An Object-Oriented Datamodel for the VLSI Design System PLAYOUT

Ernst Siepmann, Gerhard Zimmermann

University of Kaiserslautern

D-6750 Kaiserslautern, West - Germany

Abstract

A complete datamodel for an integrated VLSI design system is developed in a stepwise manner. This datamodel introduces a unified view of all design domains and hierarchy levels that result in a considerable simplification of the communication among the design tools. We defined five basic objects: *cell, interface, contents, instance* and *configuration* for high level description of all design data. The model also covers views, types, alternatives, versions and the design history.In particular our handling of the configuration demonstrates reasonable response times, due to the strict avoidance of redundant information and the storage of complex objects. This datamodel can be directly used for the implementation of an efficient object-oriented design database.

1. The PLAYOUT Architecture

The design process of the PLAYOUT system /Zi 86, Zi 88/ takes place on a design plane (Figure 1.1), with the **hierarchy levels** and the **domains** (in our system behavior, structure, foorplan, masklayout) as dimensions. The design process traverses the design plane in a top-down (for chip planning), a bottom-up (for chip assembly), and a mixed (for logic design) manner. The design process is nondeterministic and in some phases highly parallel.

Different design tasks are handled by **toolboxes.** A toolbox consists of a number of algorithms (tools) and a common internal data structure. Typically, this is an abstract data type (ADT). Thus the tools in a toolbox can communicate very fast. A typical toolbox is the Chip Planner. It contains tools for placement, sizing, global routing, pin placement, and analysis. For placement several alternatives exist.

Besides the Repartitioner all toolboxes execute transformations between domains and therefore cover two domains. Toolboxes also cover two hierarchy levels. The upper level is the cell-under-design (CUD) which consists of subcells at the lower level.

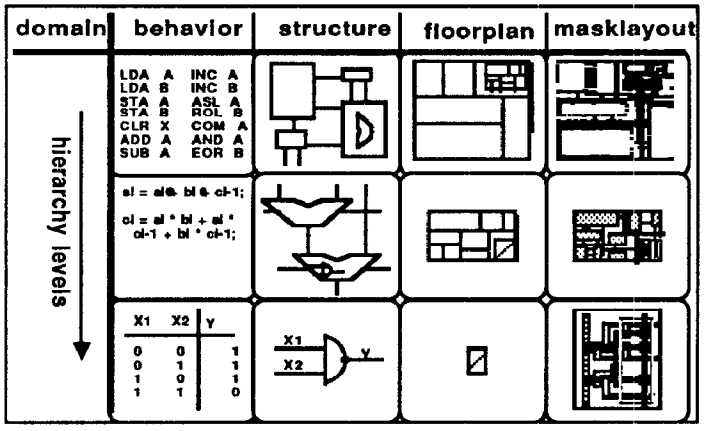

Fig. 1.1 Design Plane

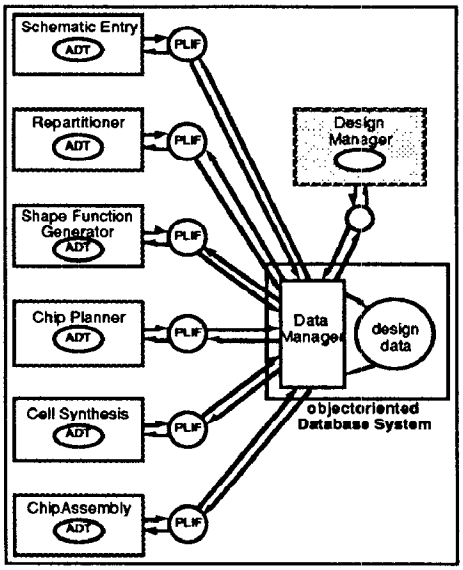

Fig. 1.2 PLAYOUT System Architecture

Most toolboxes can be used **recursively** at different levels of the hierarchy. Therefore the number of levels can be adjusted to the complexity of the problem. For example, the Chip Planner at the top level (the chip) generates frames for its subcells. These frames are the input for the Chip Planner at the next lower level where the former subcells have become the CUDs, see Figure 1.3.

The communication between toolboxes is always through the central **Design Database**, even in the case when it is applied recursively, see Figure 1.2. The reason is that the toolboxes get their input data not only from one other toolbox, but from several. So the Chip Planner receives both the net list and module list from the Schematic Entry or the Repartitioner, the shape functions from the Shape Function Generator and the CUD-frame from the Chip Planner employed the level above.

On the other hand, the same shape functions, the same net list and the same module list are used for the Shape Function Generator, so that a complex n:m relation exits among the toolboxes regarding their communication. With direct communication it would be hard to guarantee consistency, particularly if you have various alternatives, ver-

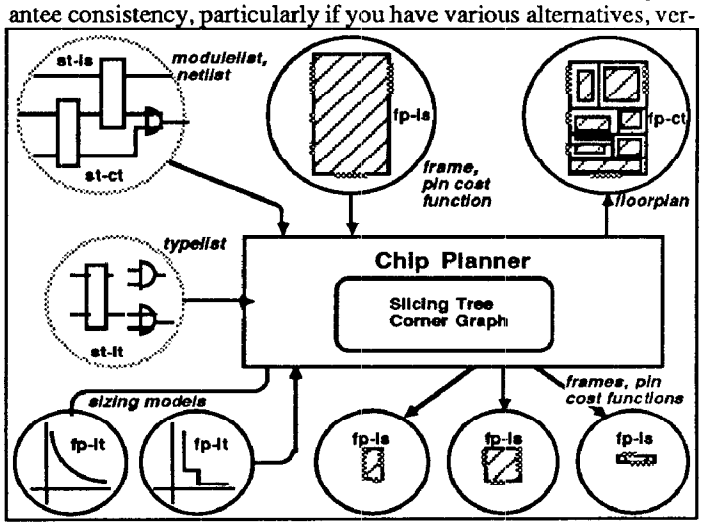

Fig. 1.3 The Communication Objects for the Chip Planner

sions and configurations of the design data.

The data are exchanged in the form of ASCII files. The format is a object-oriented language PLIF (PLAYOUT Interchange Format) /Si88/. The input and output files for each toolbox application consist of a number of different **communication objects**, shown in Figure1.3 as circles. The database breaks the files up into the communication objects, stores and manages these objects and the assembly of objects into files.

2. The PLAYOUT Datamodel

In this chapter, the data model will be derived in a stepwise manner from a hierarchy model. An object-oriented approach is chosen. The concept also avoids redundancy of data. This not only reduces the amount of data, but greatly supports data consistency.

2.1 Model of the Hierarchy

This basis of the datamodel is the hierarchy as described in Chapter 1. Each **cell** can be part of another cell (supercell) or contain other cells (subcells). This complex relation is called *part* and corresponds to the abstraction concept of aggregation /Mi88a/. A cell cannot contain itself and no cell of which it is a part itself. The *part* relation is not defined between all cells. Therefore it defines a partial order.

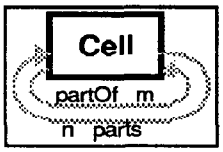

Fig. 2.1 Hierarchy

Figure 2.1 shows the *part* relation as Molecule Atom Data (MAD) model /Mi88b/. In the MAD-model boxes are the entities, arrows the relations. The relation name is split in two roles, in Figure 2.1 for example in *parts* and *part-of*. Numbers or variables at the arrows describe the cardinality of the relation. Thus the part relation in Figure 2.1 is an n:m relation. The arrows of the *part* relation in Figure 2.1 are shaded, because this relation is between different hierarchy levels.

Normally a hierarchy is represented as tree, but this is only a special case of the representation as a directed acyclic graph (DAG) which we will call the **hierarchy graph**. The database uses a model of the DAG for consistency checks. If a new *part* relation is to be added, it is first checked if this would result in a cycle. This would violate the above conditions and could be a new or previously made error. The user is then asked to solve this problem.

2.2 Classification

This model has to be extended for the complete description of schematics. A cell very often contains several copies of the same subcell. For the purpose of describing the connections between the subcells (netlists), each copy has to be identified. The description of the subcell is only needed once. Therefore we use the principle of classification and assign a type to each cell which describes it and an instance to each copy. The *instance* relation uniquely assigns a cell type to each cell instance. Thus the complex n:m *part* relation in Figure 2.1 is now split into two functional relations as shown in Figure 2.2. A cell (type) is composed of the cell instances of other cells. We can now generate a complete instance tree called the **hierarchy tree**.

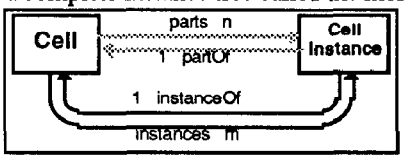

Fig. 2.2 Hierarchy Tree

This simple hierarchy model is used in many existing CAD tools. The disadvantage is the missing construct for alternatives. Each alternative forms a new cell. This results in a very early decision as to which alternative of a subcell is chosen. Otherwise no netlist can be generated. We wish to postpone this decision until it becomes necessary. Therefore we further extend our datamodel.

2.3 Alternatives

Let us first define cell and alternative more precisely. A **cell** is characterized by its function, for example a 2-input-nand or a 1-bit full-adder. For the implementation or realization of this function we have many structural, floorplan and layout alternatives. All of them are to belong to the same cell. We call these **realization alternatives**. Often in a design process, many of the alternatives are a valid choice and the

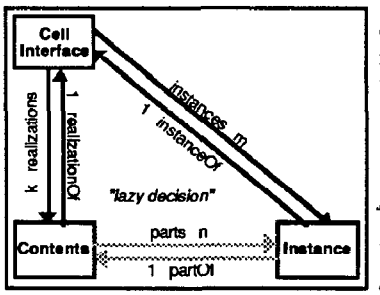

Fig. 2.3 Realization Alternatives

designer or a toolbox can determine the best one.

Let us for the moment select one domain, for example the structural domain. We further specify a cell type by its *interface* and its *contents*. The same approach has been used by many authors, for example /BBN 85/ and /MDK 87/ and is also part of EDIF. An *interface* can represent an abstraction of several *contents* alternatives with the same abstract function. This is expressed by the relation *realization* in Figure 2.3. A *contents* may contain *instances* (*part* relation). These are instances of the *interfaces* of the subcells. The *interface* typically contains enough information for a tool about the subcell. For example, a schematic diagram is sufficiently described by the interface symbols of its subcells. The internal structure of a subcell (contents) is only necessary if we want to refine a subcell.

This seperation has the decisive advantage that the use of a cell does not automatically require a decision about the selected realization alternative. This supports the principle of "**lazy decision**".

The division into *interface* and *contents* also provides the framework for hierarchical top-down designs. If we look again at Figure 1.3, we see that the Chip Planner only needs the *structure contents* (st-ct) of its CUD and the *structure instances* (st-is) and *interfaces* (st-it) of its subcells. In addition, it needs the *floorplan instance* (fp-is) defining the frame of the CUD and the subcell *floorplan interfaces* (fp-it) describing the shape functions. This principle avoids expansion of the hierarchy down to the leaf cells.

2.4 Domains

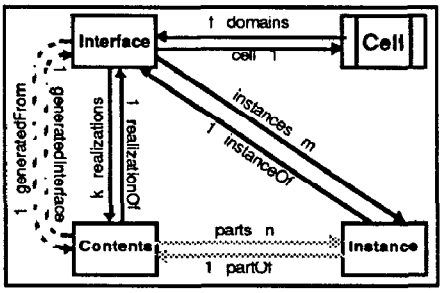

Fig. 2.4 Cell as Complex Object

We now extend the data model to several domains. As shown in Figure 2.4 a *cell* is now an abstract object which itself contains no design data. In each domain it may be represented by one or more *interfaces*. *Interface*, *contents*, and *instance* remain in the already known relations to each other within each domain.

We now need a relation between domains. In Figure 2.4 this is shown as the relation *generated* with dashed arrows. A design step has an initial cell description as its input and generates a resultant cell description. From one original *contents* we can generate several resultant *contents* alternatives. But all *contents* alternatives have the same resultant *interface*. Thus we can use a 1:1 *generated* relation between original *contents* and resultant *interface* as shown in Figure 2.4.

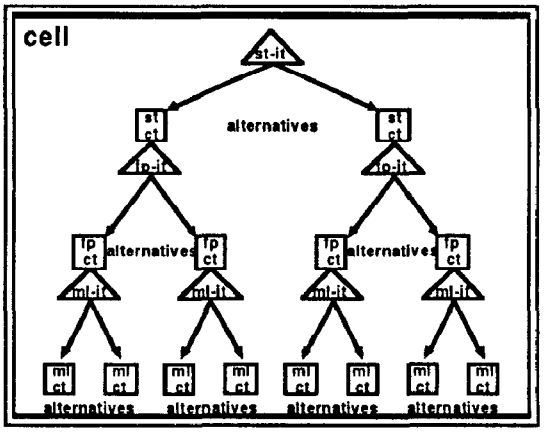

Fig. 2.5 DesignTree with Realization Alternatives

Figure 2.5 gives an example. Each *structure interface* (st-it) may be realized by st-ct alternatives. The Shape Function Generator generates one *floorplan interface* (fp-it) for each st-ct. The Chip Planner can generate several fp-ct for each fp-it. These are again alternatives. This is carried on to the masklayout domain. We call this tree of alternatives in Figure 2.5 the **design tree**.

2.5 Configuration

So far we have postponed the selection of alternatives for as long as possible. But the decision has to be made and we need a relation to express the decision. The process itself is called configuration and can be handled by the designer with a configuration editor or by the tools.

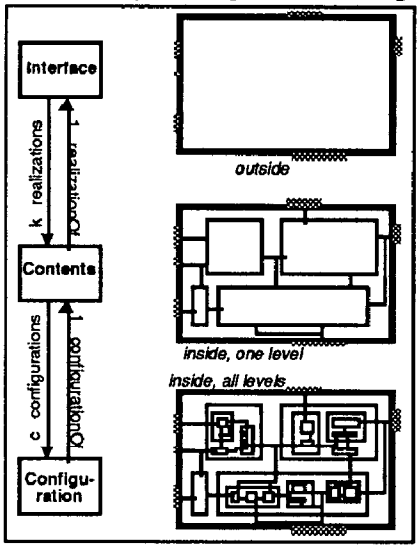

Fig. 2.6 Selected Alternatives

The configuration process is split into two levels. First, we can select a *contents* alternative for each *instance* and the corresponding *interface*. We call this an **initial configuration**. On the right side of Figure 2.6, the middle floorplan shows the initial configuration of the top cell. We do not know how each of the subcells is composed in itself. This has the following reason. Since we represent only the types of cells and the instances of a type once in the database, each instance may occur several times in the design tree (see chapter 2.4). Thus each of these occurrences of an instance may be configured differently.

We model this by a relation *configuration* that assigns several *configuration objects* (entities) to each *contents*. The left side of Figure 2.6 shows this relation. An initial configuration means that for each instance we select one contents. A **final configuration** is reached if we make a more detailed selection and chose a *configuration object* for each *instance*.

If we do this at all levels of the hierarchy, the complete refinement of a cell as shown in the floorplan example of Figure 2.6, bottom right corner, can be expressed.

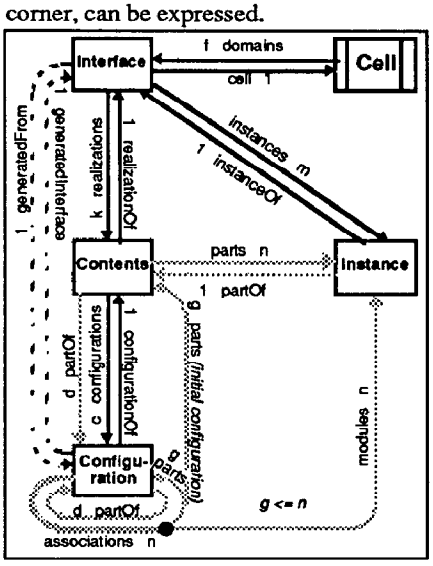

Fig. 2.7 Configuration Graph

A more complete datamodel is shown in Figure 2.6. An additional ternary relation *association* is introduced. Therefore it is necessary to extend the MAD-model. We define branched arrows for two cases. First branch type is an exclusive or, which in Figure 2.7 is used to select either a *contents* (initial configuration) or a *configuration* (final configuration). The arrows are drawn in the same style before and after the branching.

The more interesting second type is used to represent a ternary relation. The bold drawn line branches at the black dot in two arrows. They represent the two binary relations, in which each ternary relation can be split by losing some information.(in Figure 2.7 *parts* and *modules*). You cannot discover the association between each *instance* and the selected *configuration* (or *contents*) from these partial relations. The retrieval of the linkage information must be supported by the datamodel and implemented on the Design Databaase. In Figure 2.7 the relation *association* provides this information.

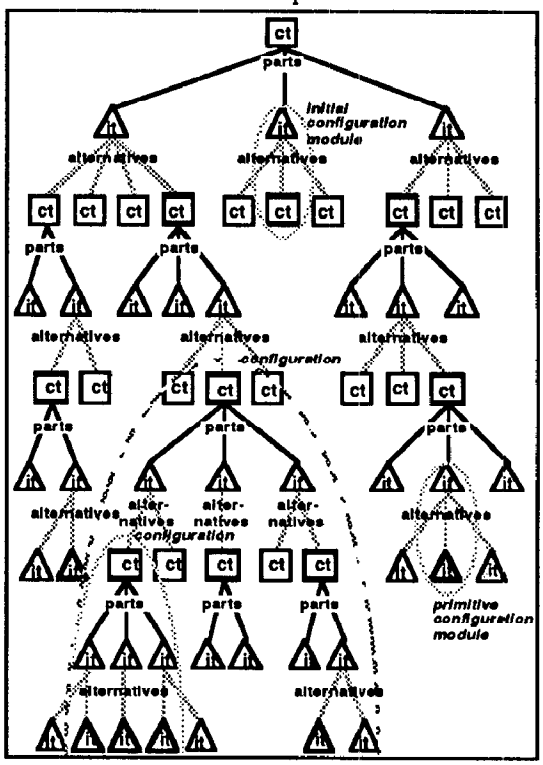

Fig. 2.8 Hierarchy Tree with Alternatives and Configurations

If we look at a specific *configuration* of a *contents* of cell A, the *association* links each *instance* of the subcells of A (*part* relation) either to the subcell *contents* alternative (initial configuration) as shown by the *association* in Figure 2.6 or to the subcell *configuration* alternative (final configuration).

At the bottom of the hierarchy, cells will not be composed of subcells, but are primitive or leaf cells. For these cells no configuration in the previously described sense exists. Instead, a reference to a corresponding leaf cell from a domain further to the right can be used as configuration. We call this a **primitive configuration**.

For example the Sizing Function Generator at the lowest level has *masklayout interfaces* (ml-it) as an input. This differs from the fp-it input to the same toolbox at the next higher level. The primitive configuration defines this special relation at the bottom level of the hierarchy.

In the case of *configurations* the *generated* relation also has to be changed. The original data for a toolbox is a configured *contents*. Therefore we draw the relation between *configuration* and *interface* as shown in Figure 2.7.

The ternary relation *association* in Figure 2.7 is now well represented by the MAD-model and therefore easy to understand. Let us try a different interpretation. *Association* has similarities to the *part* relation because it also links different levels of the hierarchy. If we consider the *part* branch of the *configuration* in Figure 2.7 instead of the cells in Figure 2.1, we get a similar DAG, the **configuration graph**. This is the subgraph of the hierarchy graph. The hierarchy graph contains all alternatives, the configuration graph only the selected ones.

The configurations are the vital part of a design database. In our opinion, it is not possible to find a simpler model to represent the configurations with the same expressive power. The hierarchy tree in Figure 2.8 may help to understand the meaning. For the sake of clarity the *instances* are not shown and are represented by the corresponding *interfaces* (it). The tree shows a *contents* (ct) with all its alternatives in one domain and three hierarchy levels. The *contents* that were selected for a design (configured) are emphasized by heavy print. *Interfaces* (it) from a domain to the right (primitive configurations) are shaded. *Configurations* are indicated by open ellipses. For some *interfaces* no *contents* exists so far. These incomplete configurations

are indicated by checkered open ellipses. We also note again that a intial configuration only spans one level of the hierarchy and a final configuration several levels.

2.6 Views

A further extension to the model are the *views*. They are the place for the storage of the design data or for pointers to secondary storage of the data. Several *views* of an object can exist. In principle these *views* represent the same data in a different form or different aspects of an object. Examples are the *views* "schematic" and "netlist". Both represent the structural information. A schematic entry tool can generate both *views*, but needs only read the netlist *view*. On the other hand, the complete floorplan information is represented by the *views* "frame", "iopins", "slicing tree", and "fp netlist". Different tools need different assemblies of these partial *views*.

2.7 Versions

The management of versions is another important task of a design database. Versions are generated sequentially and only the last version is, in general, valid. This is in contrast to alternatives which are all valid. Normally, a new version is produced if the last one had an error.

Nevertheless, old versions may have to be stored if they are origins of other cells (generated relation). The corresponding transaction may have to be reset and all generated cells may have to be newly generated with the new version as origin. If the error does not affect the generated cell, old versions may still continue to exist.

Versions are introduced in the datamodel for all *interfaces*, *contents* and for some *configurations*. We currently do not provide versions for *instances*, because in our experience a change in an *instance* always caused a change in the corresponding *contents* and a new *version* there. The *views*, containing the design data, are now related to the *versions*.

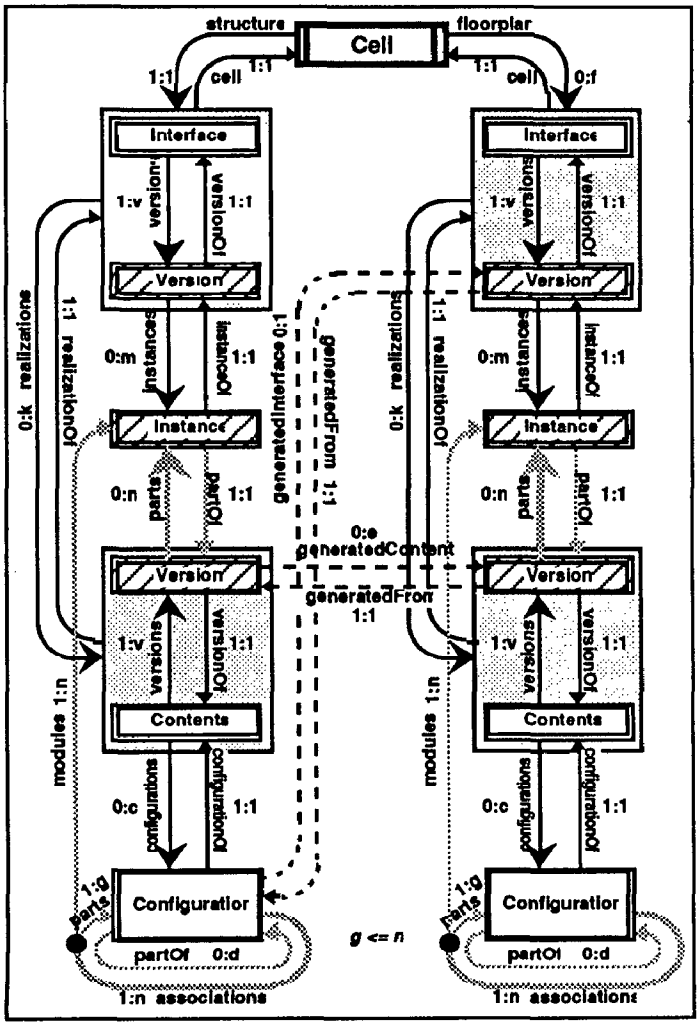

Fig. 2.9 Datamodel for two Domains

2.8 The Nearly Complete Datamodel

Figure 2.9 shows the datamodel for the domains structure and floorplan in the notation of an extended MAD model. Only the *views* are missing in order to simplify the drawing, objects with *views* have a hatching. The *versions* are included. In principle, each domain is represented by the datamodel which we developed in the previous chapters.

The *generated* relation relates *versions* to each other. An additional *generated* relation has been introduced between *contents versions*. We can now reconstruct the design history.

Figure2.9 also shows the limitation of datamodels. Many semantic details cannot be properly expressed. The figure tries to show some of the special characteristics by different shadings. It distinguishes relations between hierarchy levels and between domains. All black arrows represents relations within the same cell, the same hierarchy level, and the same domain. The bold type arrows point to dependent objects, for example a *contents* can only be defined to an existing *interface*. If you delete this *interface*, all dependent *contents* must be deleted too.

3. Conclusion

The datamodel has been implemented in Smalltalk 80 as a prototype. So far the experience is positive. The database does not represent a performance bottleneck. Assembly and disassembly of design files is performed in a few minutes. This time is shorter than the toolbox execution times. Smalltalk 80 provided a suitable environment for the object orientation of the data model. An implemented **object browser** could be used for ad hoc retrieval. The language also provides excellent functions as a retrieval language in the case of some extensions. Currently the complete PLAYOUT system is used for a VLSI design.

This research is part of the Sonderforschungsbereich 124 "VLSI Design Methods and Parallelism" which is financed by the Deutsche Forschungsgemeinschaft and the Center of Computer Aided Engineering Systems.

4. References

/BBN 85/ A. Beetem, J. Beetmen, A. Nigam, "HDMS: A Hierarchical VLSI Design Data Management System", IBM Research Report, Yorktown Heights 1985

/Ka 85/ R. Katz, "Information Management for Engineering Design", Springer Verlag, Berlin Heidelberg New York 1985

/MDK 87/ J. Mülle, K. Dittrich, A. Kotz, "Design Management Support by Advanced Database Facilities", Proc. of the IFIP WG 10.2 Workshop on Tool Integration and Design Environments, Paderborn 1987

/Mi88a/ B. Mitschang, "The Molecule-Atom Data Model", in: "The PRIMA Project, Design and Implementation of a Non-Standard Database System", T. Härder (ed.), SFB 124 Research Report No. 26/88, Kaiserslautern 1988

/Mi88b/ B. Mitschang, "Towards a Unified View of Design Data and Knowledge Representation", in: Proc. of the 2. Int. Conf. on Expert Database Systems, L. Kerschberg (ed.), pp 33 - 49, Virginia 1988

/Si 88/ E. Siepmann, "PLIF - Ein objektorientiertes Datenaustauschformat zur Kommunikation in PLAYOUT", SFB 124 Research Report No. 32/88, Kaiserslautern 1988

/Zi 86/ G. Zimmermann, "Top-down design of digital systems", in Logic Design and Simulation, E. Hörbst (Editor), Elsevier Science Publ. B.V., Amsterdam 1986

/Zi 88/ G. Zimmermann, "PLAYOUT - A Hierarchical Layout System", GI-18. Jahrestagung, Informatik Fachberichte 187, Springer Verlag, Berlin Heidelberg New York 1988