Trait: An Attribute Management System for VLSI Design Objects

Tzi-cker Chiueh, Randy Katz

Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720

Abstract

There are two aspects of engineering design data: internal data representation and abstract attributes. Most previous design database systems focused on the representational issues of design objects. This paper presents an attribute system called *Trait* that manages the abstract aspects of engineering design. We introduce a *weakly-typed* data model that supports both *object typing* and *arbitrary attribute attachment*. It features an active-value mechanism that performs demand-driven attribute computation, which is useful when attributes are measurement of certain aspects of design objects. Furthermore, to avoid unnecessary attribute recomputation, an attribute management subsystem is developed that takes into account the semantics of tool execution and automatically maintains the consistency between the abstract and the concrete aspects of design data. Finally an index subsystem is built into *Trait* to support associative accesses to design objects. We briefly described two applications that exploit the features of *Trait*.

1. Introduction

Engineering design objects typically have both an internal data representation and a set of attributes that either abstract particular functional characteristics or contain the administrative status of the objects. As shown in Figure 1, a VLSI layout object contains a complex data structure that represents the physical masks of the underlying circuit, as well as attributes such as area, power consumption, and etc. We call the data representation the *concrete content* and the set of attributes the *abstraction set*. Accordingly, an engineering design database should support this partitioning of design data for the following reasons. Most CAD tools manipulate the concrete contents of design objects. Users and high-level design management systems, on the other hand, generally are only concerned with the abstractions. By separating the management of the concrete and abstract aspects of design data, ordinary CAD tools and design management tools can work on their own domain without interfering with each other. Moreover, because the internal representation of design objects typically have much more complex data structures than the abstract counterpart, it is also advantageous to have distinct management subsystems optimized for each. Most previous VLSI design databases [SING89] [SIEPS89] [WOLF88] did not make explicit distinctions between the concrete and abstract sides of design data. As a result, these systems seem to be biased towards the management of *concrete* data and lack a flexible mechanism to associate design objects with attributes. *Cactis*, [HUKI88] is to our knowledge the only design database that provides significant attribute management. In particular, *Cactis* features an interesting attribute update mechanism that is based on attribute graphs. In this paper, an attribute system called *Trait* is developed that stores, manipulates, and accesses the abstractions of design objects, and maintains the consistency between the abstract and concrete aspects of design data.

We start the design of *Trait* with a type-based object-oriented data model because it has been generally recognized [BATO85] as a nice match to the semantics of engineering design objects. The central theme of a type-based object-oriented system is the notion of *type*. All objects that share similar static (property) and dynamic (method) behaviors form a type (or class). The notion of type serves as the basis for two fundamental mechanisms in object-oriented systems: *encapsulation* and *inheritance*. However, because type-based object-orientation requires every object to belong to a certain type,

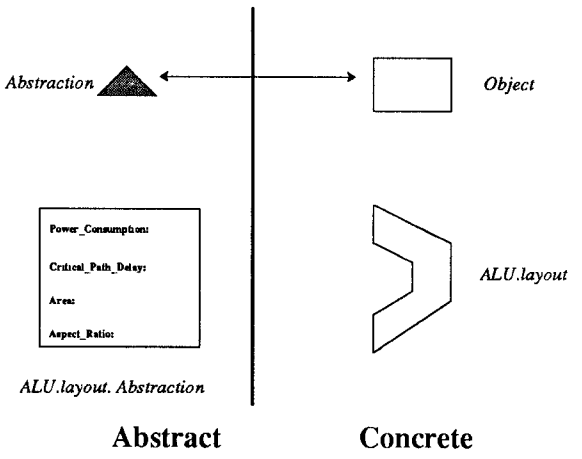

Figure 1 Dual Aspects of Engineering Design Data

the only way to accommodate objects with a different structure or behavior is by creating new types or by refining existing types. In engineering design, it is usually difficult to impose a type system on the design objects *a priori* because the structures of attributes (the object descriptions) become refined over the course of the design process. A fundamental issue in engineering design data modeling thus is to provide a conceptually simple and practically easy-to-use *dynamic modeling* mechanism: the ability to model dynamically evolving objects at run-time. Consequently pure type-based object-oriented database models are abandoned because they are considered too restrictive to be useful in the context of engineering design.

In the object-orientation literature, the concept of *schema evolution* [SKAR86][BANE87] [KIM88] has been proposed to solve the dynamic modeling problem. Its central idea is to provide a version mechanism for the type specifications. In other words, there could be more than one definition associated with each type. To fit schema evolution into the type system, complicated mechanisms are invented to smooth out the interaction between alternative versions of types. We considered these works unlikely to be successful because end users tend to be intimidated by the complicated side effects involved in changing the data schema. It appears that schema evolution is invented simply to circumvent the rigidness of the type system.

On the other hand, prototype-based object-oriented systems [LIEB86][LALO86] replace the notion of type with the notion of *prototype* and thus avoid the rigidness of type-based systems altogether. In this case, every object can act as a prototype, from which new objects with minor schema changes can be created. Prototype inheritance, together with *delegation* offers a powerful paradigm for modeling dynamically evolving objects. However, the performance of prototype-based systems degrade correspondingly because there is no clustering of similarly-behaved objects. Consequently, there is no sharing of meta-information such as data schema. Taking into account the modeling power and the database performance, a hybrid approach seems to be the logical choice. *Trait* is based on a hybrid model which we call the *weakly-typed* model.

This paper is divided into six sections as follows. Section 2 outlines the functional requirements of an attribute system. The model of *Trait* is introduced in section 3. The underlying implementation issues and applications of *Trait* are discussed in section 4 and 5. We conclude this paper with a discussion of possible extensions in section 6.

2. Functional Requirements

The most important requirement of an attribute system is that it allows attachments of arbitrary attributes to design objects. This capability offers the maximum dynamic modeling flexibility in that it allows designers to create arbitrary abstractions out of a design as it evolves. In our opinion, there shouldn't be artificial restrictions on the constituents of the abstraction sets. Using this flexibility, users and design management tools can choose to deal with any intended levels of abstraction by *dynamically* creating arbitrary attributes. For example, one design tool may need to know the delay of a design object while another needs to know the exact critical path. By allowing arbitrary attribute attachment, the abstraction can be refined *dynamically* to accommodate the needs of the tools that manipulate the objects. This flexibility is really what distinguishes a general attribute system from a type-based object-oriented database system.

Conceptually an abstraction set is just a set of attribute name-value pairs. Without some kind of schema information, it is difficult to store and manipulate the attribute sets as efficiently as records in relational database system, or even objects in type-based object-oriented databases. Therefore an attribute management system should aim to minimize the performance penalty entailed

by the generality of the first requirement.

One of the uses of an attribute management system is to provide an attribute-based naming scheme, in addition to the object identifiers. That is, users can specify an object in terms of its attribute values, thus performing associative accesses to design data. It is therefore important to have an effective indexing mechanism that can be used to step through the objects that share the same subset of attributes. Furthermore, this indexing scheme should be able to accommodate the hierarchical data organization that typically exist in an engineering design environment.

The attribute system should be implemented on the same physical database model as the *concrete* design object system so that a uniform treatment of both kinds of data is possible in a single application. *Trait* is implemented under the OCT data model[HARR86], a design database developed in U.C. Berkeley. We will discuss the data model of OCT and our implementation under OCT in section 4.

3. The Architecture of *Trait*

The attribute system *Trait* is based on a *weakly-typed* model, which supports both a type system and an attribute attachment facility. The type system is used to model the "invariant" abstract behavior of the design objects such as the generic properties of a particular design representation (e.g. a physical mask). The attribute attachment facility, on the other hand, allows users or applications to create arbitrary abstractions by manipulating attributes. An important result of this model is that those which access only typed attributes do not suffer a performance penalty because schema information is available. However, applications that enjoy the power of the dynamic modeling capability would pay the price of performance degradation.

3.1. The Structure of Attributes

Although our original goal is to avoid the rigidness of the standard type model, we believe that the concepts of type and arbitrary attribute attachments are not necessarily incompatible. Unlike the case with programming languages, the notion of type in a database model should be thought of as a default template, from which an object can start to flesh out other details. In our model, whenever an object is created, a set of default attributes are assigned to the object according to its type. Subsequently any number of attributes can be attached to it. From the user's point of view, type templates serve as a short hand for building attributes. From the system's standpoint, the implementation for the default attributes can be optimized in terms of speed and space. As a result, users of *Trait* only have to pay the performance penalty for those user-added attributes.

Trait provides a type definition facility for users to build up the type systems for different databases in different application domains. In VLSI, the notion of type corresponds to the data representation at various levels of abstraction. Within a data representation domain, there may be various data formats, tailored to the manipulation of specific CAD tools. For example, a typical VLSI design database may have three types: *textual, logic,* and *layout.* Associated with each type is a set of generic attributes that form an "invariant", "minimal" abstraction of an object of that type. For example, the generic attribute set of a *layout* object may include power consumption, area, speed and so on. All objects of the same representation type, even with different data formats, share the same set of generic attributes.

Every time a design object is created, *Trait* attaches to it the generic attribute set of its representation type. If the object is generated by a CAD tool, the attributes will be attached automatically by *Trait.* If the object is created manually (for example through a text editor), the users need to explicitly declare its representation type. As a side-effect of this declaration appropriate attributes are attached. To add additional attributes, users simply use

attach object, attribute, attribute_value[,tool]

to specify the object name, the attribute name-value pair, and possibly the name of the measurement tool, if the attribute value is unspecified. *Trait* will ensure that the attribute be attached to the designated object.

The value of an attribute can be in **active** or **passive** mode. Passive value means that the current attribute value is *valid.* For example, the values of administrative attributes such as *owner* and *time-of-creation* are typically passive. Active values, on the other hand, require explicit measurements: The value of an attribute has to be calculated explicitly by invoking a measurement tool. The mode of an attribute value is active only if the attribute needs measurement and the value

itself is either outdated or uninitialized. After the value is computed and cached, the attribute value changes from active to passive. Conceptually, each attribute has an attribute name, an attribute value, a mode bit, and a measurement tool name. Some of the fields may be unapplicable to some attributes.

This ability of specifying active values is particularly important in VLSI application because the measurements typically take substantial amounts of time. With the active value mechanism, attribute measurement can be delayed until the value of the attribute is actually needed. Whenever an attribute of an object is accessed and its value is in active mode, *Trait* will invoke the measurement tool transparently and cache the returned value, after which the value becomes passive. In other words, the attribute value is computed *lazily*. Consequently the attachment of the generic attribute set upon object creation is asynchronous in that actual computation of active attribute values is decoupled from the attachment operation. Furthermore, the values of those attributes that never get accessed never need to be computed.

Active value can be viewed as a restricted form of the *trigger* mechanism. The triggering condition in this case is attribute access. The attribute model also implements the essential idea of *access-oriented programming* [STEF86] by invoking a certain computation transparently as a result of attribute access. Alternatively, an active value can also be regarded as a view of the original design object because it displays a certain property of the object through some computation process. Just as in the view update problem, the abstraction set of an object has to be updated accordingly whenever the underlying object is modified. We will discuss this issue in the next subsection.

3.2. Attribute Management

In this section we discuss how attributes and their values react to tool executions. There are two cases to consider: *attribute consistency maintenance* and *attribute propagation*. Because part of an object's abstraction set is obtained through measurement, it is essential to re-measure some of the attributes when the underlying object is modified. Attribute consistency maintenance ensures the consistency between the concrete content of a design object and its abstraction set.

If an object is derived from another object through a tool execution, it is natural for the new object to share some of the attributes (and their values if possible) of the original object if they are regarded as "similar" in some sense. By such sharing, unnecessary measurement computation can be avoided. The question is *which* attributes (and their values) can be preserved across a tool execution. Attribute propagation manages the propagation of attributes and their values across tool executions by taking into account the tool execution semantics.

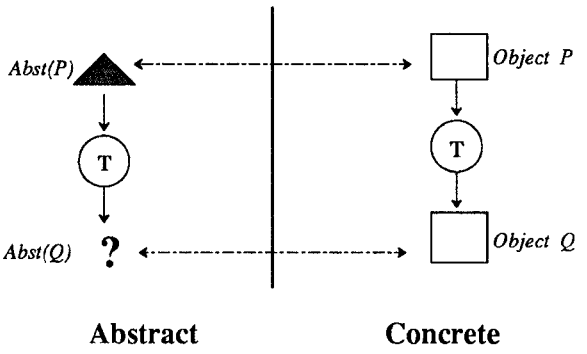

Figure 2 Propagating Attributes Through Tools

Because a VLSI design process is characterized by ubiquitous use of Computer-Aided-Design (CAD) tools, these two issues reduce to the same question: how do tools affect the values of attributes? As shown in Figure 2, object Q is obtained by applying tool T to object P. The question is what attribute values of object P can be propagated to object Q. The difference between attribute consistency and attribute propagation lies in whether tools modify the input objects in-place (attribute consistency) or produces new outputs (attribute propagation). Suppose one can somehow capture the tool execution semantics. Then it is possible for the attribute management system to determine how to modify or propagate attributes accordingly. However, certain tools are more difficult to characterize than others. A layout editor, for example, could potentially modify arbitrary portions of a layout object. Without user's hints, it is necessary to update every measurable attribute just for safety sake. A layout compaction tool on the other hand will presumably affect only area-related attributes and leave others intact.

In *Trait*, the semantics of CAD tools are defined in terms of the impacts of the tools on the

attributes of the input/output objects. The tool semantics impact specifications are part of the definition of the design environment configuration. The tools are generally classified into two distinct categories: *synthesis* and *analysis*. Analysis tools extract certain information from a design object. The outputs of analysis tools are NOT regarded as design objects. For example, a timing simulator extracts timing characteristics of design objects, but timing information itself is not a design object as far as *Trait* is concerned. Analysis tools also include verification and testing tools. The attribute management system simply ignores a tool execution when the tool in question is an analysis tool.

Synthesis tools are those that operate on design objects and produce design objects. According to the representation domains of the input/output objects, synthesis tools can be further classified into *same-domain* and *cross-domain* transformation tools. A logic minimizer takes a logic description and generates a minimized version of it, and therefore is a same-domain transformation tool. A module generator takes a logic gate description and produces a piece of layout, and thus is a cross-domain transformation tool.

For cross-domain transformation tools, with inputs and outputs in different domains, *Trait* simply attaches a generic attribute set to the output objects according to their representation type. Associated with each same-domain transformation tool are two lists of attributes: MODIFIED and ADDED . The MODIFIED list contains all the attributes that may be affected by the execution of this tool; the ADDED list contains the list of new attributes that can be added as a result of the tool execution. For each same-domain synthesis tool execution, *Trait* consults with these lists to determine the abstraction set of the output object and their values. For those attributes that are to be modified or added, their values become active. For example, a Place_I/O_Pads tool would affect the I/O interface behavior but not the internal characteristics such as the critical path. Thus only those I/O interface attributes are in the MODIFIED list of Place_I/O_Pads. To reason about the evolution of attributes and their values across tool execution, *Trait* must interact with the design process manager, which controls the execution of tools. The overall algorithm is summarized in the Figure 3.

IF T is an ANALYSIS
THEN ignore; exit
ELSEIF T is a CROSS-DOMAIN SYNTHESIS
THEN attach the generic attribute set
ELSE copy abstraction from input to output
invalidate attributes in MODIFIED(T)
add attributes in ADDED(T)

Figure 3 Attribute Management Algorithm

The attributes discussed above are those in the generic attribute set associated with a representation type. Because *Trait* allows arbitrary attribute attachments, there are attributes, other than the generic attribute set, called *user-added* attributes. *Trait* does not reason about the behavior of user-added attributes. For cross-domain transformation tools, *Trait* simply ignores the user-added attributes. These are typically specific to a representation domain and usually not applicable to another domain. For same-domain transformation tools, *Trait* implements the *prototype inheritance* protocol by propagating all the user-added attributes of the input object to the output object. However, it is the user's responsibility to decide whether the values of user-added attributes can propagate through. This interactive control mechanism is also useful in overriding the default attribute-propagation semantics of individual CAD tools.

3.3. The Indexing Scheme

In an engineering design database, users and applications interact with objects, which are design units that have distinct names. On top of this we add an attribute system, which represents design objects in terms of abstractions --- a set of attributes, the abstraction set. Suppose all design objects have the same set of attributes. Then each abstraction set can be viewed as a tuple in the sense of relational data model. All the abstraction sets thus form a big relation (or table). Similar things can be said when the design objects are arranged in an object-oriented type hierarchy. Here all the instances of a given type correspond to a relation. Furthermore, an instance may belong to more than one relation because an object of one type is also an instance of its supertype. Conventional indexing techniques can be applied to the above two models. Unfortunately this is not the case for our model.

In *Trait,* every design object has an abstraction set, which is a union of the generic attribute set and user-added attribute set. The contents of the latter are arbitrary. Therefore it is impossible to treat the set of all abstraction sets as records in a table. Moreover, there is another dimension of complexity because the space of design objects is organized as a collection of partitions. As shown in Figure 4, the design data space is partitioned into three distinct portions: *archive, group,* and *private.* The archive workspace stores stable design objects that are never modified; the group workspace is a middle ground for members of a design project to put together their designs; the private workspace is where a designer actually carries out his work. Designers usually cannot access objects in aother designer's private workspace. The design objects can be moved among these partitions by explicit checkin/ checkout operations. Within a private workspace, the objects are structured as a set of activity workspaces [CHIU90], which is a further refined unit for collecting related data. For example, the design objects related to an ALU module, a shifter, and a decimal arithmetic unit can be organized as three separate activity workspaces, which collectively form the designer's private workspace.

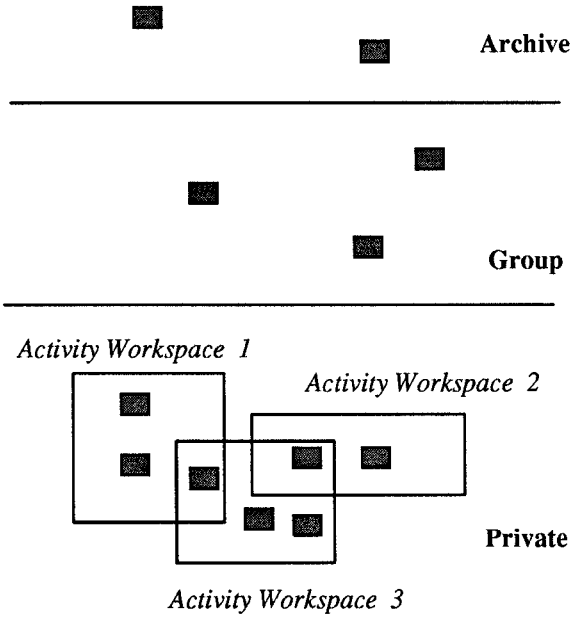

Figure 4 Partitioning of Design Data Space

There are two issues related to indexing: the scope of an index and the organization of the indices. In the relational data model, the scope of an index is a relation: an index provides a fast access path to the entries in a relation along the indexed attribute. Depending on the access characteristics, the organization of the indices vary. In *Trait,* the indexing scope is chosen to be the set of objects in an activity workspace for the following reasons. The design process model of our system [CHIU90] forces users to issue CAD tool invocations with respect to a specific activity. As a result, all objects are created or deleted with respect to a specific activity workspace. In view of the inherent locality, the index scheme should exploit the data partitioning to speed up access. On the other hand, the index scheme should also observe the access protection boundary imposed by the data partitioning structure. By providing indexing at the activity-workspace level, the management of each individual index can be separated, thus allowing multiple concurrently evolving activities and reducing cross-activity interference. Indexing at the activity-workspace level also has the advantages offered by *partial indexing* schemes as discussed in [STON89]. As for the index organization, *Trait* chooses to use standard B-tree structures.

Because each design object has a distinct name, object access queries can be classified into *nameless* and *named* accesses. A query such as "find all layout objects with area greater than X" is a nameless query. The query "find the versions of ALU that has a delay less than Y" is a named access because the name of the module (ALU) is listed in the query. In our data model, all versions of a design entity share the same object name. As shown in Figure 5, the abstraction of a design object actually consists of three parts: an administrative , a generic and a user-added attribute set. All objects in the database share the same set of administrative attributes; objects of the same type share the same set of generic attributes. User-added attributes are instance-specific. If the indexed attribute is a user-added attribute, the objects that are included for indexing are those that have that attribute. If the indexed attribute is one of the generic attributes of a particular representation, then all the objects of that representation type are included in indexing. If the indexed attribute is an administrative attribute, e.g., time-of-modification, then all the objects in the activity workspace are included. By default, *Trait* provides two indices which are based on object-name and time-of-modification attributes, both of which are administrative attributes. These index files called **non-local** indices

because they are separated from the indexed objects and are used to handle nameless queries.

For named queries, our system provides a separate navigation facility based on the notion of *link*. These links are called **local** indices because they are built into the abstraction set of the design objects. For each generic attribute that can define a total ordering on its values, there may be a corresponding link pointing to the next and previous element in the order defined by the attribute AND with the same object name. This linking mechanism is used to traverse the versions of a design entity. As a result, versions of a design entity can be organized along useful "merit-of-figure" attributes such as performance and cost, as opposed to a fixed derivation relationship in most previous version models such as [KATZ86]. A hypertext-like graphical interface can be used to explore the versions along different dimensions. Because the number of versions of an entity is relatively small, users can use this traversing scheme without suffering the "lost in hyperspace" syndrome found in large-scale hypertext systems.

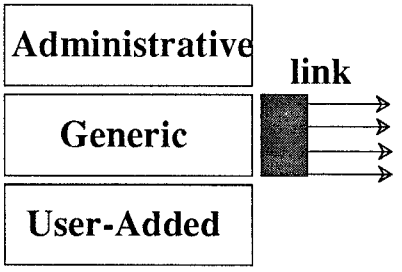

Figure 5 Structure of An Object's Abstraction

Because the indexed attribute value must be known for a newly created object to be included in the index, it has to be computed (assume it is necessary to compute) immediately upon the object creation. The index manager takes care of the attribute computation and pushes this computation to the background if that computation does not require user interaction. Consequently object creation is still asynchronous from user's point of view. In this case, the index manager is involved in computing the value of the attribute and putting the associated object in the appropriate position of the corresponding index.

Because the indexing scope is at the activity-workspace level, there are cases in which more than one index is involved since the objects of interest spread over multiple activity workspaces. The need for merging several non-local partial indices into one dictates the data structure of the individual index file. For example, a B-tree organization would be easier to merge than a hash index. For local indices (links), the merge process is dynamically performed and is similar to the merging phase of merge-sort.

4. The Physical Data Model

In OCT, a generic design entity is represented by a *cell* . The various representations of the object are represented as different *views* of the cell. For each view of the object, there is a *content facet* that represents the concrete contents of the view of the cell. There could be other facets that represent various "aspects" of a view. A facet can have different *versions*. Therefore, the OCT name space of the objects is hierarchical and consists of the names of cell, view, facet and version. For example, ADDER.PHYSICAL.INTERFACE.2 represents a design object which is the second version of the interface facet of the physical view of the cell ADDER. The single way to associate related objects is through the **attachment** mechanism. An object (called the *content* of the attachment) can be attached to another object (called the *container* of the attachment). OCT provides procedural interfaces that allow dynamic creation and deletion of attachments. Furthermore, OCT allows navigating from one object to another through an attachment relationship in either direction (attaching and attached).

Take a design cell ALU layout for example. The concrete part of it is represented as the **content** facet of the *physical view* of a cell called ALU. The abstract part of a design object is implemented as a **abstraction** facet of the same view of the same cell as the design object. OCT also provides a *bag* entity, which purely serves as a place-holder. In *Trait* the administrative, the generic, and the user-added attribute sets are clustered under different bags. The global picture of the attribute implementation is shown in Figure 6.

Every object with a unique name has the administrative attribute set. Every design object has both generic and administrative attribute sets. User-added attributes can be added to every object in the database. We separate the implementation of these three kinds of attributes to exploit their different nature. User-added attri-

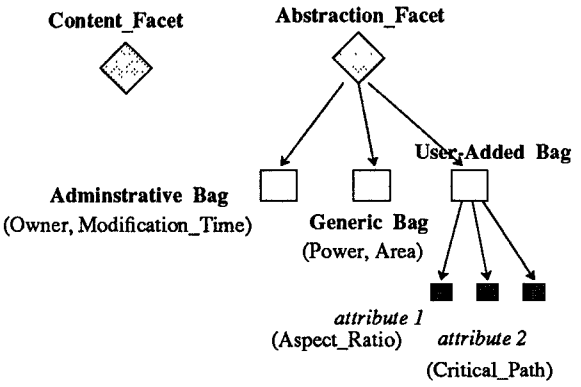

Figure 6 Implementation of An Attribute Set

butes are the most expensive since associated with each user-added attribute are attribute name, attribute data type, attribute value, measurement tool name (if applicable) and a status bit to indicate whether that attribute is currently active or passive. For generic attributes, only the attribute value and a status bit need to be maintained. For the administrative attributes, only attribute value is kept. Furthermore, the access to generic and administrative attributes does not need associative comparison with the attribute names. The meta information (type specifications for generic attributes or schema for administrative attributes) kept by *Trait* can be shared by all objects with the same abstraction. Consequently user-added attributes are expensive both in terms of storage and speed. But this is exactly the price that one paid for the dynamic modeling power of the weakly-typed data model.

5. Applications of *Trait*

5.1. Attribute-based Naming

In a conventional file system, the only way to specify a file is through the unique name of the file, possibly through some kind of name-completion mechanism. In a (relational) database system, users can access records associatively, i.e., through some combinations of the contents of the fields of the records. Attribute-based naming is an associative access scheme applied at the object level.

Attribute-based naming is particularly useful when the unique names of the objects are not available or when the needed objects share certain properties. For example, suppose a designer wants to access something he worked on last night but could not figure out which one it is. He can simply use the time index to access that file. Another example: suppose a designer wishes to access all ALU objects with power consumption less than a certain threshold. He can access these objects at once through the power-attribute index.

The attribute-based naming resolution system is designed on top of the attribute management system. Instead of being a full-blown query language, users interact with the naming system through a fill-in-the-form graphical interface. The language syntax is shown in Figure 7. The IN clause specifies the scope to which the query is applied. The default is the current activity workspace. Users can specify more than one workspace, or even large-granularity workspaces such as **Private, Group,** or **Archive.** The NAME clause can be used to specify the partial name string of the objects of interest. This facility is useful when users want to further restrict the access scope. The WHERE clause specifies the conditions on the attributes which must be satisfied. The language also supports powerful predicates such as "the biggest" and "the smallest." All the conditions in the WHERE clause are ANDed to get the aggregate condition.

FIND
IN *Activity Workspace {Archive, Group, Private}*
NAME *Partial Object Name*
WHERE *Attribute* **OP** *Value*
[Attribute is {Smallest, Biggest}]

Figure 7 Syntax of Attribute-based Naming Query

5.2. Intelligent Design Process Management

We have proposed a design process model [CHIU90], in which routine tool execution sequences can be aggregated into what we have called *tasks.* Tasks encapsulate both the details of individual CAD tool execution (e.g. parameter setting) and the interaction among tools. As a result, circuit designers are completely shielded from the peculiarities of the individual tools and their interrelationships. In other words, designers only need to interact with high-level tasks and the system will either automate the underlying tool execution or lead the designers through the execution sequences. The next step beyond this process model is to provide a more intelligent control of tool execution based on the status of the

design.

There are several scenarios in which this kind of conditional design flow control is particularly useful. First, with the emerging trend of CAD tool specialization, it is important to take advantage of the tool capability as much as *and* as transparently to the users as possible. Based on the attributes of a design object, the process management system can determine the most appropriate tool (or the same tool with the most appropriate parameter setting) to apply. For example, the logic minimizer MIS can optimize either performance or area metrics. Now suppose a design object has four attributes: PERFORMANCE, AREA, PERFORMANCE_GOAL, AREA_GOAL. Further assume that the performance criterion is already satisfied (PERFORMANCE < PERFORMANCE_GOAL), then the process manager will choose to optimize the area criterion by directing MIS with appropriate parameter setting. In this scenario, the AREA and PERFORMANCE attributes are automatically updated to reflect the current status of the object.

In many cases, designers are refining the design by iteratively applying the same tool sequence until a certain goal is achieved. The design process manager can take over such an iterative process by comparing the performance attribute with the desired goal after each step, and stopping when the goal is satisfied. For example, a compaction tool based on iterated relaxation may need a series of trials to attain the intended goal. With the goal and the current status attached to an object, the process manager can control the progress of iterative improvement by comparing the status against the goal.

The intelligence mentioned above is procedurally coded as control statements in a task description language, which is previously developed to encapsulate pre-defined tool execution sequences. The task manager that interprets the task specifications interacts with the attribute system to examine the status of the design and direct the flow of tool execution based on that information.

By separating attribute management from tool sequencing control, the system architecture becomes more modular. Moreover, the services provided by the attribute system can be shared by the design process manager with other applications.

6. Conclusion

Engineering design objects consist of two components: the concrete contents and the abstract properties. Most existing design database systems do not make explicit distinctions between the two. In this paper we have developed an attribute system *Trait* that is dedicated to the management of the abstract aspect of design objects. By adopting a weakly-typed data model, *Trait* supports both object typing and arbitrary attribute attachment. The latter is particularly useful in modeling dynamically evolving objects that often arise in engineering design. *Trait* features an active-value mechanism that performs lazy attribute value computation. This is an important feature because measurement of VLSI design attributes typically takes a substantial amount of time. Furthermore, to avoid attribute recomputation, an attribute management subsystem takes into account the semantics of the tool executions and automatically manages the propagation and consistency of the attributes and their values. Finally an index subsystem is built into *Trait* to facilitate the accesses to the associated objects. We briefly described two applications that are constructed on top of *Trait*.

One possible direction of future work includes extending the notion of active value to a more general concept called "procedural attachment". Instead of just computing the values of attributes, it is sometimes desirable to achieve certain side effects as a result of attribute accesses. Procedural attachment mechanism can be viewed as an extension of the attribute attachment facility, in the sense that both static and dynamic behaviors can be built into the design objects in an incremental fashion. The transparent creation of side effects can be used in VLSI design to automate routine trial and error process, background checking and clean-up, thus freeing circuit designers from tedious routine tasks. They are thus freed to focus their energy on more important design issues. The other future direction is to incorporate a constraint system that can automatically maintain the declared relationships among intra-object or inter-object attributes. An automatic constraint validation system provides a basis for maintaining the consistency among design submodules and is an important component for supporting cooperative team work.

REFERENCE

[BANE87] Banerjee,I., Kim,W., Kim,H-J, Korth,H.F., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," Proc. 1987 SIGMOD Conference, San Francisco, 1987.

[BATO85] Batory,D.S., Kim,W. "Modeling Concepts for VLSI CAD Objects", ACM TODS, vol 10, no.5, Sep. 1985, pp 322-346.

[CARE88] Carey,M., et al, "A Data Model and Query Language for EXODUS," Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago,Ill., June 1988.

[CHIU90] Chiueh,T.C., Katz,R.H., King,V. "A History Model for Managing VLSI Design Process," Proceedings of 1990 International Computer Conference on Computer Aided Design, Santa Clara, November 1990.

[HARR86] Harrison,D., Moore,P., Spickelmier,R., Newton,R., "Data Management and graphics editing in the Berkeley design environment", 1986 International Computer Conference on Computer Aided Design, Santa Clara, pp 20-24.

[HUKI88] Hudson,S.E., King,R., "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System", TR 88-20, Department of Computer Science, University of Arizona.

[KATZ86] Katz, R.H., Chang,E., Bhateja,R., "Version Modeling Concepts for Computer-Aided Design Databases," Proc. 1986 SIGMOD Conference, Wahington,DC, 1986.

[KIM88] Kim, W., Chou, H-T, "Versions of Schema for Object-Oriented Databases," 14th VLDB Conference, Los Angeles, CA 1988, p 148-159.

[LALO86] Lalonde,W.R., Thomas,D.A., Pugh,J.R., "An Examplar Based Smalltalk", 1986 Object-oriented Programming Systems, Languages and Applications Conference, Oregon, pp 322-330.

[LIEB86] Lieberman,H., "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", 1986 Object-oriented Programming Systems, Languages and Applications Conference, Oregon. pp 214-223

[SIEP89] Siepmann,E., Zimmermann,G. "An Object-Oriented Database Model for the VLSI Design System PLAYOUT", Proc. 26th. ACM/IEEE Design Automation Conference, Las Vegas 1989, pp 814-817.

[SING89] Singhal,A., Parikh,N., Dutt,D., Lo,C.Y. "A Data Model and Architecture for VLSI/CAD Database", 1989 International Computer Conference on Computer Aided Design, Santa Clara. pp 276-279.

[SKAR86] Skarra,A., Zdonik,S.B. "The Management of Changing Types in an Object-Oriented Database", Proc. 1986 Object-oriented Programming Systems, Languages and Applications Conference, Oregon. pp 483-495.

[STEF86] Stefik,M., Bobrow,D.G., Kahn,K.M. "Integrating Access-Oriented Programming into a Multiparadigm Environment", IEEE Software, vol. 3, no. 1, Jan., 1986, pp 10-18.

[STON89] Stonebraker,M. "The Case for Partial Indexes", Memo no. UCB/ERL M89/17, Electronics Research Laboratory, University of California, Berkeley, CA 94720.

[STON86] Stonebraker,M., Row,L., "The Design of Postgres", Proceedings of International Conference on the Management of Data, May, 1986, pp 340-355.

[WOLF88] van der Wolf, P. van Leuken, T.G.R., "Object Type Oriented Data Modeling for VLSI Data Management", Proc. 25th ACM/IEEE Design Automation Conference, Anaheim 1988, pp 351-356.