Reuse of Design Objects in CAD Frameworks

Joachim Altmeyer Stefan Ohnsorge Bernd Schürmann

University of Kaiserslautern
D-67653 Kaiserslautern, Germany

Abstract

The reuse of well-tested and optimized design objects is an important aspect for decreasing design times, increasing design quality, and improving the predictability of designs. Reuse spans from the selecting cells from a library up to adapting already designed objects.

In this paper, we present a new model for reusing design objects in CAD frameworks. Based on experiences in other disciplines, mainly in software engineering and case-based reasoning, we developed a feature-based model to describe design objects and their similarities. Our model considers generic modules as well as multi-functional units. We discuss the relationships of the model to the design process and to the configuration hierarchy of complex design objects. We examined our model with the prototype system **RODEO**.

1. Introduction

The complexity of VLSI designs increases rapidly. Contrarily, the design times must be reduced to resist the growing pressure of competition. To solve this problem, new design management concepts are necessary which reduce superfluous design activities and concentrate the designer's work on the essential and ambitious problems.

An examination of large design databases shows that many functions are realized more than once. In addition, cells are laid out in many alternatives. Therefore, a chance to reduce the design time is reusing already existing results. We can do this reuse by instantiating design objects which exactly match a given requirement specification or by adapting similar objects. In this paper, we present a feature-based reuse model which expresses the suitability of reusing design objects for the actual design.

The Reuse Problem

Reuse of design information is one of the best opportunities to increase the productivity. Reuse decreases the design time and increases the product quality by using well tested design objects [8]. In ECAD systems, reuse spans from selecting cells from a library [6] up to adapting already designed objects to a given requirement specification.

Generally, we discern three modes of reuse:

- *Reuse by Instantiation*
  The main idea is to reuse often used components, e.g. registers, instead of redesigning them from scratch. Here, candidates for the reuse are frequently used components or components which implement standards, e.g. the IEEE floating-point standard [8].
- *Reuse by Parameterization (Generation)*
  A parameterized object (e.g. like a net in [3]) is instantiated with fixed values. For example, instead of designing a new 32-bit multiplier an existing n-bit multiplier can be instantiated with a width of 32 bit. A parameterized object builds an equivalence class over a set of concrete objects. At least before generating the final layout, a generator or a compiler must flatten the parameterized objects. Examples of possible parameter classes are the bit-width, the arity, the function, and the microprogram memory.
- *Reuse by Adaptation*
  Already designed objects are adapted to a given requirement specification. For example, it is much more easier to write a VHDL program of a 32-bit adder by adapting the code of a 16-bit adder instead of designing the adder from scratch. The correctness of such designs is not kept automatically and the result of the adaptation process must be validated.

In the first and second case, we search for objects which match the given requirement specification exactly. In the third case, we search for objects which are suitable for an adaptation process that means they are only *similar* to the given specification. The degree of similarity of a requirement specification and a given object should correspond to the costs of the complete adaptation process.

Reuse in other Disciplines

Of course, reusing components is not specific to ECAD. The problem of reuse is also well known in soft-

ware engineering and in artificial intelligence by the term case-based reasoning. In software engineering, a survey of interesting works can be found in [4]. In [2], a framework for a "software life-cycle technology that allows *comprehensive reuse* of all kinds of software-related experience" is introduced.

Case-based reasoning deals with the reuse of available solutions [10]. The basic process of case-based reasoning is presented in [5]. First, all existing cases are indexed in a *case memory* so that they can suitably be retrieved (*indexing*). A search algorithm finds the adequate reuse candidates. The most appropriate candidate or candidates are selected and adapted to the new situation. After evaluating the result, the case memory is extended with the new cases. The used similarity measures for the retrieval process [12] base on the *Tversky contrast model* [17].

Tversky defines for two objects x, y:

A = The set of features belonging to x

B = The set of features belonging to y

A similarity scale S of x and y is defined as

$$S(x, y) = \theta \cdot f(A \cap B) - \alpha \cdot f(A - B) - \beta \cdot f(B - A)$$

with θ, α, and β are positive real numbers. This model expresses the similarity of two objects as a function of their common and their different features (see figure 1).

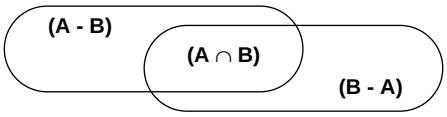

Figure 1: Venn Diagram to Illustrate Tversky's Contrast Model

If, for example, $\theta = 1$ and $\alpha = \beta = 0$, we only look for common features. On the other hand, if $\alpha = \beta = 1$ and $\theta = 0$, we only take the different features into account.

Reuse in ECAD

The process of reuse in an CAD systems is determined by both the complexity of the design objects and the complexity of the design processes. [9] describes useful relations between design objects. Obviously, if we permit the reuse of all objects stored in the design database (and we avoid redundancy), the relations between the design data then become complex.

An important difference between ECAD and many traditional CAD systems is the subdivision of a design process into several abstraction levels (*domains*). A VLSI design process may begin with an abstract description (in form of an HDL-program), continues with a netlist representation, and ends with the final layout of a cell.

To date, the support of reuse in existing ECAD systems is only rudimentary. In [6], a cell selection method is presented where for each cell the relevant data are normalized and ranked before a search algorithm is used. In [8], the problem of reuse is motivated and a classification of reuse strategies is given. Although there are first publications, much more work has to be done before "comprehensive reuse" is supported by ECAD frameworks.

Structure of this Paper

The rest of this paper is organized as follows: In section 2, our reuse model is presented. Section 3 shows a prototype implementation of a reuse tool. An examination of the reuse potential and first experimental results are shown in section 4. At the end, section 5 summarizes the presented reuse method. This leads to a survey of interesting future works.

2. The Reuse Model

In this section, we characterize design objects and a common design model. After defining specifications, we focus on the similarity between these specifications and existing design objects. We also describe the design space, the design process, and the configuration hierarchy of complex objects.

Design Objects

In contrast to most publications (especially [9]), we call every object resulting from a design step a *design object (DO)* because from the viewpoint of the reuse process we need not differ between alternatives or versions. Each design object is characterized by several *properties* or *features* which are defined as follows:

Definition: Feature (Property)

Let DO be a design object. A feature (property) p is a predicate with

$$p(DO) = \text{true}.$$

We define a *set of features* (*set of properties*) *P of DO* as

$$P = \{p \mid p(DO) = \text{true}\}.$$

Note that the features do not depend on the representation of the design object (e.g. VHDL program or EDIF netlist). Every feature is an instance of a *feature class*. For example, the feature 'Aspect Ratio is 1.5' is an instance of the feature class Aspect Ratio. Of course, more than one instance of a feature class can be assigned to a design object (e.g. {'Function is Adder', 'Function is Multiplier'} are features of an ALU with 'Function is Adder' and 'Function is Multiplier' are instances of the feature class Function).

Features can be classified into *simple* or *complex* features. A feature is regarded as an attribute/value combination. Simple features have atomic values (e.g. 'Size is 0,04 mm^2') whereas complex features represent aggregates (e.g. in form of a truth table). Simple features are *nominal*, *ordinal*, or *cardinal*. For example, a feature class Technology contains nominal, Design State ordinal, and Size cardinal features.

A feature is called *generic* if it summarizes other features of the same feature class (e.g. 'Arity is Even'). Therefore, a generic feature defines a subset of the corresponding feature class. In our case, the feature 'Arity is Even' corresponds to the set {'Arity is 2', 'Arity is 4', 'Arity is 6', ...}.

Specifications

Similar to design objects, requirement specifications are also characterized by a set of features. The input of a design process is a *system specification*. The features of this specification span different domains in the design space. For examples, in the specification {'Function is Multiplier', 'Width is 16-Bit', 'Technology is CMOS', 'Size $\leq$ 0,15 mm^2', 'Aspect Ratio is 1'} the function belongs to the domain behavior whereas the aspect ratio belongs to the domain layout. Therefore, only some of these features make the input specification of the first design step, e.g. writing a VHDL description for the 16-bit multiplier. Together with other features of the system specification, this description is the specification of the next design step. We call these specifications for single design steps *tool specifications*. The system specification is a tool specification at the meta level.

A Common Design Model

Figure 2 shows the history of a design process of a module documented in the product model. The design began with an abstract description (e.g. in form of a high level hardware description language) represented by DO_1 and ended with the final layout DO_5. The module will generally be designed at different abstraction levels which we call domains (similar to the representations in Gajski's Y-Chart [7]; the geometric representation is divided into floorplan and layout). Based on specifications, the first design steps could be a behavioral design and a netlist generation. Later on, we perform area estimation and floorplanning, and, finally, the construction of the layout. All realizations (design data) of a module are part of a

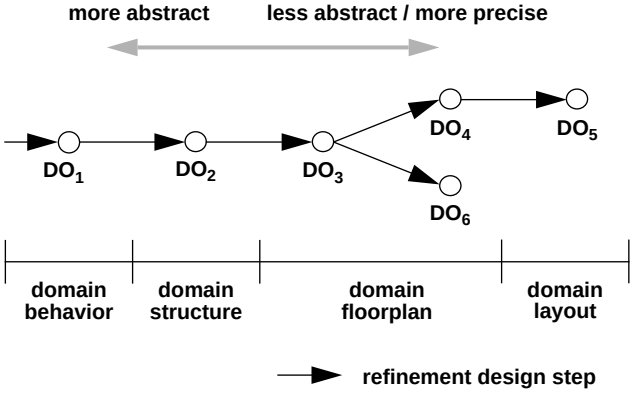

Figure 2: Example of a Design

refinement tree. This tree is similar to the version tree of Katz [9]. Branches represent *design alternatives* and the tree levels represent different *refinement levels*.

If a system specification S_j is characterized by a set of features P_j, a design is successful if a designed object DO_i with features P_i fulfills the specification S_j, i.e. $P_j \subseteq P_i$. A *refinement design step* (*synthesis step*) is characterized by a function φ with $\varphi(DO_{i1}) = DO_{i2}$ and $(P_{i1} \cap P_j) \subseteq (P_{i2} \cap P_j)$ (see also [1] and [15]). $(P_{i1} \cap P_j)$ and $(P_{i2} \cap P_j)$ are the sets of features of the system specification S_j which are fulfilled by the design object DO_{i1} and DO_{i2}, respectively. Of course, during a design there are other design steps, too, e.g. the correction of errors, but these are no synthesis steps.

We also say that two arbitrary design objects DO_{i1} and DO_{i2} are *alternatives with respect to a design object DO_j (or a specification S_j)* if $P_j \subseteq P_{i1}$ and $P_j \subseteq P_{i2}$. In figure 2, we see that DO_3 can be regarded as part of a specification for DO_4 and DO_6, and, therefore, DO_4 and DO_6 are alternatives with respect to DO_3.

Reuse of Specifications

Before starting a design, it is useful to ask: "Has someone tried a design with the same requirement specification before?". An answer to the question may help the designer to reuse experiences of other designers with the same specification. Most designs cannot fulfill their requirement specification in its entirety. So, the answer gives some information about the reasons why the previous design failed or it provides the "best possible" design for the given requirement specification. Specifications of ongoing designs may inform designers about the progress of similar works. This represents a simple but effective instrument to support reuse in a concurrent engineering environment and to avoid unnecessary parallel work. Therefore, we store all specifications together with the resulting design objects and the experiences of the designers in the database. (An adequate data model for this purpose can be found in [14]). There is no difference between design objects and specifications for the retrieval process because of the same representation by a set of features.

Similarities of Design Objects

Before we define the similarity of design objects and specifications, we define the similarity of two features of a feature class C by a function

$$sim_C: C \times C \rightarrow [0, 1]$$

which roughly expresses the expense of converting the first feature to the second feature. For example, $sim_C(p_i, p_j) = 1$ corresponds to equivalent features. The linguistics calls these synonyms. $sim_C(p_i, p_j) = 0$ means that the features are totally different. Note that this function is asymmetric because it could be easier to transform a feature p_i into a feature p_j than vice versa. For example, it is easier to

enlarge a cell by adding empty area than to shrink the cell.

A problem is the mapping from *qualitative*, i.e. nominal and ordinal features to *quantitative*, i.e. cardinal features. For nominal features, the function sim_C may yield 1 for synonyms and 0 otherwise. For ordinal features, the mapping may be given explicitly, e.g. in form of a matrix (with the value 1 on its diagonal) which represents a directed graph. For instance, for the qualitative feature class Function we roughly express that 'Function is Subtractor' is more similar to 'Function is Adder' than to 'Function is Multiplier'. For quantitative feature classes the return values of the function sim_C must be normalized to the range [0, 1].

Now, we define a *similarity function* SIM for design objects (and specifications) as

$$SIM(DO_i, DO_j) = \sum_{C \in \Gamma} r_C \cdot SIM_C(DO_i, DO_j).$$

Γ is the set of all feature classes which have instances in P_i. Each r_C is a positive real number. It represents a *relevance factor* (weight factor) which expresses the importance of the corresponding feature class for the comparison. The sum of all weight factors has to be less or equal than 1. In our implementation (section 3), we use default weight factors based on our current experiences. If a feature class of an instance in P_j has no corresponding instance in P_i, this feature class is not taken into account. We discuss the reason for this below.

The Goal

The goal is to find a design object DO_j (or several design objects) with SIM (S_i, DO_j) is maximal for a given requirement specification S_i. If P_i contains a feature which has no corresponding feature (i.e. a feature of the same feature class) at the design object, the feature is undefined. In this case, we assume that the function SIM_C returns 1/2. However, if the design object has a feature which has no counterpart at the specification this feature is not used, i.e. the corresponding feature class is not in Γ.

The reason for this is that we do not want to find similar objects in the sense of Tversky's contrast model (see section 1). Our goal is to find design objects which **fit** the current specification. Therefore, the set (B - A), i.e. the set of features of the design object which has no corresponding feature at the specification, will not be considered because they do not influence the fitness. On the other hand, the set (A - B) is considered with the value 1/2 for SIM_C (S_i, DO_j).

Multi-Functional Modules

If a feature class C is represented by only one instance in P_i and P_j, respectively, the function SIM_C is equivalent to the function sim_C. If there is more than one instance, as it is possible for the feature class Function (see above), we define the similarity between two sets of features of the same feature class as:

$$SIM_C(DO_i, DO_j) = \frac{\sum_{p_i \in C_i} max(\bigcup_{p_j \in C_j} sim_C(p_i, p_j))}{|C_i|}$$

with $C_i = \{p \mid p \in P_i \wedge p \in C\}$ and $C_j = \{p \mid p \in P_j \wedge p \in C\}$.

Example: DO_i has two features of the feature class Function: 'Function is Adder' and 'Function is Multiplier'. If another object DO_j has the feature 'Function is Adder' and $sim_{Function}$ ('Function is Multiplier', 'Function is Adder') is 0 then $SIM_{Function}(DO_i, DO_j) = 1/2$. On the other hand, if DO_i has the feature 'Function is Adder' and DO_j has the features 'Function is Adder' and 'Function is Multiplier' then $SIM_{Function}(DO_i, DO_j) = 1$.

Generic Features

We define a generic feature g as a feature that summarize other features. It defines a subset G in the feature class C. If P_i and P_j contain generic features g_i and g_j, and G_i and G_j are the sets of features represented by g_i and g_j, respectively, we use the formula above with

$$C_i' = (C_i - \{g_i\}) \cup G_i \text{ instead of } C_i$$

and

$$C_j' = (C_j - \{g_j\}) \cup G_j \text{ instead of } C_j,$$

respectively. In short, a generic feature g is replaced by the features of the corresponding subset G. If the cardinality of one of the sets G_i and G_j is infinite, the implementation has to guarantee that the calculation terminates. This can always be ensured for practical applications. We take the same formula if only one of the feature sets P_i and P_j contains a generic property. Then, one of the subsets G_i and G_j contains only one element. For example, if DO_i has the feature 'Arity is 4' and DO_j the generic feature 'Arity is Even' then $SIM_{ARITY}(DO_i, DO_j) = 1$.

Rules

One problem of the formulas above is the fact that each feature is regarded separately. Dependences between features in the form of "An n-bit adder can easily be constructed of an n/2-bit adder" cannot be expressed. Here, the transformation of the feature of the class Width from 'Width is N-Bit' to 'Width is N/2-Bit' depends on the feature 'Function is Adder'. The example "An multiplier can be constructed of an adder and a register" shows that the feature 'Function is Multiplier' can be realized by two other features: 'Function is Adder' and 'Function is Register'. To handle these cases, we propose two kinds of rules:

- rules which influence the relevance factors (*factor rules*) and
- rules which induce other search processes (*substitution rules*).

A rule consists of a left hand side (LHS) and a right hand side (RHS). The LHS contains a condition while the RHS represents an action which will be evaluated if the LHS condition is true. In the case of a factor rule, a rule is represented as

$$P \rightarrow C_1 : r_1, C_2 : r_2, ..., C_3 : r_n$$

with P is the condition in form of a feature set and r_k, $1 \leq k \leq n$, are the relevance factors which influence the feature classes C_k. If, at the start of a search, the condition is true, i.e. a specification has all features of P, the relevance factors are set to the relevance factors of the RHS. For example, we know that a multiplexer can easily be built by smaller multiplexers. We therefore decrease the relevance factors of the feature classes Arity and Width if we retrieve a design object with the feature 'Function is Multiplexer'.

Substitution rules are represented as follows:

$$P \rightarrow P_1 \,\&\&\, P_2 \,\&\&\, ... \,\&\&\, P_n.$$

The rule is also activated if the features P_i of the current specification S_i cover the feature set P. Then, additional search processes are started beside the current search process which must all be successful. The feature sets of the new specifications S_k are built by substituting the features P by the features P_k, i.e. $S_k = (S_i - P) \cup P_k$, $1 \leq k \leq n$. Now, we concurrently search with the old specification S_i and the new specifications S_k. (The sign '&&' symbolizes the parallelism.) The retrieval is successful if either the old search process terminates successfully or all processes caused by the RHS yield a result. Later, we give an application example of such a substitution rule.

Dimensions of the Design Space

The design space spans three dimensions. Firstly, design objects may be aggregates which are built of other design objects as submodules (*configuration hierarchy*). For example, an 8×4 multiplexer may consist of two 4×4 multiplexers and one 2×4 multiplexer (see figure 3.a). Both, the 4×4 multiplexers and the 2×4 multiplexer are primitives in the sense that they are no aggregates of other design objects but they are complex objects in the sense that they contain netlist or layout descriptions.

Secondly, the design objects may be part of a refinement tree (see figure 2) which represents the evolution of the design synthesis process (*design hierarchy*).

Thirdly, different objects may be similar in the sense that they possess common or similar features. We call this dimension *feature network*, because it characterizes the

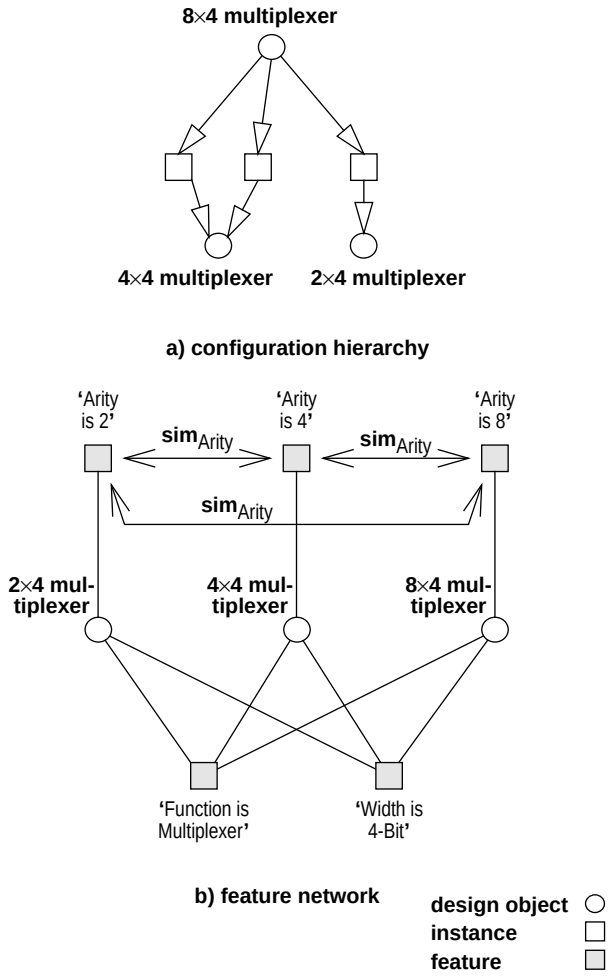

Figure 3: Design Space of a Multiplexer

similarities of design objects with regard to their features (see figure 3.b). Of course, these dimensions are not independent. For instance, objects with a common predecessor in the refinement tree have common features (see above) and so they are connected in the feature network by these common features. In this paper, we focus on the feature network. Our data model of the design hierarchy and the configuration hierarchy is described in [16].

Considering the Design Hierarchy

Let us return to the example of figure 2. Regarding the design process, we make two observations:

- The abstraction of the design decreases during the design process, i.e. the design becomes more and more precise. The degree of abstraction expressed by the refinement level can be regarded as a rough measure for the design progress.
- The expense of changing early decisions increases with each design step. For example, it is less complex to change an n-bit adder to an m-bit adder in the

domain behavior and to compute the layout of the m-bit adder than to change the adder in the domain layout (figure 4).

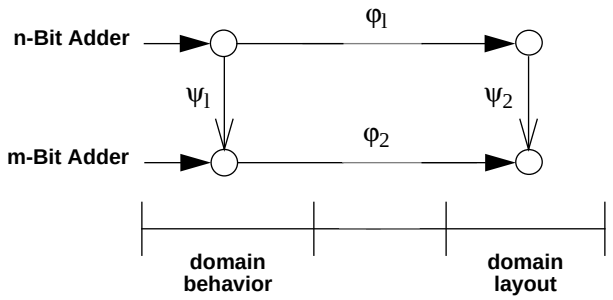

Figure 4: Example of a Change of an n-Bit Adder to an m-Bit Adder (Step ψ_1 followed by the refinement step φ_2 is easier to perform than the step ψ_2)

Due to the latter observation, we define the similarity function 'sim' of the ordinal feature class Refinement Level as follows:

Assume a refinement tree with levels 0 (root) to n (leafs). The feature p_k represents the refinement level k. The similarity function $sim_{RefinementLevel}$ (or short: sim_r) must meet three conditions:

(1) $sim_r(p_i,p_k) < sim_r(p_i,p_j), \quad 0 \leq i \leq j < k \leq n$
(2) $sim_r(p_k,p_i) < sim_r(p_k,p_j), \quad 0 \leq i \leq j < k \leq n$
(3) $sim_r(p_j,p_k) < sim_r(p_j,p_i), \quad 0 \leq i \leq j < k \leq n.$

Conditions (1) and (2) state that the similarity of two design objects becomes smaller with increasing distance in the refinement tree. For instance, the layout DO_5 in figure 2 is more similar to the floorplan DO_4 than to the netlist DO_2. We need two conditions to describe this aspect because the similarity function of a feature class is not symmetrical. Condition (3) reflects the second observation described above. For a given design object at the refinement level j, all objects at a smaller level i are more similar than the objects at a larger refinement level k.

The typical shape of the similarity function $sim_{RefinementLevel}$ is shown in figure 5. For the example of figure 2, the order of similarity to a specification S = {'Domain is Structure'} would be DO_2, DO_1, DO_3, (DO_4,DO_6), and DO_5.

Considering the Configuration Hierarchy

For the reuse process, three aspects of the configuration hierarchy can be exploited. Firstly, to determine the (re)use frequency of a module we can examine how often it is instantiated. Secondly, we can use features of the submodules to determine the function of the aggregate. Thirdly, we can examine which features of the feature class Function the submodules of the design objects have. This information can be used to share functions of the submodules to reduce the chip size or to search for fitting submodules when the search for a design object fails.

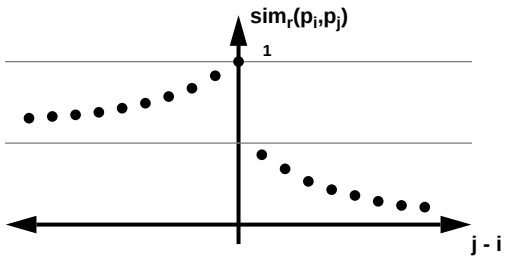

Figure 5: Similarity function $sim_{RefinementLevel}$ The feature p_j is more similar to p_i in the case that the refinement level j is smaller than i than for the case that j is larger than i. The similarity decreases with increasing distance of the refinement levels. The shape of the two partial functions may be any strongly monotone curves.

Example: Assume we have a specification S_i = {'Width is 16-Bit', 'Function is Multiplier'} and we know that a multiplier is composed of a register and an adder. The function 'Function is Multiplier' can be realized by the function 'Function is Adder' and the unit 'Function is Register'. So, we generate an additional substitution rule: {'Function is Multiplier'} → {'Function is Adder'} && {'Function is Register'}.

If we search for a multiplier we also search for an adder and a register. If we find both subcells, adder and register, an explanation component advises the designer to build the multiplier with these modules.

3. The Prototype System RODEO

The model described above is the basis of our prototype implementation **RODEO** (**RODEO** is an acronym for reuse of design objects). It is part of our VLSI CAD system PLAYOUT [18]. **RODEO** is implemented in C++. It works on an abstracted view of our design database. The data of this view are stored in main memory. In the implementation, the different feature classes are implemented in a C++ class hierarchy (see figure 6). Classes in the left subtree (qualitative features) have special methods which perform the casting of the qualitative feature instances to quantitative instances. So, the implementation is a mirror image of the feature classes described in section 2.

Retrieval Strategy

First, the user defines a requirement specification in form of mandatory and desirable features. He has the possibility to set relevance factors or to reduce the search space by defining threshold values for the similarity function SIM and the similarity functions SIM_C. Predefined threshold values and relevance factors have been deter-

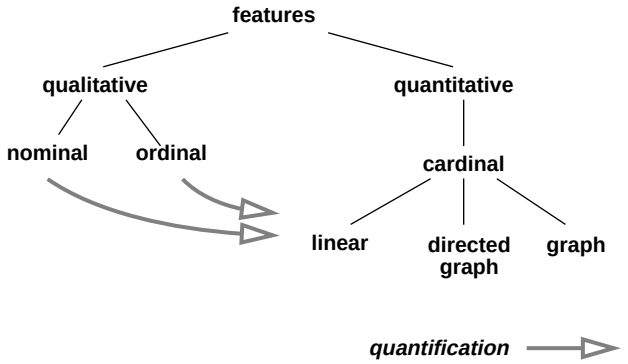

Figure 6: Section of the RODEO C++ Class Hierarchy

mined on the basis of our experiences. Before starting the retrieval, **RODEO** also examines which rules can be applied.

The retrieval returns a set of reuse candidates based on the mandatory features. It works similar to the general retrieval strategy of case-based reasoning [5]. To compute the search space, the features of each feature class are ordered with respect to their similarities to the features of the specification defined by SIM_C. Then, new specifications are composed step by step in decreasing order of the result of the similarity function SIM (e.g. see figure 7).

feature class	ordered features	start specification
C_1	(a, b, c, ...)	(a, A, α)
C_2	(A, B, C, ...)	
C_3	(α, β, γ, ...)	

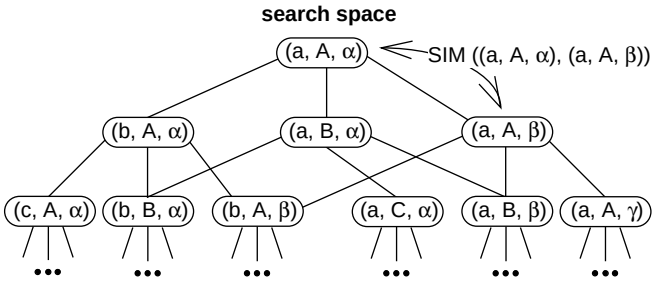

Figure 7: Search Space of a Specification

The search strategy of **RODEO** bases on an A*-algorithms [13]. From the features of the generated specification we can directly access the design objects. Since design objects, specifications, and substitution rules have the same representation, there is no difference from the viewpoint of the search process. They are therefore treated in the same manner. Several heuristics improve the run time behavior of the search algorithm. Currently, the relevance factors and substitution rules are defined manually. It is our future goal to determine these values and rules automatically. Because of the changing similarities by different relevance factors, we cannot manage the similarities by a grid file or something similar. After the search, an explanation component informs about the reasons for the differences between the original specification and the results. For example, it reports the use of factor and substitution rules.

4. Reuse Potential and Experimental Results

RODEO supports all three types of reuse mentioned in section 1. Reuse by instantiation is supported by retrieving objects which have all mandatory features of the specification. For reuse by adaptation, we can perform a nearest neighbor search or an interval search using tolerances of features. Reuse by parameterization is supported by every retrieval step since generic features are examined.

To assess the possible amount of reuse we must study our design activities over a long time. A quantitative analysis of these studies is very difficult because we are looking not only for objects which fit completely but also for similar objects. Statements about reuse of similar objects are only possible if we can compare the adaptation time with the corresponding design time of new objects. However, in real designs, only one of these two time values is available. We therefore examined the reuse by instantiation and the retrieval times quantitatively. For the amount of possible reuse we examined our design databases because they contain much more analysis data than we have with our design traces to date.

Only large databases can provide the necessary precondition for finding candidates for reuse. Table 1 shows the sizes of the PLAYOUT databases which we examined by **RODEO**. All design objects were computed before we used **RODEO** so that few objects are available twice.

database	I	II	III
no. of design objects	5625	4020	4243
no. of different features	2330	2152	2642

Table 1: PLAYOUT Databases

Currently, we use 12 feature classes: Function, Technology, Width, Arity, Area, Aspect Ratio, Refinement Level, ObjectType, Designer, Library, Phase, and SubCells which is more than looking for the function of a cell only. The size of the feature classes used by **RODEO** are on average as follows: Function 33, Technology 3, Width 29, Arity 13, Area 760, Aspect Ratio 756, Refinement Level 5, ObjectType 16, Designer 33, Library 17, Phase 4, and SubCells 103. A data-

base contains about 360 different cells but only 33 features of the class Function are available. For most cells we do not know the function. One reason is a necessary repartitioning step that changes the circuit hierarchy of the behavioral design to a new hierarchy that meets the requirements of the physical design phase. Since this step splits and combines cells, it is not always possible to specify the functions of the new cells. However, this shows the importance of exploiting the configuration hierarchy during the reuse process as described in section 2.

We examined the reuse potential by analyzing different feature classes. For instance, table 2 shows the layout alternatives of different cells. One cell even has 25 layout alternatives. If we further consider that the typical tolerance of a layout synthesis step is about 10%, the reuse potential is very high here.

no. of layout alternatives	2	3	4	5	6	7	8	11	13	14	17	20	25
no. of cells	6	2	9	3	1	2	3	1	1	2	3	1	1

Table 2: Available Layout Alternatives

Besides the possible degree of reuse, we examined the retrieval times of typical applications. The retrieval to one requirement specification depends on the number of features of the specification. Table 3 shows the average times needed to retrieve a design object.

The table shows that the retrieval times are small compared to the design times which may be minutes, hours, or even days. The gain in time increases with the number of the levels of the configuration hierarchy. This has two reasons: a) the design becomes larger and b) the number of iterations on higher levels is larger than on lower levels.

no. of features	1	2	3	4	5	6
first object (sec)	0.02	0.09	0.09	0.09	0.03	0.05

Table 3: Average Retrieval Times (on a HP730)

5. Conclusions

In this paper, we gave an overview of our reuse method and its relationships to the design process. We developed a feature-based reuse model to find similar design objects for a given requirement specification. The presented model considers the special purposes of an hierarchical and multi-domain design space which is present in most ECAD systems. We examined our model with the prototype implementation **RODEO**.

We see four directions for future works. Firstly, it is very important to study the reuse experiences of designers over a long time. Here, the reuse frequency of modules must be recorded. Secondly, we will try to handle complex features, e.g. features which describe pin assignment. Thirdly, we want automatically determine the relevance factors, the function sim_C, and the rules. To date, our rules are user-defined. And finally, we want to reuse not only design objects but also documentations, design experiences, and whole design processes in the sense of a comprehensive reuse. Our current reuse results are hopeful, but much more work has to be done before a comprehensive reuse, i.e. a reuse of all design information, is possible.

6. References

[1] J. Allen, “Performance-Directed Synthesis of VLSI Systems”, Proceedings of the IEEE, February 1990

[2] V.R. Basili, D.D. Rombach, “Support for Comprehensive Reuse”, IEEE Software Engineering Journal, Sept. 1991

[3] B. Becker, G. Hotz, R. Kolla, P. Molitor, “Hierarchical Design Based on a Calculus of Nets”, Proc. 24th Design Automation Conference, 1987

[4] T. J. Biggerstaff, A. J. Perlis (Ed.), “Software Reusability / Volume I / Concepts and Models”, ACM Press Frontier Series, 1989

[5] E. L. Rissland et. al., “Case-Based Reasoning“, Proc. Case-Based Reasoning Workshop (DARPA), 1989

[6] S. Y. Foo, Y. Takefuji, “Database and Cell-Selection Algorithms for VLSI Cell Libraries“, IEEE Computer, February 1990

[7] D. D. Gajski (Ed.), “Silicon Compilation”, Addison-Wesley, 1988

[8] E. Girczyc, S. Carlson, “Increasing Design Quality and Engineering Productivity through Design Reuse”, Proc. 30th Design Automation Conference, 1993

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

[10] J. L. Kolodner, “An Introduction to Case-Based Reasoning“, Artificial Intelligence Review, 6, 1992

[11] R. Prieto-Diaz, P. Freeman, “Classifying Software for Reusability”, IEEE Software Magazine, January 1987

[12] M. M. Richter, “Classification and Learning of Similarity Measures”, Studies in Classification, Data Analysis and Knowledge Organization, Springer, 1992

[13] E. Rich, K. Knight, “Artificial Intelligence”, McGraw Hill, 1991

[14] B. Schürmann, J. Altmeyer, M. Schütze, “On Modeling Top-Down VLSI Design”, Proc. Int. Conference of Computer Aided Design, San Jose, California, 1994

[15] E. Siepmann, “Entwurfstheorie und Entwurfsdatenmodellierung fuer CAD-Frameworks“, Ph.D. Dissertation, University of Kaiserslautern, 1991, in German

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

[17] A. Tversky, “Features of Similarity”, Psychological Review 84, 1977

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