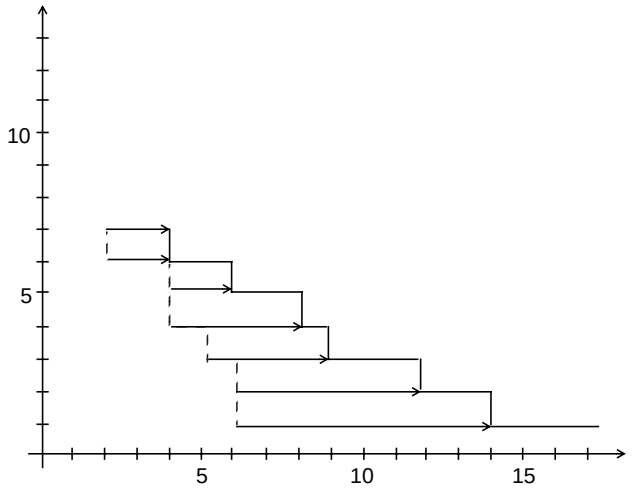
sizing function of module M1



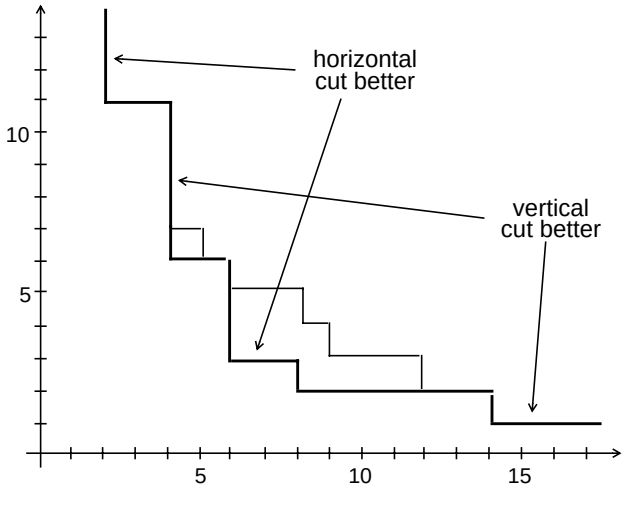
sizing function of module M2



addition of M1 and M2
for a horizontal cut



addition of M1 and M2
for a vertical cut



sizing function of the cluster

Fig. 6: Evaluating the sizing function for the cluster of two modules

design phase	task	input information
bipartitioning	slicing tree evaluation	module list and net list from the structural description
bottom-up sizing	sizing function evaluation	slicing-tree and sizing functions of the primitive elements
top-down dimensioning	h/v orientation in the slicing tree, dimension estimation for all submodules	slicing tree, sizing functions, cell sizes
global and local routing	planning the wiring areas	slicing tree, sizing functions, net list, pin information

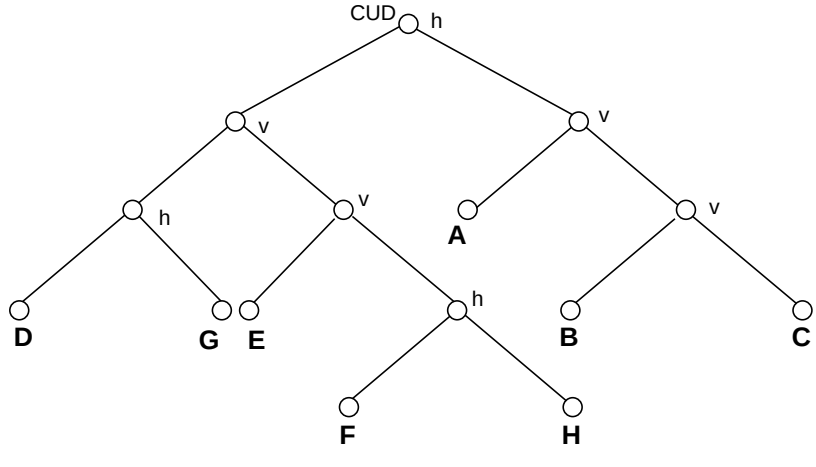
Tab1.: The main phases of the chip planning process

y- and x-direction (horizontal or vertical cut) results in a sizing function for the cluster representing both modules ..(Fig. 6)

Accordingly, bottom-up sizing evaluates the sizing function for each CUD from its submodules by determining the sizing function of each inner node in the slicing tree, and then evaluating the sizing function of the root node which corresponds to the CUD module. Note, that there is still no h/v orientation in the slicing tree.

In the next phase, the top-down dimensioning, the side lengths of all submodules are estimated from the given CUD size, the slicing tree and the evaluated sizing functions. Generally, if one side length of a module is given, the other side length can be evaluated from the sizing function (Fig. 6). Each point of the sizing function has a h/v orientation, depending on whether the subcells have been added in y- or x-direction for this point. If the cut is vertical, the sons in the slicing tree have the same y-dimension as the father, otherwise they have the same x-dimension. From this value and the sizing function of the sons, the other dimension is evaluated. Finally, the side lengths of the submodules (the leaves in the slicing tree) are determined.

However, we have not yet taken into account the wiring areas. In order to do this, a certain increase is evaluated and generally added to the module dimension /17/. Consequently, top-down dimensioning de-



h = horizontal cut ; v = vertical cut

Fig. 5: Example for a slicing tree of the chip in Fig. 4

livers a raw channel distribution. In a latter phase (global and local routing) the wiring areas are exactly determined.

After this brief introduction in the VLSI-design process, with special emphasis on chip planning, we present our NDBS based prototype system PRIMACHIP.

3.2 PRIMACHIP - a Chip Planning System

PRIMACHIP /18/ is a non-standard application which realizes some of the chip planning phases listed above. The PRIMACHIP system is structured into three layers: the NDBS kernel PRIMA, the application layer, and the actual application (compare Fig. 1). Fig. 7 sketches the entire architecture of PRIMACHIP. At the data model interface, a set of neutral yet powerful operations is provided (MAD model, see /1/). After a MAD retrieval statement is evaluated, the result is stored in the object buffer, which is a repository for the complex, heterogenous set of atoms. Primitive operations allow access to the atoms and molecule structure information, which reflects the internal structure of every molecule.

Fig. 8 shows the MAD schema for PRIMACHIP. A detailed discussion of MAD schema definition can be found in /1/. The central atom type is MODULE. Each MODULE is either a standard module (not further divided) or consists of certain submodules. For modelling the slicing tree, the atom type GRAPHNODE is required. The sizing function is modeled by the atom types CURVE and POINT. The relationships among modules are recorded in NET. In order to store the wiring information, two other atom types are of interest: CHANNEL to model the wiring area attached to each module and PIN to store the I/O-connections to other modules.

The application layer achieves a kind of tailoring mechanism for the VLSI chip planner (PRIMACHIP). The programs in the AL transform the objects and operations of the application model into those known at the data model interface. The application model interface provides an object-oriented application model: Some objects are e.g. module list, net list, slicing tree, etc., for which the corresponding operations are offered by the AL. Therefore we have introduced the classical ADT (abstract data type) concept to organize the AL. The following four ADT operations may serve as an illustrative example:

- READ_AND_STORE_LIST takes module lists and net lists as input information and stores them in the database using an INSERT statement.
- The BIPARTIONING operation selects the affected modules and net information from the database and executes the cluster algorithm in order to determine the topological placement of all submodules. Thereby, new graph nodes are created and arranged in the slicing tree.
- BU_SIZING determines the sizing function of each module from its submodules sizing functions and the slicing tree.
- TD_DIMENSIONING estimates recursively the size of each submodule from the given CUD size.

Finally, the application uses the objects and operations provided by the application layer according to the performing task. It delivers a graphic, menu-driven end-user interface. The top menu presents all available design phases in PRIMACHIP. In PRIMACHIP, the structural chip description, as it is normally de-

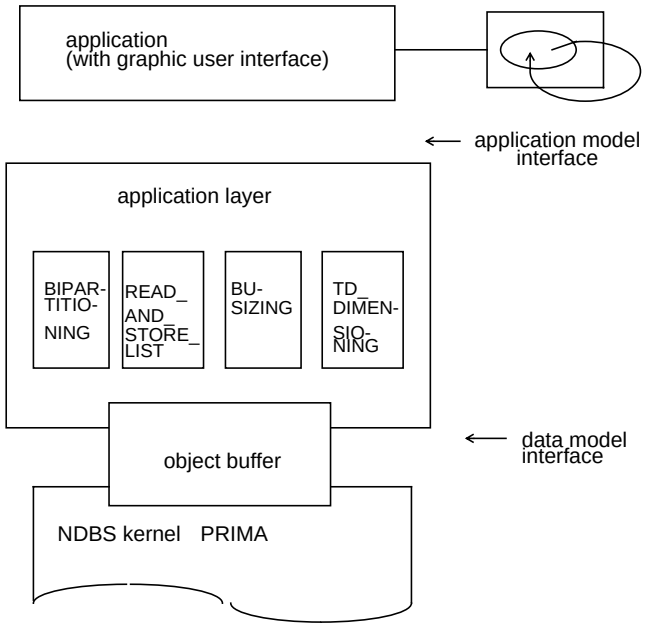


Fig. 7: The system architecture of PRIMACHIP

livered by the structural domain, is supplied by an interactive, graphic user dialogue. The result of the individual ADT operation is also represented in a graphic manner.

3.3 Processing Complex Objects in PRIMACHIP

Now we investigate the object processing in the application layer in more detail. It is the task of the application layer to transform the VLSI objects such that they are known at the kernel interface (molecules

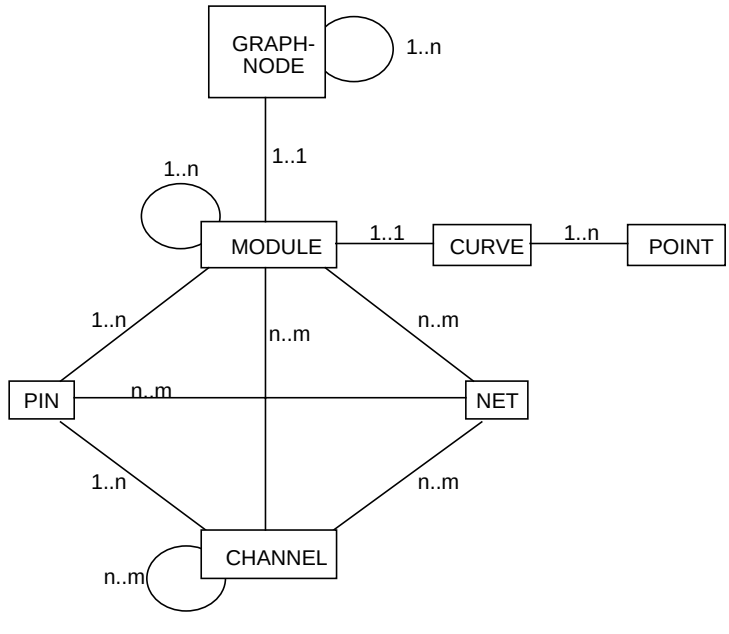


Fig. 8: The MAD schema of PRIMACHIP

and sets of molecules), and to execute the application operations by transforming them into a sequence of primitive object buffer operations and MAD statements.

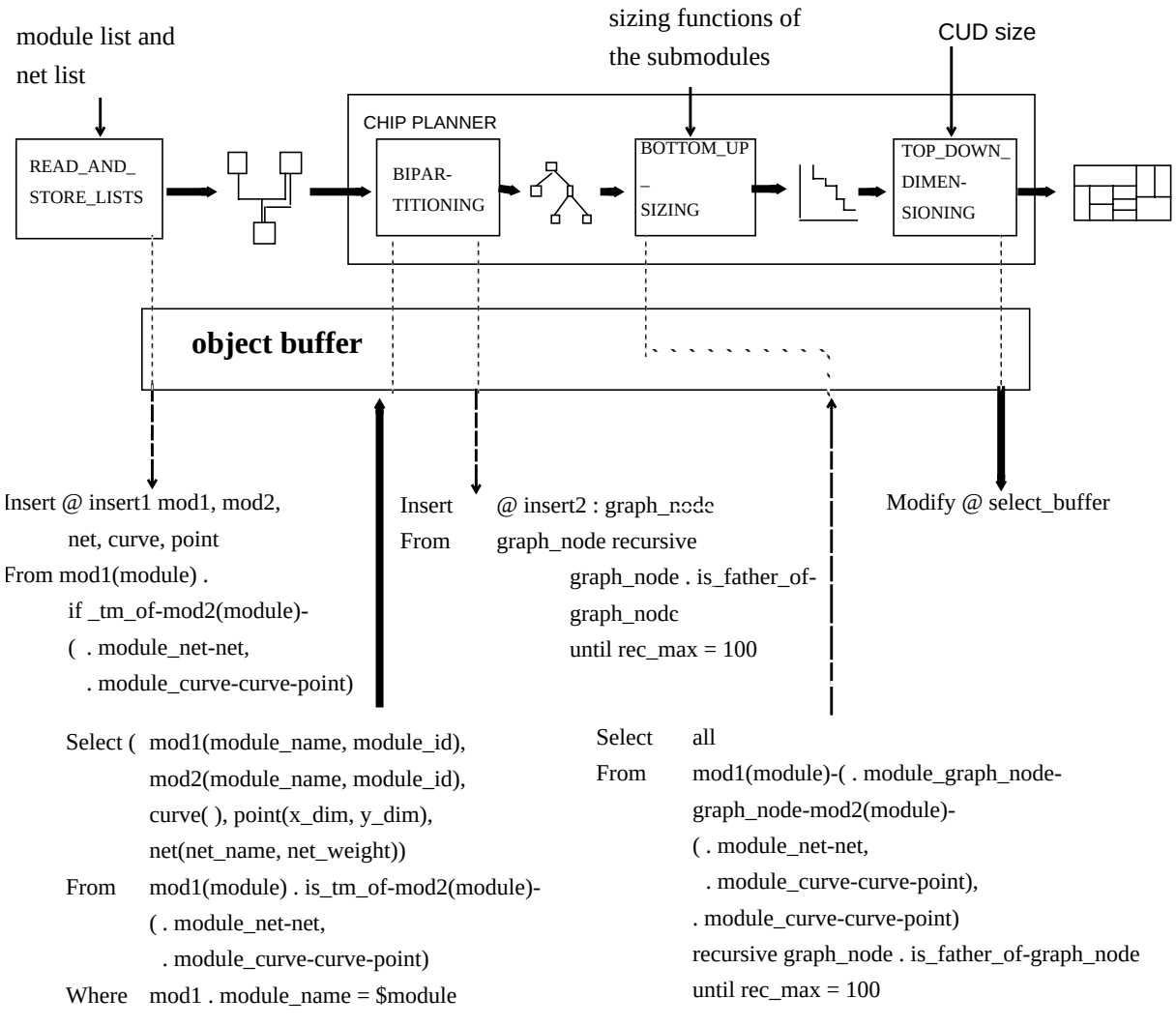


Fig. 9: The DB interaction of PRIMACHIP

An object buffer, as presented in chapter 2, involves some characteristic steps in executing an application operation. Firstly, the required data are downloaded from the database into the object buffer using the SELECT statement of MAD. Then, object buffer operations allow access to the basic components of the molecules, i.e. the atoms and their attributes. There are two possible ways to change the molecules in the object buffer and in the DB: The executing of MAD modification statements (INSERT, UPDATE, DELETE) reflects directly the modifications in the database, whereas the MODIFY statement of MAD is utilized to propagate changes to the database which have been firstly made in the object buffer using special buffer operations**.** This processing model is well known as the checkin/checkout mechanism /13/ , /19/.

Execution of a MAD statement

Before describing the usage of the object buffer, we have to clarify the result of a SELECT statement. The execution of a MAD statement is divided into a definition and an evaluation phase:

- Defining a MAD statement

Defining a MAD statement is a data model interface call activating the MAD compiler, which will check the statement for syntactic and semantic correctness and generate an access module. Furthermore, the MAD compiler delivers a type description of the latter statement result containing information about

- the molecule type structure which describes the molecule type assembly (atom types and the related reference attributes, see Fig. 9),
- the atom type structure which describes the related reference attributes, and attributes, as well as their sequence for every atom type, and
- the attribute structure (e.g. data type, attribute length).

• Evaluating a MAD statement

Evaluating a MAD statement initializes the execution of the corresponding access module generated during the definition phase. A SELECT statement delivers a set of molecules in the object buffer. Every MAD statement has its own object buffer to deposit the corresponding molecules.

Representation of molecules in the object buffe

The result of a query is a set of molecules. Every molecule has its own molecule structure which describes the relationships between the atoms in the molecule (Fig. 10). A molecule can be identified by a singular atom (root atom). Starting from this atom, every other atom in the molecule can be reached following the corresponding atom references, since the molecules are defined to be coherent. A molecule can therefore be considered as a special view on the database. If two molecules have overlapping views, they share common atoms in the buffer.

A MAD statement enables the definition of different atom roles /1/ with different atom projection lists. The qualified projection, another important concept in MAD, allows that the same atom in the database can have different atom relationships in different molecules. For these reasons, the representation of an atom in the object buffer is divided into an atom descriptor, the appropriate attribute values, and the molecule structure information which belongs to this atom. Hence, if an atom exists in different molecules with deviated atom relationships or if it has different roles, then it is represented in the object buffer by multiple atom descriptors (comprising the reference information) and uniquely stored attribute values. The atom descriptor is organized as a list of pointers to the molecule structure information and the associated attribute values. The structure of the atom descriptor is defined by the above mentioned atom type structure information. For each relationship, recorded in the molecule type structure, the molecule structure information is organized as a reference list to the atom descriptors of all related atoms. This implies, that

MAD query

```
SELECT ALL
FROM  mod1 (module) - (.module - graphnode - graphnode - mod2 (module) -(.module_net - net,
        .module_curve - curve - point), .module_curve - curve - point)
        RECURSIVE graphnode.is_parent_of - graphnode
                        UNTIL REC_MAX = 100
```

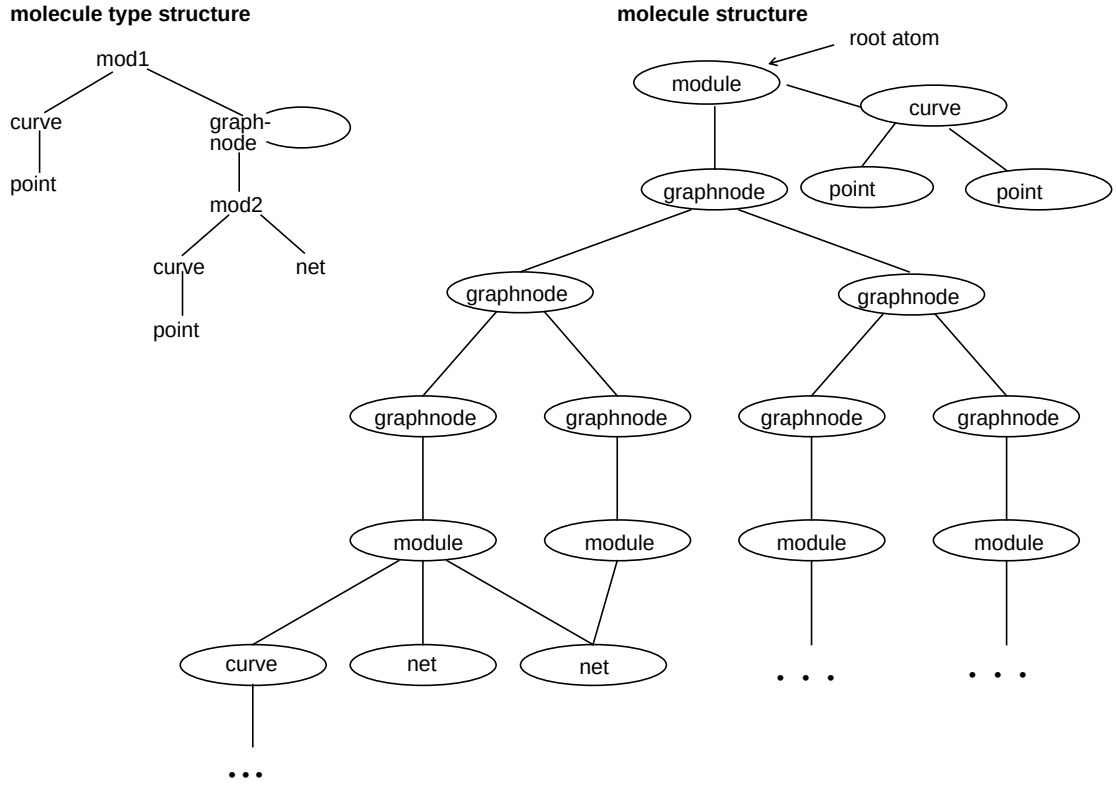


Fig. 10: Example for the molecule type structure and molecule structure of a SELECT statement

the system-defined atom identifier in the database is not sufficient to identify an atom descriptor uniquely in the object buffer.Therefore, a structure identifier is introduced, which is**r**

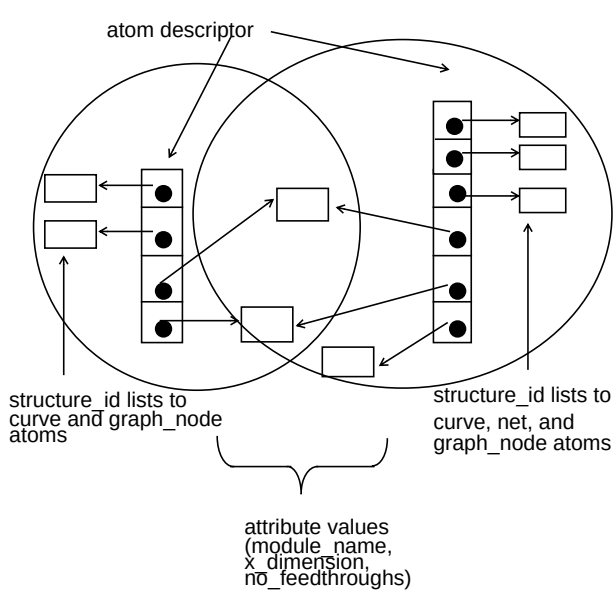
unique for every atom descriptor, and from which the molecule structure information is derived. Fig. 11 shows an example for the object buffer representation of an atom in two different roles.

Access to the molecule data

The operations on the objects in the buffer are summarized in Tab. 2 /20/. In order to select the appropriate molecule, the operations first_root_atom and next_root_atom carry out a scan across the set of molecules and deliver the structure identifier of the root atom. Once a molecule is selected, get_atom delivers a pointer to the atom descriptor of the root atom, identified by a given structure identifier.

For accessing the atoms in a certain relationship, the structure identifier list of this relationship must be determined, and the identifiers must be extracted from that list. The operation get_atom then delivers a pointer to the corresponding atom descriptor. The access to an attribute value is similar. Once the corresponding index in the atom descriptor is determined from the atom type structure information, the at-

tribute can be reached via the corresponding pointer in the atom descriptor. Then, the attribute can be accessed in the same way as a program variable for further processing.



data structures of the atom descriptors:

```
MODULE_ROLE1    =
        record
          ref_to_curve              :  pointer;
          ref_to_graph_node_ptr     :  pointer;
          module_name_ptr           :  pointer;
          x_dimension_ptr           :  pointer;
          end;
MODULE_ROLE2    =
        record
          ref_to_graph_node_ptr     :  pointer;
          ref_to_net_ptr            :  pointer;
          ref_to_curve_ptr          :  pointer;
          module_name_ptr           :  pointer;
          x_dimension_ptr           :  pointer;
          no_feedthrough_ptr        :  pointer;
          end;
```

Fig. 11: Example for the object buffer representation of aone atom in two different roles

The above stated atom access shows the usage of the type description delivered by a query definition. Every time an attribute or a structure identifier list are accessed, their position in the atom descriptor must be determined from the atom type structure. As a consequence, we choose another approach. Once a MAD statement is defined, every query compilation results in the same type description. Therefore, we hardwired the atom descriptor index for each attribute or structure identifier list in our programs (Fig. 11).

for RETRIEVAL	for INSERT
first_root_atom next_root_atom get_atom	give_new_structure_id propagate_atom sign_root_atom

Tab. 2 : List of object buffer operations

Modification of molecules

So far, we only discussed operations to retrieve the objects and deposit them in the object buffer. The propagation of molecule modifications in the database can be achieved in two ways:

- The MAD modification operations such as DELETE, UPDATE, and of course for inserting the INSERT statement reflect the changes directly in the database. One variant of the INSERT command works similarly to the SELECT statement. After the molecule type is specified by an INSERT command, the application programmer has to build up the molecules using the operations listed in Tab. 2. The operation give_new_struct_id provides a unique, temporary structure identifier, which allows a new atom to be composed in the application program. The propagate_atom operation places the atom in the object buffer and, as an option, the sign_root_atom operation labels an atom as root atom. However, the programmer has to build up a molecule on his own. For example, to store a module and all related

nets, the programmer has to deposit all nets in the object buffer and then, before deposing the module atom, the reference information of all related net atoms must be established in the module atom.

- The second possibility is to make changes in the object buffer, which contains the result of a previous SELECT statement. This comprises of the general checkin/checkout mechanism as described above. The insertion of new atoms in the buffer occurs in the same manner as described by the INSERT command. At checkin time, all changes (e.g. modification of an attribute value, insertion or deletion of various atoms, modification of relationships) are propagated simultaneously using the MODIFY statement.

However, the arbitrary usage of modification operations on objects in the buffer leads to serious semantic problems. Each molecule is a materialized view of the database as mentioned above and the modification operations are performed on these views. A detailed examination of these problems can be found in /21/. However, the current prototype system does not support this kind of object processing at all.

3.4 Analyses and Evaluation of the PRIMACHIP Processing Concept

After having explained our initial approach in handling molecules, we would like to outline an evaluation. Therefore, we characterize the main processing properties which we have introduced above. First of all, we can state that there is a clear division between data preparation (evaluation of MAD statements) and data processing. The algorithms of the AL retrieve mainly all the molecules which represent the whole *processing subject*. In doing this by MAD statements, preparation activities are triggered, and all the corresponding data are loaded into the object buffer. The processing phase is then reached and the algorithms manipulate the molecules placed in main memory. This can be implemented very efficiently, since locality can be exploited, implying that the data access is normally provided by a few pointer references. After finishing the manipulations, the modified objects are passed back into the database. In comparison with a similar chip planning tool (called DBCHIP), which is based on a conventional database system, PRIMACHIP shows a significant reduction in database access frequency. The main reason for this is the above illustrated preparing and processing model. For example, the algorithm to build up the slicing tree requires two DBMS statements only, whereas the same task in the DBCHIP system yields some thousand database operations (for about 12 submodules of a CUD) /22/. Certainly, these operations are less complex than the corresponding MAD statements, but despite this, each simple operation produces a fixed processing overhead.

The structure of data, constituting the input as well as output argument of a MAD statement, is determined by the *meta-data* known only inside the kernel system. To work with this data, the AL programs must be supplied with this knowledge. In our prototype system this is done by the programmers themselves. They freeze the concrete knowledge in programs' coding, defining compatible data structures, and including the operations for some MAD-specific data types (list type, set type,...).

Having in mind these observations, we can state two main issues of criticism, which constitute the starting point for the improvements proposed in chapter 4:

- After checking out the data from the central database, there is only elementary support for molecule processing.
- The conversion from internal and persistent data to external and temporary data (from the DBMS point of view) is not supplied. That means, there is no support for adaptation of data structures and data types (language binding).

4. General Issues of Molecular Objects' Linkage to the Application

After a detailed description of the implementation model of our VLSI application system, we concentrate on the general issues for application linkage, generalizing for this purpose some important processing properties of the PRIMA kernel interface. We explain then a refined approach of interfacing the AL and the NDBS-kernel, focusing in particular on concepts for AL organization and efficient molecule processing as well as suitable methods for language embedding. Furthermore, we touch on some issues of the application model interface which have to be carried out by the AL. Finally, we consider a few basic aspects of *modelling the AL as ADTs.*

4.1 Generalization of Processing Properties

For the purpose of generalizing processing properties, it should be advantageous to contrast the simplified processing scheme (used in PRIMACHIP) with the alternatively used concepts. The typical processing scheme in PRIMACHIP can be characterized as a scheme for external processing (Fig. 12a). It is determined by three steps :

- Retrieve the whole (or nearly the whole) subject of processing (checkout).
- Manipulate this subject 'nearby' the application.
- Propagate all modifications into the database (checkin).

A processing scheme, which obeys a different philosophy is shown in Fig. 12b. Here, the work is done inside the kernel system (scheme for internal DB processing). In other words, the differences among the underlying ideas are embodied in data supply for the algorithms (Fig. 12a) and, on the other hand, in transmitting the algorithms to the data (Fig. 12b). Considering this dualism, it becomes obvious that the external processing scheme seems to be typical, not only for our prototype system, but also for all applications which are based on complex algorithms manipulating complex objects. This is mainly caused by the fact that these algorithms cannot be formulated in the data model language (e.g. MAD language). For example, it is impossible to describe the evaluation of a slicing tree by means of a data model language

only. However, the scheme for internal DB processing has more advantages in the case of simple algorithms manipulating simple objects (e.g. increment the salary of all employees by 5%).

In the environment of non-standard applications, the central point of improvement is obviously the checkout/checkin processing scheme. In particular, we have to eliminate the weaknesses in handling molecules by the AL programs, which are pointed out in section 3.4 (molecule processing, language binding).

4.2 Molecule Processing

Having in mind the above observations, we can state that molecule preparation does not cause a problem, since this can be handled adequately by the MAD query language: a few MAD statements are grouped together to checkout the subject of processing, or alternatively to checkin the modified data. However, it is obviously difficult to handle retrieved molecules, since there is no efficient addressing support to access molecule components. This problem depends on the complexity of the retrieved objects. Therefore, conventional DBMS applications do not have a similar problem because retrieved records or tuples have simple data structures. So, the main memory address of each attribute can be evaluated from the object's base address by adding a fixed offset.

Improvement of molecule processing facilities

The improvement of molecule processing facilities forces extensions of the main memory data structures, which represent molecules and atoms (in/out parameter of MAD statements), by access and control information. This information is mainly used to support new access functions (e.g. address evaluation and navigational access). In conventional DBMS, data structures used in a similar way are called *cursors.* Cursors are known in the environment of the network data model /23/ and in the programming interfaces of several implementations of the relational data model /24/, /25/. The concrete semantics of them, however, is different in several data models. For example, in the network data model, implicit cursors are used to support navigational access and to define explicitly the semantics of data model operations. In the context of the relational data model, cursors are only required for accessing query results

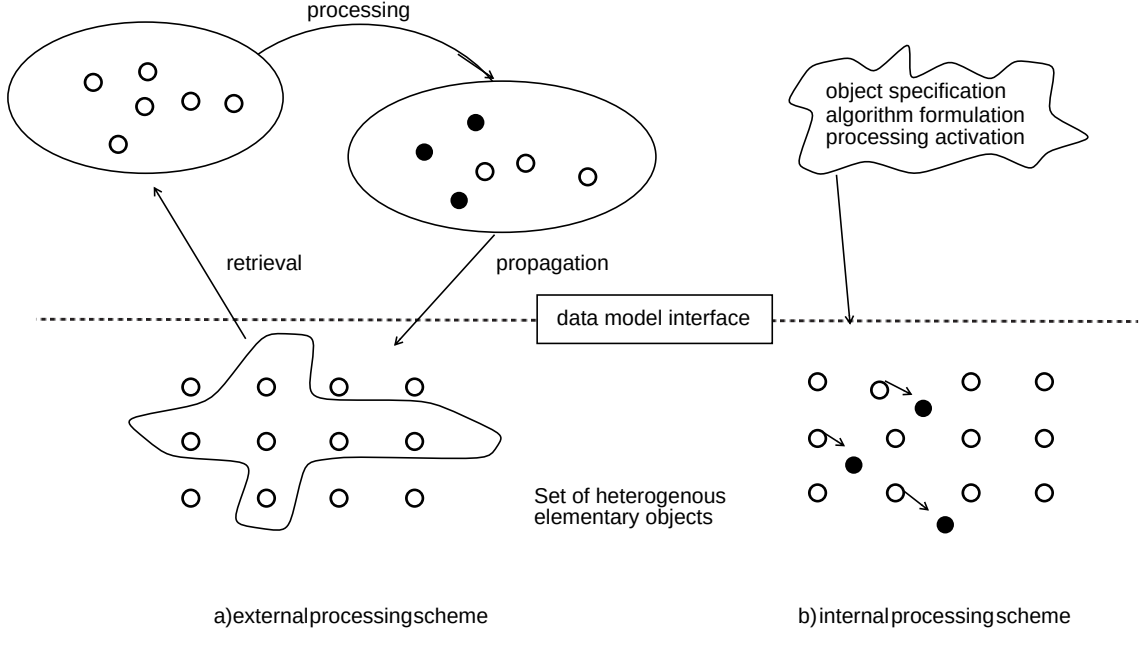


Fig. 12 : Schemes for external/internal processing

by the application. This means, that application programs can employ them to handle the set-oriented data model objects (relations) step by step, one set element after the other. Hence, the programmer will run into some problems if he requires more than one set element at a time. All these cursor semantics and the associated problems are worth more detailed considerations, which however, lie beyond the scope of this discussion.

Molecule processing requires multiple cursor types in different semantics. Handling molecules in a set-oriented manner, cursors are used to indicate a single molecule within the set. Such a cursor usage is similar to the cursor concept in the relational environment. Having determined one molecule, we must employ the cursor in another way, since navigation through the corresponding atom network is required now. Therefore, we have developed the cursor concept with a special semantics, which fulfills the requirements of set *and* network processing. Additionally, our approach allows explicit cursor definition, that is, the programmer can dynamically generate cursors as often as required. In order to define a cursor, he must specify at least one atom type, determining the type of atom instances which he desires to access via the cursor. Specifying exactly one atom type, the programmer will get a *flat* cursor, which may only be used to scan a set of homogeneous atoms. When the programmer determines the root atom type of the corresponding molecule type, the cursor provides some kind of molecule scan. Thus, this special case serves the same function as do the operations first_root_atom and next_root_atom (compare chapter 3). In addition, the definition of *hierarchical* cursors is allowed, in order to support the navigational processing of hierarchical molecule components. The cursor hierarchy is determined by a list of atom type names, marking the cursor paths. The concept of hierarchical cursors may be implemented by a hierarchy of dependent flat cursors. Navigation via one cursor automatically affects the cursors on subordinate levels. The binding between cursor and data to be referenced, is provided by the programmer. He defines the scope of data which may be referenced, associating the cursor with a set of molecules delivered by MAD statement evaluation. Then, the cursor may be moved in the atom network by explicit cursor operations enabling access to each atom. For detailed explanation of the implementation issues, the reader is referred to /21/.

Administration of the object buffer

The essential condition for efficient molecule processing consists in buffering molecules nearby to the algorithms operating on them. Therefore, the considered prototype system was already based on an object buffer at the kernel interface. The issue we would like to illuminate now, is how this central data structure should be organized in order to satisfy the discussed requirements in an efficient way. Note, the object buffer is associated with the workstation site (cf. chapter 2). The data loaded into the object buffer are usually kept there for a long span of time. It will therefore become necessary to move the objects from the main memory to the private DB existing on secondary storage (local for this workstation). Thus, we organize the object buffer using so-called *main memory areas.* The required *relocation* can be easily managed, if all logical references among molecules or atoms, included in the object buffer, are substitut-

ed by some kind of area relative addresses. The relocation can then be achieved without need of recalculation by simply moving whole areas.

Fig. 13 illustrates the representation of a molecule set within the object buffer. Based on the *molecule set descriptor,* the overall structure can be divided into three parts:

- The *cursor table* contains all cursors, which are linked with molecules within the considered molecule set, and is mainly used for cursor administration.
- The *area table* and the set of *areas* obtain structures to support the control of memory which is allocated for atoms. The area concept supplies not only relocatability, but also prevents the scattering of main memory by numerous small atoms.

Furthermore, the *atom type table* and *atom table* allow us to calculate atom addresses by using logical atom identifiers. The atom type table separates the atoms by their corresponding atom type and enables efficient type-oriented access. Each entry bears an internal type key and the address of the atom table. This table is organized as a dynamic hash table, since the number of stored atoms can shrink or expand significantly. The atom table entries contain further administrative information. The modification flag indicates the type of modification (insert, delete, update) and is later used for propagating back to the central database. Additionally, there are also the fields *area index* and *area offset.* The area index determines an entry of the area table, which finally leads to the area containing the required atom. The area base address incremented by the area offset delivers the atom's main memory address.To maintain and use all these data structures, we need efficient algorithms for address calculation and memory management. Moreover, a further component becomes necessary to deposit/reload molecule objects to/from the private DB. Fig. 14 shows all the components which are necessary to support adequate molecule pro-

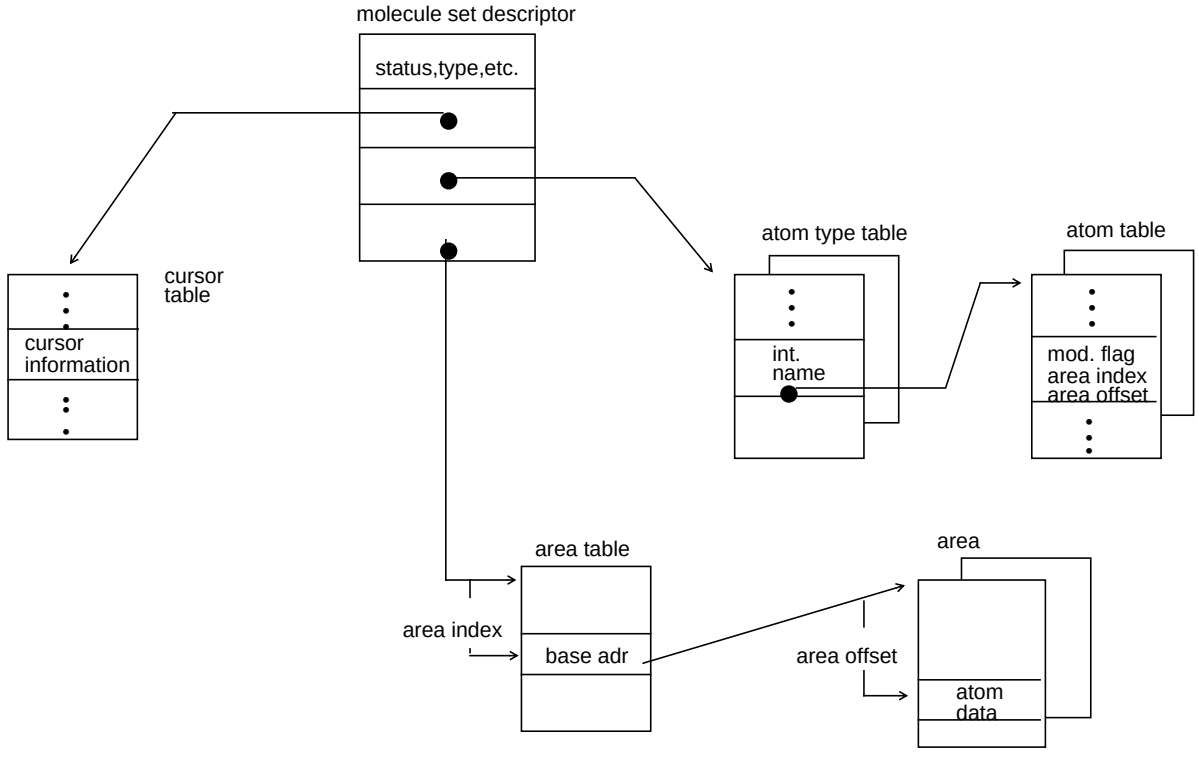


Fig. 13 : Representation of a molecule set within the object buffer.

cessing. These modules cooperate with a preparation component, which handles the checkin/checkout operations and offers the functions of the MAD language.

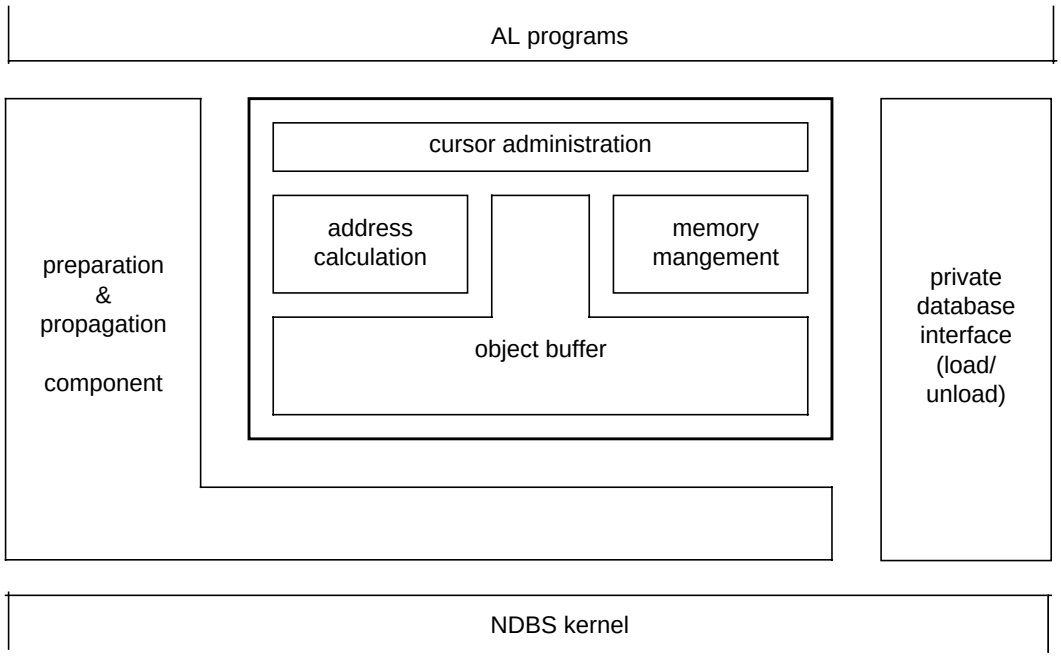


Fig. 14 : Interface architecture supporting adequate molecule processing

4.3 Language Embedding

Another problem derived from our prototype observations concerns data structure adaptation. During the implementation of the VLSI application system, the AL programmers have to define a great number of internal data structures, which must be compatible with the one delivered or expected by the NDBS-kernel. Additionally, there is no adaptation of MAD-specific data types (list type, set type, hull, time,...). The programmers have to know internal representation issues and must carry out the associated operations manually. These unpleasant properties are typically related to the concept of a call interface, the simplest concept for language binding. In the context of conventional DBMS, four different approaches for binding database and programming languages are well known:

- call interface (e.g. sketched above),
- simple host language extension (e.g. CODASYL COBOL-DML),
- embedding database languages in general purpose languages (precompiler, e.g. embedded SQL),
- integrated languages (new data types, e.g. PASCAL/R).

/26/ contains a detailed consideration of these binding concepts. Here, we would only like to state the results: the most advantages lie in the fourth approach, because the internal (with respect to the program) and temporary data are compatible with the external and persistent data (since they have the same logical structure). However, it is not the best procedure from an implementation point of view, i.e. a new language must be designed, a compiler must be written, etc.

A precompiler approach

Therefore, our own proposal for an application programming interface (API) is based on a mixture between the third and the fourth approach. We have designed a precompiler to handle the inclusion of MAD

statements, to enrich the underlying host language in order to handle MAD-specific data types, and additionally to integrate special language constructs for supporting the molecule processing concepts mentioned in the previous section (cursor, object buffer).

The use of precompiler statements is sketched in Fig. 15. It depicts the scheme of an AL program (ADT) in a PASCAL-like programming language. We distinguish between the definition of *query types* and *cursor types*, as well as the declaration of corresponding instances. The AL programmers can therefore generate multiple instances of the same type. These are handled like host language variables. The query type declaration is the place for MAD statement specification. Therefore, all functions of molecule preparation are included in the program by query types. The *query variable* corresponds to the object buffer introduced above. Note that the instance of a query represents the data which is the parameter of MAD statement's activation. The linkage between variable and data is done via the built-in function *eval.* Eval directs actual parameters to the kernel interface and triggers the evaluation and preparation activities. Similar to the definition of query instances, the programmers can generate *cursor variables* supporting the processing of object buffers (query variables). The binding of cursor variable and object buffer is provided by the *attach* function. The bounded cursor can be moved in molecules around atoms, which are currently in the object buffer, and is also used to specify the required molecule-oriented access (cf. Fig. 15).

Steps of compilation

The transformation of AL programs into executable object code is divided into two phases. In the first phase, the precompiler replaces the definitions of queries and cursors by the declaration of corresponding host language data structures. In particular, data structures are generated for all atom types. The required meta-data can be extracted from the internal schema information (known in the NDBS-kernel) by pretranslating all the MAD statements which occur in the program. Query evaluation and cursor binding statements are converted into host language operations (subroutine calls). These subroutine references are later satisfied by a *run time system (RTS)* of the NDBS-kernel interface, which supports molecule processing and preparation (compare Fig. 14, e.g. cursor maintenance component). Moreover, the MAD-specific data types are taken into account. Therefore, the precompiler includes an extension of the normal host language run time system. In the second phase of transformation, the standard host language compiler runs finally generating the object code.

Detailed considerations of syntactical issues and further information about the precompiler are given in /21/, where the reader can additionally find the implementation model for the RTS of the NDBS-kernel interface.

```
PROGRAM example;
   definition of 'normal' types and variables
QUERY TYPE      mcp_query_type =
                'Select  module, curve, point
                From     module-curve-point
                Where    module.x_dim < $max'
                (max : integer);
   •••
CURSOR TYPE     mcp_cursor_type =
                         module, curve, point > mcp_query_ type
   •••
mcp_cursor  :   mcp_cursor_type;
mcp_buffer  :   mcp_query_type;
   •••
BEGIN
      •••
   EVAL ( mcp_buffer, ... max_value ...);
      •••
   ATTACH (mcp_cursor, mcp_buffer);
      •••
   DISPLAY (mcp_cursor @ module @name);
      •••
END.
```

Fig. 15: An AL program scheme

4.4 ADT Modelling

At the beginning of this chapter, we introduced the AL as the linkage between NDBS-kernel and application. So far, the interface between AL and kernel system as well as the AL organization were explained above. Now, we sketch our ideas about interfacing the application with AL.

All the concepts just explained should support the AL to offer application objects at the level of the application model. The application objects can be described by their specific data structure and their associated operations. One method which allows an integration of data structure and operation specification is provided by the well-known ADT concept. The sum of the existent ADTs forms the application model. That means, it embodies the user's mental image of the behavior of relevant application entities (behavioral object orientation /26/, /28/.

Mapping of ADTs

The application often distinguishes entity types from entity instances. Thus, we correspondingly map entity types to ADTs, and in turn, entity instances to instances of ADTs. Not only application-specific operations are required (defined by ADT types), but also a few other ones which are more general. Simple examples are given by the *create, destroy,* and *assign* operations. Furthermore, *storing* and *loading* of ADT instances must be supplied, and descriptive methods for selection are desirable. We propose a direct mapping of the ADT instances to the molecules delivered by the MAD model. This has the advantage

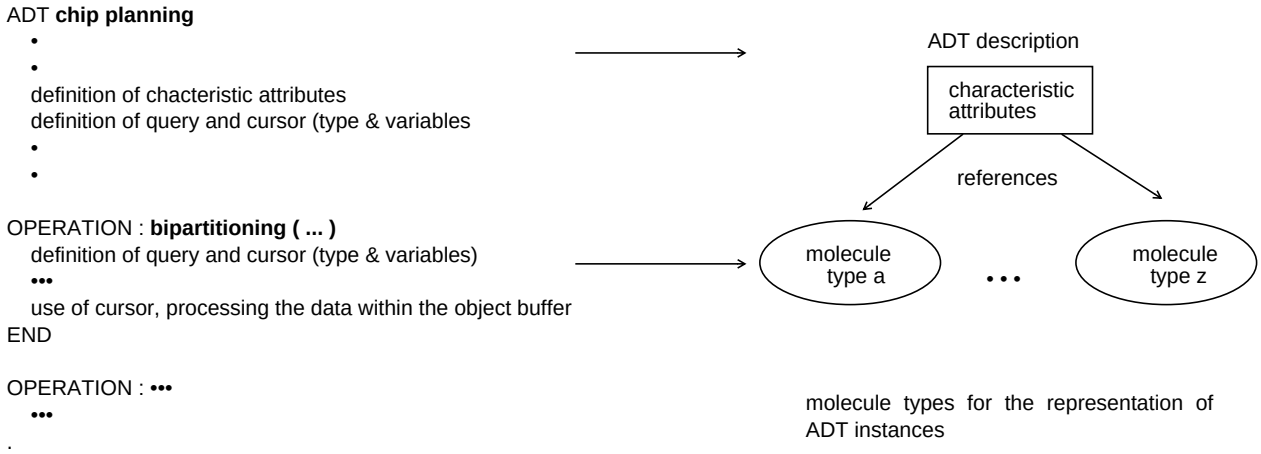


Fig. 16: ADT molecule mapping

that we can use the full MAD functionalism to maintain ADT instances. For example, we can select ADT instances by an SELECT.. FROM.. WHERE..-Statement giving an instance identifier as the qulification criteria.

Fig. 16 illustrates the mapping principles. An ADT is defined as a program module which offers some operations and characteristic attributes. Internally, there exist further attributes for state description, as well as some query and cursor definitions. The type of molecules (i.e. molecule instances) which represent the ADT instances is determined by a special atom type, the ADT descriptor. This atom type contains all the characteristic attributes as well as the references to all molecule types, which are defined by MAD statements inside the ADT definition.

Further considerations are required, if we want to represent the ADT themselves (i.e. ADT types, **not** ADT instances) by the molecules. The relevant issues concern mainly the handling of ADT operations. Of course, these operations can be represented as attributes of type 'code', but the associated values must be loaded, linked together, and executed dynamically which is not supported by the MAD model.

Taking into account the characteristic properties of processing, we have to deal with some other aspects. The ADTs themselves may be used by further application programs at the application model interface. In the most simple case, the application programs carry out the user's system image. For this reason, the application model offers additionally some application-independent operations to organize users' activities. For instance, an engineering user can determine the beginning, the end, the suspension, and the resumption of a design process. Furthermore, he can activate an ADT, making the corresponding operations available, and he can save or restore the state of ADT instances. All these activities must be carried out by a further refined processing model for ADTs and, in particular, by suitable transaction concepts at the kernel interface. These related issues are outlined in /13/ and must be refined by future work.

5. Conclusions

In this paper, we have presented our ideas for supporting engineering applications by the NDBS kernel PRIMA and have stated experiences which we gained with the prototype system PRIMACHIP. Firstly, we have pointed out the modelling and processing problems in handling complex database objects in an engineering environment. Due to the special requirements of engineering applications, we proposed a

confederate hardware architecture, which based on a loose workstation/host coupling. This hardware architecture supports the division of the non- standard database system in an application independent kernel and an application layer in a natural way. According to the suggested processing model, a central host processor works as a database server and an autonomous workstation download the required data from the database server in a local object buffer (checkout), where the complex objects are manipulated independently. During a commit phase, all changes are propagated to the central database server (checkin).

The PRIMACHIP system, which comprises some of the chip planning phases of the VLSI design process, was implemented to obtain experience in the usage of the NDBS kernel architecture. The processing model of PRIMACHIP involves an object buffer and realizes a checkin/checkout mechanism. From this prototype system, we observed some demands for efficient object processing:

- a generalized hierarchical cursor concept for navigating in complex object structures, and
- an adequate concept for object buffering.

Based on these experiences, we outlined our generalized application programming interface, a mixture of language embedding and integration, which embodies language constructs for effective object processing in a conventional programming language. A precompiler is employed to transform the application programs into equivalent host language code. Our API allows for:

- access to program variables and buffered data in a homogeneous manner
- a 'natural' handling of queries and cursors in the application programs.

Future work will be concerned with the development of a application layer for CAD/ CAM applications in order to valid the utilization of the database kernel architecture in engineering applications.

Acknowledgements

We would like to thank T. Härder for his helpful comments and our colleagues N. Mattos, B. Mitschang and A. Sikeler for their careful reading of an earlier version of this paper. They have contributed to clarify and improve the preparation of important issues. The help of I. Littler polishing our english style is greatfuly acknowledged.

References

/1/ Mitschang, B.: The Molecule-Atom Data Model in: The PRIMA Project, Design and Implementation of a Non-Standard Database System, T. Härder (ed.), Report Nr. 26/88, SFB 124, University Kaiserslautern, März 1988.

/2/ Lorie, R., Kim, W., et. al.: Supporting Complex Objects in a Relational System for Engineering Databases, IBM Research Laboratory, San Jose, CA, 1984.

/3/ Schek, H.-J., Scholl, M. H.: The Relational Model with Relation-Valued Attributes, in: Information Systems, Vol. 2, No. 2, 1986, pp. 137-147.

/4/ Pistor, P., Anderson, F.: Designing a Generalized NF2 Data Model with a SQL-Type Language Interface, Proc. 12th VLDB Conf., Kyoto, 1986.

/5/ HMMS87 Härder, T., ...

/6/ Härder, T.: Overview of the PRIMA Project, in: The PRIMA Project, Design and Implementation of a Non-Standard Database System, T. Härder (ed.), Research Report No. 26/88, SFB 124, University Kaiserslautern, 1988

/7/ Liskov, B., Zilles, S.: Programming with Abstract Data Types, in: Proc. ACM SIGPLAN, Conf. on Very High Programming Language, SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 50-59.

/8/ Stonebraker, M.: Quel as a Data Type, Proc. 1984 ACM SIGMOD Conference on Management of Data, Boston, 1984.

/9/ Dadam, P., et. al.: Managing Complex Objects in R^2D^2; in: HECTOR - Heterogeneous Computer Together (Krueger, G., Mueller, G. ed.), Berlin, Heidelberg, 1988.

/10/ Deppisch, U., Günauer, J., Küspert, K., Obermeit, V., Walch, G.: Überlegungen zur Datenbank-Kooperation zwischen Server und Workstations, in: Proc. of the 16th GI-annual Conf., Berlin, 1986, IFB 126/127, Springer Verlag, 1986.

/11/ Deppich, U., Obermeit, V.: Tight Database Cooperation in a Server-Workstation Environment, in: Proc. 7 th Int. Conf. on Distributed Computing Systems, Berlin, 1987.

/12/ Härder, T., Rahm, E.: Mehrrechner-Datenbankssysteme für Transaktionssysteme hoher Leistungsfähigkeit, in: Informationstechnik it, No. 4, 1986, pp. 214-225.

/13/ Härder, T., Hübel, Ch., Meyer-Wegener, K., Mitschang, B.: Coupling Engineering Workstations to a Database Server, in: Proc. Conf. on Data and Knowledge Systems for Manufacturing and Engineering, Hartford, Connecticut, 1987, pp. 36-44.

/14/ Küspert, K., Dadam, P., Günauer, J.: Cooperative Object Buffer Management in the Advanced Information Management Prototype, in: Proc. VLDB '87, Brighton U.K., Sept. '87, pp 483-492.

/15/ Zimmermann, G.: A new Area and Shape Function Estimation Technique for VLSI Layouts; in: Proc. of the 25th Design Automation Conference, pp. 60-65, 1988.

/16/ Schürmann, B.: Hierarchisches Top Down Chip Planning; in: Informatik Spektrum, Bd. 11, Heft 2, S.57-70, Heidelberg, 1988.

/17/ Klein, A., Schreiner, F., Zimmermann, G.: Ein Sizing-Modell für den VLSI-Entwurf, SFB 124, Report No 25/87, University Kaiserslautern, 1987.

/18/ Krück, D.: Realisierung eines NDBS-basierten Werkzeuges für den VLSI-Chip-Entwurf, University Kaiserslautern, 1988 (in preparation).

/19/ Lorie, R., Plouffe, W.: Complex Objects and Their use in Design Transactions, in: Proc. of the Data Base Week: Engineering Design Applications, pp. 115-121, 1983.

/20/ Müller, Th., Schöning, H.: Entwurf und Implementierung des Datensystems von PRIMA, University Kaiserslautern, 1987.

/21/ Hübel, Ch., Sutter, B.: Verarbeitung Komplexer DB-Objekte in Ingenieuranwendungen, University Kaisersslautern, 1988 (in preparation).

/22/ Härder, T., Hübel, Ch., Pahle, H., Zimmermann, G.: Ansätze Zur DB-Unterstützung für den VLSI-Entwurf, Research Report, No. 31/88, SFB 124, University Kaiserslautern, 1988.

/23/ CODASYL Data Description Language Comittee Report, Information Systems, Vol. 3, No. 4, 1978, pp. 247-320.

/24/ Astrahan, M.M., et. al.: System R: A Relational Database Management System, IEEE, Vol. 12, No. 5, 1979, pp. 42-48.

/25/ Date, C.J.: A Guide to Ingres, Addison-Wesley Publishing Compony, 1987.

/26/ Dittrich, K.R., Dayal, U. (eds.): Proc. Int. Workshop on Object-Oriented Database System, Pacific Grove, 1986.

/27/ Lacroix, M., Pirotte, A.: Comparison of Database Interfaces for Application Programming, in: Inf. Systems, Vol. 8, No. 3, 1983, pp. 217-229.

/28/ Hübel, Ch., Mitschang, B.: Object Orientation Within the PRIMA-NDBS, appears in: Proc. 2nd Workshop on Object-Oriented Database Systems, Sept. 1988, Bad Münster am Stein.