An Object Model and Algebra for the Implicit Unfolding of Hierarchical Structures[0]

Matthew C. Jones and Elke A. Rundensteiner
Software Systems Research Lab
University of Michigan, Ann Arbor
{mjones,rundenst}@eecs.umich.edu

Abstract

Design applications typically require radically varied views of shared design data, ranging from highly compact hierarchical design artifacts to flat and unfolded structures. Because current OODB view technology is not capable of providing this requisite variety of representations, we address this problem by providing special-purpose support for manipulating and querying hierarchical structures. Our object model lays the foundation for the implementation of meta-classes required to support hierarchical set operations. For example, the algebra defined over the data model allows the implicit unfolding of hierarchical sets, providing users with a (virtual) flattened and unfolded view of shared data without the penalty of having to maintain replicated data. It thus forms the basis for powerful extensions to the view capabilities of the OODB view system MultiView. Our query operators can be used to derive unmaterialized, unfolded, and updatable views from folded hierarchical sets. These views are updatable using two types of update operations, namely, in-context and out-of-context updates. For this purpose, we present an algorithm to optimally perform in-context updates on an unfolded view via selective unfolding. In order to evaluate the advantages and limitations of interoperating through such complex hierarchical and flattened views of design data, we also present empirical performance results comparing queries on implicitly and explicitly unfolded hierarchical sets.

Keywords: Hierarchical Data, Hierarchical Object-Oriented Views, Data Transformations, Hierarchical Set Algebra, Folded Design Data.

0. This work is supported in part by the NSF RIA grant #IRI-9309076, NSF NYI grant #IRI-94-57609, and the University of Michigan Faculty Award Program. We are also grateful to Digital Equipment Corporation for fellowship support during early development of this work.

1 Introduction and Motivation

Motivation. *Object-Oriented Databases* (OODBs) are often chosen to support the needs of advanced applications for manufacturing, ECAD, ,MCAD and design, because they support the modelling of complex data [7,11,18]. Although integrated software tools benefit from these database services, such as versioning and access control, these tools often manipulate different representations of shared data. In design applications, for example, a design entry tool manipulates hierarchical design data. On the other hand, a design analysis tool may require an unfolded and flattened structure. For the tools to cooperate in an integrated environment, the data and operations on the data must be translated or transformed for each tool. Even with current integration standards [8] the burden of transforming design data between different formats is frequently the responsibility of a designated translation tool. Sometimes tool developers are charged with the responsibility to transform the data. Either approach results in *ad-hoc* systems that do not support incremental update of shared and transformed data.

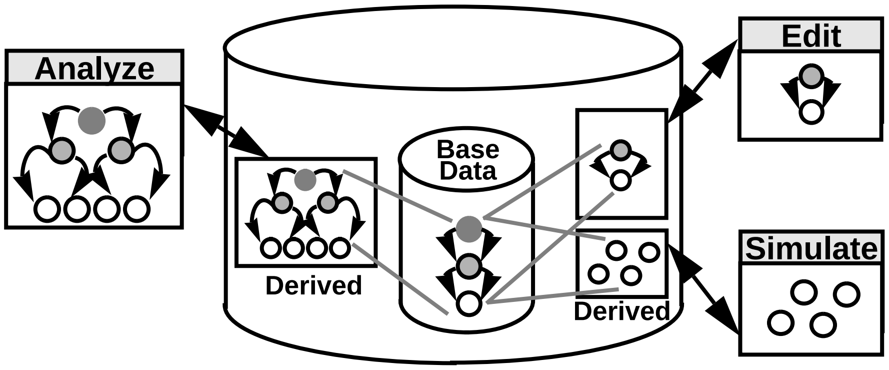


Figure 1: Database Views Supporting Interoperability under Complex Hierarchical Transformations

OODB Views. Database views have been suggested as a means to reduce the effort required to restructure data shared by integrated applications [21,10]. Figure 1 illustrates the database view system that provides custom restructuring of *base data* into a specific *derived* format required by each of the tools in the system. Our OODB view system, MultiView, [22] provides a *view definition language* (VDL) based upon a generic object algebra that is capable of restructuring data for each tool in the system. As such, it automates the transformation of data between the central format and the tool specific formats. In addition, the system provides the services necessary to maintain consistency between the central and materialized derived data [15]. MultiView assures the correctness of data transformations and reduces undesired coupling between integrated tools. As a consequence, MultiView represents a powerful enabling technology for integrating design tools that has the potential of increasing the productivity of tool developers and integrators. However, current OODB view system technology (including the MultiView system) is not yet capable of performing complex data transformations, such as the computation of closure, the traversal of paths, or the flattening of hierarchical graph structures [1,5,13,26]. Because the ability to support complex transformations on hierarchical data is important for achievinginteroperabilitiy of design tools, in this paper we extend the underlying database system with the data model, algebra, operations, and update semantics necessary to perform optimized queries and updates on hierarchically structured data.

Hierarchical Views. Hierarchical descriptions are compact because they re-use portions of the data in the form of *abstractions*. As an illustrative example, consider the description of a 64-bit integer adder that is constructed solely of two primitive elements – a single bit full adder (FA) and a 4-bit carry unit (CU) (Figure 2). In the figure, each large design object has a border pattern and shape that matches the interface abstraction associated with the design object, e.g. the 64-bit adder contains four instances of the 16-bit adder abstraction. The final design representation consists of 64 *implicit* occurrences of the 1-bit adder, even though the design description contains only 4 instances of the 1-bit adder abstraction.

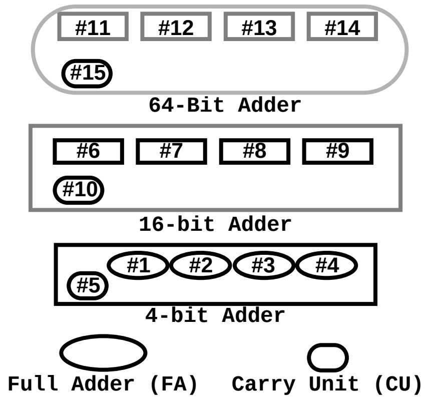


Figure 2: A Folded Hierarchical Representation of a 64-bit Adder.

This hierarchical description is compact, and consequently easier to maintain and manipulate than an equivalent unfolded description. Many complex and powerful operations can be performed very efficiently on hierarchical structures [16]. The disparity the design size and the representation size suggests that there are numerous opportunities to improve the performance of queries on the hierarchical description. We expect to gain the following advantages when employing hierarchical representations:

- Because the size of the description is logarithmic in the size of the design, many fewer disk accesses are required to fetch a hierarchical description from disk than would be for an unfolded design.
- Many queries posed on a folded representation can be answered more efficiently than would be possible on the unfolded data structure, such as: “How many Full Adders occur in this design?”.
- The “concentration” of data into a folded structure permits radical changes to the design with a single operation. For example, in Figure 2, a change to the full adder structure can impact every bit of the 64-bit adder.

The ubiquitous nature of hierarchical structures and their advantages in retrieval and query performance as well as powerful update capabilities suggests that an OODB system should provide support for the hierarchical design representations suitable for the software tools and users.

Problem Description and Our Approach. Despite the clear advantages to using a hierarchical structure, some design tools must perform operations on the *unfolded* or *flattened* (or both) data that is based upon a folded, hierarchical description [27]. The goal of this paper is to support such tools by providing the database support required to perform unfolding and flattening transformations *implicitly*, through a systematically derived, unmaterialized database view, rather than the more commonly employed method of performing an ad-hoc translation on the design. The resulting solution shields the tool developer and integrator from problems of data replication that complicates data consistency maintenance and inhibits the data exchange between tools. Our approach simplifies software tool integration by permitting tools in the environment to operate on their preferred *derived* representations, while the *base* representations are under the care of a central data manager.

Although the implicit unfolding of hierarchical structures can save enormous space and query processing time, the later stages in the design process often require an explicitly unfolded design. Unfolding provides the designer with the capability to distinguish between the small variations in initially identical design objects. We propose an in-context unfolding algorithm that permits the design objects to “diverge” in a controlled fashion, while still maintaining many of the efficiencies of the folded representation. This support called “selective unfolding” defines the necessary algorithms to store and transparently access the selectively unfolded portions of a design.

Experiments. To validate the performance gains for implicitly unfolding hierarchical structures we conduct experiments to measure access time required for aggregation queries on both folded and unfolded sets. The results of our initial experimentation verify the dramatic efficiency, both in time and space, of queries on implicitly unfolded sets. They also indicate the importance of choosing a good clustering strategy for folded sets.

Contribution. It is the goal of this research to provide the database services essential for supporting the implicit unfolding of hierarchical sets. To this end, we have developed an object model for hierarchical sets, including an algebra of operations that permits query optimization and the definition of updatable unfolded views. The new meta-classes, operations, and query operators will be integrated as base classes into the MultiView OODB view system to increase its effectiveness as an integration tool for applications requiring hierarchically structured data.

Structure of this Paper. We begin with an overview of concepts for folded and unfolded structures in Section 2. In Section 3, we describe the object-oriented model upon which our hierarchical object model is based. Section 4 describes our object model for representing and manipulating hierarchical sets. Algebraic operators that exploit the implicit unfolding within our model are presented in Section 5, with update operations presented in Section 6. A performance evaluation of the proposed structures and operations is presented in Section 7. We describe related work in Section 8, and conclude in Section 9.

2 Folded and Unfolded Hierarchical Structures

In this section, we introduce the concepts of folded/unfolded and hierarchical/flat structures commonly used in design applications. We construct design objects by composing *primitives* drawn from a *library*. Primitives may be grouped together to form more complex design objects. *Abstractions*[1] of these more complex design objects are used to compose new design objects, which are said to *own* the abstractions. Figure 3 indicates the *owns-abstraction-of* relationships between the complex design objects (16 and 4 bit adders) and the primitives of the design. This figure closely parallels the relationships shown in Figure 2. Note that the presence of four of the four-bit adders in the design is represented by the 16-bit adder *own*ing four abstractions of the four-bit adder, rather than by having four copies of the four-bit adder in the design. Even though the design description only contains a single copy of a four-bit adder, we say that the design has four *occurrences* of the four-bit adder that are *implicitly* defined by the relationships in the owns-abstraction-of (OAO) DAG.

1. In the ECAD community, these are often referred to as *Interface Instances*. The terms are both similar to and different enough from the instance described in the OO model to cause confusion. We avoid potential confusion by using the term Abstraction.

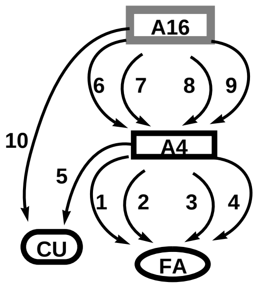


Figure 3: Folded, Hierarchical DAG Representing a 16-bit Adder.

A design description can be characterized as *hierarchical* and *folded* (Figure 3). It is hierarchical if the OAO DAG representing the description has a height greater than one. The description is called folded because each design object may own more than one abstraction of the same design object.

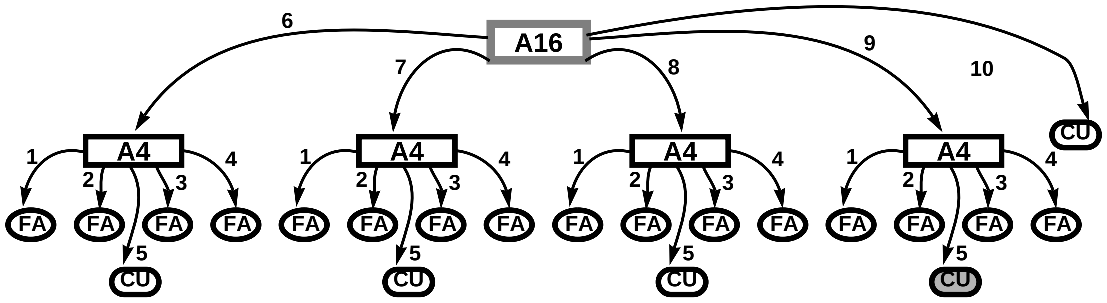


Figure 4: Unfolded Hierarchical Design (Tree).

We can unfold a design by creating a unique replica of the design object associated with each abstraction. Because of the replicated design objects, the unfolded design has space explicitly allocated for each occurrence of an object in the design (Figure 4). This is important if for instance we need to maintain desing data with each unfolded design object. The description remains hierarchical after unfolding, because the height of the design tree resulting from an unfolding operation is the same as the height of the folded design DAG.

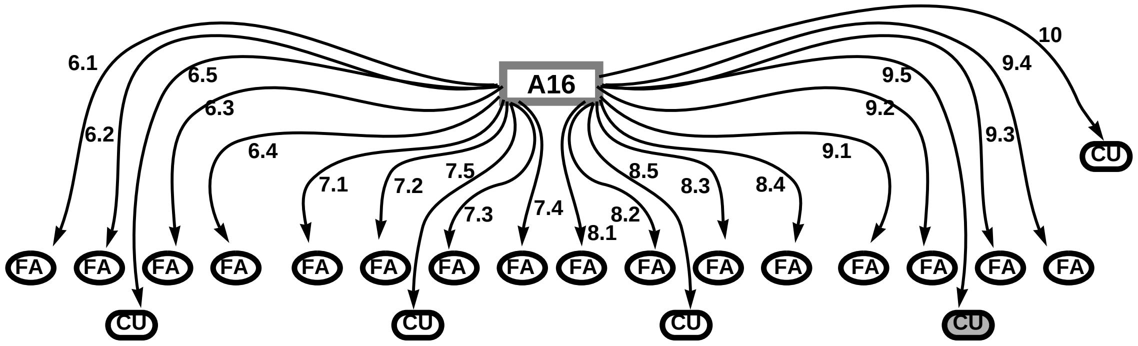


Figure 5: Unfolded Flattened Design

A folded design can be flattened by removing all of the intermediate objects in the hierarchical description and retaining only the root and the leaves of the folded tree. This is practical because, in many cases, hierarchical elements are merely artifacts that make it possible to fold a design. To distinguish between the edges in the flattened DAG, we attach unique labels derived from each path in the original hierarchical and folded DAG. The resultant unfolded and flattened design is illustrated in Figure 5.

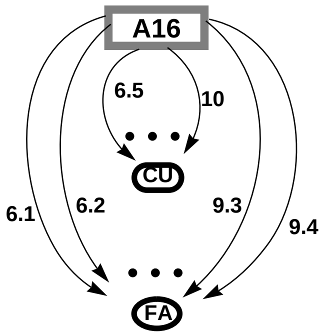


Figure 6: Folded and Flattened Design

The folded and flattened design can be a convenient form for certain design and analysis tools that do not require personalized primitives. This is because the layers of hierarchy in the design are hidden from the design tool, making the compact, folded design more easily accessible (Figure 6).

Note that the distinction between primitive entities in a flattened and folded design is made by way of a path identifier on each edge of the folded, flattened DAG (Figures 5 and 6). Each distinct path in the folded DAG corresponds to a unique *occurrence* of a design object within the design.

The four views of hierarchical design data presented in this section are suited to different applications and to different times in the design cycle. Table 1 compares the forms, their characteristics, and describes when they are most likely to be used.

Table 1: Forms of Design Data and their Uses.

	Folded	Unfolded
Hierarchical	• Used early in design cycle. • Used by tools that do not need access to distinct primitives, e.g., design entry tools. • Most compact representation. • Primitive elements are identical.	• Used later in the design cycle. • Used by tools needing both hierarchy and distinct primitives, e.g., hierarchical placement tool. • Least compact representation. • Primitive elements may all be distinct.
Flattened	• Used early in design cycle. • Used by tools requiring removal of hierarchy, but not requiring distinction between primitives, e.g., early design rule checkers. • Compact representation. • Primitive elements are identical.	• Used later in design cycle. • Used by tools requiring no hierarchy and distinct primitives, e.g., simulators. • Large representation. • Primitive elements may all be distinct.

3 Object-Oriented Data Model

The foundational elements in this work are an object-oriented data model [1, 25, 26] augmented with system classes that support operations on hierarchical sets. In this section we briefly review the object-oriented terminology we utilize in the remainder of this paper.

Classes in the model are arranged in a generalization hierarchy permitting multiple inheritance. Subclasses inherit the type characteristics of their superclasses. Objects of a class may be *substituted* anywhere that objects of a superclass could otherwise be used. The classes that comprise the generalization hierarchy combine to form the *global schema*.

Objects have attributes that are encapsulated. Domains of attributes are either primitive built-in types or references to other objects. Attributes are accessible using the dot (.) operator. In general the (.) operator invokes a method of the same name as the attribute, returning an object that represents the value of the attribute. For an object *a* with an attribute *name*, *a.name* accesses the value of the attribute. It is assumed that a method is invoked to compute the attribute. Several invocations of the (.) operator may be used to access data along an attribute path (i.e., *a.name.length* invokes the *name* method on the object *a* and then the *length* method on the object returned from the *name* method.).

Objects have a unique identifier associated with them. It can be used for comparison when trying to determine if two references to an object are to the same object. Objects can be compared for equality using two distinct comparisons. The $=_{id}$ operator determines if two references are to the same object. The = operator determines if all of two objects' attributes are $=_{id}$.

A class may define a *prototype* instance from which copies are made when instances of the class are created. An instance created from a prototype inherits both the type and the values from the prototype instance. All instances made from a prototype are initially = but *not* $=_{id}$ to the originating prototype.

A *view* in our model is a virtual class [22], defined and named by a query operation on *base classes*. All virtual classes are automatically integrated into the global schema by the view system. For the purposes of this work, views are not *materialized*. We say we can update an unmaterialized view if updates made to the view can be unambiguously propagated to the associated base classes [26].

Because objects and classes are *introspective*, they both have methods to determine if a particular attribute or if a specified operation is valid for the class or object. For a given object *o* and a function *f*, for example, *o.accepts(f)* determines if the method (or predicate) *f* is valid for the object.

4 **`HierSet`** Concepts: The Model

The representation of folded sets requires a means to specify the nesting of sets. To keep the representation compact, this nesting is accomplished using an instance of a set *abstraction* rather than a copy of the set itself. In the simplest case, the abstraction is simply a reference or pointer to the set, but there may also be data stored with the abstraction to distinguish it from others. To ensure the acyclic properties of the OAO DAG, we impose constraints on the nesting of sets.

In the following sections class names appear in a bold, typewriter font, for example the class **`HierSet`** is so indicated. When we refer to the class itself, we will always refer to it as the **`HierSet`** class. We will refer to instances of the class as **`HierSet`** instances, instances of **`HierSet`**, or just **`HierSet`**s. We set off instance names, such as Adder, using a normal-weight sans-serif font.

4.1 The Meta-Data Classes

The meta-classes for modeling hierarchical sets are shown in Figure 7 using OMT notation [24][2]. The class **`Primitive`**, whose instances are maintained by a **`Library`** instance represents the primitives in the set. We build hierarchical sets by inserting instances derived from the class **`Abstr`** into a multi-set, an instance of the **`HSet`** class. The insertion of a **`PrimAbstr`** indicates nesting of a **`Primitive`** instance within an **`HSet`**, while

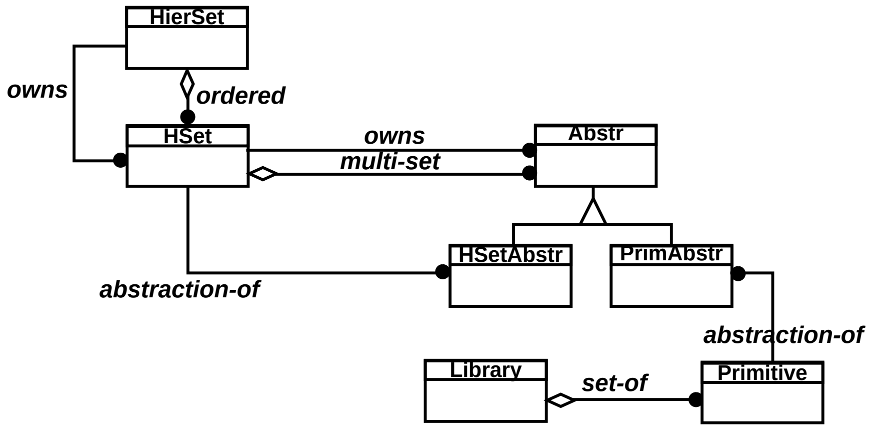


Figure 7: OMT Diagram of the System Classes Supporting Hierarchical Sets.

an **`HSetAbstr`** instance indicates that an **`HSet`** is nested within another **`HSet`**. An instance of the class **`HierSet`** maintains constraints upon operations on the instances of the **`HSet`** class.

A **`HierSet`** manages an ordered collection of **`HSet`** instances. It sequences the **`HSet`**s, and imposes a non-circularity constraint on the composition of **`HSet`**s into **`HierSet`**s. For every instance of an **`HSet`**, there is a corresponding instance of the **`HSetAbstr`** class. This instance serves as the prototype for all other **`HSetAbstr`**s associated with the same **`HSet`**. When a **`HierSet`** creates an **`HSet`** instance, it also creates a corresponding **`HSetAbstr`** prototype for the **`HSet`**. For simplicity, we limit the construction of **`HierSet`**s to be from the bottom-up, however, the top-down construction could be supported with some additions to the system classes.

The HierSet Model

Definition 1. A **`HierSet`** *H* is an ordered set of **`HSet`** instances h_j, namely:

$$H = [h_0, h_1, ..., h_j, ..., h_k] . \qquad (1)$$

Most of the constraints on the construction of **`HierSet`**s are actually imposed on the construction of member

2. OMT, the Object Modeling Technique is one of several popular object-oriented design methodologies that include graphical modeling languages specifically tailored for object-oriented design.

HSets. As shown in Figure 8, an **HSet** h_j is a multiset constrained to contain two types of elements. The first kind of element is an instance of an **HSet** *abstraction* (**HSetAbstr**). Each instance of **HSetAbstr** is associated with only one **HSet** via the *abstraction-of* relationship.[3] The second kind of element, is a *primitive abstraction* (**PrimAbstr**). The **PrimAbstr** relates via *abstraction-of* to a **Primitive** instance stored in a **Library** instance. **HSetAbstr** in an **HSet** represents the nesting of an **HSet** associated with the **HSetAbstr**.

For example, for the 16 bit adder design shown in Figure 3, we have the **HierSet** Adder defined as $\text{Adder} = [h_0 = \text{A16}, h_1 = \text{A4}]$. The **HSet** A16 is defined as $\text{A16} = \{a_1^{\text{A4}}, a_2^{\text{A4}}, a_3^{\text{A4}}, a_4^{\text{A4}}, a_5^{\text{CU}}\}$, indicating the nesting of A4 within an A16 four times and nesting of a CU primitive within the **HSet**. Figure 8 shows the OMT instance diagram for the **HierSet** Adder.

To prevent infinitely nested **HSet**s we number each **HSet** h_k with a unique topological number k and define the following constraint.

Definition 2. For an **HSet** h_k containing abstractions $a_i^{h_j}$, each related to the **HSet** h_j via the abs*traction-of* relationship, the following condition holds:

$$\forall a_i^{h_j} ((a_i^{h_j} \in h_k) \rightarrow j > k) . \tag{2}$$

For our Adder example, the constraint prevents the nesting of an A16 within an A4, or an A4 within itself, because A16 has a smaller topological index than A4.
Additionally, the constraint permits the designation of an **HSet** as the root of a **HierSet** as follows:

Definition 3. In a **HierSet** H, there exists an h_0 such that no **HSet** in H contains an **HSetAbstr** $a_i^{h_0}$.We designate h_0 as the *root*, and access it via the **HierSet** method *H.root()*.

For the Adder example, A16 is the root of the **HierSet** Adder. We impose an additional constraint on the construction of a **HierSet** to assure that it is well–formed. The constraint assures that all required **HSet**s are elements of the **HierSet**.

Definition 4. For a **HierSet** H defined as in Definition 1, we have:

$$\forall h_k \forall a_i^{h_j} ((a_i^{h_j} \in h_k \wedge h_k \in H) \rightarrow h_j \in H) . \tag{3}$$

The Adder example meets this requirement because A4 is a member of the **HierSet**. Figure 8 shows the resulting OMT instance diagram for our Adder example. The diagram shows the *owns* relationship associating **HSet**s with their elements and the *abstraction-of* relationship associating each **Abstr** to its **HSet** or **Primitive**. If we combine the owns and the abstraction-of relationship into a single relationship called *owns-abstraction-of*, we can use the derived relationship to construct the *owns-abstraction-of* (OAO) DAG. The OAO DAG for the Adder is illustrated in Figure 3.

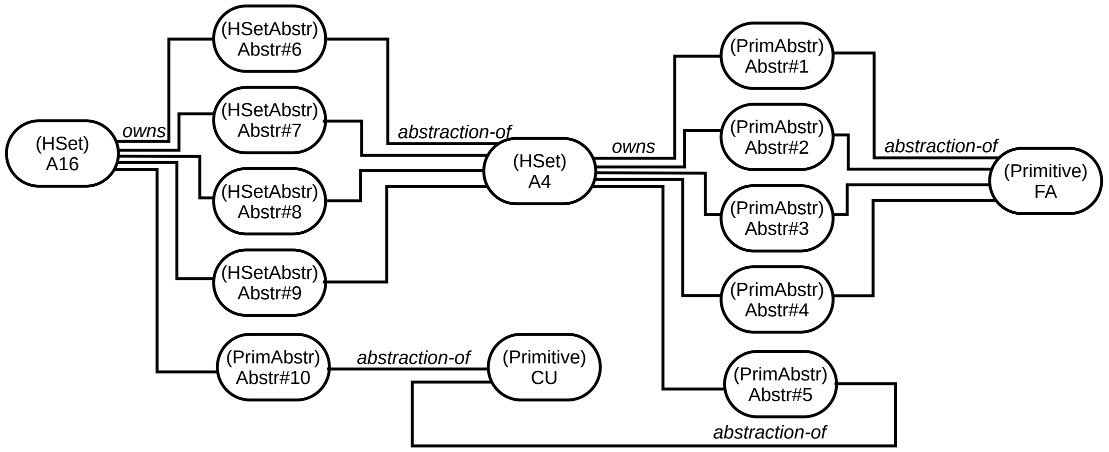


Figure 8: OMT Instance Diagram for the 16-bit Adder portion of the HierSet *Adder*.

3. The association of an **HSetAbstr** with different **HSet** alternatives is called the configuration problem. We assume in our current model that a configuration has already been selected.

Definition 5. The *owns-abstraction-of* (OAO) DAG $G = (N, E)$ of a **`HierSet`** $H = [h_0, h_1, ..., h_j, ..., h_k]$ as defined in Definition 1 and a collection of nodes *N* and edges *E* that satisfy the following conditions:

a. There is a root node h_o.
b. For every h_i in *H*, there is a node in *N* labelled h_i.
c. For every $a_n^{h_j}$ in h_i there is a unique *owns-abstraction-of* edge labelled $a_n^{h_j}$ in *E* from h_i to h_j.
d. For each **`PrimAbstr`** a_n^p in all h_i, there is a single leaf node *p* in *N*.
e. For every **`PrimAbstr`** a_n^p in h_i, there is a unique *owns-abstraction-of* edge in *E* from h_i to *p*.

Figure 3 shows OAO DAG for the **`HierSet`** representing a 16-bit Adder design. The DAG includes the A16 and A4 **`HSet`**s as well as the FullAdder (FA) and Carry Unit (CU) **`Primitive`**s. The OAO DAG has important properties that permit us to define two new concepts *context* and *occurrence*.

Definition 6. An *occurrence* is an object O formed by a unique path in the OAO DAG from an object S to an object D. An occurrence inherits the type of both the path and the object D that ends the path. We define the method *source* to return S, the method *dest* to return the object D, and the method *context* to return the path in the occurrence associated with the parent of D in the OAO DAG.

Referring to the instance diagram illustrated in Figure 8, we see that the **`PrimAbstr`** Abstr#5 is on several paths that originate at the root **`HSet`** A16. One of the four occurrences (occurrence 9.5) in which Abstr#5 participates is shaded in the unfolded/flattened design illustrated in Figure 5.

Definition 7. A *Context* C for an occurrence O in a **`HierSet`** is the path from the root of the **`HierSet`** to the parent of O.

The context is so named because it provides a context in which sibling objects in the OAO DAG may be related via simple relationships in the **`HierSet`**. For example, the context for the occurrence corresponding to the shaded CU in Figure 5 is 9. This context information, coupled with the identity of the **`Abstr`** instance, comprises an occurrence in the implicitly unfolded **`HierSet`**.

4.2 Properties of HierSets

Lemma 1. For **`HierSet`** $H = [h_0, h_1, ..., h_j, ..., h_k]$ rooted at **`HSet`** $h_0 = \{a_1^{h_{i_1}}, a_2^{h_{i_2}}, a_3^{h_{i_2}}, a_4^{h_{i_3}}, ..., a_n^{h_{i_n}}\}$, every **`HSet`** h_{i_j} is the root of a **`HierSet`** consisting of h_{i_j} and all **`HSets`** reachable from h_{i_j} via the transitive application of the *owns-abstraction-of* property.

Proof: This follows from the properties of a DAG as well as the preservation of properties (2) and (3) of **`HierSet`**s. It can be shown using a proof by contradiction on property (2).

To support the definition of the Flatten operator, we define the multiset union, denoted by $\overset{multi}{\bigcup}$, as preserving both membership and duplicate counts. Multiset union meets conditions (4) and (5) for the multi-sets A and B as follows:

$$\forall x \, (\, (x \in A \vee x \in B) \rightarrow x \in (A \overset{multi}{\bigcup} B) \,) \, . \tag{4}$$

If we define a function $dupcount(S, x)$ that returns the number of times the element x occurs within the multiset S, we can express the multiset union as preserving counts with the following condition:

$$\forall x \, (dupcount(A, x) + dupcount(B, x) = dupcount(A \overset{multi}{\bigcup} B, x)) \, . \tag{5}$$

Definition 8. We define the overloaded operation *Flatten* on a **`HierSet`** *H* with the following recurrence:

$$Flatten(x) = \begin{cases} \{x\} & \text{if } (x.class() = \mathbf{Primitive}) \\ \overset{multi}{\bigcup\limits_{i \in x.owns}} Flatten(i.abstraction\text{-}of) & \text{if } (x.class() = \mathbf{HSet}) \\ Flatten(x.root()) & \text{if } (x.class() = \mathbf{HierSet}) \end{cases} \tag{6}$$

5 An Algebra for the HierSets Model

Although there are apparent time and space advantages to using folded **`HierSets`** to represent large, regular structures, there are often times when the structure described by a **`HierSet`** is best viewed in its unfolded splendor. In fact, it is common for applications to require a view of hierarchical/folded data as if it were flat/unfolded (see Section 2). For example, the 64-bit adder structure illustrated in Figure 2 might be subjected to queries such as: "How many full-adder cells does this design contain?" and "What is the total size of all cells in the design?". Although these kinds of queries are most easily posed to a flattened and unfolded representation, the cost to explicitly unfold and flatten a design can be prohibitive. For this reason, we provide query operators that support the querying of hierarchical and folded structures as if they were unfolded and flattened.

In this section, we describe an object algebra for **`HierSet`**s [22,25,26]. Unlike previously proposed object algebras, our operators are unique in that they provide implicit flattening and unfolding of **`HierSet`**s and are essential to query optimization in the presence of hierarchy. We are motivated by the following goals:

- Development of query operators that provide the appearance of operating on unfolded/flattened sets, but have the necessary capabilities to perform the queries on folded/hierarchical structures.
- The need to identify algebraic identities that will enable the development of query optimizations for hierarchical structures.

5.1 Query Operators on `HierSet`s

In this section, we present the algebraic operators defined for **`HierSet`**s. In general, any of the operations with names beginning in "H" take a **`HierSet`** as an argument. As a shorthand notation, they may take an **`HSet`** *h* as an argument, when they are interpreted to mean "the **`HierSet`** rooted at the **`HSet`** *h*".

Unfold

The Unfold operation is the basis for defining most of the algebraic operations for the **`HierSet`** class. Unfold returns a set of all occurrences in a **`HierSet`**. This consists of all paths in the OAO DAG originating from the root. Recall that the OAO DAG consists only of **`HSet`**s and **`Primitive`**s at the vertices and that every occurrence inherits the type of both the path and the final vertex in the path. Because of this, the set returned from the Unfold operation can be treated just as if it consisted of **`HSet`**s and **`Primitive`**s. Unfold for a **`HierSet`** H is formally defined as:

$$Unfold(H) = \{o \mid (o \text{ is an occurrence}) \wedge (o.source() = H.root())\}. \tag{7}$$

Select

Select returns a set of objects *s* from a set *S* for which the predicate *q* is both valid and true. The *accepts()* method establishes the validity of *q* as a method or predicate.

$$Select(S, q) = \{s \mid (s \in S) \wedge q.accepts(s) \wedge q(s)\} \ . \tag{8}$$

HSelect

The HSelect operator finds all paths that are reachable from the **`HierSet`** H starting at the root and ending at an **`HSet`** or **`Primitive`** *s*, such that *q(s)*. We define HSelect in terms of the Select operation and the Unfold operation.

$$HSelect(H, q) = Select(Unfold(H), q). \tag{9}$$

To retrieve a set of all FA occurrences in the Adder design, we ask for all elements in the **`HierSet`** that have the name "FullAdder".

```
Full-Adders = HSelect(Adder, λv v.name = "FullAdder").
```

This returns a set of all occurrences of full adders in the implicitly unfolded design. Because the **`Primitive`** class has a *name()* method, this query operation returns 64 occurrences, each ending with a FA **`Primitive`**.

Image

Image applies a function to each element of a set, creating a set of new objects comprised of the return value from each function application. Only elements in the set *S* for which the function *f* is valid are considered by Image. Invalid elements are discarded. For the set *S* and the function *f*, we define Image as:

$$Image(S, f) = \{f(s) \mid (s.accepts(f) \wedge (s \in S))\} \ . \tag{10}$$

The function *f* may accept the set itself as an argument. This permits a more expressive composition of nested operations, namely,

$$Image(Y, f) = \{f(s, S) \mid (S = Y) \wedge s.accepts(f) \wedge (s \in Y)\} \quad . \tag{11}$$

This form of Image permits the binding of the set Y to the second argument of *f*. Because the function *f* may compute a set, the resulting multiset may be a nested set. For example, to construct a set composed of sets of elements that all have the same name, we can express this as:

```
NameGroups = Image(Y, λs,S Select(S, λx x.name == s.name)).
```

If Y = {A, B, B, A, C, A}, containing objects with single letter names, then NameGroups takes on the value:

```
NameGroups = {{A,A,A}, {B,B}, {B,B}, {A,A,A}, {C}, {A,A,A}}.
```

HImage

The HImage operator is the Image operator over an implicitly unfolded **HierSet**. It is defined as:

$$HImage(H, f) = Image(Unfold(H), f). \tag{12}$$

To obtain a set of all of the sizes of the occurrences of primitive cells in the Adder design we can apply the function λ, which invokes the size method of the argument, to all occurrences:

```
Sizes = HImage(Adder,  λi i.size).
```

The result is a set of the sizes of all occurrences in the Adder for which the size method is valid.

Reduce

Reduce applies a binary operator *op* repeatedly to each element *s* of a set *S* and an accumulated value in order to produce a single value. Defined procedurally, we have:

```
Reduce(S, op) {
   accum = identity(op);
   foreach (s in S) accum = accum op s;
   return accum;
}
```

Reduce does not consider objects for which the reduction operator is not valid, namely:

$$Reduce(S, op) = Reduce(Select(S, opIsValid), op) \tag{13}$$

Our definition requires the following conditions are met:

- *op* must be a binary operator and must be associative, since ordering of the set is arbitrary.
- The domain of *s* must be closed under *op*. This makes it possible to accumulate a value.
- *op* must have an identity value, for example, identity(addition)=0, identity(multiplication)=1.

For convenience, we name commonly used reductions on a set of numbers *S*, using the + operator:

$$Count(S) \equiv Reduce(Image(S, 1), +). \tag{14}$$

We count the objects in *S* by applying the constant function *1*, which returns the value 1 for each element in the set. Recalling the definition of Image from Equation 11, we require that all objects accept constant functions.

$$Sum(S) \equiv Reduce(S, +) \equiv \sum_{i \in S} i. \tag{15}$$

For example, to compute the total size of all primitive cells in the adder design, assuming that each primitive element has a *size()* method that returns a numeric value, we compute the sum over the size of the primitive elements:

```
Size(Adder)=Sum(Image(HSelect(Adder, λs s.isPrimitive), λi i.size)).
```

HReduce

The HReduce operator applies a reduction using the operator *op* to the **HierSet** *H*. We define the HReduce operator using the Unfold and Reduce operators as follows:

$$HReduce(H, op) = Reduce(Unfold(S), op). \tag{16}$$

DupEliminate

This operator is used to eliminate the duplicates from a multiset. This is particularly important when the set property depends upon the type of equality used to define uniqueness. A set defined with $=_{id}$ as the uniqueness criterion can have elements that are value-equal (=), and thus with respect to the = comparison, the set is a multiset. Since the notion of duplicate is completely dependent upon the notion of equality, the operator requires that the kind of equality be specified in the operation [23]. To obtain sets of unique names from the NameGroups set define

```
UniqNames=DupEliminate=(Image(Y, λs,S Select(S, λx x.name == s.name))).
```

If Y={A, B, B, A, C, A}, representing objects with the single letter names as listed in the set, then UniqNames assumes the value:

```
UniqNames = {{A,A,A},{B,B},{C}}.
```

Note that because Image creates unique objects with new identities, $DupEliminate_{=id}$ would not have eliminated any elements from the set returned by Image.

Project

The Project operator permits the specification of multiple functions on a set. The values returned form a new object with the attribute names specified in the operator. Project is defined by:

$$Project(S, \{(a_1, f_1), (a_2, f_2), \ldots, (a_n, f_n)\}) = \{[a_1, a_2, \ldots, a_n] \parallel a_i = f_i(s), s \in S\}. \tag{17}$$

In this definition, the [] operator creates an object with the specified attribute names. The function f_i determines the value of the attribute a_i. For example, we create a histogram set for UniqNames using Project to record the name and the number of occurrences for each name.

```
Histogram = Project(UniqNames,{(count, λs Count(s)),
                                  (name, λs DupEliminate=(s))}.
```

This returns a set of new objects each with a count and name attribute as follows:

```
Histogram = {[count:3,name:A], [count:2,name:B], [count:1,name:C]}.
```

HProject

Similarly to HReduce, HProject can be defined by extending project to operate on implicitly unfolded **Hier-Set**s. HProject is defined as:

$$HProject(H, p) = Project(Unfold(H), p). \tag{18}$$

5.2 Query Optimizations

Because folded structures have a built-in means for identifying duplicate subpaths, we can systematically factor out multiply occurring subpaths and recombine them in a more efficient manner. In this section, we present an optimization that can be applied to aggregation queries that obey the distributive property. For this explanation we will consider the implementation of the query:

```
Area(Adder) = Reduce(HImage(Adder, area), +)
```

We assume that for the 16-bit adder design illustrated by the OAO DAG in Figure 3, both primitive library elements have an area attribute that reveals the area of the component. For the purposes of this example, the FA area is 7 and the CU has an area of 13.

This operation clearly could be carried out by enumerating all paths in the OAO DAG and totalling the area attributes for all paths ending in primitives. However, this can result in redundant computation as many paths have common subpaths that could otherwise be systematically merged. Our approach to optimize this query is to factor out common subpaths and use multiplication to combine them in the reduction.

First we factor out the duplicates in an **HSet** in order to compute the frequency of commonly nested elements in the current **HSet**. We group the elements together that are equal (=) using the Group operator defined as:

```
G = Group(X) = DupEliminate=(Image(X, λs,S Select(S, λx x = s))).
```

The Histogram operator determines frequency counts for elements in the groups formed by the Group operator.

```
Histogram(G)=Project(G, { <count, λs Count(s)>,
                          <id, λs DupEliminate=(s)> }.
```

This creates the tuple object consisting of the number of times each distinct abstraction is represented in the **HSet** X. For the Adder this corresponds to a set of the form:

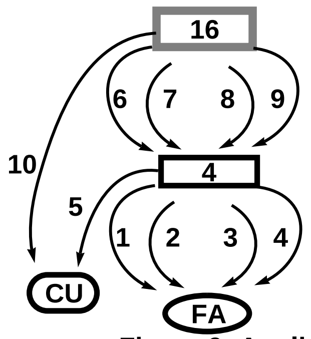
```
Histogram(Group(A16)) = {<count:4,id:A4>,<count:1,id:CU>}
```

Duplicates have been removed, so this has the effect of pruning all duplicate sub-paths from the operation. Next we define a function Distribute that will be used to apply the optimization of distributing multiplication over addition. Distribute takes an instance of the Histogram object as defined in the project operation above and computes the frequency count times the value returned from a recursive call to Reduce. Figure 9 shows the values for the Group, Histogram, and Distribute operators along with the data on which they are applied.

```
Distribute(j) = j.count * Reduce(j.id.abstraction-of, +)
```

By using the Image operator, we can apply the Distribute function to the results of the Histogram operation.

```
Area(A16) = Reduce(Image(Histogram(Group(A16)), Distribute),+)
```



Group(A16) = { {A4,A4,A4,A4}, {CU}, {A4,A4,A4,A4}, ...

Hist(G(A16)) = { <4,A4>, <1,CU> }

Area(A16) = 1 * CU.size + 4 * Area(A4)

Area(A4) = 1 * CU.size + 4 * FA.size

Area(A16) = 177

Figure 9: Application of Distributive Optimization to Area(A16)

This optimization uses the distributive property to improve query performance in folded structures and can be applied using any pair of operators for which the distributive property holds. This applies to reduction on the addition operator, the logical or operator, and the parallel path operator. These operators have distributive counterparts in multiplication, logical AND, and the serial path operator (concatenation), respectively.

6 Updates on HierSets

In addition to querying implicitly unfolded and flattened structures, we also need to perform update or deletion operations. We classify these updates into two categories: *out-of-context updates*, and *in-context updates*. Out-of-context updates correspond to operations on the **`HSet`**s directly. They are called out-of-context updates because updates directly to **`HSets`** apply to all occurrences in the implicitly unfolded set, rather than to a specific object in a specific context. For example, these kinds of operations are common in design applications when the user of the data wants to make changes to the entire hierarchical structure. The most common out-of-context updates are the removal or insertion of **`Abstrs`** from **`HSet`**s. Additionally the attribute values of **`HSetAbstr`** instances may be changed. If a single attribute is changed out-of-context, it affects all of the paths in the implicitly unfolded structure that contain the attribute. In-context updates, on the other hand, are updates to the implicitly unfolded/flattened view of the **`HierSet`**. They are, in a sense, updates to data that does not exist explicitly. In-context updates are so called because they effect only a specific path (in a specific context) in the OAO DAG.

We take advantage of function overloading to disambiguate between out-of-context and in-context updates. In-context updates take occurrences as arguments, out-of-context operations take **`Abstr`**s as arguments. It is this difference in the argument types that determines the type of operation performed.

6.1 Out-of-Context Update Operations

Out-of-context updates are subject to constraints because they may result in the violation of constraints (i.e., non-cycle constraint) imposed upon the **`HierSet`**. In general, the operations described in this section are rejected if they result in the violation of the basic **`HierSet`** constraints.

Delete

The deletion of an **`Abstr`** from a particular **`HSet`** corresponds to the deletion of a single edge in the OAO DAG. Many occurrences from the unfolded must be removed as a result. **`Abstr`**s are deleted from an **`HSet`** using the **`HSet`** method *delete()*. To delete an **`Abstr`** *a* from the **`HSet`** h_k:

$$h_k\texttt{.delete(a).} \qquad (19)$$

Even though the deletion of a single **`Abstr`** corresponds to the removal of one edge in the OAO DAG, the operation may remove a entire sub–DAG from the design. For example an out-of-context update that removes a single FA **`PrimAbstr`** from the **`HSet`** A4 in Figure 2 removes four FAs from the unfolded design.

Insert

Similarly, inserting an **`Abstr`** into an **`HSet`** corresponds to adding an edge into the OAO DAG. This again has far-reaching consequences and may result in the addition of many occurrences into the unfolded **`HierSet`**, since an entire sub–DAG can be connected into the **`HierSet`** using the following operation.

$$h_k\texttt{.insert(a).} \qquad (20)$$

The insertion of an abstraction into an **`HSet`** is subject to the non-circularity constraint imposed upon the **`HierSet`** (see Equation 2). For example, the insertion of an A4 abstraction into the **`HSet`** A4 would not be permitted, but the insertion of an A4 abstraction into the A16 results in the addition of four new occurrences of FAs into the design.

Modify

Modifications to the membership of **`HSets`** can be accomplished with a deletion and insertion. However, the **`HSet`**s in a **`HierSet`** may also have other modifiable attributes that are used by software tools requiring hierarchical information. Updates to these **`HSet`** attributes do not cause topological changes to the OAO DAG, but they still can effect many occurrences in the unfolded set. For example, we may attach an *area* attribute to an **`HSet`** such as A4 in Figure 2. This property then applies to all occurrences of the **`HSet`**.

6.2 In-Context Update Operations

To make views defined on **`HierSet`**s updatable, we define the semantics of updates on implicitly unfolded **`HierSet`**s. For simplicity and clarity, we specify update semantics in terms of operations on the OAO DAG. Because there is a definitive mapping between the instance graph and the OAO DAG, operations on the OAO DAG are unambiguous.

Delete

Deleting a single occurrence from the design corresponds to removing a single path from the OAO DAG. In order to perform the delete operation, topological changes must be made to the OAO DAG. In propagating the update we strive to impact the DAG as little as possible. An occurrence *O* is removed using three basic steps.

1. Identify the critical section of the DAG that must be removed or changed.
2. Remove all paths that begin with *O.source()*.
3. Reconstruct all paths that begin with *O.source()* excluding elements of *O* on the critical section.

```
Delete(D : oaoDAG, c : oaoPath)
int top = index of oaoPathNode closest to root with indegree > 1
int btm = index of oaoPathNode closest to leaf with outdegree > 1
if (top > btm) then
    // only one path between c[btm].node and c[top].node, and it
    // it can be removed without disturbing other paths. (Figure 11)
    D.deleteEdge(c[btm+1].edge to c[top].edge)
    D.deleteNode(c[btm].node to c[top].node if disconn. from DAG)
else
    // record the parent of the split node in p
    // remove edge in the path at the split point (Figure 12(b))
    p = c[top-1].node;
    D.deleteEdge(c[top].edge);
    // “split” each node between top and btm inclusive
    for (i=top; i<=btm; i++)
            newnode = D.cloneNode(c[i].node); // add copy to DAG
            // clone all but the corresponding edge in the path being removed
            D.deleteEdge(from newnode to c[i].node);
            D.addEdge(from p to newnode);
            p = newnode;
    end for
end if
```

Figure 10: Algorithm to Delete a Single Occurrence.

The algorithm shown in Figure 10 demonstrates the steps required to accomplish the deletion of a single occurrence from an implicitly unfolded **`HierSet`**. It first finds the top and bottom of the critical section of the DAG. The index number of these two vertices are stored in the variables top and btm, respectively. The first case the algorithm addresses is when the branching points in the DAG result in the value of top > btm. This means that the path to be removed contains a subpath that is not shared by any other occurrence. This case is illustrated in Figure 11, in which only the “critical portion” between the btm and top markers is removed from the DAG. Removing only the critical portion of the DAG is sufficient because it does not affect any other occurrence path and because it removes the desired path from the DAG.

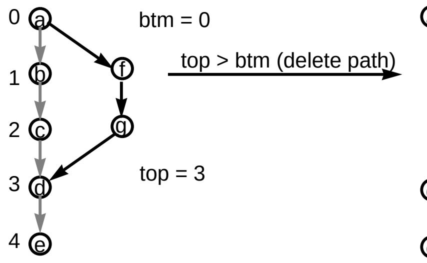
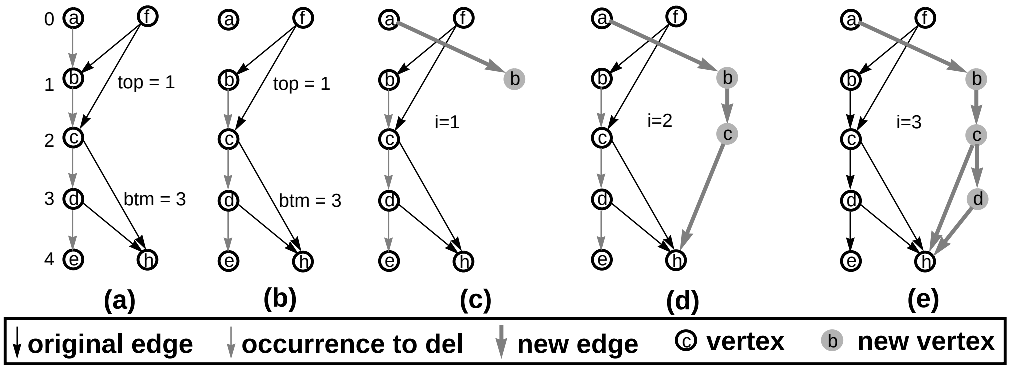


Figure 11: Removal of path <abcde> in DAG where top > btm

The sequence of frames illustrated in Figure 12 shows the operations required for removing a single path <abcde> from a DAG when top < btm. The figure shows the removal of all paths starting with the vertex (a) and the systematic reconstruction of all paths beginning with (a) but do not contain the occurrence path <abcde>. This reconstruction corresponds to the creation of copies (versions with small modifications in terms of which **`Abstr`** they own) of all design objects between the *top* and *btm* markers.



a. top and btm are located by traversing the path from the root and from the leaf.
b. First edge in path is removed.
c. Element b is cloned and the deleted edge is restored to point to b.
d. Element c is cloned (including its edge to h) and an edge from b to c is added.
e. Element d is cloned (including its edge to h) and an edge from c to d is added.
f. Costs of Queries and Updates in HierSets

Figure 12: Removal of path <abcde> in DAG where top < btm.

Insertion

The insertion of an occurrence in the implicitly unfolded design is similar in complexity to the deletion operation. A more simple method is to add the path directly to the DAG by attaching it to the root. A more complex method attemptd to find a "closest fit" path (so that the inserted path can be folded into the existing DAG) and add the path at the appropriate point in the DAG. For our approach we simply add the new path by attaching it to the root of the DAG.

Update

An in-context update corresponds to the creation of a new version of the modified object in the folded DAG. To accomplish this update, we delete the occurrence and then add another with the new attribute value. For example, to change an attribute value on the occurrence <abcde> as shown in Figure 12(a), we perform a deletion as shown, and then add another version of element *e* with the new attribute value. Because the update algorithm knows where the deletion occurred, it knows the "closest fit" point to insert the new, modified element *e*. The resulting OAO DAG is shown in Figure 13.

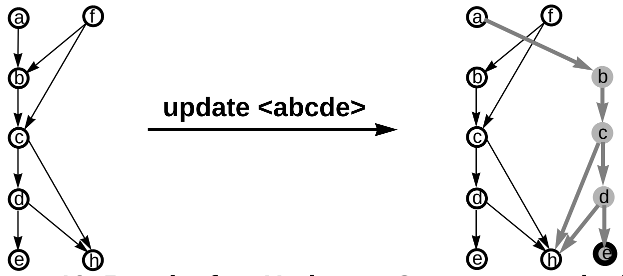


Figure 13: Result of an Update to Occurrence <abcde>.

While complex updates to occurrences may require topological changes to the OAO DAG, it may be possible (and practical) to update parameterized values in the **`Primitive`**s that apply to a single occurrence in the **`HierSet`**. This capability is not fully developed in our current model, but we plan to include it in future work.

7 Performance Evaluation

To evaluate the cost of query operations in a **`HierSet`**, we specify our model of the disk paging system, as well as the parameters that characterize the **`HierSet`**s. For our evaluation, we measure the cost in terms of disk accesses of queries on both the folded **`HierSet`** and the flattened and unfolded **`HierSet`**.

7.1 Disk Model

We model the persistent storage of the **`HierSet`** as a sequence of fixed size pages of *B* bytes each on disk. The storage system buffer has a capacity of *P* pages. Pages are replaced in the buffer using an LRU replacement policy. We assume an object table that fits into main memory and provides constant time to identify the location (page) of an object, given its object identifier. The time to read data already in the cache is *H* time units. To read a page that is not in the cache requires M time units.

Data can be clustered onto the disk in several different ways. We consider a **`HierSet`** that is clustered onto a disk using either depth-first or breadth-first clustering schemes. In depth-first/breadth-first clustering, **`HSets`**, **`Abstr`**s and **`Primitive`**s are assigned to disk pages based upon a depth-first/breadth-first traversal of the OAO DAG.

7.2 Characteristics of the HierSet

Sizes of the objects that comprise a **`HierSet`** are designated in bytes as follows:

b_a : The size of an **`Abstr`** instance. **`PrimAbstr`** and **`HSetAbstr`** instances are the same size.

b_h : The size of an **`HSet`** instance (minus the size of its **`Abstr`** instances).

b_p : The size of a **`Primitive`** instance.

To determine performance properties of folded and unfolded **`HierSet`**s for various queries, we identify characteristics of the OAO DAG that may influence the performance of operations.

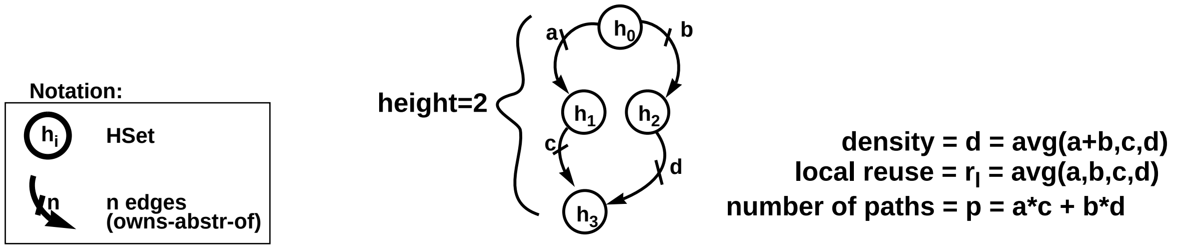

Notation:
h_i HSet
n n edges (owns-abstr-of)
height=2
h0
h1
h2
h3
a
b
c
d
density = d = avg(a+b,c,d)
local reuse = r_l = avg(a,b,c,d)
number of paths = p = a*c + b*d

Figure 14: OAO DAG Characteristics from the Performance Model

Density (d)

The density d of a **`HierSet`** is the average number of edges in the OAO DAG that originate from each **`HSet`**. This average fanout or branching factor corresponds to the complexity of each level of the design. For our evaluation, we consider OAO DAGs that have the same number of edges originating from each **`HSet`** in the **`HierSet`**. For the DAG illustrated in Figure 14 the density is determined by the computation: $d = avg(a+b, c, d)$.

Local Reuse Factor (r_l)

This factor measures how often a specific **`HSet`** references the same **`HSet`** or **`Primitive`**. We compute the local reuse factor r_l for a specific **`HSet`** by computing the average number of edges between every pair of **`HSet`**s for which at least one edge exists. In the **`HierSet`** illustrated in Figure 14, the local reuse factor is determined by computing: $r_l = \mathrm{avg}(a, b, c, d)$.

We consider the local reuse factor because we expect a high reuse to improve performance due to caching behavior. When an **`HSet`** retrieves all of its owns-abstraction-of relationships, a high local reuse factor in a folded **`HierSet`** can reduce the cost to retrieve only the unique ones.

Height of OAO DAG (*h*)

We determine the height h by calculating the longest path from the root of the **`HierSet`** to any of the primitives. We consider the height of the DAG as indicating how deeply nested a hierarchical description is. Because of the multiplicative factor of d at each level, the height contributes exponentially to the size of the design. The height of the OAO DAG in Figure 14 is 2.

Number of Paths in the OAO DAG (*p*)

The number of paths in the OAO DAG are determined by many characteristics of the DAG, including the re-use factor, the density, and the height of the DAG. The following recursive function determines the number of paths in a **`HierSet`**. A trace of the definition shows that it performs a depth-first enumeration of all of the paths in the DAG originating from the root and ending at a primitive. Each time the traversal ends a path it adds one to the total.

$$p(x) = \begin{cases} p(x.root()) & if\ (x.class = HierSet) \\ \sum_{i \in x.owns} p(i) & if\ (x.class = HSet) \\ p(\text{x.abstraction-of}) & if\ (x.class = HSetAbstr) \\ 1 & if\ (x.class = Primitive) \end{cases} \tag{21}$$

7.3 Performance Experiments

In this section, we present the costs to perform query operations on **`HierSet`**s in either folded or unfolded and flattened form. Recall that the reduce operation permits the application of a single operator cumulatively over the entire **`HierSet`**. We consider aggregation queries and operations that return a single value for two reasons.

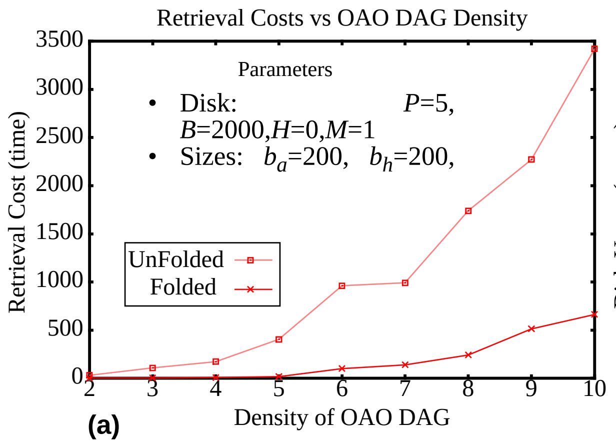
1. Because of the absence of other relationships in our current model, we are limited to traversals of the design by traversing the *owns-abstraction-of* DAG.
2. Queries that return single values offer a more realistic evaluation of the implicit unfolding of hierarchical sets. This is because operations that return entire designs must explicitly perform unfolding to return the unfolded values.

In keeping with other examples presented earlier in this paper, we evaluate the performance associated with answering the following query operation to compute the total power dissipation for a design represented by the **`HierSet`** H. Assuming that each **`Primitive`** in the **`Library`** has an integer attribute *power*, we express this query as:

```
Reduce(HImage(H, λs s.power),+).                    (22)
```

7.3.1 The Effect of DAG Density on Aggregation Query Performance

For this evaluation, we construct **`HierSet`**s characterized by different densities and measure the cost to do the power dissipation query operation on the **`HierSet`**. The density measures the branching factor in the DAG, and we see the multiplicative effects of the branching factor on the retrieval cost of the design. We see similar shapes for many configurations of the DAG, with the unfolded cost growing quadratically, and the folded cost growing linearly as the DAG is more dense. Figure 15(b) shows similar, but even more dramatic differences when the sizes of the two representations are compared.



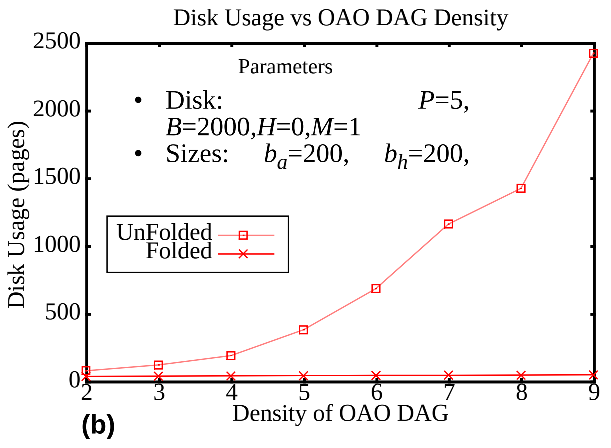


Figure 15: Retrieval Performance (a) and Disk Usage (b) for Folded and Unfolded Data.

7.3.2 Effect of Query Optimization on Retrieval Costs

In this experiment, we again measured retrieval performance on the power dissipation query for an unfolded and folded designs of varying density. This time, however, we exploited the distributive optimization for a folded design that we presented in Section 5.2.

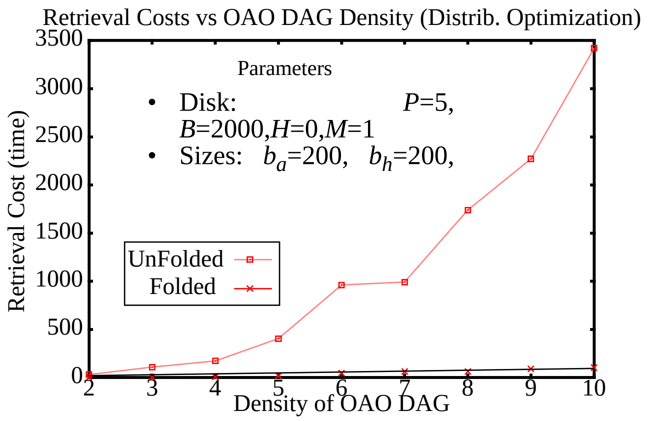


Figure 16: Retrieval Performance for Unfolded and Optimized Folded

We see in Figure 16 that the distributive optimization possible for reduction operations makes the retrieval cost very insensitive to changes in the DAG density. We note that the optimization shows dramatically improved performance across a broad range of parameters. Although we did not verify this with an exhaustive set of experiments, we believe that the distributive optimization, because it eliminates the traversal of paths multiple times, renders the query performance insensitive to the size of the cache.

7.3.3 Effect of DAG Height on Retrieval Costs

We measured the retrieval cost for both unfolded and folded DAGs of varying heights in this test.

Figure 17 illustrates the exponential growth of the unfolded design as the height of the DAG increases. This is consistent with the relationship between the size of folded and unfolded designs. It is important to note, though that in electrical design applications for example, the height of the design DAG is rarely more than 5, so we may not achieve as dramatic a performance gain as the graph suggests.

7.3.4 Effect of Relative Sizes of Abstractions and Primitives on Retrieval Performance

For this experiment, we determined the retrieval time on folded and unfolded data while the relative sizes of **`Abstr`** (b_a) and **`Primitive`**s (b_p) were varied. b_p remained fixed while b_a varied from 25 to 2000 bytes.

The size of **`Primitive`**s is assumed to be substantially larger than **`Abstr`**s, otherwise there would be little incentive to construct a hierarchical and folded representation. Therefore, for most practical applications, the sytem operates in the region to the right side of the plots. Note, for example, that to the right in Figure 18 the folded representation clearly outperforms the unfolded. Even with **`Abstr`** instances as large as the **`Primitive`**s, the folded representation performs well. However, if the **`Abstr`** instance is much larger, the performance degrades substantially. This effect is observable in many different configurations, with the crossover point determined by how much of the folded design fits in the disk cache. Note the different crossover points when 50 per-

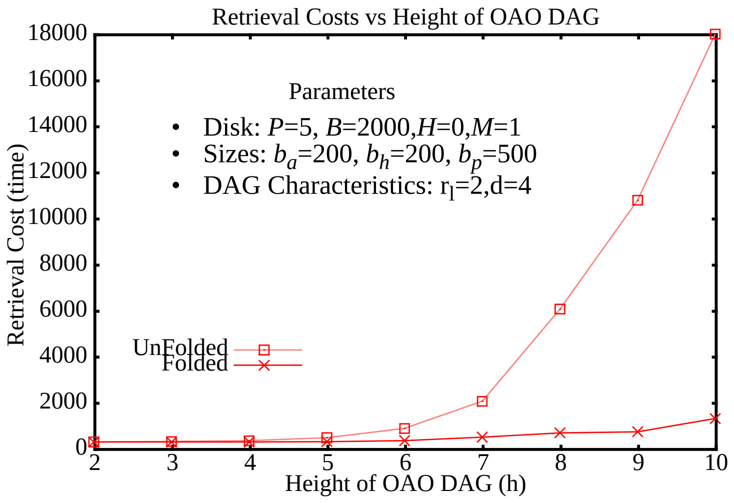


Figure 17: Retrieval Cost For Unfolded Design is Exponential in DAG Height.

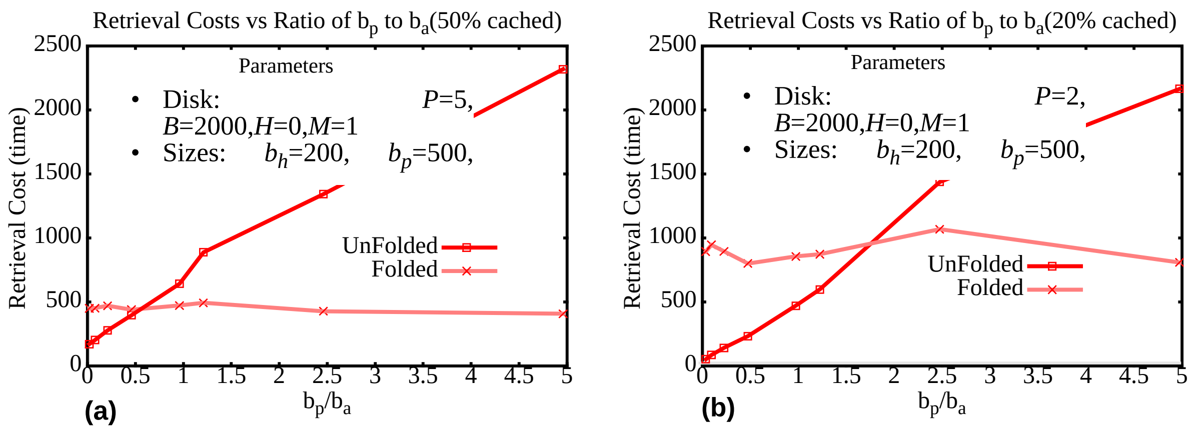


Figure 18: Retrieval Performance for Relative Sizes of b_p and b_a.

cent (Figure 18a) and 20 percent (Figure 18b) of the folded design fits in the cache.

7.3.5 Effect of Clustering on Retrieval Performance

For this experiment, we measured the performance for traversing the folded design in a breadth-first order for an ideal clustering (breadth-first), and a depth-first clustering. We varied the size of the disk cache for the tests.

The plot of Figure 19 shows the effect on query performance of having a clustering order different from the traversal order as the proportion of the `HierSet` that fits in the cache is varied. Breadth-First clustering performs very well when 40 to 50 percent of the `HSet` fits in the cache. The poorer Depth-First clustering requires a much higher (70 percent) fit before it sees comparable performance gains. This clearly demonstrates the need for clustering strategies that are compatible with the traversal algorithms [4].

7.3.6 Experimental Summary

Our initial experiments indicate that aggregation queries have the potential to be dramatically faster on folded than on unfolded structures. The substantially smaller size of folded structures reduces disk access costs from quadratic and exponential time in DAG characteristics to linear time. Furthermore, we have quantified the tradeoff of the abstraction size and the primitive size. Our experiments also demonstrate the importance of clustering algorithms to the performance of queries on folded structures.

8 Related Work

Related work falls mainly into three distinct categories. The first is work on domain specific constructs for databases, the second is work on database views, and the third is research specifically related to electrical CAD.

Recent research in databases for specific domains has suggested that additions to data models and query languages may be appropriate to enhance the effectiveness of the databases [2, 9]. In fact, it has been proposed to confer first-class citizenship to new entities such as paths [9], or hyperwalks [2]. Paths are an important part of our model, but they exist primarily to represent the occurrence of unfolded objects. Our work suggests extensions to

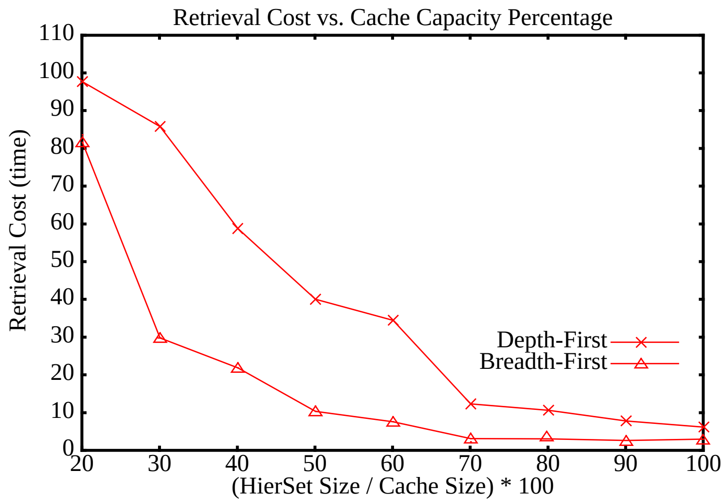


Figure 19: Breadth-First Traversal Costs for Different Clustering

the data model through the introduction of new meta-data classes that can be used effectively in domains requiring hierarchically structured sets.

Earlier work in OODB database views has established data models, object algebras, update policies and materialization strategies, all motivated by the desire to provide either real or virtual restructuring of data for database applications [5,1,26]. Most current work on OODB views studies traditional OO query languages similar to SQL rather than employing query extensions for complex views. Similary, the MultiView OODB view system [22] currently employs an object-preserving algebra as a query language for view definitions. To model the more complex restructuring employed in design tools, such as flattening hierarchical graphs and deriving transitive relationships, we are extending MultiView with more powerful view operators [13]. The work presented in this paper continues this effort of extending MultiView with complex transformation support.

In addition to extending the object algebra to support more powerful transformations, we are addressing the problems of how to make these views updatable. This is an important topic for view mechanisms that has for instance been studied by Scholl et. al. for object-preserving algebra views [26]. For our implicitly unfolded data, the update problem is how to propagate the updates from objects which only exist implicitly to the base data. In this paper, we solved this problem by transforming updates on the implicitly unfolded structure into updates on paths in the folded design DAG.

Work on the HS system [19] describes an API capable of implicitly flattening netlist data. Updates to the implicitly flattened data are limited, and require a re-initialization of the database. Additionally, the work does not present a data model and query operations capable of defining implicitly unfolded views.

Research on hierarchical attribute grammars [12] presented incremental update schemes to propagte changes from a folded representation to an explicitly unfolded representation. Our work also propagates updates from the unfolded to the folded representation as well as saving the space that is otherwise wasted on an explicit unfolding of the data (see Section 7.3.1). The FICOM system [3] maintains complex constraints across various abstraction domains, but also requires that the two distinct representations are stored separately. The system addresses update propagation in both directions, but the same problems of space and performance overhead remain.

Recent research in enabling technology for electronic design frameworks has focused on information modelling of folded and unfolded design [6,8,27]. These models are used to define APIs, to develop data structure generators, or to formalize the exchange of data between systems. In general, the work on information modelling does not present how a data manager in the database system can provide support for implicit unfolding of data.

9 Conclusions

We achieved an important step toward improving interoperability of design tools that operate on hierarchical data that is folded and unfolded. In this paper, we have presented new meta-data classes appropriate for the efficient representation and querying of folded, hierarchical sets. Included in our model are operations that can be efficiently performed on these **`HierSet`**s by fully exploiting the constrained characteristics of the model. We have presented algebraic operators that enable the implicit unfolding and flattening of hierarchical sets, and have shown an algorithm capable of propagating updates from the implicitly unfolded view to the folded view via “selective unfolding”. Additionally, we have presented opportunities for query optimization for performing aggregation queries on hierarchical sets. Finally, we have conducted experiments that validate the dramatic impact that the folded representation has on retrieval performance and on disk storage utilization, as well as demonstrate some of the issues that we must consider while continuing our work.

10 References

[1] S. Abiteboul and A. Bonner, “Objects and Views,” in *Proc. of the ACM SIGMOD 91*, 1991.

[2] B. Amann and M. Scholl, “Gram: A Graph Data Model and Query Language”, *ECHT 92*.

[3] R. Armstrong and J. Allen, “FICOM: A Framework for Incremental Consistency Maintenance in Multi-Representation, Structural VLSI Databases,” in *Proc. IEEE International Conference on Computer-Aided Design (ICCAD)*, 1992, pp. 336-343.

[4] J. Banerjee, W. Kim, S.-J. Kim and J. F. Garza, “Clustering a DAG for CAD Databases”, *IEEE Transactions on Software Engineering*, 14(11):1684-99, 1988.

[5] J. Blakeley, “Efficiently Updating Materialized Views”, SIGMOD Record, 15(2):61-71, June, 1986.

[6] A. Bredenfeld, “A Generator for Graph-Based Design Representations,” in *4th International Working Conference on Electronic Design Automation Frameworks*, (EDAF 94) 1994.

[7] M. A. Breurer, W. Cheng, and e. al., “Cbase 1.0: A CAD database for VLSI circuits using object oriented technology,” in *Digest of IEEE International Conference on Computer-Aided Design (ICCAD)*, 1988.

[8] CFI-DR-TSC, “Design Representation Electrical Connectivity Information Model and Programming Interface,” 121, October 23, 1991.

[9] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl, “From Structured Documents to Novel Query Facilities,” in *Proc. ACM SIGMOD International Conference on Management of Data*, 1994.

[10] D. Garlan, “Views for Tools in Integrated Environments,” in *Advanced Programming Environments*, Springer-Verlag, 1986, pp. 314-343.

[11] S. Heiler, “An Object-Oriented Approach to Data Management: Why Design Databases Need It”, in *Proc. IEEE/ACM Design Automation Conference (DAC),* 1987, pp 335-40.

[12] L. G. Jones, “Fast Batch and Incremental Netlist Compilation of Hierarchical Schematics,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(7):922-31, 1991.

[13] M. Jones and E. Rundensteiner, “Extending View Technology for Complex Integration Tasks,” in *Proc. 4th Intl. Working Conf. on Electronic Design Automation Frameworks, (EDAF 94),* pp. 71-80, 1994.

[14] H. F. Korth and A. Silberschatz, “Database System Concepts”, Second Edition. 1991, McGraw-Hill.

[15] H. A. Kuno and E. A. Rundensteiner, “Materialized Object-Oriented Views in MultiView,” in *Proc. Fifth International Workshop on Research Issues on Data Engineering: Distributed Object Management (RIDE-DOM ’95),* March 1995.

[16] T. Lengauer, “Combinatorial Algorithms for Integrated Circuit Layout,” John Wiley and Sons: pp. 105-121.

[17] R. A. Lorie, “Issues in Databases for Design Applications,” in *File Structures and Data Bases for CAD,* J. Encarnacao and F.-L. Krause, Editor. 1982, North Holland.

[18] D. Maier, “Making Database Systems Fast Enough for CAD Applications”, in *Object-Oriented Concepts in Databases and Applications,* W. Kin and T. H. Luchovsky eds., ACM Press, 1989.

[19] N. Parikh, C.-Y. Lo, N. Singhal, and K. Wu, “HS: A Hierarchical Search Package for CAD Data,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):1-5, 1993.

[20] Y. G. Ra, H. Kuno, and E. A. Rundensteiner, “A Flexible Object-Oriented Database Model and Implementation for Capacity-Augmenting Views”, Electrical Engineering and Computer Science Dept., University of Michigan, Ann Arbor, Tech. Rep. CSE-TR-215-94, May 1994.

[21] E. A. Rundensteiner, “Design Tool Integration Using Object-Oriented Database Views,” in *Proc. IEEE International Conference on Computer-Aided Design (ICCAD),* 1993, pp. 104-107.

[22] E. A. Rundensteiner, “MultiView: A Methodology for Multiple Views in OODBs,” *in Proc. IEEE Intl. Conf. on Very Large Data Bases,* 1992, pp. 187-198.

[23] E. A. Rundensteiner and L. Bic, “Set Operations in New Generation Data Models”, in *IEEE Transactions on Knowledge and Data Engineering*, 1992, pp. 382-98.

[24] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, “Object-Oriented Modeling and Design”, Prentice Hall, 1991.

[25] G. M. Shaw and S. B. Zdonik, “An Object-Oriented Query Algebra”, *IEEE Data Engineering,* Sept. 1989, pp. 23-36.

[26] M. H. Scholl, C. Laasch, and M. Tresch, “Updatable Views in Object-Oriented Databases,” in *Proc. DOOD Conference,* Germany, Dec. 1991.

[27] G. Scholz and W. Wilkes, “Information Modelling of Folded and Unfolded Design”, in *Proc. European Design Automation Conference (EDAC),* 1992.

[28] G. Zimmermann, “PLAYOUT - A Hierarchical Design System,” in *Information Processing 89,* Elsevier Science Publishers B.V. (North Holland), 1989.