On Modeling Top-Down VLSI Design

Bernd Schürmann Joachim Altmeyer Martin Schütze

University of Kaiserslautern
D-67653 Kaiserslautern, Germany

Abstract

We present an improved data model that reflects the whole VLSI design process including bottom-up and top-down design phases. The kernel of the model is a static version concept that describes the convergence of a design. The design history which makes the semantics of most other version concepts, is modeled explicitly by additional object classes (entities types) but not by the version graph itself. Top-down steps are modeled by splitting a design object into requirements and realizations. The composition hierarchy is expressed by a simple but powerful configuration method. Design data of iterative refinement processes are managed efficiently by storing incremental data only.

1. Introduction

Due to the complexity of current and future VLSI circuits hierarchical design approaches become more and more important. Even systems like TimberWolf which seem to favor the flat approach change to a hierarchical design style to get acceptable run times for very large circuits [7]. There are two possible hierarchical approaches, *top-down* and *bottom-up*.

The bottom-up design style is the simpler one, and it is used by most current design systems. Cells at the lower hierarchy levels are designed first by considering internal restrictions only. The layouts of these cells will then be composed to larger cells. The disadvantage of this approach is the possible waste of chip area and wiring length. Since all subcells are designed independently, the shapes of adjacent cells generally do not fit and result in empty space when they are combined to larger modules. Wiring lines become relatively long because the pins are mostly at 'wrong' sides of the subcells.

These disadvantages can be avoided by a top-down design. Here, the cells at upper hierarchy levels will be designed first to get the global minimal layout and shorter wires [5], [12]. Further, the shapes and pin positions of the subcells are determined at the upper level and must be met by the design steps at the lower levels. Because of tolerances between estimations and final results the top-down design is more an iterative process. Cells must be improved several times until they meet the global restrictions. Often backtracking over several hierarchy levels becomes necessary [8].

The hierarchical design in general and the top-down design in particular are possible only by using a design database that manages the huge amount of data which are the result of the various design steps. However, standard database systems, which are very successful in application domains as banking and flight reservations, are not suitable for the VLSI design. Design objects are very complex and have a lot of interdependencies. The transaction concept has to be improved because design transactions often need very long time and may be nested.

All these requirements of the CAD domain resulted in new database research projects. Two basic problem classes have to be solved. One topic addresses new transaction concepts which allow concurrent and cooperating design steps and new storage concepts for complex objects. The other research topic is at the modeling level.

Several groups are looking for data models which describe the whole VLSI design process in an appropriate manner (esp. at the meta model level). Most of the research is based on the first works of R. Katz [3] and the definition of the EDIF data format.

Using the basic concepts of EDIF, our research group defined a data model with object-oriented features that supports hierarchical designs very well. It has been published in [9]. We implemented the model in a prototype database system and tested its quality by performing several (large) designs. The key concepts and our experiences with that model are briefly described below.

In parallel to our works, several more or less similar models have been developed by other groups. Important contributions are the works of R. Katz [4], E. Kupitz [1], W. Wilkes [10], the Nelsis group [11], and the CFI (CAD Framework Initiative) which is working on a standard data model for the VLSI design [2]. This list is by far not complete. What is common to all approaches is the fact that

Permission to copy without fee all or part of this material is granted, provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1994 ACM 0-89791-690-5/94/0011/0508 $3.50

they support the bottom-up design style very well. The top-down style on the other hand, which is made up of many (iterative) improvements, cannot be expressed so clearly. This design style needs the integration of a convenient version and configuration concept that is different from the design history and the ability to express requirements. The description of a suitable data model is the subject of this paper.

The rest of this paper is organized as follows: Section 2 gives an overview of our first data model. Our experiences and differences to other approaches are described in section 3. There, we also address the requirements of hierarchical top-down designs. In section 4, we present our improved data model that supports the requirements very well. Section 5 concludes the paper with some final remarks.

2. The Old Model

In this section we describe the key ideas of a data model that is implemented in our current design database. We present this model because it comprises most of the commonly applied concepts. Further concepts and differences between our and other approaches can be found in the following section.

The notation used in this paper is an extended ER diagram. Boxes describe entity sets or object classes and arrows describe relationships between the object classes. Arrows with a filled head represent functional relations which are defined in both directions. The cardinality is written at the arrow end. A '*' means an arbitrary number including zero, '+' at least one, '?' zero or one, and '1' exactly one. The name of the relation against the arrow direction is formed by exchanging the ending 's' of the given relation name with the pattern 'Of' (e.g. `part`**`s`** → `part`**`Of`**).

The basis of the data model is the description of a cell which is described by its interface and contents (white boxes in figure 1). The interface is the abstraction of the cell to its environment while the contents describes its structure or realization. In most cases, we have several realization alternatives which are all abstracted by the same interface. A contents is generally the composition of several parts (subcells) which are `Instances` of other cell `Interfaces`.

So far, we are able to describe the realization of a cell at one hierarchy level. It may be sufficient to reflect a flat approach. However, if we have alternative realizations (contents) for the subcells, which again may be compositions of smaller parts, we have to decide which alternatives should be used for the instances. This selection is called configuration. A contents has one or more configuration alternatives. The configuration is a list of `Associations` which form the relations between the instances (parts of the corresponding contents) and the selected objects. It is allowed to select either a contents or another configuration. If we select a contents we connect exactly two hierarchy levels. Otherwise, we connect three or more hierarchy levels by selecting a configuration.

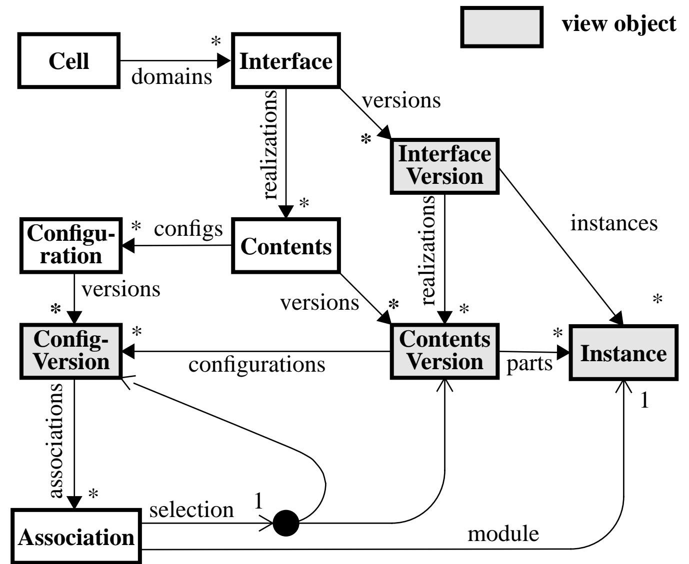

Figure 1 The current model

Now, we can express the composition hierarchy of a cell. But this is not enough to describe a whole design. Besides the composition hierarchy, we need an abstraction hierarchy, a version concept, and the ability to build views.

A cell will generally be designed at different abstraction levels which we call domains. The first design steps are the behavioral design (domain behavior) and the netlist generation (domain structure). Later on, we perform area estimation and floorplanning (domain floorplan), and we assemble the layout (domain masklayout). This distinction between different domains is modeled by inserting the relation `domains` between the root object `cell` and the `Interfaces` of the different domains.

The gray boxes in figure 1 show the remaining concepts. An important concept is the version management. Versions are generated sequentially by correcting design errors or by improvement steps. In contrast to alternatives, only the last version is generally valid. Versions are used for the object classes `Interface`, `Contents`, and `Configuration`. Instances need not be versioned individually. They exist with their surrounding contents only and, therefore, have the same version as the contents.

The last major extension is the view concept. Views are the place for storing the actual design data. They act as an interface between the meta model and the detail data. In

our current database implementation, the design data are stored in files. Views are pointers to the file system only. Each object may have several views. They are used to represent the same data in different forms (e.g. a circuit can be expressed in form of a schematic or an equivalent netlist) or to partition the object data into different aspects (e.g. a cell interface may be partitioned into the views `frame` and `pins`).

With figure 1, we described the main parts of the data model. The complete model further has object classes for cell libraries, technologies, etc. Data objects of different domains are linked by a `generated` relation which basically models the execution of a tool. For instance, the floorplan of a cell is *generated* from the netlist by executing the chip planner.

3. Experiences and Related Works

We implemented the model in the design database of the PLAYOUT VLSI design system [12]. During the last two years, we then designed several test circuits with various number of hierarchy levels. Some of the designs were performed bottom-up while in most cases we preferred the top-down approach. The largest design was a circuit with about 280.000 standard cells. In that time, we got a lot of experiences with our data management.

In the remaining part of this section, we will discuss three topics which are important for top-down design: the composition hierarchy, the iterative refinement process, and the distinction between requirements and realizations. For each of these topics, we will describe the strengths and weaknesses of our model and we will show different solutions.

3.1 Composition Hierarchy

The basic object classes `interface`, `contents`, and `instance` can be found in nearly all flat design systems and exchange formats, e.g. high-level synthesis systems as MIMOLA [13], one-level Place&-Route tools, as well as hardware description languages like YAL [6] and Calcos.

For hierarchical designs, we must introduce a configuration concept. The most simple way is storing the configuration directly at the instance as it is done in the CFI connectivity information model [2]. But this simplification has several restrictions. For instance, it allows the description of one fixed hierarchy tree only. More flexibility is available with the introduction of the fourth object class `Configuration` as shown in figure 1. Such an object can be found in several other hierarchical models as for instance the VHDL data model [14] or the Cadlab data model [1] that shall become part of the JESSI Common Framework.

The configuration makes the data management of top-down designs possible. While designing a cell at one hierarchy level, it is not necessary to decide which realization alternative (contents) will be used for a given instance. The decision can be postponed until the whole hierarchy of an actual design is needed. This is known as *lazy decision*.

Currently, we allow a change of the circuit hierarchy, that is called *repartitioning*, in the domain `structure` only. In all other domains the hierarchy must be fixed because we have a lot of references to the circuit netlist which is stored in the domain structure. On principle, this is a technical problem only. Most of the physical design tools work at one hierarchy level and need no knowledge of the hierarchy meta model. However, this becomes different if tools like a floorplanner perform repartitioning. Data models which support the change in hierarchy during the whole design process can be found in [1] and [11].

3.2 Iterative Refinement Process

A top-down design often consists of many incremental improvement steps. Currently, we can model the results of the different steps as independent alternatives or linear versions only. Using the alternative concept, we do not know the derivation history of the objects. Modeling the improvement steps as linear versions we cannot express alternative versions which are derived from the same predecessor.

Solutions to this problem can be found at various places in the recent literature. Many data management research groups are working on versioning. Nearly all authors describe trees or directed acyclic graphs (DAGs) as possible structures. They differ in the semantics of the relations only. A good overview of the most important version concepts is given in [4].

3.3 Requirements and Realizations

Using a top-down approach, a contents needs relations to two different interfaces. The first is a requirement that is the result of the design at the upper hierarchy level. For example, the chip planner determines the placement and frames of all subcells in a floorplan. These subcell frames are inputs or requirements for the chip planning steps at the next lower hierarchy level. The floorplans of the subcells have to meet these requirements (frames) as good as possible. However, because of estimation tolerances there will always be differences.

So, the contents of the subcells (i.e. the subcell floorplans) have relations to the requirements they are derived from and to abstraction data of the realizations. The requirements and the abstraction data are both interface descriptions. Since most data models support bottom-up designs we do not find distinctions between requirement and realization.

4. The New Model

With our new data model we present solutions for all three problems. As far as possible, we adapted the concepts of other approaches. Nevertheless, several key ideas described below are new. Although we combined different ideas, our new model seems to be simpler than most other models. We first address the iterative refinement process, then the difference between requirements and realizations, and finally the decomposition hierarchy.

At the end of this section, we discuss an additional topic: the data inheritance. Data inheritance is the key aspect for the definition of a simple but powerful data model without storing many redundant data.

Our tool execution model describes the dynamical design aspects and completes a version concept that models the static refinement relationships only. However, these aspects are beyond the scope of this paper.

4.1 Iterative Refinement Process

We already mentioned that the a top-down design often results in many iterative refinement steps. This is especially true for the physical design phase. Due to estimation tolerances, it is not possible to perform pure top-down chip planning. Without adjustment steps at upper hierarchy levels, the final layout consists of a large amount of wasted area or long critical paths [8].

Our designs have shown that in practice each cell exists in many refinement states. These different instances of a cell are generally called versions [4]. As we described above, there generally exist versions which are derived from others and versions which are alternative to others. So, we need a version tree as proposed by Katz.

For our needs, the tree structure is sufficient. There is no need for a DAG which allows the merge of versions. In practice, the merge of two or more versions is a too complicated task.

We now discuss our version semantics in more detail. We first define the refinement of attributes or views of design objects, and then we define the `refinement` relation of the meta data model. This refinement relationship describes one possible version semantics.

Refinement of Attributes

Each version of a cell has several simple attributes (e.g. cell name) and more complex attributes (e.g. a netlist). The complex attributes correspond to the views described above. The actual design data are stored in files while the data management system uses the file name as attribute value. For the context of this paper it is not necessary to distinguish between attributes and views. We will use the term attribute only.

For documenting the design progress, we assign a state value to each attribute.

Definition (attribute states): Each attribute is in one of the states: *unknown*, *default*, *predicted*, *preliminary*, or *final*. We further define a partial order on these state values which represents the refinement of the attribute:

{*unknown*, *default*} < *predicted* < *preliminary* < *final*

where (`s1 < s2`) means that `s2` is a refinement of `s1`, i.e. `s2` is more precise than `s1`. ❏

As an example, let us choose the geometrical frame of a cell. This frame is *unknown* as long as we do behavior or structure synthesis. During top-down chip planning the computation of the subcell frames is based on a preceding area estimation so that the state changes to *predicted*. The frame of a floorplan is in state *preliminary* because the geometry of the floorplan may still change. Only the frame of a layout may be in state *final*.

The refinement states of attributes are the basis of our version semantics.

Version Semantics

A design tool gets one version of a cell as input and generates a new version as output. In general, the input and output objects are linked by the `version` relation.

This version concept describes the execution order of design tools as we can see it in most publications. However, such a version graph may be different to the refinement steps of a cell. A continuous refinement process may go into a 'dead end' if the estimation errors are too large. Then, it may be necessary to exchange a more precise design object (e.g. a layout frame in state *final*) by a less precise object (e.g. the frame of a flexible cell in state *predicted*) to increase the flexibility of a design. This problem will be explained with figure 2.

Example (refinement): Figure 2 shows a floorplan with four subcells. In floorplan `1` all subcells are flexible, i.e. their shapes are based on an area estimation. The next steps are chip planning and layout generation of the subcells. This can be done for all subcells in parallel or sequentially. In the example, we first generate the layouts of subcells `A` and `B`. After that, we adjust the top-level floorplan (floorplan `2`). We assume that for cells `A` and `B` the estimation is correct so that floorplan `2` does not differ from `1`. Next we generate the layout of cell `D`. For this cell, the shape of the layout is different from the shape based on the estimation. The adjustment at the top level is now more complicated. For minimizing the wasted area, we put cell `A` on top of cell `D` and discard the layout of cell `B`. Cells `B` and `C` get new shapes which fit to the layouts of cells `A` and `D` (floorplan `3`). After that, we compute a new layout for cell `B` that fits in the given area. ❏

In our data model, we distinguish between the static refinement or version relation and the modeling of the dynamic aspects of tool executions. We define the version semantics as follows:

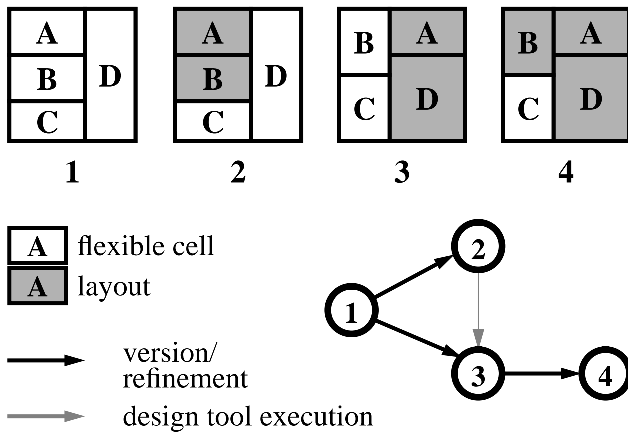

Figure 2 Floorplan Refinement

Definition (Versions): A descendant in the version tree always represents a more precise (i.e. refined) object than a predecessor that is nearer to the root of the tree. In other words, a design object o2 is a version of o1 (o1 $\longrightarrow$ o2) if o2 is more precise than o1.
The preciseness of the design objects, and with that the version relation, is defined on the states of the object attributes. An object o2 is a version of another object o1 only if all of its attributes are equally or more precise than the attributes of o1. ❑

This definition determines the position of a new design object in the version graph. After performing a design step, we first compare the input and output versions. In the case that all attributes of the output version are equally or more precise than the attributes of the input version both versions will be linked, i.e. the output version becomes a direct descendant of the input version. In the case that at least one attribute becomes less precise, we backtrack through the version tree toward the root until we find a version for which the condition is true (all attributes of the new version are equally or more precise than the attributes of the considered version). The new version will be linked as a descendant to that version.

Example (Version Tree): For our example, we define the preciseness of the floorplan on the configuration, i.e. on the state of the subcell shapes (see subsection 4.3 for more details). A floorplan is a refinement of another floorplan only if all subcell frames are more precise. With that definition, floorplan `2` is a version of floorplan `1` because we replaced two *predicted* shapes (flexible cells) by *final* layouts. Since we then discard the layout of cell B, the floorplan `3` is **not** more precise than floorplan `2` and therefore no descendant in the version tree. To find the right location in the tree, we traverse the tree back towards the root until we find an object that is less precise than the new version. In our case, this is the root (floorplan `1`). The new floorplan `3` becomes a version of floorplan `1`. The relation between the floorplans `2` and `3` (`3` is computed from `2`) that is annotated by the dotted line is part of the tool execution model but no version relationship. ❑

4.2 Requirements and Realizations

A second important topic for modeling the top-down design process is the distinction between requirements and realizations. Analyzing our designs, we can see that in most cases the relationship between interfaces and contents were used in a requirement-realization manner. For instance, high-level synthesis systems specify the netlist interfaces of cells which are not provided by a subcell library. To these interfaces we later have to generate realizations (contents), e.g. by using module generators. Another example is the top-down chip planning [12]. The chip planner specifies the frames (geometrical interfaces) of the subcells for which it later computes one or more floorplans (realization alternatives).

Note that there is a difference between our semantics of a requirement and the more generally used term ‘specification’. In principle, all input data of a design step can be seen as specification data. These are the interface data computed at a higher hierarchy level (which we call requirement) but also the more abstract data which were computed by an earlier design step of the actual cell. The latter are the cell attributes of preceding versions.

Definition (requirement): The term *requirement* represents the interface data of a cell which are (top-down) generated at the hierarchy level of the parent cell. ❑

All other specifications like the netlist input data for chip planning are called *realization* data because they are the result of a previous design step. While these *realization* data **must** be valid for the output data of following design steps the top-down computed *requirements* **should** be met as good as possible. If a top-down requirement is not exactly met an adaption step at the upper hierarchy level becomes necessary. An example will be given below.

The basis of our new model is the splitting of a `Cell` into two object classes: `Requirement` and `Realization` (figure 3). A cell is described by all its realizations and requirements. The version tree is defined on the class `Realization`. Each of these versions has one or more attributes/views which abstract the object to its environ-

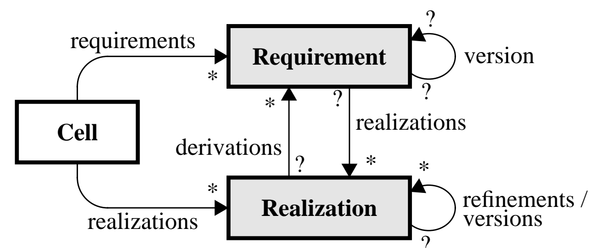

Figure 3 Requirements and Realizations

ment (interface data).

The object classes `Requirement` and `Realization` are linked by two functional relations `realizations` and `derivations`. A requirement may result in several alternative realizations and we can derive different requirements from one realization. This should be illustrated by the example shown in figure 4.

Example (requirement - realization): The upper part of the graphic shows a section of the geometrical refinement process of a cell `X` [8]. First, `X` is input to the planning step of the father cell. At that time, `X` is only estimated by a shape function. The chip planner derives a first requirement (frame) from that estimation. The requirement is input for a planning step which generates a floorplan f_1 of cell `X`. f_1 is then used for an adjustment of the father cell that results in a new requirement of cell `X` which itself is input to a new planning step (computing a refined floorplan f_2).

The lower part of figure 4 shows the relationships between the design objects generated by these steps. The first requirement (frame) of `X` is derived from the estimated shape function and it results in a realization (floorplan f_1) which is a refinement of the estimation. It has a further version f_2. ❑

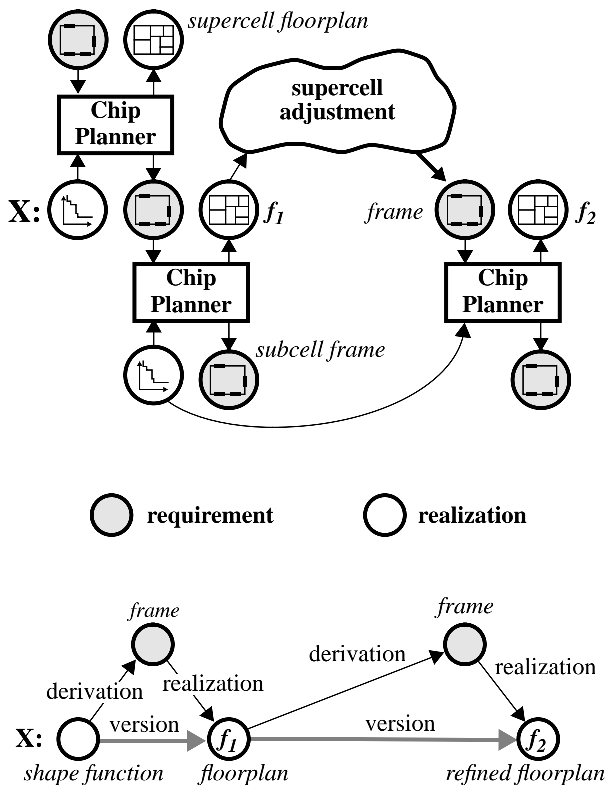

Figure 4 Design Example

Further Version Semantics

The version tree described so far expresses the refinement of design objects and design alternatives (branches in the tree). A third kind of versions mentioned by Katz models the correction of wrong data [4]. We model this semantics by the same version tree. This is done by annotating the relation between two `realization` versions as *refinement* or *correction*. If we correct a given version we have to mark that version and all descendants as invalid. The new version becomes a sibling node of the corrected version in the version tree.

In general, there exists no direct relation between different requirements. Nevertheless, we versioned these objects. We use a linear relationship that expresses modifications or corrections of requirements. It is not necessary to model alternative requirements.

4.3 Decomposition Hierarchy

In the previous two subsections, we described the modeling of requirements and iterative refinement. Now, we add the decomposition hierarchy for which we need one further object class `Instance` as shown in figure 5. This results in a simple data model that is as powerful as all other models known by us in describing the cell hierarchy.

In section 3, we described the configuration as the 'glue' for connecting the hierarchy levels in a top-down design. In our new model, the configuration is done by the objects of the class `Instance` that has relations (`associations`) to one or more objects the instance is configurated with. The `associations` relation configures a subcell with a `Requirement` (which may be an imprecise interface) or a `Realization`. This realization object can be any node (version) in the refinement tree. It may be the root of the tree which describes an interface in most cases, or it may be a very precise leaf node object which implies the whole decomposition subtree. An instance may have associations to more than one object. In some cases, it is desirable to select several alternatives for one instance and to postpone the restriction to one object to a further refinement step.

The configuration is strongly related with the refinement tree. A realization object can only be as precise as its parts. This has already been mentioned in subsection 4.1. We get following consistency condition:

Definition (subcell-based refinement): A realization object R is a refinement of R' if all subcells of R are associated with the same or more precise objects than the subcells of R'. An association A is called more precise than A' if A refers to a realization object that is a descendant (in the refinement tree) of the realization object referred by A'. A reference to a requirement is as precise as the realization the requirement is derived from. ❑

This condition is sufficient for the data management

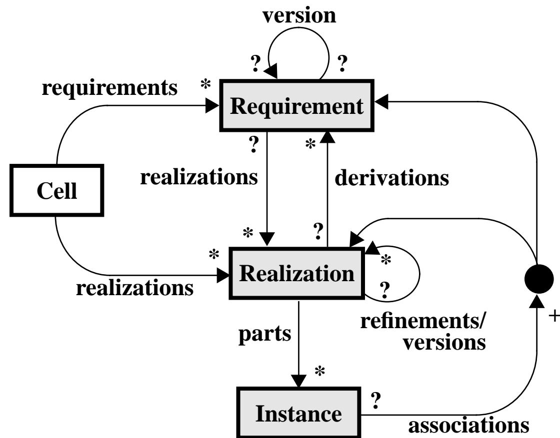

Figure 5 Basic Data Model

system to insert a new realization object (version) at the correct position in the refinement/version tree (see subsection 4.1). Generally, we get a new realization version due to a refinement of one or more subcells. Then, the new version will be linked to the version that was input to the design step. However, in some cases it is necessary to exchange an association by a less precise one for improving the design flexibility (because less precise objects generally are less rigid). In that case, the new version will be linked to a predecessor of the version that was input to the design step.

4.4 Data Inheritance

Subsections 4.1 to 4.3 described the basic object types and relationships of a new data model. This model is very simple and powerful. Interface data, contents data, and the configuration are modeled (with different attributes) by one object type, the `Realization`. The iterative refinement of a cell is expressed by a version tree. However, one may argue that we have to pay for this advantage with an overhead in the design data. For instance, several configuration alternatives could not share the data of one contents. All contents data seem to be stored redundantly with each configuration alternative. Even if we change the association of one subcell only, we get a new realization object that would have all attributes and all subcell associations. Except the changed association, all data would be the same as for the previous version.

We avoid this multiple storage of the same data by applying inheritance. In contrast to the structural inheritance of the object-oriented approach, we use the data inheritance of object attributes. Attributes may be simple data as names, sizes, etc., or complex data structures (see above). Only attributes whose values have changed are stored with the object. A `Realization` object inherits all other attribute values from a predecessor that is located nearer to the root of the refinement tree. The retrieval of an attribute value starts at the actual version and follows the refinement tree towards the root until it reaches a version where the value is stored.

Attribute values may change several times during a design process. In that case, several design object versions containing different values of the same attribute exist on a branch of the refinement tree.

Example (data inheritance): Figure 6 shows the refinement tree of an adder as an example. The circles represent the versions of the adder. Beside these circles we find the attributes which are stored with these versions. The example uses the attribute `name`, several `views` (complex attributes), and the `parts` relation which is treated similarly as attributes (see below). At the beginning, the function and the structural interface (i.e. the ports in view `s-frame`) of the cell are known. They are all stored with the root version. Two realization alternatives are derived from the root as refinements: the netlists of a carry-look-ahead adder and a ripple-carry adder. Both inherit the function and the interface data from the root but they overwrite the cell name. They further store the additional `netlist` and `schematic` (`RC-adder` only). The `RC-adder` is composed of two half adder (`HA1`, `HA2`) and an `OR` gate which refer to their interfaces (i.e. they refer to nodes of their refinement trees). The next step is configuring the half adders down to the leaf nodes of the composition tree (version `4`). We then perform chip planning twice which results in two alternative floorplans (versions `5` and `6`).

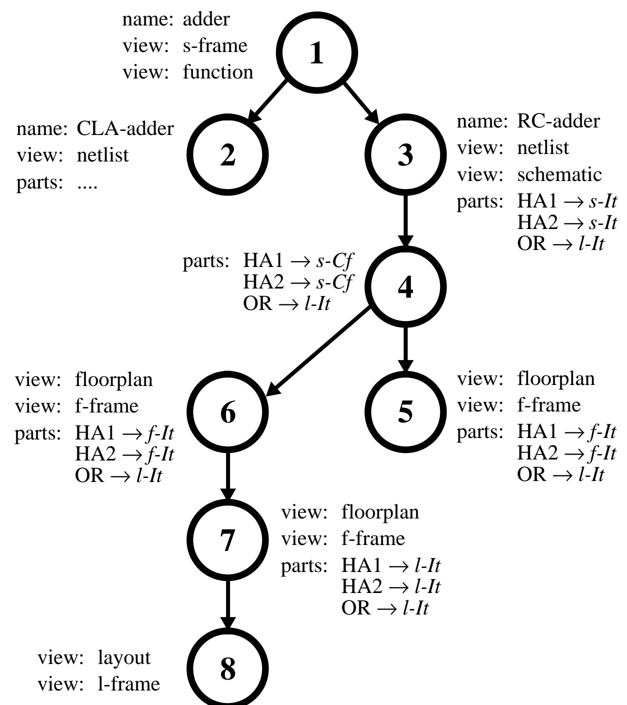

Figure 6 Data Inheritance

The corresponding versions store the floorplan and floorplan-frame and inherit the structural data from their common ancestors (versions 1, 3, and 4). The subcells are configured with floorplan interfaces (frames). The final two refinement steps are an adjustment of the floorplan with subcell layouts (version 7) and the assembly of the final layout (version 8). Although the leaf node stores the layout data only, it inherits the cell name (RC-adder) and the structural data from versions 1 and 3, and it inherits the floorplan data as well as the subcell configuration with layout data from version 7. ❑

The inheritance concept is not only used for attributes but also for the parts relation. Relations are treated like attributes as it is the case for the object-oriented approach, too. A new Instance object (association) may supersede an older version of the same subcell. Whenever we change the association of a given subcell, we simply supersede the part reference that belongs to the subcell identifier. The associations of unchanged subcells will be inherited.

The inheritance concept can be implemented by assigning unique identifiers to the subcells of a given cell. These identifiers allow changing the decomposition hierarchy (*repartitioning*) during the whole design process. For changing the hierarchy, we add new subcell identifiers and 'delete' the identifiers of subcells which are no longer part of the current cell. Subcells are deleted by setting their part relation to NIL.

Inheritance is also applied to requirements. For the data retrieval, we follow the versionOf and derivationOf relations (figure 5). A requirement version inherits data not only from previous versions but also from the realization the requirement is derived from. For instance, the requirement object of a module generator stores the netlist interface itself, but it inherits the specified behavior from a realization object.

4.5 Conclusions

With this paper, we presented an improved data model for the VLSI design. We described the particular problems of modeling top-down design steps. We solved the problems by defining a versioning concept that models the iterative refinement of cells, and by splitting design objects into requirements and realizations.

We only described the most important aspects of the model which are different to current approaches. Other aspects like libraries and technologies which are modeled similarly to all other models have been neglected. The inheritance concept avoids storing redundant design data and makes the model simple. Tool executions will be modeled independently of the static version tree.

Our new model is a result of performing and analyzing several large test designs. We examined our old model but also other models if they support all design objects and design steps. Since no model meets all requirements, we had to define a new model.

Examining our chip designs, we found out that our new model supports the top-down and refinement steps very well without any disadvantages for bottom-up steps and the usage of library data.

5. References

[1] M. Brielmann, E. Kupitz, "Representing the Hardware Design Process by a Common Data Schema", Proc. Int. European Design Automation Conference, 1992

[2] "Design Representation Electrical Connectivity Information Model and Programming Interface", CFI Pilot Release Document, CFI-92-P-6, 1992

[3] R.H. Katz, "Information Management for Engineering Design", Springer Verlag, 1985

[4] R.H. Katz, "Towards a Unified Framework for Version Modeling in Engineering Databases", ACM Computing Surveys, Vol. 22, No. 4, 1990

[5] M. Pedram, B. Preas, "A Hierarchical Floorplanning Approach", Proc. Int. Conference on Computer Design, Cambridge, 1990

[6] B. Preas, K. Roberts, "YAL Language Description", part of the MCNC benchmark distribution, MCNC Research Triangle Park, NC, 1987

[7] W. Sun, C. Sechen, "Efficient and Effective Placement for Very Large Circuits", Proc. Int. Conference of Computer Aided Design, 1993

[8] B. Schuermann, J. Altmeyer, G. Zimmermann, "Three-Phase Chip Planning - An Improved Top-Down Chip Planning Strategy", Proc. Int. Conference of Computer Aided Design, 1992

[9] E. Siepmann, G. Zimmermann, "An Object-Oriented Datamodel for the VLSI Design System PLAYOUT", Proc. 26th Design Automation Conference, 1989

[10] G. Scholz, W. Wilkes, "Information Modelling of Folded and Unfolded Design", Proc. Int. European Design Automation Conference, 1992

[11] P. van der Wolf, N. van der Meijs, T.G.R. van Leuken, et.al., "Data Management for VLSI Design: Conceptual Modeling, Tool Integration and User Interface", Proc. IFIP Workshop on Tool Integration and Design Environments, 1988

[12] G. Zimmermann, "PLAYOUT - A Hierarchical Design System", Information Processing 89, G.X. Ritter (ed.), Elsevier Science Publishers B.V. (North Holland), IFIP, 1989

[13] G. Zimmermann, "The MIMOLA Design System - A Computer Aided Digital Processor Design Method", 25 Years of Electronic Design Automation, A compendium of papers from the Design Automation Conference, 1988

[14] "IEEE Standard VHDL Language Reference Manual", The Institute of Electrical and Electronics Engineers, Inc., New York, 1988