JUNE 1988 VOL. 11 NO. 2

a quarterly bulletin of the Computer Society technical committee on

Data Engineering

CONTENTS

SPECIAL ISSUE ON WHATEVER HAPPENED TO SEMANTIC MODELING

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC
IEEE

THE COMPUTER SOCIETY

The LOTUS Corporation has made a generous donation to partially offset the cost of printing and distributing four issues of the Data Engineering bulletin.

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering . Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Letter from the TC Chair

My two years as the chair of the Technical Committee on Data Engineering are about over. I am pleased to announce the appointment of Larry Kerschberg as my sucessor. Larry is Professor of Information Systems and Systems Engineering at George Mason University in Fairfax, Virginia. His research interests include expert database systems, knowledge support systems, and systems engineering.

We have increased the scope and improved the quality of services to our members over the last two years. Below are some of our accomplishments:

- We have attempted to better serve the technical interests of our members by sponsoring or cosponsoring relevant conferences, symposia, and workshops. This has increased the TC visibility and resulted in addition of several members to its rolls.
- The TC was renamed to Technical Committee on Data Engineering to more accurately reflect the evolving nature of database technology.
- **Data Engineering** is now being published in a timely manner (thanks to Won Kim and the Associate Editors). We were successful in getting the financial support from the IEEE Computer Society to publish five issues last year, and I am pleased to announce that we have received a very generous donation from the **Lotus Development Corporation** to help meet the publication costs this year. Now that Lotus has broken the ice, I sincerely hope that other corporate sponsors will come forward and offer financial help.
- Our TC played a vital role in the development of the proposal for the new **Transactions on Knowledge and Data Engineering** and its approval by the Computer Society. It is scheduled to begin publication in July, 1989 on a quarterly basis.

One issue that still remains unresolved is the establishment of a TC membership dues mechanism. It will not only provide a solid financial basis for the TC activities but strengthen the ties between the TC and its members. A task force headed by Mario Barbacci has been set up to address this issue, and I will continue my involvement as one of its members.

As you can see, our TC is in good shape. I look forward to Larry Kerschberg's leadership with optimism. Under his leadership, it is assured of a bright future. Quoting Oliver Wendell Holmes, "the greatest thing in the world is not so much where we stand as in what direction we are moving."

Finally, I would like to mention the establishment of a new home for database proposals within the Division of Information, Robotics, and Intelligent Systems at the National Science Foundation. The Knowledge and Database Program, which was established in FY 1986, was recently divided into two programs: the Knowledge Models and Cognitive Systems Program and the Database and Expert Systems Program. Each has its own research community, and each now has its own Program Director. For additional information, you may contact Dr. Y. T. Chien, IRIS Division Director at 202-357-9572 or by email at ytchien@note.nsf.gov.

Best wishes for a healthy and stimulating summer ahead.

Sushil Jajodia
May 16, 1988

Letter from the Editor

What ever happened to Semantic Modeling?

That's the question this issue is meant to examine. Indeed, during the seventies and very early eighties, a large number of semantic models were introduced in the literature. This was followed by a wave of related work, in such areas as database physical implementation and theoretical investigations of semantic models. Researchers have discovered that semantic models, as they have natural visual representations, are an interesting medium for studying graphical interfaces to databases. And, the incorporation of behavioral specifications in semantic models has become an active area.

The papers in this issue represent some of the more recent efforts in these areas. Indeed, semantic modeling is alive and well, in the sense that it has been absorbed into the mainstream of database research. In fact, two of the papers describe commercial products.

The first two papers discuss theoretical results. Abiteboul and Hull investigate update propagation in the context of their IFO model, and Lyngbaek and Vianu study the equivalence of semantic and relational schemas, and use the IRIS model of Hewlett-Packard Laboratories as their representative semantic model.

The third and fourth papers describe physical implementations. Guck et al. give an overview of a commercially-available product from Unisys; their system is based on the SDM model of Hammer and McLeod. Weinreb et al. describe a database management system based on Shipman's Daplex model; this system also contains behavioral extensions in the spirit of object-oriented programming. It also interfaces to Symbolics Common Lisp.

The last two papers describe graphical interface tools. Rogers and Cattell discuss a commercial product based on the Entity-Relationship model of Chen. Finally, Ege describes a tool for constructing user interfaces; unlike the research projects described in the other five papers, this work is not based on a semantic model. However, it supports the semantic concepts of aggregation and specialization, and the specification of complex object types.

I would like to give the authors of these papers my sincere thanks. They were given very little time to produce and rewrite their papers, and everyone cooperated to get the issue out on time.

[illegible]

Roger King
May, 1988

Update Propagation in a Formal Semantic Model

Serge Abiteboul[1]
Institute National de Recherche
en Informatique et en Automatique
Rocquencourt, 78153
France

Richard Hull[2]
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782
USA

Abstract: The update propagation methodology of a semantic database model, IFO, is described. An informal review of IFO, a mathematically defined model which subsumes the structural components of most prominent semantic models in the literature, is also presented.

1 Introduction

There is currently wide interest in a variety of advanced database models which provide explicit constructs for structuring data. These include the semantic models [HM81,HK87,KP76], object-oriented database models [CM84,AH87b,RV87], nested relation and complex object models [Hul87,HY84,JS82], and data models for engineering design and version management [KC87,Zdo86]. In all of these models, the explicit constructs used imply that intricate semantic relationships hold between the stored data. It is critically important to perserve these relationships when the underlying data is updated.

While the general problem of determining update propagation for arbitrary collections of semantically motivated constructs is largely unsolved, some important special cases have been studied in connection with the semantic database models. Notable examples include [AH87a] and [HK81], both of which describe formal approaches for computing the propagation of updates in a semantic model. While the models and overall frameworks used in these investigations differ, both use an underlying two level paradigm which focuses on (a) local propagation through individual constructs, and (b) global propagation, i.e., how the various forms of local propagation are combined. This paper provides a brief survey of one of these investigations [AH87a], which is based on the IFO database model.

Although no one semantic model has gained the significance of, say, the relational model, several fundamental principles of semantic database modeling can be identified [AH87a,HM81,HK87,Ken79]. With regards to the structural component, these include (a) being "object-based", in that they focus on object- or entity-sets; (b) providing an explicit construct for *attributes* or stored functions; (c) providing explicit constructs for *ISA* relationships, which permit the inheritance of structural information (e.g., of attributes and type); and (d) providing hierarchical mechanisms for building object types out of other object types (primarily using the *aggregation* (tuple) and *grouping* (set) constructs). The IFO model, introduced in [AH87a], is a mathematically defined data model which supports these four principles, and thereby subsumes the structural components of most of the prominent semantic models in the literature. (Other important components of semantic models include derived data and integrity constraints; see [HK87].)

Since its introduction the IFO model has been used in a number of research investigations. It has been used to characterize the object types which can arise in semantic models [AH87a], and as the basis for an interactive, graphics-based database schema manager [BH86]. The survey [HK87] of semantic models and research uses many aspects of IFO to provide the framework for a tutorial on semantic models. Finally, a current investigation focused on the development and study of deductive query languages for semantic models [AG87,AH88] is based in part on IFO.

The intent of this paper is to give an overview of the update propagation methodology for IFO. A complete description of the methodology, along with theoretical results demonstrating its correctness, is given in in [AH87a] (see also [AH84]). The current paper has two main sections, presenting informal descriptions of the IFO model, and of the update semantics.

[0]Work supported in part by PRC BD Avancee.
[1]Work supported in part by NSF grant IST-85-11541

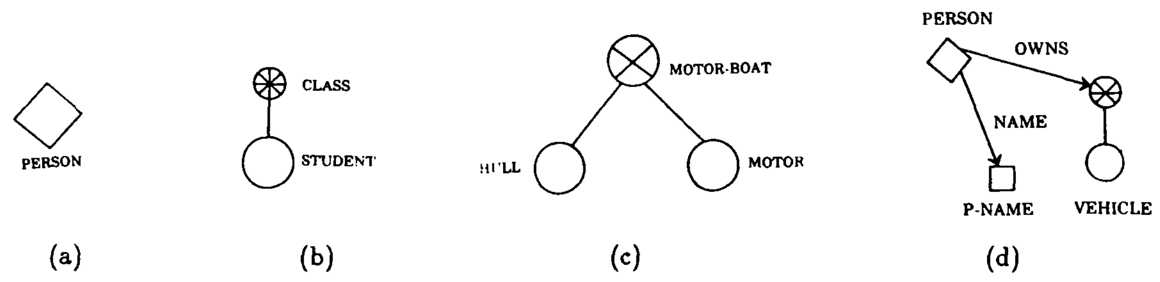


Figure 1. Three object types and a fragment

2 Informal Description of the IFO Model

This section introduces the various components of the IFO model in an informal manner. The emphasis of the discussion will be on the intuition and motivations behind the model, rather than on the precise definitions. While considering the definition of the model, it is important to note that there are restrictions on how the various constructs can be combined. These correspond to natural intuitions concerning the meaning typically associated with the constructs, and also provide the foundation for the update propagation methodology.

Briefly, an IFO *schema* is a directed graph with various types of vertices and edges. A family of IFO *instances* is associated with each schema, which correspond to the structured (or formatted) versions of data sets that can be represented using the schema. In an IFO schema, a set of object *types* are specified; attributes are associated with these types to form *fragments*; and fragments are connected using ISA relationships. Complete (albeit simple) IFO schemas are shown in Figures 2 and 3 below; illustrations of various IFO components are presented in Figure 1.

The object types of the IFO model are constructed from atomic types using *aggregation* (tuple), denoted using a ⊗-vertex, and *grouping* (set), denoted using a ⊛-vertex (see Figure 1). In IFO two atomic types are used: *printable* types are shown using squares, and *abstract* types are shown using diamonds. Abstract types correspond typically to entity sets such as PERSON that have no underlying structure (at least, relative to the database designer or user), although they may have attributes and subtypes. Also, as illustrated in Figure 1, a leaf of an IFO object type can be a *free* node (depicted with a circle). In a full IFO schema each free node has at least one associated ISA edge, through which the type of objects associated with the free node is defined. Indeed, ISA edges are closely related to the inclusion dependencies found in the relational model and referential constraints found in other models.

Speaking informally, an *instance* of the type PERSON is a finite set of people. In the formal model, a *domain* is associated with each abstract type, e.g., the domain of PERSON could be a set of abstract symbols that corresponds to the set of people in the world. An *instance* of this type would then be a finite set of these abstract symbols. Turning to the grouping construct, each (finite) set of students is an "object" having the structure of the type shown in Figure 1(b); an instance of this type is a finite set of such objects. Finally, Figure 1(c) illustrates the aggregation construct, representing the object-type MOTOR-BOAT as an ordered pair consisting of a MOTOR and a HULL. Although not shown here, the grouping and aggregation constructs can be used recursively.

The second main structural component of IFO is the direct representation of functional relationships using *fragments.* Figure 1(d) shows a simple fragment in which the abstract type PERSON serves as the underlying domain with two attributes, one yielding printable name values, and the other yielding sets of vehicles. An instance of this fragment consists in a finite set of PERSONs, along with two stored functions, one mapping this set of PERSONs to printable P-NAMEs, and the other mapping it to VEHICLE-sets. Object types with arbitrary complexity can serve as the domain or range of such attributes. In general, attributes are viewed as partial functions; constraints such as being total or 1-1 can also be incorporated. (IFO also supports *context-dependent* attributes, which are not considered here.)

The final structural component of the IFO model is the representation of ISA relationships. As in most semantic data models, these are based fundamentally on subset inclusion, and are thus somewhat different than the notion of ISA found in the object-oriented paradigm of programming languages [BKK*86,GR83]. In IFO, an ISA relationship from a type SUB to a type SUPER indicates that each object associated with SUB

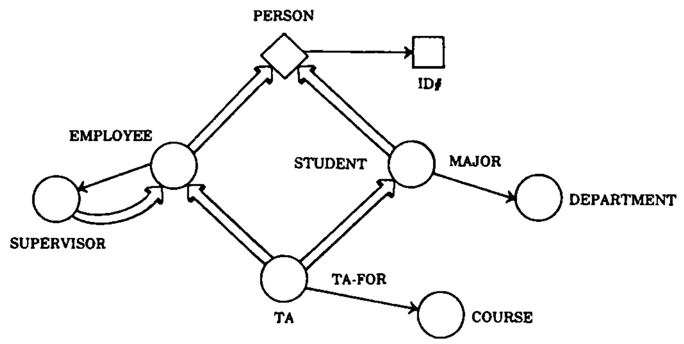


Figure 2. The TA example

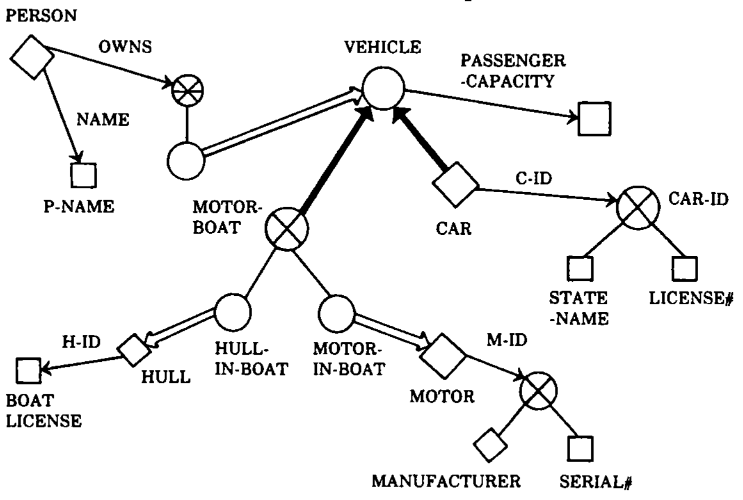


Figure 3. The vehicle example

is also associated with SUPER. This immediately implies that each attribute defined on the type SUPER is automatically defined on SUB; in this sense attributes of SUPER are *inherited* by SUB.

Because of the interplay between object structures and ISA relationships, in IFO two types of ISA relationship are distinguished: *specialization* and *generalization.* (Similar distinctions are made in several investigations, including [BLN86,Cod79,DH84].) The specialization relationship is depicted using a broad double-shafted arrow, whereas the generalization relationship is depicted using a bold-shafted arrow (see Figures 2 and 3). Speaking intuitively, *specialization* can be used to define possible *roles* for members of a given type (e.g., a person might be a student). Furthermore, an object may change such roles without changing its underlying or fundamental identity (e.g., a student might become an employee and/or cease to be an employee without losing his or her underlying identity as a person). In contrast, *generalization* represents situations where distinct, pre-existing types are combined to form new virtual types (e.g., the types CAR and MOTOR-BOAT might be combined to form VEHICLE, as in Figure 3). In such situations it is not appropriate to permit an object of one subtype to migrate into another subtype. Also, it is typical to require that a generalized supertype be *covered* by its subtypes (e.g., that in any instance the set of vehicles equals the union of the sets of cars and motor-boats).

As noted above, if there is an ISA relationship from type SUB to type SUPER, then attributes on SUPER are inherited "downwards" by SUB. In cases of specialization, object type (i.e., internal structure) is also inherited "downward," in the sense that the type of SUB is inherited from that of SUPER. On the other hand, in cases of generalization, object type is inherited "upward." By distinguishing specialization and generalization, the IFO model provides an inheritance framework in which each object has a single type (internal structure), and where both attributes and type can be inherited. Although not discussed here, it is possible to include constraints on ISA relationships, such as requiring that subtypes of a type be disjoint.

Specialization in IFO is also used to restrict attribute ranges and free vertices in types, as shown in

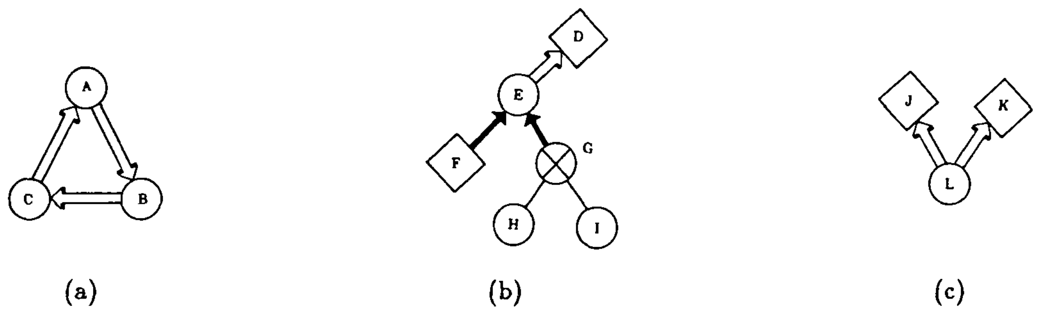


Figure 4. Three invalid IFO "schemas"

Figures 2 and 3. Thus, IFO schemas use *distinct vertices for distinct roles* of an entity set. This is a departure from most previous graphical representations of semantic data models. In IFO this approach permits a more modular view of schemas, with fragments serving as hierarchical structures which can be considered in their entirety, independent of the rest of the schema. Furthermore, this approach provides a natural framework for studying update propagation, and provides a natural basis for a graphics-based query language [BH86].

An important aspect of IFO is the articulation of resstrictions for how the various constructs can be combined. We have already discussed how the type constructors and attributes can be combined to form fragments. There is also a need for restrictions concerning ISA relationships. For example, the "schema" of Figure 4(a) suggests that A is a subtype of B, which is a subtype of C, which is a subtype of A. Thus, there is no way to determine the underlying type of any of these. Other, more complex cycles of "object determination" can arise from the interplay of ISA relationships and nonatomic types; and is prevented by rule ISA4 below. In the "schema" of Figure 4(b), the type E is a subtype of D, and so objects associated with E are supposed to be elements of the domain of type D. On the other hand, the type E is formed by unioning the types F and G, and so objects associated with E are supposed to come from either the domain of F or be ordered pairs associated with G. Thus, the specialization edge and the generalization edges cause conflicting requirements on the type E. The "schema" of part (c) has similar problems.

We now state the specific restrictions imposed on how ISA relationships can be used in IFO schemas. In addition to preventing the anomolous "schemas" of Figure 4 above, these rules are essential to the update methodology. Here a vertex schema is *primary* if it is the root of a fragment or type; these correspond to the primary entity sets stored in the database.

ISA1: Each free vertex is either the tail of at least one specialization edge, or the head of at least one generalization edge, but not both.

ISA2: For each specialization edge, the tail is free and the head is primary.

ISA3: For each generalization edge, the tail is primary and the head is both primary and free.

ISA4: (informal) There is no cycle of "object determination".

ISA5: Two directed paths of specialization edges sharing the same origin can be extended to a common vertex.

We now indicate the formal notion of *instance* of an IFO schema. Specifically, an instance of a schema **S** is a mapping **I** from the primary vertices of **S**, which maps a given primary vertex p to an instance of the fragment (or type) rooted at p, subject to two conditions (stated informally here) which enforce the intuition behind the specialization and generalization constructs: (a) if (p, q) is a specialization edge, then each object associated with p is also associated with q; and (b) if p is a generalization head, then the set of objects associated with p is equal to the union of the sets of objects associated with the vertices of $\{q \mid (q, p)$ is a generalization edge $\}$.

Although not required by the basic IFO model, we restrict our attention here to IFO schemas whose generalization relationships satisfy an additional restriction, called *separable generalization.* Informally, a schema **S** has this property if: for each maximal connected subgraph **G** of **S** involving only generalization

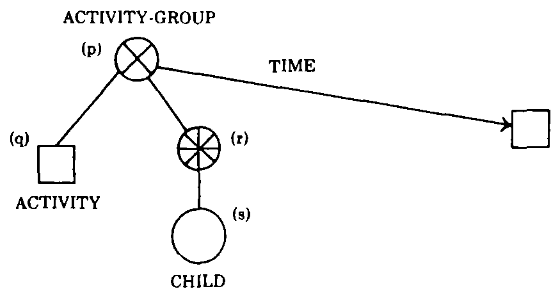


Figure 5. The kindergarden example

edges, if p and q are distinct topologically minimal vertices of **G**, then in each possible instance of **S**, the domains of p and q are disjoint. It appears that most schemas arising in practice satisfy this property; and as illustrated in [AH87a], this restrictions simplifies the semantics of update propagation through generalization relationships.

3 Update Semantics in the IFO Model

This section gives an overview of the update propagation methodology for IFO as defined in [AH87a]. The discussion focuses primarily on characterizing the complete effect of a request to perform a single change at an entity set or attribute value.

To illustrate the overall flavor of the update propagation methodology, suppose that **I** is an instance of the schema of Figure 3, and that a certain boat hull H1 is damaged beyond repair, and is to be deleted from the database. As a result of the specialization edge (HULL-IN-BOAT, HULL) and the type-construction edge (MOTOR-BOAT, HULL-IN-BOAT), this deletion will imply the deletion at MOTOR-BOAT of each motor-boat for which H1 is the first coordinate. And as a result of the generalization edge (MOTOR-BOAT, VEHICLE), each of these pairs must also be deleted from the VEHICLE vertex. Finally, if (H1,M1) $\in$ OWNS(x) for some person x, then it must be removed from this set, in effect performing the replacement OWNS(x) := OWNS(x)$-\{$(H1,m)$|$ (H1,m) is present at the VEHICLE vertex $\}$. Importantly, this illustrates how a deletion at one part of the database can lead to the modification (replacement) of an object at another part of the database.

There are three stages in the development of the update propagation methodology: (i) specification of how updates propagate within object types; (ii) the combination of this with propagation through ISA relationships; and finally, (iii) the incorporation of attributes. In each case, the discussion here first illustrates the intuitive properties which the methodology should have, and then describes how these intuitions are formalized in [AH87a]. This discussion is intended only to give the overall flavor of the methodology; many of the technical details are omitted.

3.1 Updates on Objects

In this subsection, we focus exclusively on update semantics for object types and object instances. Two fundamental issues will be discussed, these being (i) updates directed at the primary vertices of types, and (ii) indirect updates caused by updates occuring at the (nonprimary) leaves of a type (as might be implied by ISA relationships and updates to other parts of an IFO schema).

We begin with an informal discussion that illustrates both of these issues. Consider the fragment shown in Figure 5, which might be used to store information about the daily schedule of activity-groups in kindergarden. Specifically, an activity-group is an ordered pair, with the first coordinate being an activity-name (such as 'nap' or 'art') and the second coordinate being a set of children. For each activity group the function TIME gives the time at which the group will begin the given activity.

The most direct type of update to an instance I of this type is one that deletes a given activity group (say O) from the instance (yielding the new instance I−{O}). In the syntax developed in [AH87a], this deletion is requested by the triple (p, O, $\perp$). The insertion of a new activity group, say P, into I is requested by (p, $\perp$, P)). Finally, the replacement of an existing activity group O by O′ is requested by (p, O, O′). In general, if $\mathcal{M}$ is a (consistent) set of update triples directed at p and I is an instance of the ACTIVITY-GROUP type, then $\mathcal{M}$[I] denotes the *effect* of applying all members of $\mathcal{M}$ to I.

We now consider how updates at the leaves of the type in Figure 5 affect instances of that type. Intuitively, modifications and deletions to the active domain of a leaf of the type will permeate upwards in elements of the active domain of in the natural manner. For example, suppose that the kindergarden has run out of paint, and that all 'paint' groups are to become 'draw' groups. This modification can be requested by the triple (q, 'paint', 'draw'). If ('paint', {Nancy, Bobby}) is an element of an instance I, then under the update it will be replaced by ('draw', {Nancy, Bobby}). On the other hand, if the activity 'paint' is to be dropped, this is requested by (q, 'paint', $\perp$), and in the resulting instance ('paint', {Nancy, Bobby}) will simply be deleted.

The situation is more interesting if objects associated with the vertex s are to be modified or deleted. For example, suppose that Bobby has moved to another town, and so he is to be deleted. (This might be implied by a specialization edge connecting the vertex s with some other vertex modeling the townspeople). This deletion is requested by (s, Bobby, $\perp$), and has the effect of replacing ('paint', {Nancy, Bobby}) by ('paint', {Nancy}) in the set of activity groups.[3] Now suppose that there is some specialization edge from a vertex t to p. Then ('paint', {Nancy, Bobby}) must also be replaced by ('paint', {Nancy}) whenever it occurs at t. Similarly, the modification (s, Bobby, Jimmy) will will imply the modification (p, ('paint', {Nancy, Bobby}), ('paint', {Nancy, Jimmy})) in the overall instance.

In this framework the semantics of a modification is different than that of deletion followed by insertion. For example, in the above paragraph, the modification (PERSON, Bobby, Jimmy) implies the modification (p, ('paint', {Nancy, Bobby}), ('paint', {Nancy, Jimmy})). On the other hand, if Bobby is first deleted from PERSON then the modification (p, ('paint', {Nancy, Bobby}), ('paint', {Nancy}) will be applied to p. If Jimmy is then inserted at PERSON, this will have no impact on the ('paint', {Nancy}) tuple in ACTIVITY-GROUP.

The formal notation developed in [AH87a] permits the specification of multiple deletions and modifications to the leaves of object types. In particular, if **R** is a type, I and instance of **R**, and $\mathcal{M}$ a set of deletion and/or modification update triples directed at the leaves of **R**, then $\mathcal{M}$[I] is defined in the intuitively natural manner. It is shown in [AH87a] that the updates in $\mathcal{M}$ can be executed either simultaneously or leaf-at-a-time, in either case yielding the same result.

3.2 Update Propagation through ISA Relationships

In this subsection we focus on updates involving types and ISA relationships. This is the most intricate aspect of the update propagation methodology for IFO, in part because ISA relationships can carry the effect of an atomic update request to all parts of the schema.

We begin our discussion by stating two intuitively motivated Principles of Propagation, and then discuss a number of examples. The principles are:

PP1: Modification to an object at a free vertex can occur only as the result of changes to that object at a nonfree vertex.

PP2: An object can be inserted at p only if it already exists at all q such that (p, q) is a specialization edge

The first principle captures the intuition that free vertices correspond to roles that objects can have, whereas non-free vertices correspond to their principle "location" in the database. The second principle is introduced primarily to simplify the basic update propagation methodology. Relaxation of these restrictions are discussed in Subsection 3.4.

The first set of examples focuses on local update propagation resulting from specialization edges between primary vertices. Consider again the IFO schema of Figure 2. Three kinds of direct updates

[3] An alternative would be to simply remove ('paint', {Nancy, Bobby}) from the overall instance if Bobby is to be deleted. The development of update semantics presented here can easily be modified to satisfy this alternative interpretation of the implications of object deletion on sets.

involving the primary vertices in this schema are permitted. First, objects can be deleted, inserted, or modified at the vertex PERSON. If an object is deleted (or modified), then that deletion (modification) is propagated downwards (i.e., from the head of a specialization edge to its tail) in the obvious fashion. On the other hand, if an object is inserted at PERSON, then no other insertions occur. Second, objects associated with any of the vertices EMPLOYEE, STUDENT or TA can be deleted. (Direct modifications here are prohibited in virtue of PP1.) In such cases, the deletion again propagates downwards, and it has no impact on vertices above the vertex where the deletion occured. Third, an object can be inserted at the EMPLOYEE, STUDENT, or TA vertex if it is already present in the immediately more general vertices (PP2). This insertion has no propagational effect.

We now consider the propagation resulting from specialization edges eminating from the leaves of object types. We have already considered one such example, in which the deletion of the hull H1 in Figure 3 resulted in the deletion of all motor-boats (H1,m) present at the MOTOR-BOAT vertex. In general, the impact at a vertex p with underlying type **R** will be determined by the set of updates implied at the leaves of **R** by specialization edges.

We now consider the kind of local propagation resulting from generalization edges. Again referring to Figure 3, objects associated with the vertices MOTOR-BOAT and CAR can be deleted, inserted, or modified. Such updates have the obvious propagational effect on the set of VEHICLEs. On the other hand, the semantics of generalization imply that no direct insertions, deletions, or modifications be permitted at the VEHICLE vertex.

We now turn to the formal framework for capturing these intuitions. Suppose now that **S** is an IFO schema with no attribute edges, and **I** is an instance of **S**. A class of *permissible* atomic updates to **I** are defined in [AH87a]; these are atomic update triples which intuitively satisfy PP1 and PP2. As shown in [AH87a], determining whether an update triple is permissable can be done by examining the instance in a local part of the schema.

Suppose now that **I** is an instance of **S**, and that u is an update triple directed at vertex r of **S**. The strategy for computing the effect of u on **I** involves describing how $\mathbf{I}(p)$ should be modified for each primary vertex p of **S**. To do this, the methodology describes a set $\mathcal{U}(p)$ of update triples directed at p. The final instance $\mathbf{I}'$ is defined so that $\mathbf{I}'(p) = \mathcal{U}(p)[\mathbf{I}(p)]$ for each primary vertex p. In general, $\mathbf{I}'$ may violate the ISA edges of **S** and thus not be an instance of **S**. A basic result of [AH87a] states that if u is permissable, then $\mathbf{I}'$ is necessarily a valid instance.

At this point there are two possibilities. The first is to specify an algorithm for computing $\mathcal{U}$. This kind of approach is taken in connection with the Functional Data Model in [HK81]. The second possibility, taken by IFO, is to begin by specifying the properties which $\mathcal{U}$ should satisfy (UD1 through UD4 below). These provide a formal standard against which specific algorithms can be judged. Furthermore, the intuitive merits of these rules can be analyzed independently of procedural considerations.

A fundamental concept used in describing the desired properties of $\mathcal{U}$ is that of the *update propagation graph* UPG(**S**) of a schema **S** = (V, E) (with no attribute edges). This is defined to be the graph (V$'$, E$'$), where V$'$ is the set of primary and leaf vertices of **S**, and E$'$ consists in: (i) edges from the leaves to roots of object types in **S**; (ii) an edge for each generalization edge in **S**; and (iii) an edge, *in the opposite direction*, for each specialization edge in **S**. Intuitively, update propagation will proceed along the edges of UPG(**S**). Rule ISA4 imples that UPG(**S**) is acyclic for each valid IFO schema **S**.

The desired properties of $\mathcal{U}$ are specified by:

UD1: $\mathcal{U}(r) = u$.

UD2: If there is no directed path in UPG(**S**) from r to p, then $\mathcal{U}(p) = \emptyset$.

UD3: (informal) Suppose that p is a primary vertex of a type **R** in **S**, and that p is not a generalization head. Then $\mathcal{U}(p)$ is the set of update triples directed at p which are implied by the application of $\{(q, \text{O}, \text{O}') \mid q$ is a leaf of **R**, (q, t) is a specialization edge of **S**, and $(t, \text{O}, \text{O}') \in \mathcal{U}(t)\}$.

UD4: (informal) Suppose that p is a generalization head. Then $\mathcal{U}(p) = \{(p, \text{O}, \text{O}') \mid (q, p)$ is a generalization edge and $(q, \text{O}, \text{O}') \in \mathcal{U}(q)\}$.

Here, UD1 states that the only effect of u at vertex r is the application of u, and UD2 states that u affects only on vertices "above" r in the Update Propagation Graph. Rule UD3 is concerned with propagation through specialization and object construction edges, and UD4 concerns propagation along generalization edges. (UD4 is not sufficient if separable generalization is not assumed.) Note that each rule focuses on the semantics of propagation at a local level.

To summarize the development, suppose that **S**, **I**, r, and u are given as above with u permissible. Then there is exactly one mapping $\mathcal{U}$ which satisifies UD1 through UD4; and furthermore, the mapping $\mathbf{I}'$ defined by $\mathbf{I}'(p) = \mathcal{U}(p)[\mathbf{I}(p)]$ is a valid IFO instance. In [AH87a] an algorithm is given for computing $\mathcal{U}$, and hence $\mathbf{I}'$. Importantly, this algorithm computes $\mathcal{U}$ by visiting each vertex of the schema at most once.

3.3 Update Propagation on Attributes

In the absence of integrity constraints, attributes in IFO are assumed to be partial functions; they need not be defined on all elements of their domains. This permits a relatively simple characterization of propagation through attribute edges, a welcome relief after the intricacies of the previous subsections. In particular, modifications to attribute values, either direct or indirect, do not lead to further propagational effects in the schema.

Three issues must be addressed. The first concerns what happens to an attribute f if the contents of the domain of f is modified. Let p be the vertex representing the domain of f. If a new object O is inserted at p, then f(O) is assigned $\perp$, the undefined value. If an object O is deleted, the value f(O) is forgotten. Finally, if O is replaced by O$'$, then f(O) is forgotten, and f(O$'$) retains its old value if it had one, and becomes $\perp$ otherwise. (As with the propagation of object deletions in sets, alternative semantics can be developed here.)

The second issue concerns what happens if an update propagates through a specialization edge to a free node in the range of an attribute f. In such cases, all of the implied modifications $\mathcal{M}$ at free nodes in the range of f are collected. For each object O in the domain of f, the new value of $f(O)$ is computed by applying $\mathcal{M}$ to the original value of $f(O)$, using propagation through the object constructs. This was illustrated by the example concerning the impact of deleting VEHICLEs on the OWNS attribute of Figure 3. As noted in [AH87a], it is easy to extend the formalism of the previous section to include the propagation through attributes described so far.

The third issue concerns direct updates to attributes. Two kinds of updates are supported. First, an attribute value may be directly assigned, using the syntax $f(O)$:= O$'$. In order to satisfy PP2, objects in O$'$ occurring at specialization tails of its underlying type must already occur at the corresponding specialization heads. Suppose now that f is set-valued, i.e., that the type of the range of f has a $\circledast$-vertex as root. In this case insertions, deletions and modifications can be requested using the triple notation. For example, (OWNS(John), $\perp$, (H1,M1)) inserts the boat (H1, M1) into the value of OWNS(John). Again, the restrictions of PP2 must be satisfied.

3.4 Extensions and Applications

The approach taken in this section was to define the propagation that results from the simplest possible types of update requests. The framework developed here could be used to provide the first tier in a two-tier system in which the constraints PP1 and PP2 on permissible updates are relaxed. Speaking roughly, this two-tier system might permit a request to replace an object O by O$'$ at a vertex p, where (p,q) is a specialization edge and O is not currently resident at q. Although this request does not satisfy the principle PP1, it is possible to imagine an intuitively natural semantics: the system should first find the "highest" specialization ancestor r of p, and second perform the update (r, O, O$'$). A subtlety arises if the type of r is not atomic, in which case O$'$ might violate PP2 at r. This suggests that finding the true implied effect of (r, O, O$'$) may require a nested reverse traversal of the object and specialization edges in the Object Definition Graph. Updates to generalization heads might be accomodated in an analogous fashion. The problem of precisely articulating update semantics for these nonpermissible atomic updates remains open. A second open problem is to describe the effects of two or more permissible atomic updates which are requested simultaneously.

We also mention two possible applications of the results of the update propagation methodology for IFO. The first concerns the relational model. It is possible to define a subset of IFO in which there are no generalizations and in which consecutive $\circledast$-vertices are not permitted, so that all schemas of this subset can be mapped into third normal form (3NF) relational schemas with inclusion dependencies. This subset of IFO subsumes the Entity-Relationship model [Che76]. The update propagation methodology for IFO can thus be extended to a large class of naturally arising relational schemas.

The methodology can also be used in cases where integrity constraints are included in the specification of an IFO schema. In particular, a two-step approach can be used. When a permissible update u is requested of instance $\mathbf{I}$, the system can first compute the effect $\mathcal{U}$ of u on $\mathbf{I}$. Recall that $\mathcal{U}$ specifies the set of updates directed at the primary vertices that are implied by u; as a result $\mathcal{U}$ will typically have size quite small relative to the size of the database. In the second step, the system can ckeck whether the application of $\mathcal{U}$ would violate the integrity constraints. If the constraints would not be violated, then the update should be executed. The advantage of this approach is that the decision of whether or not to execute the update can be made before the database contents are modified.

4 Concluding remarks

In the preceding sections the basic elements of the IFO model and a formal update propagation methodology for it were briefly described. This exercise yields a couple of important lessons.

The first concerns the development of data models, both semantic models and those for more complex application environments (e.g., engineering design or manufacturing). It is important that such models include a clear statement of how the basic constructs can be combined. The IFO model provides such an articulation for a relatively limited set of constructs. This can serve as a skeleton for the incorporation of other simple constructs (e.g., list) and for integrity constraints. However, the incorporation of more complex constructs such as version objects may require a considerable rethinking of the rules for construct combination.

The articulation of how constructs are combined is essential to the development of a comprehensive update propagation methodology. The methodology described here has another fundamental feature. Specifically, effort was made to separate the semantics of the methodology from algorithmic and procedural considerations. This is illustrated most clearly by the rules UD1 through UD4 above, which describe the properties desired of the methodology in connection with ISA and object construction relationships. In the context studied here the specification of these abstract properties is relatively simple; only four rules are needed, and they can be satisfied using a straight-forward algorithm.

The presence of more sophisticated constructs, e.g., version and other engineering objects, may yield rules and computational procedures which are considerably more complicated. The problem of formally characterizing update propagation in the presence of such semantically-motivated constructs remains a challenging and largely unresolved problem.

References

[AG87] S. Abiteboul and S. Grumbach. COL: a language for complex objects based on recursive rules (extended abstract). In *Proc. Workshop on Database Programming Languages*, Roscoff, France, September 1987.

[AH84] S. Abiteboul and R. Hull. *IFO: A Formal Semantic Database Model.* Technical Report TR-84-304, Univ. of Southern Calif., Computer Science Dept., April 1984.

[AH87a] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4), Dec. 1987.

[AH87b] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In *Proc. Conf. on Object-oriented Programming Systems, Languages and Applications*, pages 430–440, 1987.

[AH88] S. Abiteboul and R. Hull. Data functions, DATALOG, and negation. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1988. to appear.

[BH86] D. Bryce and R. Hull. SNAP: A graphics-based schema manager. In *Proc. of IEEE Conf. on Data Engineering*, pages 151–164, 1986.

[BKK*86] D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: merging Lisp with object-oriented programming. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–8, September 1986.

[BLN86] C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[Che76] P.P. Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, March 1976.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, 1984.

[Cod79] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. on Database Systems*, 4(4):397–434, December 1979.

[DH84] U. Dayal and H.Y. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Trans. on Software Engineering*, SE-10(6):628–644, 1984.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[HK81] M.S. Hecht and L. Kerschberg. *Update semantics for the functional data model.* Technical Report, Bell Laboratories, Holmdel, N.J., January 1981.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, June 1987. to appear.

[HM81] M. Hammer and D. McLeod. Database description with SDM: a semantic database model. *ACM Trans. on Database Systems*, 6(3):351–386, 1981.

[Hul87] R. Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*, pages 193–256, Academic Press (London), 1987.

[HY84] R. Hull and C. K. Yap. The Format model: A theory of database organization. *Journal of the ACM*, 31(3):518–537, 1984.

[JS82] B. Jaeschke and H. J. Schek. Remarks on the algebra of non first normal form relations. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1982.

[KC87] R. H. Katz and E. Chang. Managing change in a computer-aided design database. In *Proc. of the 13th Conf. on Very Large Data Bases*, pages 455–462, 1987.

[Ken79] W. Kent. Limitations of record-based information models. *ACM Trans. on Database Systems*, 4(1):107–131, January 1979.

[KP76] L. Kerschberg and J.E.S. Pacheco. *A Functional Data Base Model.* Technical Report, Pontificia Universidade Catolica do Rio de Janeiro, Rio de Janeiro, February 1976.

[RV87] P. Richard and F. Velez. An object-oriented formal data model. In *Proc. of Workshop on Data Base Programming Languages*, Roscoff, France, September 1987.

[Zdo86] S.B. Zdonik. Version Management in an Object-Oriented Database. In *Proceedings of the International Workshop on Advanced Programming Environments*, pages 405–422, IFIP WG 2.4, Trondheim, Norway, June 1986.

Relational Translations of Semantic Models: A Case Study Based on Iris

Peter Lyngbaek
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, California 94304

Victor Vianu(*)
Department of Electrical Engineering
and Computer Science
University of California, San Diego
La Jolla, California 92093

ABSTRACT

The connection between semantic database models and the relational model is formally investigated using the Iris Data Model. The results focus on properties of relational schemas that are translations of Iris schemas. Two new types of constraints, cross-product constraints and multiplicity constraints are introduced to characterize the relational translations of Iris schemas. The connection established between Iris and relational schemas also yields new, unexpected information about Iris schemas. In particular, a notion of equivalence of Iris schemas is defined using their relational translations, and a result is obtained on simplifying the type structure of Iris schemas.

1. Introduction

Relational database technology and semantic data modeling have been two major areas of database research in recent years. Relational database technology is based on solid theoretical grounds, and it is understood what constitutes a well-designed relational database schema. Semantic data modeling, on the other hand, provides a rich set of data abstraction primitives which can capture additional semantics of the application in the database schema. So far, relational database technology and semantic modeling have evolved almost separately. In practice, however, some of the better known semantic database systems have been implemented on top of existing relational systems [Ca83,TsZa84], or implemented by techniques that are similar to the ones used in relational systems (e.g., query optimization techniques). This has lead to a need for establishing and understanding connections between semantic models and the relational model. The present paper is an attempt to formally investigate this connection. The work described here is reported in more detail in [LyVi87].

The model chosen for the investigation is the Iris Data Model, a semantic model developed and implemented at Hewlett-Packard Laboratories using relational database techniques. Our approach is based on defining formally a translation of Iris schemas into relational schemas and characterizing the relational schemas corresponding to Iris schemas. Establishing this formal correspondence has several significant benefits:

1. The special properties of relational translations of semantic schemas are identified and can be used to obtain more efficient implementations.
2. Semantic database interfaces can be constructed for existing relational databases satisfying certain conditions which are described.
3. Questions concerning the semantic model can be re-formulated in terms of the relational model, where powerful theoretical tools are already available.
4. Relational translations of semantic models can be used to compare different semantic schemas and address issues such as equivalence and simplification of semantic schemas.

The results obtained in this paper focus on understanding the properties of relational translations of Iris schemas. After establishing the formal mapping between Iris schemas and relational schemas, we characterize the relational schemas which are translations of Iris schemas in terms of the constraints they satisfy. The constraints used are unary inclusion dependencies and two new types of constraints called "multiplicity constraints" and "cross- product constraints". It is shown that these constraints are not finitely axiomatizable. Relational translations of Iris schemas are then investigated with respect to Normal Forms. It is shown that such translations are not generally in Boyce-Codd Normal Form due to the presence of unexpected "side-effect" functional dependencies. A restriction on Iris schemas is then inferred, which guarantees that their relational translations are in BCNF and have additional desirable properties. Moreover, the same restriction is shown to guarantee that a certain finite set of inference rules for the

(*) This author was supported in part by the NSF under grant number IST-8511538.

constraints in the relational translations is sound and complete.

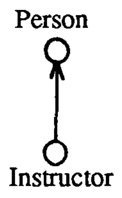
Finally, relational transactions are used to obtain new insight into properties of Iris schemas. A notion of equivalence of Iris schemas is defined based on their relational translations. This is then used to obtain a result on simplifying the type structure of Iris schemas. Other issues are briefly investigated, such as non-trivial satisfiability of Iris schemas, localeness of Iris schemas, hidden type aliasing, and hidden restrictions on the cardinality of types of Iris schemas.

2. The Iris Data Model

The Iris Data Model[1] is an example of semantic data models that supports high-level structural abstractions. The roots of the model can be found in previous work on Daplex [Sh81], the Taxis language [MyBeWo80], and earlier work at HP Laboratories on the Integrated Data Model [BeFe83]. The Iris Data Model is further described in [DeKeLy85, LyKe86, Fi87, LyDeFiKeRy87].

The Iris Data Model is based on *objects, types,* and *functions*[2]. Objects represent things and concepts from an application domain, types are sets of objects that share common properties, and functions define properties of objects. An Iris schema is defined by types and functions represented as a directed graph together with a set of constraints.

Types are uniquely named sets of objects. Types such as Integer and String are *literal types* (represented by squares). Their instances are directly representable (i.e, printable). *Non-literal types* (represented by circles) contain objects that are internally represented by surrogates. Such objects correspond to entities from the application domain that cannot be represented directly in external form. For example, non-literal objects can model persons, departments, and bank accounts. Non-literal types may overlap; for example, an object which represents a given person may be an instance of the types Employee, Taxpayer, and Subscriber. Non-literal types are organized in a type structure that supports generalization and specialization [SmSm77]. The type structure represents subtype/supertype relationships. If a given type is a subtype of another type (represented by a directed edge from the subtype to the supertype), then all the instances of the subtype are also instances of the supertype. In the example below, the type Person is the supertype of the type Instructor:



A given type may have multiple subtypes. The subtypes may be overlapping and they do not necessarily partition the supertype. A type may also have multiple supertypes. In that case, each object of the subtype must belong to all the supertypes.

Properties of objects are expressed in terms of functions, which are defined over types. A function may have any number of argument and result types. For example, the function

Enrollment: Student -> Course × Grade/String

is defined on Student objects and returns pairs of Course and Grade objects. The Iris functions considered in this paper[3] are implemented by storing the graphs of the functions. They may possibly be multi-valued and are therefore not functions in the mathematical sense. Rather, they are relations defined over the cross-products of their argument and result types. A function is graphically represented by a labeled edge that connects two cross-product vertices. The cross-product vertex in which the function edge originates is connected to all the types in the domain of the function and the cross-product vertex in which the function edge terminates is connected to all the types in the range of the function. For example, the function

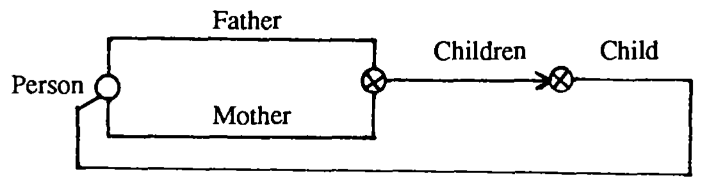
Children: Father/Person × Mother/Person -> Child/Person

is graphically illustrated below:

[1] The model described in this paper is a subset of the actual Iris Model which is currently implemented. The full model is object-oriented and supports behavioral abstractions.

[2] Types and functions are themselves represented as objects [LyKe86].

[3] In addition to functions whose graphs are stored, the Iris Data Model supports derived functions, computed functions, and functions with side-effects.



The objects that are interrelated by a function play certain roles in that function. In Children, for example, one Person object plays the role of a father, another the role of a mother, and a third object the role of a child. In order to be able to distinguish the different roles from one another, the roles are uniquely labeled (represented by labels on the edges from the type vertices to the cross-product vertices). If a role name is not explicitly specified it defaults to the name of the associated type.

A function is defined not only on the types explicitly mentioned in the function definition, but also on the subtypes of those types. This is referred to as *inheritance*. For example, the function Children defined on Person objects are automatically defined on Instructor objects.

In order to capture the precise semantics of a given application, a data model should support the distinction between multi-valued and single-valued functions and partial and total functions. In the Iris Data Model, the more general concept of *object participation* serves that purpose. As an example, consider the function Major: Student -> Department which is typically partial (i.e., a student is not required to have a major) and single-valued (i.e., a student has at most one major). These requirements can be defined in the Iris schema by specifying a *lower object participation* (LOP) of zero and an *upper object participation* (UOP) of one for objects playing the Student role in the function Major: Student [0,1]. The LOP specifies the minimum number of times each Student object must participate in the relation defined by Major. Since the LOP is zero, a Student object is not required to be related to a Department object. The UOP specifies the maximum number of times each Student object can participate in the relation defined by Major. Since the UOP is one, a Student object can be related to at most one Department object. Lower and upper object participations, which are referred to as *participation constraints*, can be defined for any subset of the argument and result roles of a function. Thus, a participation constraint on the two roles Student and Course of the function Enrollment can limit students to enroll only once in a given class: Enrollment: Student, Course [0,1]. LOP values are restricted to zero and one; UOP values must be positive integers or the special value ∞. If a participation constraint has not been explicitly specified for a given subset of roles in a function it is assumed to be [0,∞] (i.e., participation is not restricted). For literal types with infinite domains, e.g., Integer and String, the LOP must be zero. When several participation constraints are defined on the same function it is important to ensure that they are consistent with each other.

In addition to supporting semantic integrity control, participation constraints are important for query optimization and organization of data on physical storage devices.

A rigorous definition of Iris schemas is provided in [LyVi87]. Formally, an Iris schema is a directed, labelled graph together with a set of participation constraints. The graph specifies the types, functions, and subtype/supertype relationships defined for a given Iris database. The set of constraints specifies the participation constraints imposed on the roles and functions of the graph. The definition is closely related to the formal definition of the IFO model [AbHu84].

We now illustrate the Iris model by describing a schema of an example database. The database is used by an educational institution to keep track of students, courses, instructors, enrollments, and teaching assignments.

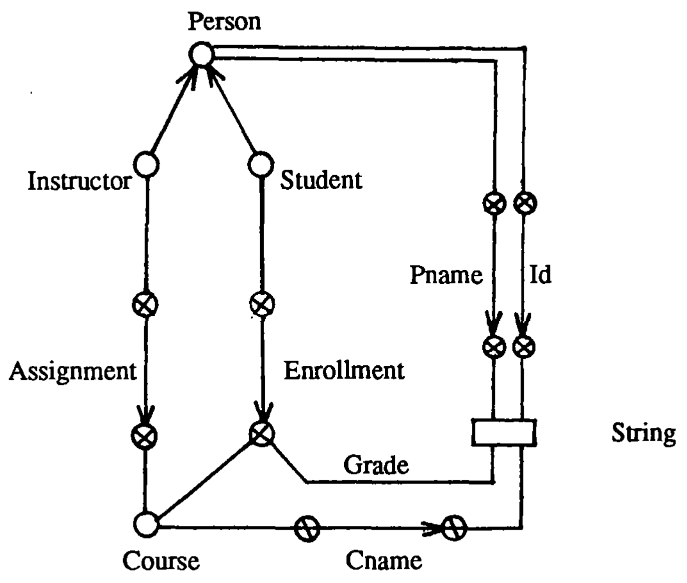
Example 2.1. The Iris schema of the example database contains the non-literal types Person, Instructor, Student, and Course, together with the literal type String. Person objects represent people affiliated to the educational institution, such as students and instructors. The types Student and Instructor are subtypes of Person. The instances of the type Course correspond to courses offered by the educational institution. Properties of the objects in the example application are captured in the database schema by the following functions:

Pname: Person -> String	Id: Person -> String	Assignment: Instructor -> Course
Cname: Course -> String		Enrollment: Student -> Course × Grade/String

The functions represent the facts that persons have names, persons have identification numbers, courses have names, students obtain grades for the courses in which they enroll, and instructors are assigned to courses.

Many kinds of requirements that are appropriate in an educational institution are specified in the database schema as participation constraints (Figure 2.1). For example, the requirement that every person must have at least one name is specified by the constraint C1. The constraints C2 and C3 specify that each person must have a unique identification number: constraint C2 specifies that every person must have exactly one identification number, and constraint C3 specifies that a given identification number is assigned to at most one person. Course names usage is also restricted by two participation constraints: each course is required by C4 to have exactly one course name, and all course names are required by C5 to be unique. Enrollments of students in courses are restricted by C6, C7, and C8: each course must be offered and the class sizes cannot exceed a maximum of 24 students (C6); students are limited to five courses (C7); but they are allowed to enroll only once in a given course (C8).

Figure 2.1 shows the graphical representation of the Iris schema and the participation constraints for the educational institution database. Iris schemas that have many types and functions tend to be complicated to represent graphically. However, database interfaces that support graphical schema representations should allow end-users to selectively view and manipulate small pieces of a schema.



Pname: Person [1,∞] (C1)
Id: Person [1,1] (C2)
Id: String [0,1] (C3)
Cname: Course [1,1] (C4)
Cname: String [0,1] (C5)
Enrollment: Course [1,24] (C6)
Enrollment: Student [0,5] (C7)
Enrollment: Student, Course [0,1] (C8)

Figure 2.1: Graph of Iris Schema and Participation Constraints for Educational Institution Example

The example Iris schema supports queries such as "In what courses did Nancy Slick receive the grade A?" It is beyond the scope of this paper to formally define the Iris query language.

3. Mapping Iris Schemas to Relational Schemas

The Iris model is currently implemented using relational database techniques. Thus, every Iris schema is mapped to a relational schema with appropriate constraints, and every Iris instance is implemented as a corresponding relational instance. Iris queries are translated into relational select-project-join queries, and Iris updates become relational transactions. The usual tools of relational databases can be used to improve schema design, perform query optimization, and maintain database integrity in the course of updates. In this section we formally examine the correspondence between Iris schemas and relational schemas. We define a mapping[4] of Iris schemas to relational schemas and characterize all relational schemas which are translations of Iris schemas. The relational translation of Iris schemas is then used to define a simple notion of equivalence of Iris schemas. Based on this notion of equivalence, we exhibit a result concerning simplification of the type structure of Iris schemas.

We first present some basic concepts and notation of the relational model, which will be used in the rest of the paper. We assume familiarity with the notions of attribute, domain, tuple, relation over a finite set of attributes, projection of a relation or a set of constraints on a set of attributes, functional dependency (fd), inclusion dependency (id) and Boyce-Codd Normal Form (BCNF), as in [Ma83, Ul82]. A relation schema is a triple <RelName, R, Σ> where RelName is a symbol called the relation name, R is a set of attributes and Σ is a set of constraints over R. A (relational) database schema is a pair <S,Σ> where S is a set of relation schemas with distinct names and Σ a set of inter-relational constraints involving relations in S. The logical closure of a set Σ of constraints is denoted by Σ*. If Σ consists of fd's or unary id's, its logical closure can be computed using known sets of inference rules [Ul82, CFP82]. We will use two types of constraints in addition to fd's and id's. These new constraints arise through the translation of the participation constraints of Iris schemas, and are defined next.

[4] The mapping defined here is a simplification of the mapping used by the Iris prototype implementation.

Definition. Let R be a finite set of attributes. A cross-product constraint over R is an expression $\otimes X$ where $X \subseteq R$. A relation r satisfies $\otimes X$ iff $\Pi_X(r) = \times_{A \in X} \Pi_A(r)$ ($\times$ is the cross-product operation[5]). A multiplicity constraint is an expression $X[\leq k]$ where $X \subseteq R$ and $1 \leq k \leq \infty$. A relation r over R satisfies $X[\leq k]$ if no more than k tuples in r have identical projections on X.

Thus, a cross-product constraint over a subset of the attributes of a relation requires that every combination of values of those attributes appear in the relation; a multiplicity constraint places an upper bound on the number of times each combination may appear in the relation. For example, the relation represented in Figure 3.1 satisfies $\otimes BC, BC[\leq 1]$, $AB[\leq 2]$, and $A[\leq 3]$. It does not satisfy $\otimes AB$ or $C[\leq 1]$.

A	B	C
0	0	1
0	0	0
0	1	0
1	1	1

Figure 3.1

We next describe the mapping of Iris schemas to relational schemas. The relational translation of an Iris schema s will be denoted by RelTrans(s). Consider an Iris schema s. The mapping is based on associating with each function edge of s a relation whose attributes are the role labels of the edges connecting the two cross-product vertices of the function edge to corresponding type vertices. For instance, the function Enrollment: Student -> Course × Grade/String of Example 2.1 is represented by a relation with attributes Student, Course, and Grade. Next, subtyping and participation constraints of s induce constraints in RelTrans(s). Specifically, participation constraints induce intra-relational cross-product and multiplicity constraints, and subtyping (together with participation constraints) induces intra and inter-relational unary inclusion dependencies (a detailed example is presented later).

The translation of an Iris schema to a relational schema proceeds in two stages, denoted RelTrans_1 and RelTrans_2. In the first stage of the translation we will have, in addition to relations corresponding to functions, one relation corresponding to the cross-product of all non-literal types, called the "types" relation. While the relational schemas provided by RelTrans_1 are capable of representing the same information as Iris schemas, they are not entirely satisfactory because they cannot be characterized solely in terms of the kind of integrity constraints used in the schemas. This is due to the presence of the "types" relation, which is distinguished from the other relations in that it can not satisfy arbitrary cross-product and multiplicity constraints and plays a special role in the schema. The second stage of the translation, denoted RelTrans_2, eliminates this inconvenience by removing the "types" relation[6] and adding to the remaining schema the inclusion dependencies which follow from the presence of the types relation. The final result of the translation is the composition of RelTrans_1 and RelTrans_2, denoted RelTrans. The formal definition of RelTrans_1 and Reltrans_2 is provided in [LyVi87]. We now illustrate the translation.

Example 3.1. We will show how to translate a sub-part of the schema defined in Example 2.1 to a corresponding relational schema using mappings RelTrans_1 and RelTrans_2. The Iris sub-schema to be translated contains all the types of the original schema, the functions Id and Enrollment, and all the participation constraints defined on those two functions.

Applying RelTrans_1 produces the "types" relation and the function relations Id and Enrollment illustrated in Figure 3.2.

The inclusion dependencies, cross-product constraints, and multiplicity constraints induced by RelTrans_1 are shown in Figure 3.3.

Applying RelTrans_2 to the relations and constraints of Figure 3.2 and Figure 3.3 produces the relational schema of Figure 3.4. The "types" relation is eliminated together with all its inclusion dependencies. Notice how the only inclusion dependency of the schema in Figure 3.4 is deduced from the closure of the inclusion dependencies of Figure 3.3.

Remarks. (i) While RelTrans_1 is a one-to-one mapping from Iris schemas into relational schemas, RelTrans is not one-to-one. Thus, several distinct Iris schemas can be mapped to the same relational schema by the RelTrans mapping.

(ii) For each Iris schema s, there is a straightforward correspondence between the instances of s and the instances of the relational schema RelTrans(s). The correspondence is not one-to-one, because objects which do not participate in any Iris function are not represented in the relational translation.

[5] Note that cross-product constraints are special cases of join dependencies.

[6] The current Iris implementation maintains types in stored relations.

types

Person	Instructor	Student	Course

Id

Person	String

Enrollment

Student	Course	Grade

Figure 3.2: Relations produced by RelTrans_1

$[\text{Instructor}_{\text{types}}] \subseteq [\text{Person}_{\text{types}}]$
$[\text{Student}_{\text{types}}] \subseteq [\text{Person}_{\text{types}}]$
$[\text{Person}_{\text{Id}}] \subseteq [\text{Person}_{\text{types}}]$
$[\text{Person}_{\text{types}}] \subseteq [\text{Person}_{\text{Id}}]$
$[\text{Student}_{\text{Enrollment}}] \subseteq [\text{Student}_{\text{types}}]$

$\otimes \text{Person}_{\text{Id}}$
$\otimes \text{Course}_{\text{Enrollment}}$
$\text{Course}_{\text{Id}}[\leq 1]$
$\text{String}_{\text{Id}}[\leq 1]$
$\text{Course}_{\text{Enrollment}}[\leq 24]$

$[\text{Course}_{\text{Enrollment}}] \subseteq [\text{Course}_{\text{types}}]$
$[\text{Course}_{\text{types}}] \subseteq [\text{Course}_{\text{Enrollment}}]$
$\text{Course}_{\text{Enrollment}}[\leq 24]$
$\text{Student}_{\text{Enrollment}}[\leq 5]$
$\text{Student}_{\text{Enrollment}}\text{Course}_{\text{Enrollment}}[\leq 1]$

Figure 3.3: Constraints produced by RelTrans_1

(iii) The mapping RelTrans describes the correspondence between Iris schemas and relational schemas at the logical level. The actual implementation of Iris schemas differs slightly from this description. For instance, a record is maintained of the objects which are members of each non-literal type. Also, different relations are sometimes "clustered" to improve performance.

Id

Person	String

Enrollment

Student	Course	Grade

$[\text{Student}_{\text{Enrollment}}] \subseteq [\text{Person}_{\text{Id}}]$

$\otimes \text{Person}_{\text{Id}}$
$\otimes \text{Course}_{\text{Enrollment}}$

$\text{Course}_{\text{Id}}[\leq 1]$
$\text{String}_{\text{Id}}[\leq 1]$
$\text{Course}_{\text{Enrollment}}[\leq 24]$
$\text{Student}_{\text{Enrollment}}[\leq 5]$
$\text{Student}_{\text{Enrollment}}\text{Course}_{\text{Enrollment}}[\leq 1]$

Figure 3.4: Relational schema produced by RelTrans_1 followed by RelTrans_2

The following result characterizes all relational schemas which are images of Iris schemas under the RelTrans mapping. As suggested earlier in this section, the characterization involves solely the kinds of constraints present in the schema.

3.1 Theorem. For each relational database schema r whose only constraints are unary inclusion dependencies, cross-product constraints, and multiplicity constraints, there exists an Iris schema s such that RelTrans(s) = r.

The proof of Theorem 3.1 consists of finding a mapping which, for every relational schema r of the type described in the theorem, provides a translation SemTrans(r) to a (semantic) Iris schema such that RelTrans(SemTrans(r)) = r. The main component of the construction of SemTrans(r) consists of finding an assignment of attributes (roles) to types. In particular, one type is associated with all attributes belonging to a cycle of inclusion dependencies in the logical closure of the constraints of r. Consequently, the subtyping structure of the resulting Iris schema is guaranteed to be acyclic. Furthermore, it can be shown that SemTrans(r) has the property that hidden type "aliasing" cannot occur, that is, two types of the schema cannot be forced to always contain the same objects in every valid instance of the schema. (It can be verified that, in arbitrary Iris schemas, "aliasing" can indeed occur, even though the original sub-typing specification is acyclic - see Example 3.4.)

The mapping of Iris schemas to relational schemas provides a new, formal mechanism for comparing different Iris schemas. In particular, various notions of equivalence of Iris schemas can be defined based on their relational translations. We next define the simplest such notion of equivalence.

Definition. Two Iris schemas s_1 and s_2 are equivalent if and only if[7] $(\text{RelTrans}(s_1))^* = (\text{RelTrans}(s_2))^*$.

As a first application of the above definition we next compare different Iris schemas with respect to their type structure. In particular, it may be of interest to simplify the type structure of a given Iris schema according to certain criteria, such as the number of different types in the schema. This leads to the following.

Definition. An Iris schema s is type-minimal iff there is no equivalent Iris schema s′ with fewer types.

The following result shows that there is exactly one type-minimal Iris schema equivalent to a given Iris schema.

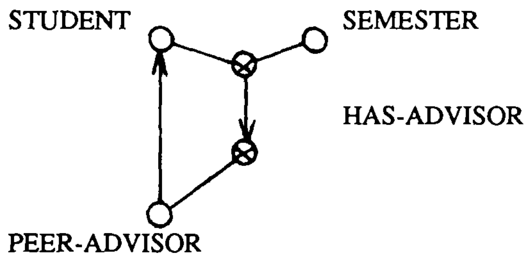
3.2 Theorem. If s_1 and s_2 are equivalent, type-minimal Iris schemas, then they are isomorphic.[8]

It can be shown that each Iris schema which is the translation SemTrans(r) of a relational schema r is type-minimal. In particular, we can use the relational translation of an Iris schema to find an equivalent Iris schema which is type-minimal. Indeed, we have:

3.3 Theorem. For every Iris schema s, SemTrans (RelTrans(s)) is equivalent to s and type-minimal.

In practice, the above results on type minimality can be used as guidelines for designing the type structure of an Iris schema. Indeed, it may be the case that the initial breakdown of objects into types is unnecessarily refined, in that the schema does not take advantage of certain type distinctions. In this case some of the types can be merged. Following is a simple example of a schema where type simplification is possible due to aliasing.

Example 3.4 The Iris schema represented in Figure 3.5 is a fragment of a schema for an academic institution, where each student must be assigned a peer advisor for each semester of the academic year. The schema contains the three types: STUDENT, PEER-ADVISOR, and SEMESTER. Each peer advisor is a student, so PEER-ADVISOR is a subtype of STUDENT. The function HAS-ADVISOR assigns a peer-advisor for every student-semester pair.



STUDENT SEMESTER $[1,\infty]$
SEMESTER PEER-ADVISOR $[0,1]$

Figure 3.5

The first participation constraint indicates that each student must be assigned an advisor each semester; the second indicates that a peer advisor can only advise one student in a given semester. It can be easily verified that, due to these constrains, each student must also be a peer advisor. Thus, the types STUDENT and PEER-ADVISOR are aliases (they contain exactly the same objects). It follows that PEER-ADVISOR can be eliminated as a type and kept simply as a role edge for students. The resulting schema (Figure 3.6) is equivalent to the first, and is type-minimal.

4. Integrity Constraints in Relational Translations of Iris Schemas

In this section we further discuss the integrity constraints induced by Iris schemas on their relational translations. We also look at the problems of side-effect fd's, non-trivial satisfiability, and localeness of Iris schemas and their translations.

[7] For each relational database schema s, s* denotes the schema containing the same relations as s and whose constraints are the logical closure of the constraints of s.

[8] Informally, two Iris schemas are isomorphic if they are the same except for the choice of labels. We omit the formal definition here.

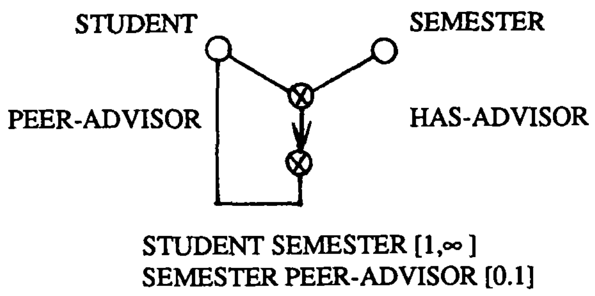


Figure 3.6

Since Iris schemas are implemented using relational translations, the constraints holding in the translations are of paramount importance for reasons familiar from relational database theory (semantic integrity, query optimization, schema improvement). As shown in the previous section, relational translations of Iris schemas can be characterized using unary inclusion dependencies, cross-product constraints, and multiplicity constraints. Thus, it is important to develop mechanisms for computing the logical closure for these constraints, such as a finite set of inference rules. Additionally, it is of interest to infer all functional dependencies implied by such sets of constraints, in order to verify that the relational translations are in normal form (BCNF). Suprisingly, our results show that arbitrary Iris schemas are not well-behaved with respect to the above goals, and that other unexpected problems arise. Indeed, we have:

4.1 Theorem. There is no finite set of inference rules for computing the logical closure of a set of (inter and intra-relational) unary inclusion dependencies, (intra-relational) cross-product constraints and (intra-relational) multiplicity constraints.

While there is no complete finite axiomatization for our constraints, it is desirable to have as powerful an inference mechanism as possible for such constraints. In [LyVi87] we provide a powerful (but incomplete) inferencing mechanism for the constraints. The inferencing mechanism is novel in that it involves a set of equations associated with the constraints, and is more powerful than any finite set of traditional inference rules.

We have seen that relational translations of arbitrary Iris schemas give rise to certain unexpected problems concerning axiomation of constraints. We now briefly discuss some additional problems arising in this context.

(i) Side-effect fd's.

It is of interest to know whether or not relational translations of Iris schemas are in Boyce-Codd Normal Form. Based on intuitive grounds, it has so far been assumed that the only fd's satisfied in these translations are key fd's which arise from explicitly constraining upper-bound participations to 1. In other words, given a relational database s corresponding to an Iris schema, it was assumed that the only fd's holding in a relation schema $<rel, R, \Sigma>$ of s are of the form $X \rightarrow R$, where $X[\leq 1]$ is in Σ. We informally call all other fd's which might hold in such a schema side-effect fd's. It is shown in [LyVi87] that side-effect fd's which violate BCNF do occur in translations of Iris schemas.

Obviously, it is of interest to infer all fd's satisfied by relational translations of Iris schemas. However, no finite set of inference rules exists for computing the fd's implied by the constraints involved in the relational translations.

(ii) Non-trivial satisfiability.

Clearly, each set of unary id's, cross-product constraints and multiplicity constraints is trivially satisfiable by a database instance consisting of a single tuple in each relation. We call a database schema non-trivially satisfiable if it has instances other than the single-tuple instance. (Thus, the original Iris schema has instances containing more than one object in each type.) It can be seen that there are relational translations of Iris schemas which are not non-trivially satisfiable. Some of these schemas can be detected using the set of equations used in the inference mechanism introduced in [LyVi87]. However, it can be shown that there are schemas which are not non-trivially satisfiable, but this cannot be detected using the equations.

(iii) Type cardinality restrictions.

Some Iris schemas imply restrictions on the maximum number of objects of a given type. For instance, consider the set of constraints $\{ \otimes AB, B[\leq 5] \}$. It is easily seen that the A column can contain at most 5 distinct values, and the type corresponding to the role A in the original Iris schema may contain at most 5 objects.

(iv) Non-localness.

We call an Iris schema local if the participation constraints for different functions are independent. It can be seen (using relational translations) that not all Iris schemas are local. Thus, participation constraints for one function may imply participation constraints for another function. This phenomenon can be undesirable and may lead to complications.

In the remainder of the section we present a restriction on Iris schemas and their relational translations which guarantees that the schemas are well-behaved with respect to all the criteria discussed so far, in particular finite axiomatization and normal forms. Clearly, different such restrictions can be found. The one we present here has the advantage of being relatively weak and the disadvantage of being rather complicated. However, a variety of simpler (but stronger) restrictions can easily be inferred from the one we present. We now formulate our restriction in terms of relational schemas. The restriction has two components: one involving the intra-relational constraints, and a second involving the inter-relational inclusion dependencies.

Definition. A relational database schema $< S, \Sigma >$ whose constraints are unary id's, multiplicity constraints and cross-product constraints is restricted iff the following conditions hold:

(i) For each relation schema $<$ rel, R, $\Sigma_{rel} >$ in S, if $X[\leq k]$ and $\otimes Y$ are in Σ_{rel} and $[A] \subseteq [B]$ where $A \in X$ and $B \in Y$, then $Y \subseteq X$, and

(ii) There is no sequence $<$ rel$_i$, R$_i$, $\sum_{rel_i} >$, $1 \leq i \leq n$, of distinct relation schemas of S, and attributes $A_1 \cdots A_n$, $B_1 \cdots B_n$, such t hat $A_i \in R_i$, $B_i \in R_i$ for each i ($1 \leq i \leq n$), $[A_i] \subseteq [B_{i+1}]$ ($1 \leq i < n$), $[A_n] \subseteq [B_1]$, and $A_1 \neq B_1$.

Informally, condition (ii) states that two distinct attributes of the same relation cannot be "connected" using the inter-relational inclusion dependencies of the schema.

For instance, the schema of Example 3.1 (Figure 3.4) is restricted.

The formulation of the above restriction in terms of Iris schemas is similar and is omitted. (Recall that a constraint of the type $X[\leq k]$ corresponds in to an Iris upper-participation k for X, while $\otimes Y$ corresponds to lower participation one for Y.)

It is shown in [LyVi87] that the restriction proposed on relational schemas eliminates the problems encountered in unrestricted schemas. First, the constraints for such schemas have a finite axiomatization. Second, no side-effect fd's occur, and each relation in the translation is in BCNF. Finally, restricted schemas do not give rise to problems concerning satisfiability, localness, or type cardinality constraints. Indeed, the following can be shown:

(i) Each restricted relational database schema is non-trivially satisfiable.

(ii) Each restricted relational database schema s is local (i.e., the constraints in s* which apply to a given relation schema $<$ rel, R, $\Sigma_{rel} >$ of s are equal to Σ^*_{rel}, and thus can be computed locally.)

(iii) Let s be a restricted relational database schema; for each integer n there exists an instance of s such that each column of a relation in the schema has at least n distinct values. Thus, no type cardinality constraints are implied by the constraints of the schema.

5. Conclusions

A formal framework for understanding the connection between semantic data models and the relational model was developed. The translation of Iris schemas to relational schemas was shown to give rise to constraints different than those usually encountered in relational databases. The results obtained so far concern mostly these types of constraints and their connection with traditional Normal Forms involving fd's. The formal approach to these problems reveals that they are more complex then was assumed originally based on intuitive grounds. For instance, relational translations of Iris schemas are not always in BCNF, as was expected, due to the presence of "side-effect" fd's; and the constraints of these schemas are not finitely axiomatizable. Other subtle problems concerning Iris schemas were also brought to light, such as type aliasing, trivial satisfiability, and non-localness. The restriction exhibited on schemas provides guidelines as to how such problems can be avoided when designing Iris schemas.

The Iris model only contains simple constructs common to most semantic models. However, the techniques developed for Iris provide insight into the problems arising when more complex constructs are allowed in the semantic model. For instance, semantic schemas allowing types which are disjoint unions or cross-products of types give rise to relational constraints for which the implication problem is undecidable. (Indeed, specialization involving types that are cross-products of types gives rise to non-unary inclusion dependencies which, in conjunction with fd's induced by participation constraints, lead to undecidability of implication (see [ChVa83]); in the case of disjoint unions of subtypes, the equations associated with the schema involve multiplication as well as addition, so implication involves the theory of integers with addition and multiplication, which is undecidable [Mo76].) In particular, it is undecidable whether relational translations of such schemas are in BCNF, whether they are non-trivially satisfiable, whether aliasing occurs, or whether type cardinality restrictions are implied. Such results provide new insight into trade-offs between expressiveness and tractability of semantic schemas.

Finally, note that the mapping between models described in this paper concerns exclusively static aspects of the models. It is of interest to include in this correspondence dynamic aspects of the models. Iris schemas with update operations would then translate to relational schemas involving both constraints and transactions, such as those discussed in [AbVi85]. The results already obtained concerning the connection between transactions and constraints in relational databases could be used to validate update semantics in Iris schemas.

Acknowledgements

The authors would like to thank Richard Hull and Bill Kent for their helpful comments on earlier versions of this paper.

References

[AbHu84] Abiteboul, S. and Hull, R. IFO: A Formal Semantic Database Model (Preliminary Report). *Proceedings ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* 1984.

[AbVi85] Abiteboul, S. and Vianu, V. Transactions and constraints. *Proceedings ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* 1985.

[BeFe83] Beech, D. and Feldman, J. S. The Integrated Data Model: A Database Perspective. *Proceedings of the Ninth International Conference on Very Large Data Bases,* Florence, Italy, October 1983.

[Ca83] Cattell, R. G. G. Design and Implementation of a Relationship-Entity-Datum Data Model. Technical Report CSL 83-4, Xerox Corporation, Palo Alto Research Center, May 1983.

[CFP82] Casanova, M.A., Fagin, R., Papadimitriou, C.H. Inclusion dependencies and their interaction with functional dependencies. *Proceedings ACM SIGACT-SIGMOD Symp. on Principles of Database Systems,* 1982, pp. 55-62.

[ChVa83] Chandra, A. and Vardi, M. The implication problem for functional and inclusion dependencies is undecidable. IBM Research Report, RC 9980 (44299), 1983.

[DeKeLy85] Derrett, N., Kent, W., and Lyngbaek, P. Some Aspect of Operations in an Object-Oriented Database. *IEEE Database Engineering Bulletin.* December, 1985.

[Fi87] Fishman, D. et al. Iris: An Object-Oriented DBMS. *ACM Transactions on Office Information Systems,* 4(2), April 1987 (to appear).

[LyKe86] Lyngbaek, P. and Kent, W. A Data Modeling Methodology for the Design and Implementation of Information Systems. *Proc. Int'l Workshop on Object-Oriented Database Systems,* Pacific Grove, California, September 1986.

[LyDeFiKeRy87]Lyngbaek, P., Derrett, N. H., Fishman, D., Kent, W. and Ryan, T. Design and Implementation of the Iris Object Manager. *Proc. A Workshop on Persistent Object Systems: their design, implementation and use,* Scotland, August 1987.

[LyVi87] Lyngbaek, P. and Vianu, V. Mapping a Semantic Database Model to the Relational Model. *Proceedings of the ACM SIGMOD International Conference on Management of Data.* San Francisco, California, May 1987.

[Ma83] Maier, David *The Theory of Relational Databases.* Computer Science Press, 1983.

[Mo76] Monk, Donald. *Mathematical Logic.* Springer Verlag, 1976.

[MyBeWo80] Mylopoulos, J., Bernstein, P. A., and Wong, H. K. T. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems* 5(2):185-207, June, 1980.

[Sh81] Shipman, D. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems* 2(3):140-173, March, 1981.

[SmSm77] Smith, J. M. and Smith D. C. P. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems* 2(2):105-133, June, 1977.

[TsZa84] Tsur, S. and Zaniolo, C. An Implementation of GEM - supporting a semantic data model on a relational back-end. *Proceedings of the ACM SIGMOD International Conference on Management of Data.* Boston, Ma., June, 1984.

[Ul82] Ullman, Jeffrey. *Principles of Database Systems.* Computer Science Press, 1982.

SIM: Implementation of a Database Management System Based on a Semantic Data Model

R. L. Guck, B. L. Fritchman,
J. P. Thompson and D. M. Tolbert

Unisys Corporation, Irvine, CA 92718

Abstract

The advantages of semantic data models over hierarchical, network and relational models have been well established. However, a comprehensive database management system (DBMS) that directly supports a semantic data model faces new implementation challenges that stem directly from the model. Some of the more significant issues are:

Data integrity: The DBMS must assume additional data integrity responsibilities due to the semantics of entity types, generalization hierarchies, relationships, and other conceptual-level constraints.

Conceptual object mapping: The conceptual schema must be mapped into lower-level physical components such as records, items, and access methods. However, high-level query parsing and optimization should not require detailed knowledge of mapping techniques so that multiple mapping methods and data sources (i.e. heterogeneous data access) can be adopted more easily.

Performance: Special considerations are needed to address performance due to the greater "distance" between conceptual and physical database objects and the corresponding increase in translation that is required.

The Semantic Information Manager (SIM) is a fully featured, commercially available DBMS implemented for the Unisys A Series line of mainframes. SIM directly supports a semantic data model similar to Hammer and McLeod's SDM [HaMc81]. SIM forms the center of the InfoExec^{tm1} system, which provides a comprehensive data management environment based on the semantic data model. This paper provides a summary of the principal InfoExec modules, and it describes some of the key implementation strategies used by SIM to address the above issues.

1 Semantic Data Models

Hierarchical, network, and relational data models possess inherent limitations that have been identified in many papers [e.g., Kent79, KiMc81, Ship81]. To overcome these limitations, a number of high-level data models have been proposed such as the binary relational model [Abri74], entity-relationship model [Chen76], extended relational model [Codd79], functional data model [Ship81], and the semantic data model [HaMc81]. Each of these models can be termed "semantic" data models because their common objective is to capture more of the meaning of data than previous data models.

1 InfoExec is a trademark of Unisys Corporation.

Precise terminology and concepts vary from one model to the next. However, a core set of semantic data model features can be described as follows:

(1) The primary objects of interest in a semantic data model are "entities" which are classified by "entity types".

(2) Entity types may be "subtypes" or "supertypes" of other entity types, thereby forming "generalization hierarchies". The "root" entity type of a hierarchy is termed a "class", while all other entity types are termed "subclasses".

(3) Entities possess "attributes" which can be "single-valued" or "multi-valued". Attributes establish characteristics of entities including relationships between entities.

(4) Data integrity constraints are expressed in several forms including semantics that are defined for hierarchies, attributes, and relationships.

Many advantages can be realized by a DBMS that is based on a semantic data model. For example, data can be modeled more closely to its real-world perception, thereby simplifying database design and maintenance. The database can assume increased responsibilities for data integrity, thereby reducing application development expense while increasing database reliability and security. A query language can be developed that is conceptually natural, expressive, and easy to use. Also, query optimization can be performed with greater scope.

2 Overview of SIM and the InfoExec System

The SIM project was initiated five years ago at Unisys with the goal of producing an advanced DBMS for our A Series systems. We chose a semantic data model similar to Hammer and McLeod's SDM [HaMc81] as the basis of SIM for several reasons. Besides offering the advantages described above, SDM is sufficiently powerful to allow a wide variety of existing database systems to be viewed with a single model. At the same time, SDM is sufficiently flexible to allow evolution into even more advanced data modeling concepts (e.g., inference mechanisms could be added to more directly support knowledge-based systems).

SIM offers a formal data definition (DDL) language for schema definition and a data manipulation language (DML) for data access. The DDL supports a rich set of constructs including powerful data types, generalization hierarchies, relationships, and integrity and security constraints. The DML is a high-level, non-procedural language that supports powerful features while addressing ease of use via an English-like syntax. These topics are fully discussed in [JFGT88].

SIM is the central component of the InfoExec system, which provides a complete data management environment based on the semantic data model[2]. The InfoExec environment addresses global data management issues, such as

[2] The term "DBMS" implies the software which is used to define, access, and maintain a database. A "data management environment" provides additional software, such as a data dictionary and ad-hoc query products, to assist the overall application development effort.

productivity, ease of use, and performance, by providing additional modules which address the development and use of both the database and its applications. In contrast to modeling or translation facilities, experimental prototypes, and hybrid implementations [e.g., AnHa87, GGKZ85, MBW80, PSM87, Ship81, TsZa84], the InfoExec system represents a "ground-up" implementation that propagates the semantic data model throughout its interfaces. We believe that ours is the first fully featured, commercially available system based on a semantic data model.

A brief description of the InfoExec modules which are integrated with SIM is provided below.

2.1 Data Dictionary

A key module of the InfoExec system is the Advanced Data Dictionary System (ADDS) which is used to define, maintain, track, and perform other functions for all SIM databases (hence, ADDS is an "active" dictionary). ADDS also provides many other definition capabilities which support the application environment. For example, ADDS can be used to define record structures, screen descriptions, program data structures, and documentation. All of the ADDS functions are offered via a menu-oriented interface which minimizes the need to know specific sublanguages (i.e., a SIM database can be defined without having to know the SIM DDL). Each menu offers extensive on-line "help" text to reduce the need to access reference manuals. ADDS is implemented as a fairly large SIM database application, thereby making use of (and giving credence to) the power of the semantic data model.

2.2 Ad-hoc Query Facilities

Two ad-hoc query facilities are provided to support on-line access to the database. The Interactive Query Facility (IQF) offers a screen-based interface via terminals while the Workstation Query Facility (WQF) offers a graphically-oriented interface via intelligent workstations. Both products preserve the data model and its expressive abilities, yet they reduce the need for the end-user to directly use the underlying DML by using prompts or graphic icons to construct query statements. Both products have dictionary interfaces for saving and reusing queries, and both products provide extensive report handling features.

2.3 Host Language Interfaces

Host language interfaces to SIM databases and ADDS are provided in COBOL, Pascal, and ALGOL. The interfaces are unique in that they provide the SIM DML as language extensions "tuned" to the flavor of each host language. This approach preserves the full expressive power of the SIM DML while allowing query statements to be intermixed with host language constructs. For example, queries may contain expressions that reference program variables and functions, and queries may be passed as parameters within and between programs. ADDS can be accessed for importing conceptual schema elements, file descriptions, data structures, screen interfaces, and other definitions.

2.4 Data Management System II (DMSII)

DMSII is a multi-user, high-performance, network-based DBMS that has been widely-used on A Series systems for nearly 15 years. DMSII provides a complete set of database features in the areas of file structures and access methods, resource management, and auditing and recovery mechanisms. SIM exploits these features by using DMSII as its primary physical access engine (although access to other data sources is also supported). This arrangement results in an integrated co-existence between DMSII and SIM. For example, an InfoExec utility is provided to view an existing DMSII database as a SIM database.

2.5 Other Database Utilities

The InfoExec system also offers a complete range of database support utilities for performance monitoring and tuning, backup and recovery, and other functions. Due to the integrated relationship of DMSII and SIM, most of these utilities will operate on either DMSII or SIM databases. A screen-oriented interface to these facilities is provided via the Operations Control Manager (OCM) to eliminate the need for learning sublanguages peculiar to each utility. The screen interface uses the same on-line help mechanisms used by other screen-based InfoExec products.

3. Implementation Strategy of SIM

3.1 Process Architecture Overview

The key implementation objectives of SIM are to (1) realize and directly support the data model, (2) provide flexible conceptual object mapping with useful alternatives, (3) provide query (DML) parsing and optimization that is independent of specific mapping techniques, (4) enforce the data integrity constraints expressed by the database schema, and (5) provide query execution performance that is suitable for databases requiring high transaction rates. Two longer-term goals are to provide access to both heterogeneous and distributed data.

As a consequence of these criteria, the SIM architecture is designed to be highly modular yet efficient. This architecture is depicted in Figure 1. As shown, SIM is composed of multiple layers that are joined at distinct interfaces. Each software module operating within each layer has a specific perspective of the database and performs its operations based on that perspective. This approach allows each module to function at the highest possible level, thereby minimizing its need to know details of lower-level modules. A key performance strategy of this architecture is the use of dynamic code generation for query execution. Also, the implementation of SIM exploits architectural advantages of A Series systems (the discussion of which is beyond the scope of this paper).

An important implementation strategy, the Logical Underlying Component (LUC) model, is described in the next section. The purpose and operation of each software module within SIM's architecture are described in subsequent sections.

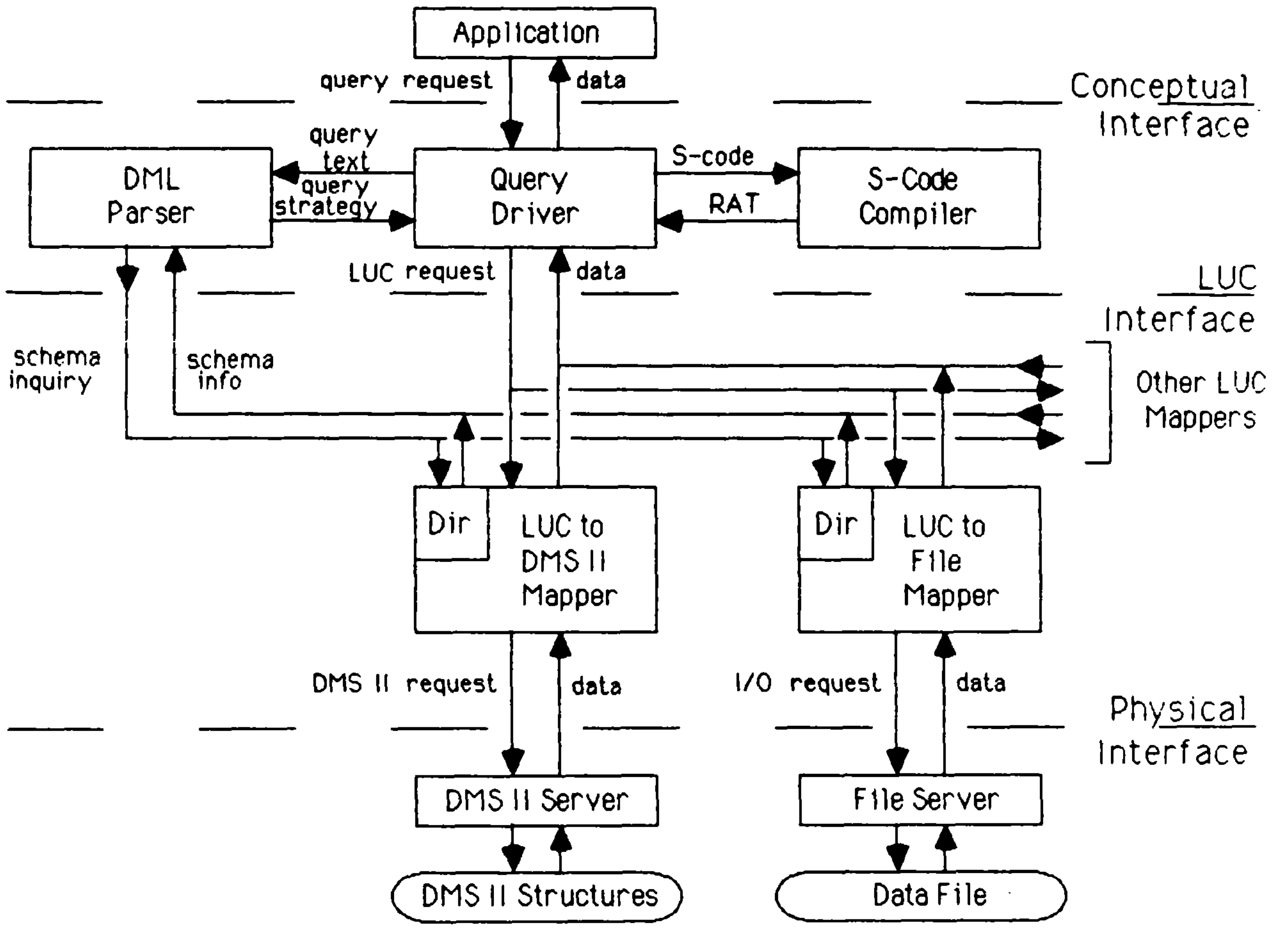


Figure 1 - SIM Architecture

3.2 The LUC Model

Because we wish to separate conceptual-level software modules from physical mapping details as much as possible, we have introduced an intermediate data model within SIM's process architecture. It is termed the "LUC" model because it consists of "logical underlying components" which are functionally simpler than the conceptual-level objects in SIM. There are three types of LUC objects:

(1) A "record" LUC consists of a fixed set of single-valued fields. Each field has a fixed size, location, and a known data type.

(2) A "relationship" LUC defines a unidirectional relationship between an "owner" record LUC and a "target" record LUC. The target record LUC may be "dependent" which means that it cannot exist without the owner LUC object. Two relationship LUCs may be identified as "inverses" of each other.

(3) An "index" LUC defines an access method on a record LUC. The index may have multiple ascending and/or descending keys, and it may be a random or index-sequential structure.

Each LUC object has a well-defined set of operations that can be performed on it and a corresponding set of integrity rules. Every conceptual schema has a precise mapping into a LUC schema that is independent of how the LUC schema is mapped into the underlying physical schema. Furthermore, the LUC model is sufficiently flexible that it can be mapped into virtually any record-based model, including conventional database systems. Thus, the software "above" the LUC level need only know how to translate conceptual-level operations into LUC level operations, and software "below" the LUC level need only know how to translate these operations into corresponding operations on the particular data source(s) involved.

3.3 Application

A SIM application is either an ad-hoc query facility such as IQF or WQF, a host language compiler such as COBOL or Pascal, or a user-written host language program. In each case, the application's view of the database is based entirely on the database's conceptual schema. Each application submits separate requests to parse and execute DML queries, thus allowing the two processes to occur independently. For example, a host language program's queries are parsed at compile-time but executed at run-time. Once a query is parsed, it can be executed multiple times, and multiple, simultaneous invocations (e.g., with different input parameters) are allowed.

3.4 Query Driver

The Query Driver is the application interface to SIM. Its responsibility is to manage the query parsing and execution requests submitted by the application program. An application requests that a query be parsed by submitting a DML query as a text string. The Query Driver submits this text to the DML Parser, which returns a query execution strategy, one part of which is an "S-code" (semi-compiled code) symbolic program. The Query Driver submits the S-code source to the S-code compiler and in return receives a block of machine code called a remap-and-test (RAT) procedure.

An application executes a query by submitting "retrieve data" or "process update" requests. The Query Driver responds to these requests by interpretively executing portions of the query's execution strategy. This causes LUC requests (including the appropriate RAT procedure) to be submitted to the appropriate LUC Mappers. The LUC Mappers respond to retrieve requests with data obtained from the database or with update results sent to the database.

3.5 DML Parser

The DML Parser parses, validates, and develops an optimized execution strategy for each DML query. While parsing a query, the DML Parser retrieves schema information by submitting schema inquiry requests to the appropriate LUC Mappers. The parser receives information on conceptual and LUC objects and the mapping between them. Information such as relationship cardinalities, available indexes, and integrity constraints is used to develop an optimal execution strategy for the query.

The information received by the DML Parser is free from physical mapping details except for the layout of LUC records. For each LUC record, the parser obtains such details as field offsets, lengths, and data types. This information is used to generate the S-code program, which is returned as part of the execution strategy.

Many integrity constraints are the responsibility of the DML Parser. For example, data type, range, and existence requirements are either verified during parsing, or the query's execution strategy is augmented such that enforcement is performed during query execution. Other integrity constraints are the responsibility of the LUC Mapper.

3.6 S-code Compiler

The S-code Compiler is a dynamic code generator. The S-code language consists of a limited set of statements, operators, and operands. A typical S-code "source" program consists of a few hundred bytes, and the S-code compiler converts this program into the RAT procedure previously mentioned. When called, a RAT procedure accepts a set of buffers (arrays) and performs testing and update operations on them as instructed by its source program. The RAT performs two basic kinds of operations which contribute greatly to the performance of query execution:

(1) The RAT can examine a record and apply a selection criterion to determine if the record should be selected. This "test" function is used to find database records during search phases of query execution.

(2) The RAT can move data from one record to another, performing expression analysis (including field-level integrity checks), data type conversion, and other functions. This "remap" function is used to map data between database records and application data structures.

Typically, the S-code compiler can compile an S-code source program into a RAT in less than one second. Since the RAT procedures perform all expression analysis and data transfer for a query, the performance impact of the interpretive nature of the remaining SIM modules is minimized.

3.7 LUC Mappers

A different LUC Mapper exists for each type of physical data server that can be accessed as part of a SIM database. Each LUC Mapper accepts LUC commands and translates them into appropriate calls on the underlying data server module. The Mapper calls the RAT procedure to filter and/or update records retrieved from the data server. The Mapper exploits any functionality that is available from the data server (e.g., transaction control, locking mechanisms, searching mechanisms, etc.). When necessary, the Mapper supplements the data server's functionality with anything that is lacking.

The LUC Mapper plays an important part in enforcing data integrity during query execution. Many of the constraints implied by the data model or the conceptual schema are automatically enforced by the Mapper. For example, existence and cardinality constraints on relationships are enforced by the Mapper during update operations. Dependent LUC objects which cannot exist without a corresponding owner are either automatically deleted when the owner is deleted, or the Mapper disallows the deletion of the owner (or the relationship to the owner) depending upon the type of conceptual object represented. For example, when an entity is deleted from a subclass role, all relationships and multi-valued attributes are deleted from both the subclass and all subordinate roles in which the entity participates. These operations are the responsibility of the LUC

Mapper so that it may exploit any physical mapping properties for greater performance.

Each LUC Mapper also services schema inquiry requests from the DML Parser. A portion of the LUC Mapper called the Directory is dedicated to this function. Directory information is stored either in the data dictionary (ADDS) or within the database itself.

Note that the use of multiple LUC Mappers allows a flexible composition and distribution of a single database. What is viewed as a single SIM database at the conceptual level can consist of data that is derived from multiple data sources. A single data server may distribute data within its own control, or data can be distributed by one or more LUC Mappers. Since the LUC Mapper's interfaces are message-oriented, it can reside on a foreign host, thereby allowing distributed data. This architecture also allows a single query to access data within multiple databases.

3.8 Physical Data Servers

Ultimately, SIM aims to allow any data source to be a data server, thereby allowing virtually anything to be accessed as a SIM database. Initially, however, the primary physical data server for SIM databases is provided by DMSII, which, as mentioned earlier, is a robust and widely-used network-based DBMS. Also, a File Mapper is available which provides access to simple data files, although only inquiry access is currently allowed.

To further enhance the performance of DMSII-mapped databases, the DMSII data server has been extended to accept and use RAT procedures during record selection. When this is done, a RAT is accessing low-level I/O buffers directly, thereby eliminating a great deal of potential data movement. Also, the RAT can search multiple records sequentially until a record which meets the selection criteria is found. This prevents returning from DMSII with unneeded records. Finally, the only records that are locked during updates are those selected by the RAT, hence, the locking of non-essential records is eliminated.

4. Conceptual Object Mapping

SIM offers multiple techniques for mapping conceptual objects into physical DMSII components. Default mapping algorithms are chosen to provide a careful balance between performance and memory/disk usage. Alternatives are available so that users may "tune" the mapping to conditions present for a specific database or host system. The default mapping techniques and some of the alternatives are described below.

Classes: Each class is mapped to a separate disjoint structure containing fixed-format records. Each record holds data items corresponding to the class's attributes plus a system-generated surrogate item. The surrogate item contains a value that gives each entity a unique identity that never changes over the entity's life. The class is spanned by a surrogate index, which is an access method whose key is the surrogate item. This provides efficient entity retrieval by surrogate value. User alternatives include structure types which employ different storage and access techniques and different surrogate implementations including the use of a suitable required, unique attribute.

Subclasses: Subclasses have two possible mappings depending upon their superclass configurations. When a subclass has a single immediate superclass and all of its sibling subclasses are mutually exclusive, the subclass is mapped to a variable-format extension of its superclass's records. A subclass with multiple superclasses and/or non-mutually exclusive siblings is mapped to a new disjoint structure, and a "copy" surrogate item and index are added to that structure. The copy surrogate item contains the same surrogate value as the entity's base class. The user may choose the disjoint structure mapping when the variable format mapping would have been the default, and he or she has the same structure type options that are available for classes.

Attributes: Attributes whose range is a system-defined class representing a printable data type (such as arithmetics, strings, and booleans) are termed "data-valued attributes" (DVAs). All other attributes range over some user-defined class and are termed "entity-valued attributes" (EVAs). SIM employs different mapping techniques for single-valued (SV) and multi-valued (MV) DVAs and EVAs.

Each SV DVA is mapped into two items within its owner's record: a data item holds the value of the DVA, and a "null bit" item indicates if the value exists. An MV DVA is normally mapped to a disjoint structure containing a data item, copy surrogate item, and surrogate index similar to that used for subclasses. When the DVA meets certain requirements, it can be mapped to an embedded structure or an array contained in the structure of the DVA owner.

Each EVA has an "inverse" EVA which is either explicitly or implicitly declared. An EVA pair forms a bidirectional relationship that is mapped with the cardinality of each EVA taken into consideration. Each EVA of a 1:1 relationship is mapped into an item within its owner's record. The item contains the surrogate value of the owner of the inverse EVA. As an alternative, the user can choose absolute address pointers for each EVA of a 1:1 relationship.

By default, both EVAs of 1:many and many:many relationships are mapped to a structure called the "common EVA structure." Each record represents a single relationship instance between two entities. Each record possesses two items containing the surrogate values of the entities plus a relationship type item. Many relationships are mapped to a single common EVA structure, thereby minimizing the number of structures consumed by the database. As alternatives, the user may choose from several absolute address and foreign key mappings to achieve better performance at a cost of an increase in the number of structures used.

5. Conclusions

Key implementation strategies have been described that are used by SIM, a DBMS based on a semantic data model. The SIM architecture is highly modular and employs an intermediate data model called the LUC model to increase the independence of the conceptual and physical layers. As a result, query parsing and optimization do not require detailed physical mapping knowledge, yet the system allows multiple physical mapping alternatives and distributed, heterogeneous data access. Data integrity is enforced by a combination of query augmentation and dynamic constraint enforcement performed by the LUC Mapper modules. Query performance techniques include high-level query optimization, dynamic code generation, and LUC Mapper exploitation of data server features.

Our experience thus far with SIM has been very satisfying. Our tests have demonstrated performance comparable with conventional database systems, and our initial customers have testified to the power and productivity provided by the model. Investigations for future work include DDL and DML enhancements, including additional integrity mechanisms, further performance improvements, on-line reorganization, increased access to distributed and heterogeneous data access, and temporal data.

References

[Abri74] J. R. Abrial, Data Semantics. In Database Management, J. Klimbie and K. Koffeman Eds. North-Holland Amsterdam, 1974.

[AnHa87] T. Andrews and C. Harris, Combining Language and Database Advances in an Object-Oriented Development Environment, In proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, 4-8, October 1987

[Chen76] P.P.S. Chen, The entity-relationship Model: Towards a unified view of data, ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976

[Codd79] E. F. Codd, Extending the Database Relational Model to capture more meaning, ACM Transactions on Database Systems, Vol. 4, No. 4, December 1979

[GGKZ85] K. Goldman, S. Goldman, P. Kanellakis, and S. Zdonik, ISIS: Interface for a Semantic Information System, In proceedings of ACM SIGMOD International Conference on Management of Data, May 1985

[HaMc81] M. Hammer and D. McLeod, Database Description with SDM: A Semantic Data Model, ACM Transactions on Database Systems, Vol. 6, No. 3, September 1981

[JFGT88] D. Jagannathan, B. L. Fritchman, R. L. Guck, J. P. Thompson, and D. M. Tolbert, SIM: A Database System Based on the Semantic Data Model, To appear in proceedings of ACM SIGMOD International Conference on Management of Data, 1988

[Kent79] W. Kent, Limitations of Record-Based Information Models, ACM Transactions on Database Systems, Vol. 4, No. 1, March 1979

[KiMc81] R. King and D. McLeod, Semantic Data Models, in S. B. Yao (Ed). Principles of Database Design, Prentice Hall, 1984

[MBW80] J. Mylopoulos, P. A. Bernstein, and H. K. T. Wong, A Language Facility for Designing Database-Intensive Applications, ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980

[PSM87] A. Purdy, B. Schuchardt, and D. Maier, Integrating an Object Server with Other Worlds, ACM Transactions on Office Information Systems, Vol. 5, No. 1, January, 1987

[Ship81] D. W. Shipman, The Functional Data Model and the Data Language DAPLEX, ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981

[TsZa84] S. Tsur and C. Zaniolo, In Implementation of GEM - supporting a semantic data model on a relational back-end, In proceedings of ACM SIGMOD International Conference on Management of Data, May 1984

An Object-Oriented Database System to support an Integrated Programming Environment

Daniel Weinreb, Neal Feinberg, Dan Gerson, Charles Lamb
Symbolics Inc.; 11 Cambridge Center; Cambridge MA 02142

Abstract: Statice is an object-oriented database system, designed to support the Genera integrated programming environment by providing objects that are persistent and shared between workstations. Statice has a clear and natural interface to the host language, Symbolics Common Lisp. We describe the basic semantics and the language interface of Statice, illustrated with examples. We then discuss the implementation techniques used to implement concurrency control, recovery, and efficient access to data. Finally, we describe some applications that have been built using Statice.

1. Genera and Data-level Integration

The purpose of the Symbolics Genera integrated programming environment [Walker 87] is to let programmers build complex and innovative software systems much more productively than they can with conventional environments. All information is represented as objects in a virtual address space shared by Genera's extensive suite of tools. The different tools are tightly integrated because they communicate with each other by sharing and manipulating these objects. Many processes can work simultaneously on many activities in many windows, and all access the same object base and shared the same data structures.

Statice is an object-oriented database system that provides shared, persistent objects to Genera. Software tools using Statice as the basis of data-level integration will be able to use objects to represent persistent state, share information between users, and efficiently manipulate large amounts of data. Statice also works as part of the Genera substrate, so applications developed using Genera can also be built on Statice.

Statice provides concurrency control and recovery (using transactions), fast associative access (using indexes), very fast direct access to object attributes, a non-procedural query language, data independence, object semantics, and a natural interface to the Common Lisp programming language. In general, Statice aims at retaining the traditional benefits of conventional database systems while adding the new benefits of object-oriented database systems.

2. Data Model

At first we thought that the natural programming goal meant that the data model of Statice should be exactly the same as that of Lisp. However, we found that this was in strong conflict with the data independence goal. Lisp does not separate real information from underlying organization. For example, lists and vectors really represent the same underlying semantics, namely a sequence. The only difference between them lies in the implementation, and mainly reflects performance (speed of insertion versus speed of accessing the *n*th element). The purpose of data independence is to hide such differences, but Lisp makes them quite visible.

Also, for many types of objects in Lisp, it's not clear what it would mean to store them as shared, persistent objects. Some objects, such as processes and indirect arrays, depend on the virtual memory for their meaning. Others objects, like most symbols and many of the pre-created structures in Lisp, must exist separately for every Lisp environment, and so cannot be shared in a straightforward manner. Some other related problems are discussed in [Vegdahl 86].

To provide data independence, and avoid the these pitfalls, we adopted a more abstract data model, described in the next section. This data model is based primarily on DAPLEX [Shipman 81] [Daplex 84] functional data model. However, it is also closely linked to Common Lisp's object-oriented programming facility, so it serves as an object-oriented data model as well. We were also substantially influenced by [Cattell 83] and [Kent 79]. A similar model is used by [Kempf 86].

A Statice database schema is made of a set of entity type definitions. An entity type has a name, a set of parent types, and a set of attributes. Each attribute has a name, a type, and various attribute options. Attributes model both the properties of entities, and the relationships among entities. For example:

```
(define-entity-type person ()
  ((name string :unique t :no-nulls t :read-only t :inverse person-named :inverse-index t)))
```

This form defines an entity type called **person**. The **person** entity type has one *attribute*, named **name**, whose type is **string**. The **:unique** option means that no two person entities in the database can have the same value for the **name** attribute. In other words, the relation between person entities and names is one-to-one. The **:no-nulls** option means that the value of the **name** attribute of a person entity cannot be the *null value*. The **:read-only** option means that the attribute's value can be examined but not changed after the entity is created. The **:inverse-index** option creates an index on the values. The **:inverse** option defines an inverse accessor function (see below).

```
(define-entity-type student (person)
  ((dept department :inverse students-in-dept)
   (courses (set-of course) :index t :inverse course-students)))
```

The **student** entity type inherits from the **person** entity type. In other words, every student is also a person. Student entities inherit the attributes of person, so every student has a name.

The type of the **dept** attribute is **department**, which is one of the other entity types of the schema, so the **dept** attribute models a relationship between entities. Since the **:unique** option is not specified, many students can be in the same department. The **dept** attribute is many-to-one.

The attributes we've discussed so far are all *single-valued*. The **courses** attribute is *set-valued*, because a student can be taking many courses. **:unique** is not specified, because many students can be taking the same course; this is a many-to-many relation.

```
(define-entity-type course ()
  ((title string :inverse courses-entitled)
   (dept department)
   (credits integer)
   (instructor instructor :inverse courses-taught-by :inverse-index t))
  (:multiple-index (title dept) :unique t))

(define-entity-type instructor (person)
  ((dept department :no-nulls t)
   (visiting boolean)
   (salary single-float)))
```

```
(define-entity-type department ()
  ((name string :unique t :inverse department-named :inverse-index t)
   (head instructor)))
```

These forms define the rest of the entity types. **:multiple-index** is an option that applies to the whole entity type, specifying an index on two attributes.

In addition to defining Statice entity types and their attributes, **define-entity-type** also defines a *class* of the Common Lisp Object System (CLOS) [DeMichiel 87] [Moon 88]. CLOS is an object-oriented extension to the Common Lisp language. Statice inheritance and CLOS inheritance work the same way, including full support of multiple inheritance. The entity classes don't have any slots; Statice attribute values are stored in the database, not in Lisp's virtual address space.

An instance of such a class is called an *entity handle*. An entity handle is a Lisp object that refers to a Statice entity. Entity handles reside in Lisp's virtual memory, whereas entities reside in databases. A particular Lisp world has only one entity handle (or none) for any entity, so the Lisp concept of identity maps directly to the Statice concept of identity. For example, the Lisp **eql** function can be used to compare the identity of Statice entities. Entity handles continue to exist outside transactions, and can be kept in Lisp data structures.

CLOS *methods* can be defined on the classes defined by entity types. The following example defines a method that returns the number of courses taught by an instructor.

```
(defmethod n-courses-taught-by ((i instructor))
  (length (courses-taught-by i)))
```

Although DAPLEX is not usually described as "object-oriented", it nevertheless posesses many of the attributes usually associated with object-oriented systems; object identity, types with inheritance, and methods (procedures attached to types, which DAPLEX calls "derived data"). The most important difference between the data models of Statice and DAPLEX is that Statice fits more comfortably into the host language, Common Lisp. The Lisp programmer treats Statice entities just like Lisp objects. The object-oriented programming syntax and semantics (such as conflict resolution for multiple inheritance) in Statice work just as they do in Common Lisp's CLOS, because Statice uses an existing object-oriented language instead of inventing a new one. The methods, which are like the "derived functions" of DAPLEX, are written in Lisp and can use the full power of the Common Lisp language, whereas DAPLEX is restricted to the small, specialized DAPLEX language. Another difference is that Statice supports true multiple inheritance, whereas DAPLEX is oriented towards single inheritance.

2.1. Types

The Statice type system maps directly into the Common Lisp type system, so the familiar Common Lisp operators such as **typep**, **typecase**, and **check-type** can be used in the natural way for entity handles. Programs using Statice can define methods on entity classes, just like any other class. The type specifiers used to name types of Statice attributes are Common Lisp *type specs* [Steele 84]. Thus, Statice fits smoothly into the object-oriented programming system already provided by the Lisp language.

Statice types have the same semantics as Common Lisp types, as well as the same names. For example, strings can be of any length, and can include style (family, face, and size) information and use many character sets. Integers can be of arbitrary precision. (Storage formats are arranged so that common cases, such as small integers and strings without style information, are stored efficiently.) Statice supports all Common Lisp numeric types, including rational and complex, as well as integer subranges, enumerated types (the Common Lisp **member** type), and others. Another type holds an arbitrary-length vector of 32-bit words, for storing arbitrary binary data. A general union type can represent a wide range of Lisp objects, by utilizing the same binary dumper that the Lisp compiler uses to store constants. Pictures in the form of diagrams from Genera's graphics system [Symbolics 88] and bitmap images can also be stored as values.

Statice application programs can extend the type system, defining their own types, in two ways. To define a *logical* type, a program defines encoder and decoder methods to translate values from the program-visible value to a value of some already-implemented type. For example, you could make a new enumerated type by encoding each element of the enumeration into a corresponding small integer. To define a *physical* type, which is a genuine new type not based on a pre-existing type, a program defines about a dozen methods that control the exact binary representation of the value. For example, the **write-value** method is given a value, an addressor (specifying a record), a word offset, and a bit offset; the method must write the value into the record at the specified location. Other methods read, compare, compute the size, and check the type of values, etc.

2.2. Accessing Attributes

define-entity-type defines an *accessor function* for each attribute. An accessor function retrieves the value of an attribute of an entity. In the example above, the accessor functions are named **person-name, student-dept, student-courses**, and so on.

Accessor functions are actually *generic functions* of CLOS. They take one argument, an entity handle. For example, suppose the value of the variable **george** is an entity handle for a particular student. The Lisp form **(person-name george)** returns the value of the **name** attribute of the student, which is a string. **person-name** can be applied to a student, by the usual rules of CLOS inheritance. The Lisp form **(student-dept george)** returns an entity handle that refers to the **department** entity that is the student's department.

define-entity-type also defines **setf** methods for updating the values of attributes. The Lisp form **(setf (instructor-salary jones) 20000.0)** sets the value of the **salary** attribute of **jones** to **20000.0**. The **person** attribute is defined with the **:read-only** option, which suppresses the definition of the **setf** method. Accessor functions for set-valued attributes return lists, and their **setf** methods accept lists to store the entire set. There are also special forms called **add-to-set** and **delete-from-set** to add or delete one element.

Statice's accessor functions thus behave and look like accessors for Lisp structures and CLOS instances. They also fit in well with the rest of Common Lisp. For example, because Statice accessor functions use the **setf** mechanism, Lisp's "place modifier" forms can be used with accessor functions. For example, **(incf (instructor-salary jones) 100.0)** means that Jones gets a raise of $100.00. The resulting programming style is immediately grasped by Lisp programmers, and fits naturally into programs.

The **:inverse-function** option to the **name** attribute defines an *inverse accessor function* named **person-named**. It takes a name as its argument and returns the person entity, thus doing the inverse of what the accessor function does. **students-in-dept** is also an inverse accessor function, which takes a department and returns a set of students. In general, the inverse of an *x*-to-*y* function is a *y*-to-*x* function. **person-named** is one-to-one, and **students-in-dept** is one-to-many.

2.3. Making and Deleting Entities

define-entity-type defines a *constructor function* that makes a new entity of this type. For example, the constructor function for the **student** type is named **make-student**. The arguments to the function specify the initial values for the attributes, using keyword arguments, and the returned value of the function is the entity handle for the new entity. This syntax is familiar and natural for Lisp programmers, because it is just like the syntax of constructor functions for Common Lisp structures. Example:

```
(make-student :name "Fred" :dept english-dept :courses (list english-101 history-201))
```

The function **delete-entity** takes any entity handle as an argument, and deletes the entity. It also removes from the database any reference to the entity. Suppose we delete entity *e*. If the value of a single-valued attribute is *e*, the value is changed to the null value. If the value of a set-valued attribute contains *e* as one of its members, *e* is removed from the set. Thus, there are never any "dangling references" to entities in a database. Statice maintains "referential integrity" [Date 82]. If any such single-valued attribute has the **:no-nulls** option, the attempt to delete the entity signals an error.

Entities are never garbage-collected, because they can never become garbage. If an entity exists, it is always reachable, because you can always use associative access (see below) to find all entities of a given entity type. This is a fundamental difference between the data models of Lisp and Statice.

2.4. Associative Access

The **for-each** special form provides associative access to entities in a database. It iterates over entities of a type or in a set, optionally selecting on the basis of attribute values, optionally sorting by some value. It serves as the non-procedural query language of Statice. Its syntax is similar to that used by Common Lisp iteration functions such as **dolist** and **dotimes**.

The following form computes the sum of the salaries of all instructors who are in the English department and whose names follow "M" alphabetically:

```
(let ((result nil))
  (for-each ((i instructor)
             (:where (and (string-greaterp (person-name i) "M")
                          (eq (instructor-dept i) (department-named "English")))))
    (incf result (instructor-salery i)))
  result)
```

The contents of the **:where** clause is always a valid Lisp form that expresses the condition of the query. **for-each** allows an extremely restricted subset of Lisp forms in a **:where**. But by making the syntax a subset of Lisp's syntax, **for-each** looks natural and is easy to understand. The body of the **for-each** can be an arbitrary sequence of Lisp forms.

for-each has several other capabilities. A query can involve more than one variable; this works like a "join". **for-each** can also iterate over the elements of a set-valued attribute. There can be an **:order-by** clause, to process the entities in sorted order.

2.5. Transactions

To provide concurrency control and recovery, Statice uses the traditional concept of serializable *transactions*. All operations on Statice databases must be done within the scope of a transaction. The programmer specifies transaction boundaries using a special form called **with-transaction**. Everything inside the dynamic scope of a **with-transaction** happens within the same transaction. If the body returns normally, the transaction commits. If the body returns abnormally (e.g. by doing a Lisp **throw** or **return**, possibly as the result of an error), the transaction aborts.

Transactions are intended to be of short duration, e.g. measured in milliseconds. Many applications in software engineering (and CAD, etc.) need some kind of longer-term concurrency control, spanning durations during which designs are altered, new versions are created, and so on. Several mechanisms for such longer-term concurrency control are discussed in the literature, such as "checking out", version control, and active user notifications [Chou 86] [Bancilhon 85] [Hornick 87]. We feel that different applications are likely to require different paradigms for handling these issues, so we have refrained from "hard-wiring" any one mechanism into the core of Statice. Instead, we plan to supply a library of alternative higher-level concurrency-control mechanisms, built on the basic transactions that Statice provides.

3. Implementation

Statice is organized into three layers of modularity. From lowest to highest:

1. *File level* provides a page-oriented file system. It supports transactions and communication between client and server hosts.

2. *Storage level* provides storage allocation for records within files, and index structures.

3. *Function level* implements entities, data types, accessor functions, and query processing.

The overall structure of the implementation is based on ideas from [Cattell 83] and [Brown 85], with various improvements.

3.1. File Level

File level provides a page-oriented file system. The caller can create and destroy files and pages within files, and can read and write the contents of pages. The caller must open a transaction before doing anything, and all side-effects happen atomically when the transaction commits. The file can be located on the local host, or on a remote host connected by a network. File level manages all network communication, allowing any number of local and remote clients to access a file concurrently.

Concurrency control is implemented with two-phase locking on page granularity. Deadlocks are detected when they occur, and cause one of the transactions to abort and restart. Recovery uses a log, storing "redo" records. A background process performs periodic checkpoint operations to propagate pages from the log to the database, and recover log space used by transactions that have committed, without delaying the operation of transactions.

Contents of pages are kept in page buffers in memory; they are read in from the disk or the network (depending on whether the file is local or remote) as needed. On any client host, there is one buffer for any database page, because all processes on the client reside in the same virtual address space and can use the buffer (under control of the locks, of course).

In most database systems, buffers can only be used for the duration of a transaction, because after the transaction commits and the locks are released, the page might be changed by some other host, rendering the contents of the buffer obsolete. However, it is important to retain these buffers in an object-oriented database, because the same objects are often used in many successive transactions. File level utilizes a novel cache coherence protocol that allows the client to use retained buffers in subsequent transactions, without the need to communicate with the server before using the page. This greatly decreases response time in typical object-oriented database scenarios.

File level also includes a facility to back up databases to magnetic tape or optical disk. The backup mechanism uses the "fuzzy dump" technique described in [Gray 78] so that transactions can continue to run while backup dumping is in progress.

3.2. Storage Level

Storage level is built on file level, and is responsible for the physical organization of databases. It provides variable-length *records* to its caller. A record is an arbitrary-length vector of words. There are entrypoints to create, delete, grow, and shrink a record, and to read words from and write them to a record. Storage level does not concern itself with the contents of records.

The implementation of records is similar to that in System R's RSS [Astrahan 76]. Records are addressed by *record identifiers*, or *RID*'s, which contain an page number and an offset into a table of descriptors at the end of that page. The RID for a record never changes, even if the record changes size, and the descriptor can indirect to another page if a record grows too large to fit on its original page. Storage level concerns itself with placement of records onto pages, and its caller never has to deal with the underlying fixed-sized pages.

Unlike System R, Statice records can grow to be extremely large. Large records have a tree of blocks, each block within one page; the branch blocks contain RIDs to the next level, and the leaf blocks hold the contents of the record. Large records are important for storing values such as long text strings and image data.

The caller can define several *areas*, and allocate any record in a particular area. Areas are disjoint sets of pages. By allocating records in the same or different areas, the caller can exert some control over locality and clustering.

Storage level provides a sophisticated B*-tree implementation, supporting variable-sized keys and multiple associated values. It uses the Prefix B-tree algorithm [Bayer 77] and the techniques described in [McCreight 77] for efficient handling of variable-size data.

A much simpler B*-tree implementation, called *B-sets*, is also provided. A B-set has one-word keys and no values. Thus, it maintains a sorted set of 32-bit numbers with fast insertion and deletion. Because of its simpler format, it attains a higher branching factor than the general B-trees, and so is more efficient for callers that only need a simple set.

3.3. Function Level

Function level provides all the database functionality of Statice, by building on storage level. Each entity is represented by a record (the *entity record*), which contains a pointer to the record representing the type of the entity, and a 96-bit identifier that is unique over all space and time (a *UID*). Values of the single-valued attributes of an entity are stored in the entity record, including values of inherited attributes. Each value of a set-valued attribute is stored in its own *tuple record*, which is like a tuple in a normalized relational database.

Function level uses the B*-trees of storage level to maintain indexes on attribute values. For example, the **:inverse-index** option of the **name** attribute of the **person** entity type tells function level to maintain a B*-tree. Each index entry's key is computed from the name string, and the entry's value is the RID of the entity record. Thus the inverse access function **person-named** is sped up.

The **:multiple-index** option of the **course** entity type makes an index whose keys are formed from the **title** string and the RID of the **dept** entity. The **:unique** option on the index ensures that only one course can have a particular pair of title and department. The key values are computed by function level, a different way for each type, combined using the algorithm described in [Blasgen 77], and handed as untyped data to storage level.

There is also a different kind of index, called a *group index*, which function level uses when indexing is based on entity values instead of data values. The **:inverse-index** option of the **instructor** attribute of **course** is an example of a group index. Given an instructor, we want to be able to locate quickly all the courses taught by the instructor. The group index associates with each instructor a set of RIDs, pointing to the entity records of the instructor's courses. When the set is small, function level stores the RIDs in a record; when it becomes larger, it converts the representation to use a B-set. The group index lets the **courses-taught-by** accessor function find the courses by directly following pointers (RIDs). (In relational databases, this kind of indexing is sometimes called *pre-linking*.) Note that finding the instructor of a particular course works by direct pointer-following regardless of indexes, since the course entity record stores the RID of the instructor's entity record.

Entity handles are created by function level when an entity is first referenced. A hash table, keyed on the internal address of the entity, makes sure that only one handle is created for any entity, to preserve the identity property. An entity handle contains the RID of the entity it references, so that accessor functions can go directly to the entity record for attribute values, without any index lookups. This provides the fast "direct access" mentioned above.

for-each uses a query optimizer that checks for the presence of helpful indexes and uses them when appropriate. Indexes can be created or deleted at any time, with no visible semantic effect, allowing the database to be tuned as requirements change over time.

3.4. Processes and Processors

Sharing of objects is accomplished by accessing databases on remote workstations, using network communication. This is done by file level. When using a database stored on a remote workstation, the application program calls down through the levels, and file level communicates with the file level on the server host. This means Statice causes little process-switching overhead on the client host: function and storage level, and the client part of file level, all run in the application's process.

There are two reasons for running function and storage levels on the client host instead of on the server. First, in many applications, there is little or no contention for database writes, thus much of the database remains cached on the client host. Second, this organization takes advantage of the ample processor power available at each client, preventing the server's CPU from becoming a bottleneck. The impressive performance consequences of such an organization were demonstrated in [Rubenstein 87].

3.5. Implementing in Lisp and Genera

The Symbolics superset of Common Lisp and the Genera operating environment worked out very well as the implementation basis of Statice. Modern Lisp provides a full range of structured programming constructs and data types. The run-time type and bounds checking, and the powerful programming environment of Genera, made development easy and fast [Walker 87]. The Symbolics implementation of Lisp is very efficient. There were no drawbacks imposed by using Lisp.

We used Lisp's object-oriented programming within the Statice implementation in many places. For example, there are two kinds of page in file level, **local-page** and **remote-page**, both of which inherit from **basic-page**. We also took advantage of Genera's user interface management system, to build a database browser and the interface to the backup dumper. Genera lets a Lisp program access operating system facilities directly, so we could use the network and the local disk efficiently.

Statice uses a high degree of multiprogramming, particularly when a host acts as a server. Because all processes run in the same address space, they can directly share Lisp objects such as the instances that represent pages, files, file systems, buffers, as well as the internal locks that protect them. This makes the implementation elegant as well as efficient.

4. Conclusions and Status

Statice provides persistent, shared storage for Genera. The data model provides data independence, expressive power, and a natural and clear interface to the host language. The implementation provides efficient access, both associative and direct, and transactions.

The implementation is about 36,000 lines, plus 22,000 lines of test suites, all in Symbolics Common Lisp. The present implementation is built on Flavors [Moon 86]. When CLOS is defined and implemented, Statice will be upgraded to use it; we expect this to be easy since Flavors and CLOS are very similar, and most of the differences do not affect Statice.

Several small applications have been written using Statice by software developers at the Symbolics Cambridge Research Center, and Statice is in alpha-test at several sites. In the future we plan to use Statice for software development tools closer to the heart of software development, including version control, configuration management, the electronic mail system, source code and object code, the relationships between procedures (*a* calls *b*, *a* uses *b* as a variable, etc), and so on. These tools will base their data integration on Statice instead of virtual memory. Statice itself will acquire new features and tools, and, like any database system, its performance will always be under improvement.

5. Acknowledgments

The authors would like to thank Dave Stryker, Bob Kerns, Sonya Keene, Penny Parkinson, and especially Dave Moon for their contributions to the design of Statice.

References

[Astrahan 76] M. M. Astrahan et. al.
System R: Relational Approach to Database Management.
ACM Transactions on Database Systems 1(2):97-137, June, 1976.

[Bancilhon 85] Francois Bancilhon, Won Kim, and Henry Korth.
A Model of CAD Transactions.
In *11th Int'l Conference on Very Large Data Bases*, pages 25-33. Morgan Kaufmann, 1985.

[Bayer 77] Rudolph Bayer and Karl Unterauer.
Prefix B-Trees.
ACM Transactions on Database Systems 2(1):11-26, March, 1977.

[Blasgen 77] Michael W. Blasgen, Richard G. Casey, and Kapali P. Eswaren.
An Encoding Method for Multifield Sorting and Indexing.
Communications of the ACM 20(11):874-878, November, 1977.

[Brown 85] Mark R. Brown, Karen N. Kolling, Edward A. Taft.
The Alpine File System.
ACM Transactions on Computer Systems 3(4):261-293, September, 1985.

[Cattell 83] R. G. G. Cattell.
Design and Implementation of a Relationship-Entity-Datum Data Model.
Technical Report CSL-83-4, Xerox PARC, May, 1983.

[Chou 86] Hong-Tai Chou and Won Kim.
A Unifying Framework for Version Control in a CAD Environment.
In *12th Int'l Conference on Very Large Data Bases*, pages 336-344. Morgan Kaufmann, 1986.

[Daplex 84] *Daplex User's Manual*
Computer Corporation of America, Cambridge, MA, 1984.
CCA-84-01.

[Date 82] C. J. Date.
An Introduction to Database Systems.
Addison-Wesley, 1982.

[DeMichiel 87] Linda G. DeMichiel and Richard P. Gabriel.
The Common Lisp Object System: An Overview.
In *ECOOP Conference Proceedings*, pages 201-220. , June, 1987.
Also published as a special issue of Bigre, No. 54, June 1987.

[Gray 78] J. Gray.
Notes on Database Operating Systems.
In R. Bayer, R. M. Graham, and G. Seegmuller (editor), *Operating Systems -- An Advanced Course*, pages 393-481. Springer-Verlag, New York, 1978.

[Hornick 87] Mark F. Hornick and Stanley B. Zdonik.
A Shared, Segmented Memory System for an Object-Oriented Database.
ACM Transactions on Office Information Systems 5(1):70-95, January, 1987.

[Kempf 86] James Kempf, Alan Snyder.
Persistent Objects on a Database.
Technical Report STL-86-12, Hewlett-Packard Laboratories, September, 1986.

[Kent 79] William Kent.
Limitations of Record-Based Information Models.
ACM Transactions on Database Systems 4(1), March, 1979.

[McCreight 77] Edward M. McCreight.
Pagination of B*-Trees with Variable-Length Records.
Communications of the ACM 20(9):670-674, September, 1977.

[Moon 86] David A. Moon.
Object-Oriented Programming with Flavors.
In *OOPSLA Conference Proceedings*, pages 1-8. ACM, November, 1986.
Also published as SIGPLAN Notices, (21):11, November, 1986.

[Moon 88] David A. Moon.
The Common Lisp Object-Oriented Programming Language Standard.
In W. Kim and F. Lochovsky (editor), *Object-Oriented Concepts, Applications, and Databases*, pages (forthcoming). Addison-Wesley, 1988.

[Rubenstein 87] W. B. Rubenstein, M. S. Kubikar, R. G. G. Cattell.
Benchmarking Simple Database Operations.
In *ACM SIGMOD Annual Conference*, pages 387-394. ACM SIGMOD, 1987.

[Shipman 81] David W. Shipman.
The Functional Data Model and the Data Language DAPLEX.
ACM Transactions on Database Systems 6(1), March, 1981.

[Steele 84] Guy L. Steele Jr.
Common Lisp: The Language.
Digital Press, 1984.

[Symbolics 88] *Programming the User Interface*
Symbolics, 1988.
Genera 7.2 documentation.

[Vegdahl 86] Steven R. Vegdahl.
Moving Structures between Smalltalk Images.
In *OOPSLA Conference Proceedings*, pages 466-471. ACM, November, 1986.
Also published as SIGPLAN Notices, (21):11, November, 1986.

[Walker 87] Janet H. Walker, David A. Moon, Daniel L. Weinreb, and Mike McMahon.
The Symbolics Genera Programming Environment.
IEEE Software 4(6), November, 1987.

Entity-Relationship Database User Interfaces

T. R. Rogers, R. G. G. Cattell

Information Management Group
Sun Microsystems, Incorporated
Mountain View, California

Abstract

We report on experience with database user interfaces that are entity-relationship oriented, rather than relation-oriented, and provide a new level of ease-of-use for information management. Our goal is to allow technical workers with little or no knowledge of database systems, query languages, or relational terminology to use databases to solve simple information management problems. Our tools also provide new capabilities for expert users, such as database browsing using a mouse and database display using bitmap graphics.

Keywords: Entity-Relationship, Ease-of-Use

1. Introduction

Relation-oriented user interfaces allow the definition and manipulation of tables. The default interfaces deal with a single table at a time; the relationship between the data in different tables is in the mind of the user instead of being supported by further semantics and primitives in the database system or its user interface. By contrast, an entity-relationship user interface supports the definition and manipulation of entities, which may be instantiated as many records from many tables, and relationships, which are instantiated as one or many records from one table. Our model supports entities and relationships similar to those defined in the entity-relationship data model [Chen 76].

In this paper, we briefly discuss features of the underlying database system that is required to support entity-relationship interfaces. We then describe the architecture and operation of two new tools that were made possible by the underlying entity-relationship platform.

2. Database System

We feel that a database system must provide *at least* four classes of features in order to support entity-relationship capabilities in a production engineering environment. First, an entity-relationship database system must support data model independent capabilities such as concurrency control, report generation, forms-based data entry and data modification, large amounts of persistent data, and transactions for recovery. Next, the system has to support all relational capabilities since these provide the necessary simplicity and power for many types of database activity. We do not want to lose or trade such important capabilites for entity-relationship features. In addition, we require the DBMS be extended to support entity definition primitives, such that an entity consists of a record from an entity table plus records from all of the other tables in the database that reference the entity record. The last feature class for entity-relationship operations involves capabilities for entity manipulation

Sun Workstation® is a registered trademark of Sun Microsystems, Inc.
UNIX™ is a trademark of AT&T Bell Laboratories.
UNIFY® is a registered trademark of Unify Corporation.
SQL™ is a trademark of International Business Machines Corporation.

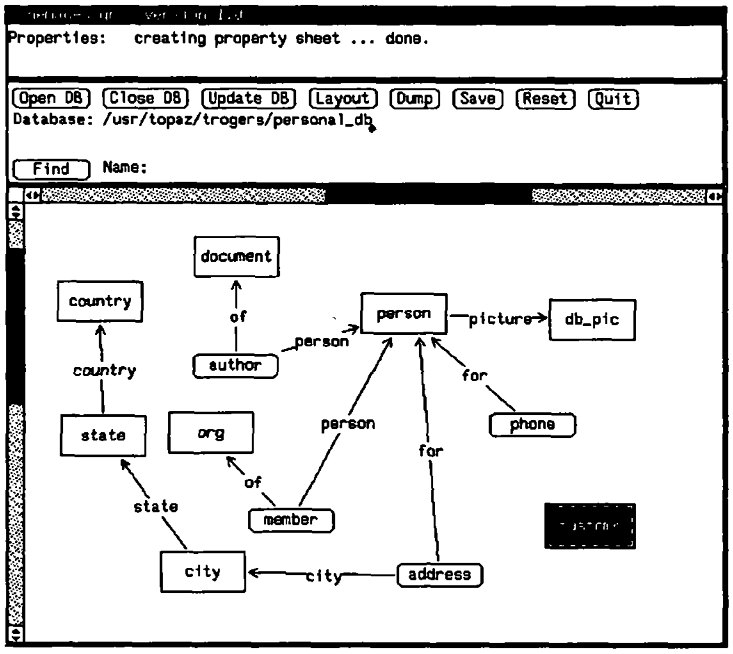
with acceptable interactive response time. Many relational systems support embedded query languages, but almost no relational systems support the performance we require for entity-relationship user interfaces [Rubenstein 87] [Learmont 87].

We now discuss the architecture and operation of two new entity-relationship user interfaces. The first tool is schemadesign, used to define the database schema. The other tool is databrowse, used for editing and browsing data in the database.

3. Schemadesign

Schemadesign is a window-based tool with which users can graphically create and display the database schema by using an entity-relationship diagram. Schemadesign's graphical representation is much easier to understand than the linear listing of tables in conventional relational systems. It is most commonly used by database administrators to define databases; however, it is also a very useful tool for viewing the schema for an existing database. The tool contains a message window for messages and a command subwindow for typing in information or clicking commands with the mouse. The editing subwindow is used for mouse and keyboard entry operations.

In Figure 1, two types of boxes are shown in the editor subwindow: entity (square) boxes and oblong (relationship) boxes. Entity boxes are used to represent tables with keys, and relationship boxes denote tables that have no key. Typically, relationship tables reference at least two entities, although this is not a requirement. Relationship tables can be transformed into entity tables by defining a key for them.



```
Properties:   creating property sheet ... done.
Database: /usr/topaz/trogers/personal_db
```

Figure 1: schemadesign with customr table selected

Constraint-checking through the use of a built-in referential integrity [Codd 81] feature provides a level of data integrity checking not found in most database management systems. The constraint checking is provided by the database system itself; it is not provided as an add-on by schemadesign. The arrows drawn between the boxes represent graphically what would be referred to as foreign keys in the relational model, although most relational database management systems do not implement them. The ability to perform integrity checks when records are inserted, deleted, or modified is recognized as a key entity-relationship enhancement to a database system [Schwarz 86]. In fact, such extensions are being considered for for inclusion in the ANSI SQL2 standard [ANSI 87]. As an example, the referential integrity facility would prevent the deletion of an organization referenced by its members. Likewise, it would not be possible to directly add a person as a member of an organization if the organization did not exist in the database.

There are three main steps to using schemadesign:

Identify 'entities'

The user identifies entities (objects) in the real world that he wants to model in the database system. Examples are people, cities, and documents.

Make entity tables for them

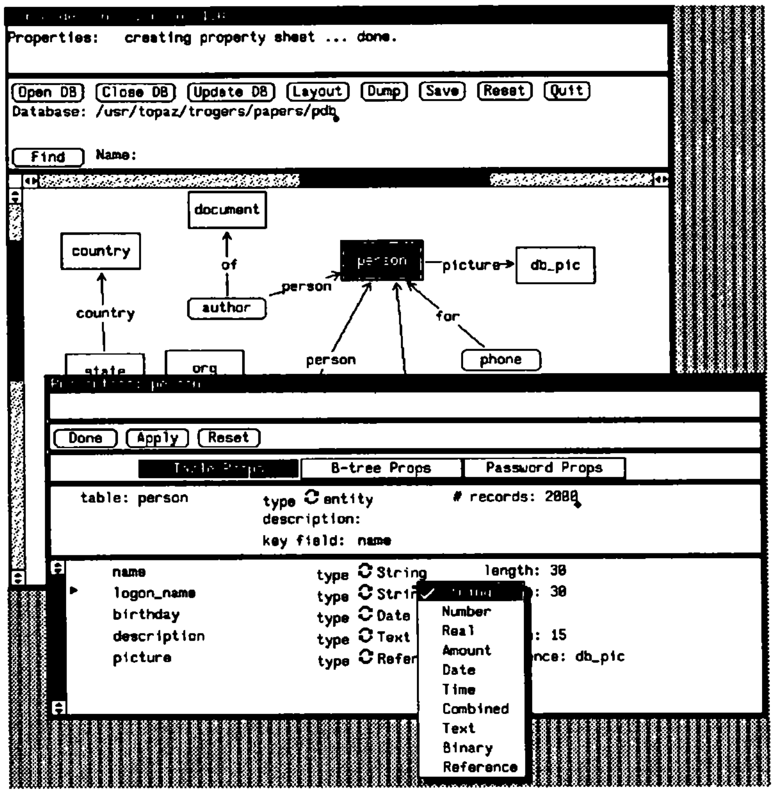
An entity table is created for each real-world object. Creating a table using schemadesign is a simple process of menu/mouse selection and table naming. A "property sheet" provides field name entry and field type selection and definition, as shown in Figure 2. Data types for fields are defined either by selecting the type with the mouse by clicking on the circle made from arrows icon or by selecting the type from a pull-down menu. For example, in Figure 2 we illustrate using a menu to select the string data type for the logon_name field.

Identify features

A feature can be thought of as more information about an entity. Examples would be the age of a person or a list of the organizations to which the person belongs. There are a few simple rules to follow when identifying features of entities. Following these guidelines encourages normalization ([Date 85] [Codd 72]) of the data without any knowledge of normalization theory on the part of the user. The user makes either fields or relationship tables for the features depending on whether they are 1-to-1 or many-to-1 with the corresponding entity, respectively. For example, since a person has only one date of birth, date of birth is said to be "1-to-1" with the person entity. Therefore, we model this relationship in our schema by making the date of birth feature a field in the entity table for person. Similarly, since an employee can have many phones (such as one at work, one in the car, and one at home), i.e. phone numbers share a "many-to-1" relationship with a person, we capture this information in the schema with the relationship table phone. It has a field in it that references the people entity. In this manner, we can add as many phones for the same person as we like. Schemadesign lets the user create many-to-1 relationships simply by selecting the reference table with the mouse, selecting connect from a menu, and drawing a line to the referenced (entity) table. The required reference field in the referencing table is automatically generated. It is also very simple to model many-to-many relationships by composing them out of 1-to-many connections: drawing two or more connections from a reference table to entity tables.

Schemadesign helps the user understand the structure of the schema by not allowing the creation of a database from an incorrectly defined schema. Although it is possible to automatically generate reference fields, the tables do not become fully defined until the entity tables have key fields defined for them. In Figure 1, we notice that the customr table is surrounded by a dotted box. This indicates that the customr table is not completely defined. In this particular case, it is not defined because no key field has been specified.

In addition to schema definition, schemadesign allows easy schema modification. Users delete or rename tables and fields by selecting commands from a pop-up menu, and they can customize the graphical layout of lines and icons. Scrollbars can be used to move about in large schemas, and a find facility is available to locate tables with particular names. B-tree and password properties can also be changed using schemadesign. After modification, schemadesign automatically reconfigures the existing database as necessary to fit the new design.



```
Properties:   creating property sheet ... done.

Database: /usr/topaz/trogers/papers/pdb
Name:

table: person        type entity         # records: 2000
                     description:
                     key field: name

  name               type String         length: 30
▸ logon_name         type Strin          : 30
  birthday           type Date
  description        type Text           : 15
  picture            type Refer          nce: db_pic
```

Figure 2: schemadesign with property sheet and menu

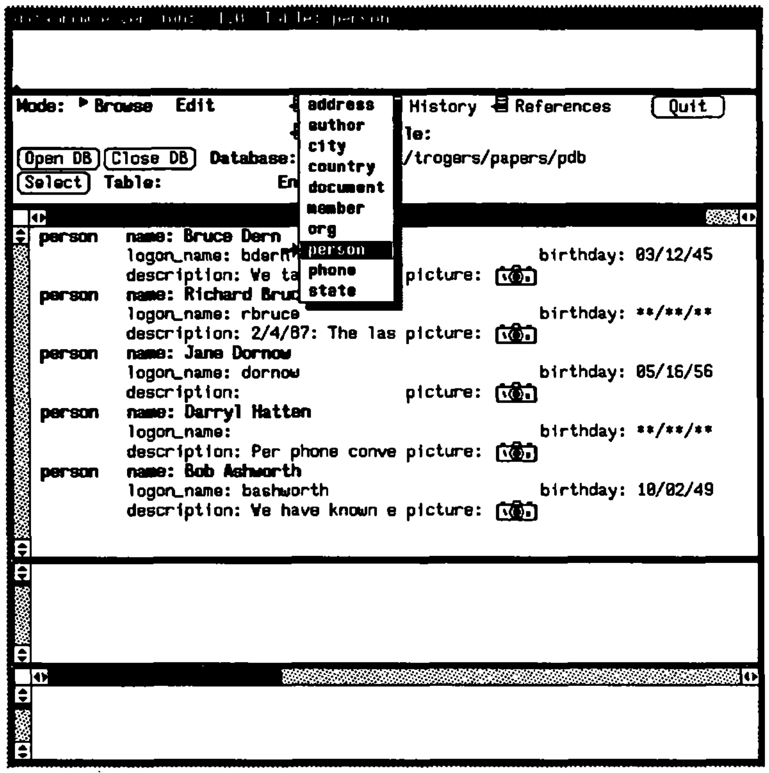
We have found that a graphical display is a useful and powerful tool for both the neophyte and expert users. We are implementing further extensions for the tool such as a "verbose mode" to allow seeing all fields for a table at once, and a group move facility to allowing moving a number of tables and references as one unit.

4. Databrowse

Databrowse is a window-based program that allows viewing and editing of logical entities instead of the single records found in conventional relational databases. It is built upon the standard SunView window system [Sun 86d] and the ERIC SunUNIFY programmatic interface [Sun 86b]. It can also be used to browse and edit information in the schema, but not change the schema itself.

Databrowse has a number of subwindows. From top to bottom they are a message subwindow for messages; a command subwindow for selecting entities and relationships, changing between databases anywhere in the computer network, and other operations; an editor-browser subwindow for viewing and editing data; a text subwindow for displaying and editing text fields; and a picture display subwindow with a picture editing subwindow. The text subwindow is only present if the database contains text fields, and the picture subwindows are only present if the database contains picture fields.

Data can be displayed in the familiar tabular format as shown in Figure 3. The person table was selected simply by using the mouse and a menu which are also displayed in the figure. Table names and entity names appear in **bold** face. Each field name (label) has a colon appended to it. Most data appears as ASCII text, but picture fields appear as camera icons. The user can scroll back and forth through the table using the scrollbars. Not shown is a property sheet that allows the user to tailor data display characteristics.



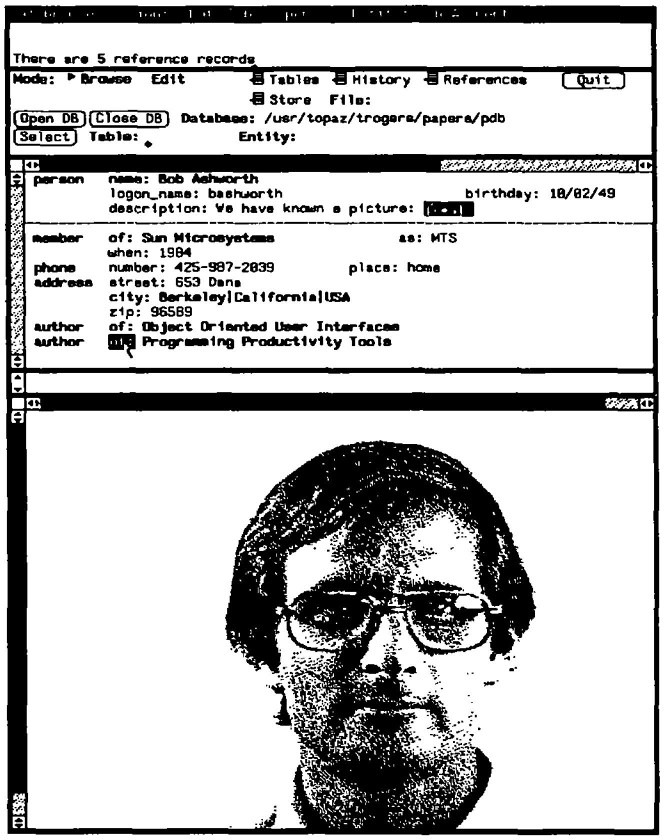
```
person   name: Bruce Dern
         logon_name: bdern                              birthday: 03/12/45
         description: We ta            picture:
person   name: Richard Bruc
         logon_name: rbruce                             birthday: **/**/**
         description: 2/4/87: The las picture:
person   name: Jane Dornow
         logon_name: dornow                             birthday: 05/16/56
         description:                  picture:
person   name: Darryl Hatten
         logon_name:                                    birthday: **/**/**
         description: Per phone conve picture:
person   name: Bob Ashworth
         logon_name: bashworth                          birthday: 10/02/49
         description: We have known e picture:
```

Figure 3: databrowse in table mode displaying person records

In addition to looking at single tables, databrowse lets the user view entities. The data in an entity consists of information from a number of records in potentially many different tables in the database. The user can click on any bold face data, with the exception of table names, to display an entity. For instance, if the user clicks on either "Bob Ashworth" or the field label "name:" for Bob Ashworth in Figure 3, the editor-browser subwindow is erased and the new entity information is displayed as shown in Figure 4. The entity record itself is above the dotted line, and the reference records are listed below the dotted line. The entity name which is a field in each referencing record is not shown, to reduce the display of redundant data. There are several ways to scroll to the other reference records for this person that are not shown in the figure.

Databrowse can be used to store pictures either in the database itself, or pictures can be stored in files, with simply the file name in the database. The underlying database system supports the binary data type, but it is only used by databrowse to store pictures. User-written programs can do whatever they want with the binary fields, however, including using them for storing binary program files, images, or other types of unformatted data. In Figure 3 the person's picture is displayed by clicking the camera icon or the "picture:" field label. The selected camera remains in reverse video.

Note that the instantiated entity is *not* one record, and it is *not* possible to get the necessary display information using a relational view. It would be difficult to get the same display information using a purely relational system. First, a simple relational view would be cumbersome because the targetlist of the view would have to be needlessly complicated. Second, a join would not work in any case where there was no matching record in any one of the join tables, unless the outer join syntax were supported [Date 83]. Third, retrieving the information would probably have to be done as a series of two-table queries which would take too long. Using the underlying capabilities of ERIC, the new display of an entity is obtained directly from the database in a fraction of a second, without read-ahead or caching in the program.



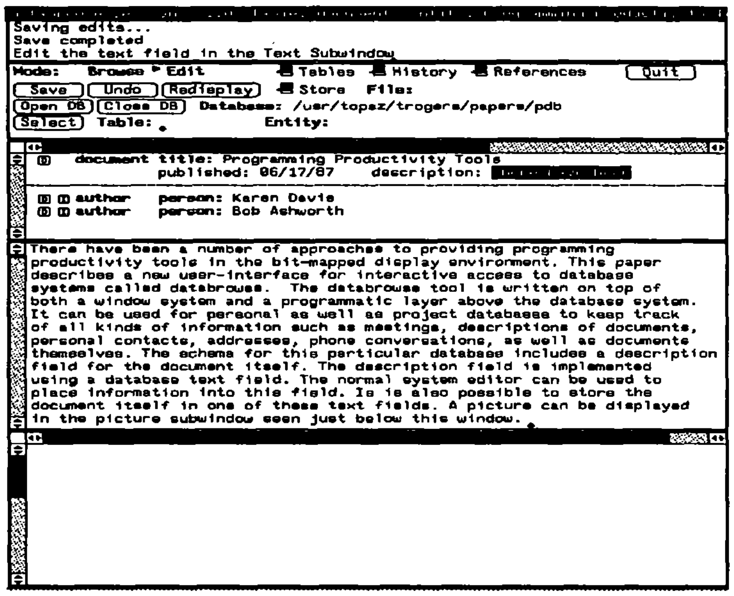
```
There are 5 reference records
Database: /usr/topaz/trogers/papers/pdb

person   name: Bob Ashworth
         logon_name: bashworth                         birthday: 10/02/49
         description: We have known a picture:
member   of: Sun Microsystems                 as: MTS
         when: 1984
phone    number: 425-987-2039          place: home
address  street: 653 Dana
         city: Berkeley|California|USA
         zip: 96589
author   of: Object Oriented User Interfaces
author   of: Programming Productivity Tools
```

Figure 4: databrowse with picture selected

After viewing the person entity, the user may want to see more information about one of the papers that the person authored. This is done simply by clicking on the name of the paper or the field label for the particular paper. In Figure 4, we show the user selecting the "of:" field label. Figure 5 shows the new entity (the paper) being displayed. There are several new features shown in this figure. Foremost is the fact that the user is now in edit mode, and can edit any data on the screen. In this mode, labels are used for browsing and data, when clicked, can be edited. Delete, "D", and insert, "I", boxes are shown. It is possible to use these to delete reference records and add reference records. These operations, of course, are subject to the referential integrity constraints applied by the underlying database system.

Note that no forms design is necessary for accessing an entity. Databrowse essentially creates a default form. In editing mode, an empty record is shown for each reference record type that has no actual records which reference the current entity. Included is a reference menu item which shows a menu of referencing from which a specific reference set can be chosen. The default form has most of the desired information about an entity, which is what the user wants most of the time.

Databrowse can also be used for text manipulation. Text fields are edited by the user in the text subwindow which has the full power of the normal Sun View text editor. It is possible to scan in entire pages of data using scanners, or an ASCII text file can be created outside of databrowse with the user's favorite text editor. The file can then be read into the text subwindow for editing. When the editing is completed, the text can be saved to the database or a file. In Figure 5 we show the contents of the description field for the document. We could just as easily have stored the entire document in the database. As with picture fields, the selected field is highlighted using reverse video.



```
Saving edits...
Save completed
Edit the text field in the Text Subwindow
Mode:   Browse ▸ Edit        Tables   History   References     Quit
Save  Undo  Redisplay        Store   File:
Open DB  Close DB  Database: /usr/topaz/trogers/papers/pdb
Select  Table:              Entity:

 D   document title: Programming Productivity Tools
              published: 06/17/87    description:

 D D author    person: Karen Davis
 D D author    person: Bob Ashworth

There have been a number of approaches to providing programming
productivity tools in the bit-mapped display environment. This paper
describes a new user-interface for interactive access to database
systems called databrowse.  The databrowse tool is written on top of
both a window system and a programmatic layer above the database system.
It can be used for personal as well as project databases to keep track
of all kinds of information such as meetings, descriptions of documents,
personal contacts, addresses, phone conversations, as well as documents
themselves. The schema for this particular database includes a description
field for the document itself. The description field is implemented
using a database text field. The normal system editor can be used to
place information into this field. Is is also possible to store the
document itself in one of these text fields. A picture can be displayed
in the picture subwindow seen just below this window.
```

Figure 5: databrowse with text selected for editing

5. Related Work

The idea of an entity-relationship diagram editor such as schemadesign is not new -- almost anyone who read the original [Chen 76] probably thought of it. Surprisingly, there have been few actual commercial implementations until recently [Chen 87] [CCA 87] [ETH 87]. There have also been some research prototypes in this direction [Chan 80] [Gold 85].

We have presented experience with the implementation and use of such a tool, and demonstrate that such a tool can easily be built on a relational system. In our experience, our users have always tended to draw entity-relationship diagrams on paper; now this task has been automated.

The databrowse tool is unique in both the research and development worlds, to our knowledge, except for the author's previous work [Cattell 83]. The two features that make databrowse unique are its entity-centric orientation, possible only through custom form design in other systems, and its browsing capability to move between database entities. The closest comparable work is probably [Motro 86], who reports on navigating entity-relationship connections by typing natural-language-like commands.

Other work on so-called database "browsers" has actually consisted of *scrolling* through records in a single table [Stonebraker 82] or of typing queries that allow a "fuzzier" specification of data [Motro 87]. Larson [Lars 86] also reports on some early similar approaches for browsing.

There is some work analogous to databrowse in the user's mode of interaction. Recently, [Delisle 86] reports work on such a system for hypertext. However, hypertext lacks the underlying database structure provided by the relational system, and thus (1) the same database of entities and relationships cannot be queried through other means, and (2) there is less uniformity of structure, so users are more likely to get confused [Mantei 82].

Unlike databrowse, some systems allow query specification and thus "browsing" from a graphical schema representation [Wong 82] [King 84] [Fogg 84] [Gold 85] [Lars 86] or from combinations of menus and type-in areas [Grap 87]. This is generally a multi-step process of moving between specification and query results or incremental query construction using the schema whereas databrowse allows the user to move directly from entity to entity based on selection of actual data values.

6. Future Work

Schemadesign and databrowse are available as a commercial product [Sun 86b]. We are looking at a number of extensions to them, to allow user-defined data types and associated procedures, control of databrowse's record display to allow custom applications, and convenience features for schemadesign. Some minor extensions can be made to databrowse to allow queries, and a more sophisticated query tool utilizing graphics is being constructed. In addition, we have built a tighter coupling between databrowse and the schema in which the portion of the schema corresponding to the current entity is graphically displayed.

We are also porting databrowse, schemadesign, and the collection of tools we build to other commercial database products. This is not a trivial task, since (1) we require extensions to the underlying relational model to incorporate new semantics, and (2) we require high interactive performance often not available at the level of SQL. However, we believe the task is feasible with most DBMSs, including some non-relational ones.

7. Summary

We have built entity-relationship user interfaces to a DBMS oriented towards ease-of-use. Schemadesign focuses on a difficult problem in DBMS ease-of-use: designing a database. It provides a simple several-step process not requiring knowledge of relational design and normalization theory. Databrowse provides a default form for database entities spanning many tables. In our experience this is the database "form" that is required for most applications, and thus provides a user interface with no further action on the part of the user after schema design. Furthermore, the browsing provides an easy-to-use alternative to a more complex query language for casual users. We use databrowse in our every-day work. Both databrowse and schemadesign are also interesting to "experts". Databrowse browsing provides a fast way to get to simple pieces of data, and schemadesign diagrams are an efficient way to view database designs. Together, they implement zooming, panning, and rudimentary filtering functionality in Larson's browsing taxonomy [Lars 86].

Acknowledgements

Mike Freedman and Bob Marti implemented schemadesign. Tom Rogers implemented databrowse. Tim Learmont, Mike Freedman, Bob Marti, Richard Berger, and Liz Chambers worked on the underlying ERIC implementation.

References

[ANSI 87]
ANSI SQL2 base document. ANSI X3H2-87-144, July 1987.

[Astrahan 80]
Astrahan, et. al: *A History and Evaluation of System R.* IBM Technical Report RJ2843, June 12, 1980.

[CCA 87]
CCA: *DB Designer - Product literature.* Cambridge, MA.

[Cattell 83]
Cattell, R. G. G.: *Design and Implementation of a Relationship-Entity-Datum Data Model.* Xerox PARC Technical Report CSL-83-4, April 1983, see Section 7.

[Chan 80]
Chan, E. and Lochovsky, F.: *A Graphical Database Design Aid Using the Entity-Relationship Model.* In Entity-Relationship Approach to Systems Analysis and Design, North-Holland, 1980, pp 295-310.

[Chen 76]
Chen, P.: *An Entity-Relationship Model -- Towards a Unififed View of Data.* ACM Transactions on Database Systems, 1, 1. January 1976.

[Chen 87]
Chen & Associates, Inc.: *ER designer - Product literature.* Baton Rouge, LA.

[Codd 72]
Codd, E.: *Further Normalization of the Data Base Relational Model.* Data Base Systems, Courant Computer Science Symposia Series, Vol. 6. Prentice-Hall. 1972.

[Codd 81]
Codd, E.: *Referential Integrity.* Proceedings of the Seventh International Conference on Very Large Data Bases, Cannes, France, September, 1981.

[Date 81]
Date, C.J.: *Referential Integrity.* Proc. 7th Int. Conf. on Very Large Databases, 1981, pp. 2-12.

[Date 83]
Date, C.J.: *The Outer Join.* IBM Technical Report TR 03.181. January 1982.

[Date 85]
Date, C.: An Introduction to Database Systems. Volume 1. Fourth Edition. Addison-Wesley. September 1985.

[Delisle 86]
Delisle, N. and Schwartz, M.: *Neptune: a Hypertext System for CAD Applications.* ACM SIGMOD Proceedings, 1986.

[ETH 87]
Institut fur Informatik, ETH: *Gambit - Product literature*, Zurich, Switzerland.

[Fogg 84]
Fogg, D.: *Lessons from a "Living In a Database" Graphical Query Interface.* Proceedings ACM SIGMOD 84, pp 100-106.

[Gold 85]
Goldman, K.J., Goldman, S.A., Kanellakis, P.C., Zdonik, S.B.,: *ISIS: Interface for a Semantic Information System.* Proceedings ACM SIGMOD 85, pp. 328-342.

[Grap 87]
Soretas-Graphael: *G-BASE - Product literature.* Waltham, MA.

[King 84]
King, R.: *Sembase: a Semantic DBMS.* Proceedings 1st International Workshop on Expert Database Systems, Kiawah Island, South Carolina, October 24-27, 1984, pp. 151-171.

[Lars 86]
Larson, J.A.: *A Visual Approach to Browsing in a Database Environment.* IEEE Computer, June 86, pp. 62-71.

[Learmont 87]
Learmont, T., and Cattell, R. G. G.: *Object-Oriented Database Systems.* To appear in Springer's *Topics in Information Systems* Series.

[Mantei 82] Mantei, M.: *Disorientation Behavior in Person-Computer Interaction.* PhD dissertation, University of Southern California, 1982.

[Motro 86]
Motro, A.: *BAROQUE: A Browser for Relational Databases.* ACM TOOIS 4, 2, April 1986, pp 164-181.

[Motro 87]
Motro, A.: *VAGUE: A User Interface to Relational Databases that Permits Vague Queries.* CS Dept, University of Southern California.

[Rubenstein 87]
Rubenstein, W. R., Cattell, R. G. G.: *Benchmarking Simple Database Operations.* Proceedings ACM SIGMOD 1987, pp 387-394.

[Schwarz 86]
Schwarz, P, et. al.: *Extensibility in the Starburst Database System.* Proceedings of the 1986 International Workshop on Object-Oriented Database Systems, September 23-26 1986, pp. 85-92.

[Stonebraker 82]
Stonebraker, M. and Kalash, J.: *TIMBER: A Sophisticated Relation Browser.* Proceedings of the Eighth International Conference on Very Large Databases, September 1982, pp 1-10.

[Sun 86b]
Sun Microsystems, Inc.: *SunSimplify Programmer's Manual.* Sun Microsystems, Mountain View, California, 1986.

[Sun 86d]
Sun Microsystems, Inc.: *Sun View Programmer's Guide.* Sun Microsystems, Mountain View, California, 1986.

[Tuori 86]
Tuori, Martin: *A Framework for Browsing in the Relational Data Model.* PhD thesis, University of Toronto CSRG, 1986.

[Wong 82]
Wong, H.K.T., Kuo, I.: *GUIDE: Graphical User Interface for Database Exploration.* Proceedings of the Eighth International Conference on Very Large Databases, September 1982, pp 22-32.

Defining Constraint-Based User Interfaces

Raimund K. Ege *

Florida International University

Abstract

We describe a new approach to constructing user interfaces by declaratively specifying the relationships between objects via filters. A filter is a package of constraints dynamically enforced between a source object and a view object.

This paper describes a specification language for filters, an implementation of filters with the aid of a constraint-satisfaction system, and a graphical interface for designing filters. We illustrate the power and flexibility of the filter paradigm with an interface example and show that it stimulates and supports the re-use of existing components and gives a design methodology for constructing interfaces.

1 Introduction

This paper presents a new approach to building user interfaces in an object-oriented environment. In such an environment, all entities of interest are represented as objects, so all aspects of user interfaces are modelled as objects. In the Smalltalk model-view-controller (MVC) Paradigm [GR83], for example, the interface consists of model, view and controller objects. The model and view are basically two different representations of the same conceptual entity [Deu86]. In Smalltalk's MVC paradigm, the model and view have procedural components that allow the controller to manage the interface correctly.

The Filter Paradigm

Our approach is to abandon procedural specification of user interfaces and relate the model (*source*) and *view* with a declarative interface specification. The idea is to use constraints to specify the conceptual relation between the *source* and *view* objects. For example, the relationship between an employee object and a bitmap object on a screen can be represented by constraints. The constraints state that the bitmap object always displays a certain rendering of the employee object. The constraints hide the procedurality of the interface. If the bitmap object on the screen is changed, then constraint satisfaction will ensure that the employee object is changed accordingly. If the employee object changes, then that change is reflected on the screen.

*This research was part of the author's Ph.D. work while at the Oregon Graduate Center under the guidance of David Maier and Alan Borning. Author's address: FIU, School of Computer Science, University Park, Miami, FL 33199. E-Mail: EgeR@servax.bitnet

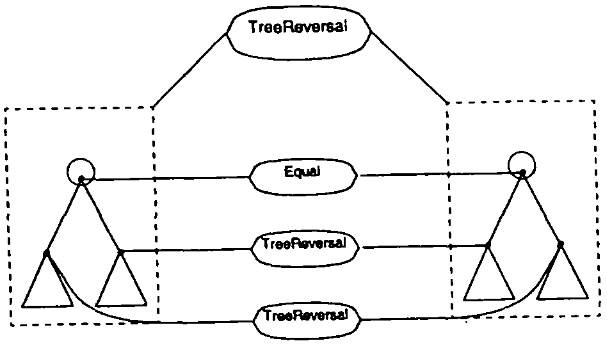


Figure 1: Tree Reversal Filter

A *filter* is an object that describes and maintains these special constraints between objects in an interface. For example, consider an interface between two binary trees. The binary trees store an integer number at each node. The interface should be constructed in a way that one tree is the reversal of the other tree, i.e., the interface can be viewed as the constraint that one tree is the mirror image of the other. This constraint can be represented as a filter between a binary tree as source and a binary tree as view object. Filters are constructed from subfilters on subparts of a source and view object. Figure 1 illustrates this tree reversal for two binary trees of height one constructed from three equality subfilters. The equality subfilters ensure that the integer numbers, stored in corresponding nodes, are kept equal. If a number is changed then the corresponding number on the other side of the filter is changed by the constraint-satisfaction system. If a subnode is added to a node on one side then a subnode is added by the constraint-satisfaction system on the other side of the filter and an equality subfilter is established between them. If a subnode is deleted from a node then the corresponding subnode is deleted from the other side of the filter and the equality subfilter is removed.

The definition of what types of the source and view objects are allowed for the filter and how the subfilters are connected to them is given by the *filter type*. Filter types specify how filters are built from atomic filters using set, iteration and condition constructors. Atomic filters are given by the implementation. The filter and object types are described by a filter specification language (*FiSpeL*) [Ege86]. *FiSpeL* is currently a theoretical tool for composing filters; a compiler and optimizer for it are planned. The *Filter Browser* is a tool for constructing filters graphically. The *Filter Browser* lets the interface designer create filters by defining and manipulating filter types. Subfilters are added interactively by connecting them with the various constructors to the object types that are displayed in the browser. The *Filter Browser* also allows the designer to instantiate a filter from its filter type definition with sample objects to test the constructed interface.

Related Work

Our approach was guided by experience with the Smalltalk MVC paradigm [GR83]. Programming experience has shown that this paradigm is hard to follow. The Smalltalk Interaction Generator (SIG) tried to add a declarative interface on top of the MVC mechanism [MNG86]. One conclusion of SIG is that display procedures need type information about the objects they display and that

Smalltalk does not provide this kind of typing. The Incense system [Mye83] uses type information supplied by a compiler to display objects. The user can influence the display format but cannot update through this system. The Impulse-86 system [SDB86] provides abstractions for objects as well as for interactions. Other approaches using algebraic techniques to specify user interfaces axiomatically [Chi85] seem manageable only for small theoretical examples. Other systems use constraints as their major construct, such as ThingLab [Bor81], which allows constraints to be expressed in a graphical manner. The Animus system [BD86] extends ThingLab with constraints that involve time. An early system that employed constraints to express graphical relations was Sketchpad [Sut63]. The language Ideal [Van81], used in typesetting graphical pictures, is based on constraints and demonstrates their power and usefulness. Bertrand [Lel87] is a term rewriting language that can specify constraint satisfaction systems. In its current implementation, however, it is not interactive and therefore not well suited for our problem.

Section 2 of this paper summarizes the filter specification language *FiSpeL*. Section 3 introduces the *Filter Browser* and elaborates an example session to generate an interface interactively. Section 4 gives details of how the objects, filters, constraints, and the *Filter Browser* are implemented in Smalltalk-80 [1] and how the constraint satisfaction is performed by ThingLab.

2 Objects and Filters

This section summarizes *FiSpeL*, a filter specification language for defining object and filter types. If we want to compose filters from subfilters, connecting objects of different kinds, it is necessary to type the objects to ensure that compositions of filters are well-defined. All entities in our filter paradigm are ultimately implemented by objects, so we put much effort in providing a comprehensive type system for the object model.

Object Types

The object type system supports the notions of aggregation and specialization. With aggregation we can build structured objects from components. Specialization allows us to refine existing objects via a hierarchy of object types and inheritance. We view object types as records. A record is a collection of typed fields. The fields have names called *addresses*. There are constant fields, which are constant for all instances of a type, and there are data fields, which are local to an instance of an object. Fields can be iterated by specifying an iteration factor; fields can be conditional by specifying a condition that must be true for the field to exist; and a field can specify its type recursively. In addition, an object type can inherit fields from other object types and can place constraints on all fields. Object types are used in the filter type definition to describe source, view and variables that are needed to connect subfilters. A field of an object is accessed by traversing a path of field names.

Figure 2 shows the two object types, `Person` and `Employee`. `Person` includes the fields that are common to a person. `Employee` is a specialization of `Person` and defines additional fields, such as `salary` and `jobDescription`. The field `supervises` is an iterated address, i.e., an employee may supervise a number of subordinates. The number of subordinates is specified by the field `subNumber`.

[1] Smalltalk-80 is a trademark of Xerox Corporation.

```
Object Type Person                          Object Type Employee
      lastName → String                           inherit from Person
      firstName → String                          salary → Integer
      socSecNum → Integer                         subNumber → Integer
      address → String                            supervises[subNumber] → Employee
end                                               supervisor → Employee
                                                  jobDescription → String
                                            end

Object Type DisplayEmployee
      icon → PersonIcon
      name → String
      subNumber → Integer
      subordinates[subNumber] → DisplayEmployee
end
```

Figure 2: Person and Employee object types

```
Filter Type EmployeeDisplay ( source : Employee, view : DisplayEmplcyee )
      make set of
            StringEquality (source.firstName, view.name)
            IntegerEquality (source.subNumber, view.subNumber)
            iteration source.subNumber times i
                  EmployeeDisplay(source.supervises[i],view.subordinates[i])
end
```

Figure 3: **EmployeeDisplay** filter type

Filter Types

The filter type system defines the structure of filters. Filters represent constraints between two objects. The filter type defines the types of the source and view objects it relates. The filter type also declares the subfilters that compose the filter. In addition, the filter type can define variables to be used as intermediate objects when subfilters are combined. A filter that is not further decomposed is called a *filter atom* and is directly supported by the implementation. For example, filter atoms are used for low-level input/output, data conversion, or error handling. A filter that has subfilters is called a *filter pack*. The subfilter constructors are: sequence, iteration and condition. The sequence constructor (**set of**) declares several subfilters of possibly different types; the iteration constructor (**iterate** p **times** i) declares a certain number of filters of the same type; the condition constructor (**condition**) declares a subfilter only if a given condition is true. It is possible to declare another instance of the same filter type as a subfilter of the one being defined, much like a recursive procedure call in a conventional programming language.

Figure 2 shows the object type definition for **DisplayEmployee** that contains displayable information for an employee. In order to define an interface that will display an object of type **Employee** we have to extract the displayable information. Figure 3 shows the filter type definition **EmployeeDisplay**. It decomposes the constraint into subconstraints. The subconstraints specify that the name, the number of subordinates and the subordinates are the same in the source object of type **Employee** and in the view object of type **DisplayEmployee**. The example uses the sequence and iteration filter constructors. The iteration names the same filter type for each of the subordinates recursively. The iteration depends on the value that is stored in **source.subNumber**. Whenever that value changes, the number of subfilters that are instantiated for the subordinates

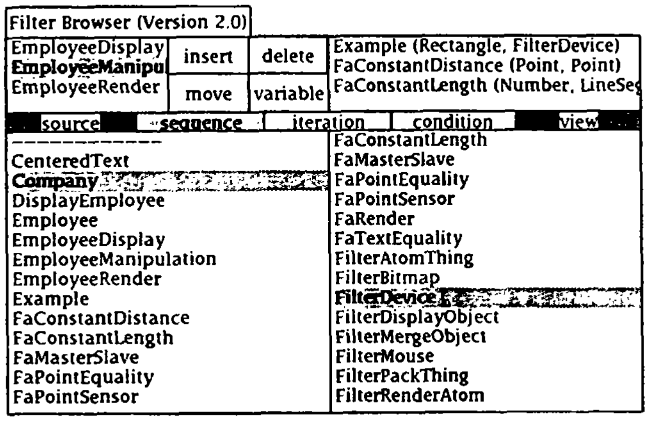


Figure 4: Filter Browser Phase One

is changed. When including a subfilter, the filter type associates source and view paths with it. For example, the subfilter `StringEquality` has `source.firstName` as its source object. When this filter type is instantiated, the filter instance will check whether the address `firstName` in fact refers to an object of type `String`. It is at this point that the path to the field is traversed to retrieve it. The newly defined filter type `EmployeeDisplay` can now be used as subfilter in other filter type definitions. The appendix contains the complete filter and object type definitions for a sample `EmployeeManipulation` interface.

3 The Filter Browser

The filter specification language is one way to define a user interface by defining filter types, but the goal here is to provide a graphical tool to build interfaces. Therefore, a tool similar to the Smalltalk [Gol84], ThingLab [Bor79] or Animus [Dui86] browser, the *Filter Browser*, is used to define, manipulate and test filter types graphically. In defining filter types, we distinguish the external and internal parts of the definition. Externally, the filter type is identified by its name and the types of its source and view object. Internally the filter type specifies subfilters and variables. These details are encapsulated inside the filter type. A session with the filter browser has three different phases. In phase one, the name of the filter type and the type of source and view objects are given. In phase two, the variables and subfilters that participate in filter constructors, such as sequence, iteration and condition are specified. Phase one represents the external, phase two the internal definition. In phase three, a constructed filter type is instantiated. Figures 4, 5 and 6 show examples of the filter browser in each of the phases as we define the filter type `EmployeeManipulation`.

Phase one (Figure 4)

The filter browser is divided into several panes. In this phase only the upper left, lower left and lower right panes are used. The upper left pane shows a list of all filter types that are known to the system. The user can add a new filter type. The current subject of the filter type definition, i.e.,

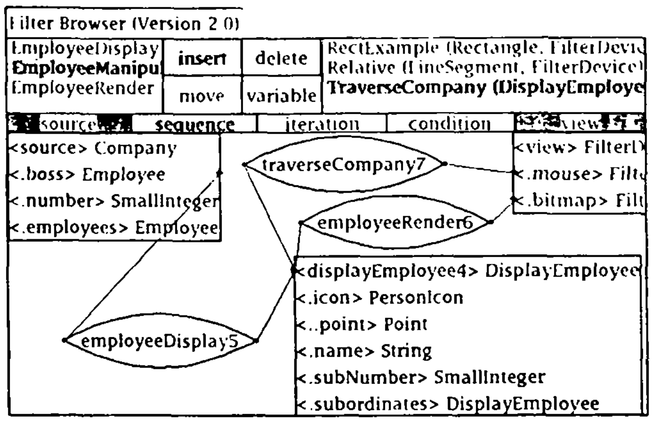


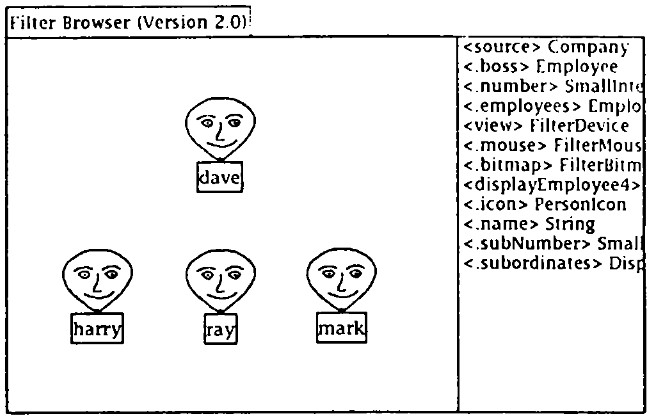
Figure 5: Filter Browser Phase Two

the filter type that will be modified or newly defined, is emphasized. The filter browser displays two lists of all object types that are available for source and view types in the two panes below. The user can inspect all object types and select one for source and view. The object type also provides a sample instance (prototype) that can be used to instantiate the filter in phase three. The **`EmployeeManipulation`** filter type is defined on source objects of type **`Company`** and on view objects of type **`FilterDevice`** (these object types are emphasized).

Phase two (Figure 5)

The upper left pane displays a list of known filter types with the current selection, **`EmployeeManipulation`**, emphasized. To the right there are four panes to select the action that is to be performed on the current filter type. The user can *insert* subfilters, add *variables*, *move* or *delete* subfilters or variables in the picture pane below. The picture pane substitutes the two object lists from phase one. If the *insert* action is selected then the upper right pane shows a list of all filter types and the types of their source and view objects. This list also contains the currently defined filter type to allow the recursive specification of a filter type. The selected element in this list represents the object of the action that is performed, i.e., it names the filter type to be inserted as subfilter in the filter type. If the *variable* action is selected then the upper right pane shows a list of all available object types, from which the user may select. The type of filter constructor (sequence, iteration, condition) is selected with one of the three panes in the middle of the filter browser.

The picture pane is used to display the subfilters, variables and their connections. A variable is placed in the picture pane by selecting the *variable* action. The variable appears as a box that shows its name and type and the paths to its fields with their types. The user selects a location and places the variable. The variable is then inserted into the current filter type definition. A subfilter is placed in the picture pane by selecting the *insert* action and a subfilter from the upper right pane. The added subfilter is then connected (linked) to paths in either the source, view or variable objects by pointing at their location on the screen. For iteration or condition constructors, an iteration or condition object has to be selected by pointing to a path that represents the iteration factor or the condition. After the subfilter is placed in the picture pane it is inserted into the current filter type



```
<source> Company
<.boss> Employee
<.number> SmallInte
<.employees> Emplo
<view> FilterDevice
<.mouse> FilterMous
<.bitmap> FilterBitm
<displayEmployee4>
<.icon> PersonIcon
<.name> String
<.subNumber> Small
<.subordinates> Disp
```

Figure 6: Filter Browser Phase Three

definition.

Figure 5 shows the filter browser as the user inserts the **TraverseCompany** subfilter into the **EmployeeManipulation** filter type. A variable of type **DisplayEmployee** has already been defined and connected to an **EmployeeRender** and **EmployeeDisplay** (see Section 2) subfilter. The **EmployeeRender** filter renders an object of type **DisplayEmployee** on the display bitmap. The **TraverseCompany** filter allows the selection of an employee within a company using a pop-up menu. These filters are already defined. After the *insert* action has been selected and the user moves the mouse cursor into the picture pane a lozenge for the **TraverseCompany** filter appears. The source link is connected to the **displayEmployee4** variable and the view link is connected to the **bitmap** address of the view object.

The picture pane represents the network of subfilters. The absolute position of the subfilters in the picture pane is not important, but it will be stored to redraw the subfilter network in the same way as it was defined. An important issue in connecting subfilters is typing. The external definition of a subfilter specifies the object type for its source and view object. Thus, when a source or view link of a subfilter is connected to addresses of source, view or variable objects of the currently defined filter type, it is necessary to check the corresponding types. These source and view links can be connected to object types that are of the same type as, or are subtypes of, their specified object types. For example, the source of the **TraverseCompany** filter has to be of type **DisplayEmployee** or one of its subtypes.

Phase three (Figure 6)

Instantiating a filter type means creating a Smalltalk object representing a filter. When instantiating, source, view and variable object instances of the correct object types have to be supplied. This instantiation operation is accessible from a pop-up menu in the upper left pane of the browser, where the filter type has been selected. The right pane of the filter browser simulates the display bitmap for the filters and all input is controlled by the filter browser, so it is possible to switch back to the previous phases. The right pane of the filter browser shows the participating instantiated objects. They can be selected, inspected and changed. Any change to participating objects will immediately change the display in the left pane. Figure 6 shows the instantiated **EmployeeManipulation** filter type for a prototype `Company`. The `FilterDevice` is simulated by the filter browser. The pane on the left displays the company's employees. We can traverse the tree of employees and make changes to the employees as they are stored for the company. The appendix lists the complete filter and object type definitions.

In phase three the user can test the filter type by observing its behavior and changing values of variables. The filter type can be changed incrementally. The user can switch back and forth between phases two and three, adding and deleting subfilters and variables or changing values of participating objects. All filter types are constructed from existing filters in a bottom-up fashion. The filter browser lists all existing filters together with their source and view type information. Filter types can only be constructed if the subfilters and variables are connected correctly and the filter can be instantiated while it is being developed to test its behavior. Thus the filter paradigm stimulates and supports the re-use of existing components and gives a design methodology for constructing interfaces.

4 Implementation

The *Filter Browser* is implemented on a Tektronix 4400 Series machine in Smalltalk-80. ThingLab [Bor79], an extension to Smalltalk, is used to do the constraint satisfaction. Filter and object types in *FiSpeL* are represented as classes in Smalltalk. ThingLab extends the Smalltalk class definition with constraints, types for instance variables and dynamic access.

Object Types

Object types are modelled as Smalltalk classes. The fields of an object are instance variables, where the instance variable name is the address of the field. Iterated fields are represented as a set of fields, while conditional fields hold the value "nil" if the condition is not true. Fields can be inherited from superclasses. Field access is done through an *access path* that stores the field names that have to be traversed. Smalltalk does not keep information on the type of instance variables, so ThingLab augments the class definition to hold the type (reference to another object type class) for each instance variable. Constraints that are defined within the object type are also stored in the class definition. An instance of an object type refers back to its class definition, so the constraint-satisfaction mechanism can retrieve the defined constraints. Note, that a subtype also inherits all constraints that are defined for its supertypes. Thus, our implemented typing mechanism does not allow an instance of a subtype to be stored in a typed instance variable. This is clearly an undesirable limitation, which we plan to remove (see [Ege87]). Removing it is not trivial, however, since depending on how the instance variable is used, we may need to ensure that the constraints on a subtype are not more restrictive than those on the specified type.

Filter Types

Filter types are represented as subclasses of the object type classes. Fields are defined for the source and view objects, as well as for the variables. The subfilters are held in instance variables that can be conditional or iterated. The subfilters have source and view object information associated with them, i.e., what fields are used as source and view object for the subfilter. This information is modelled as equality constraints[2] from the field within the source, view or variable object to the appropriate source or view object of the subfilter. For example, consider the `EmployeeDisplay` filter type in Figure 3. The `StringEquality` subfilter is held by a sequence filter constructor; its source is associated with the `source.firstName` field of the filter type. This association is modelled by an equality constraint defined for the `EmployeeDisplay` filter type class.

ThingLab

As mentioned earlier, the filter browser is implemented as an extension to ThingLab. Path access using addresses (field names) and the constraint satisfaction is handled by ThingLab. The field description within ThingLab was augmented to incorporate iterated and conditional fields. Filter atoms are directly implemented as ThingLab constraints. ThingLab's prototypes of things are reduced in their importance in that they are used to hold the graphical information used to display the filter network in phase two and to provide sample values for the instantiation in phase three of the filter browser, but they no longer are used to infer the type of an instance variable.

All objects have to exist within ThingLab. Objects outside ThingLab are the display bitmap, the keyboard, specialized input devices (mice), or existing complex objects in an application. These object are incorporated by either providing a special object in ThingLab that holds the outside object and controls all accesses to it (object holder), or by providing special filters that link existing objects within ThingLab to those external objects (implementation filter atoms). Graphical primitives, such as line rendering or input sensoring [Ege86], are examples for filter atoms to incorporate I/O-objects.

5 Summary and Conclusion

The filter paradigm represents a new approach to interfaces in an object-oriented environment. Constraints are used as the basic building block for interfaces. Constructors are provided to allow building of structured interfaces in a declarative way. The feasibility of the filter paradigm is shown by providing a working interface generation tool. In addition, the object type system represents another approach to bring typing into Smalltalk.

Acknowledgements

This research has been funded by the National Science Foundation under Grants No. IRI-8604923 and IRI-8604977.

References

[BD86] Alan Borning and Robert Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Grapghics*, 5(4):345–374, October 1986.

[2] Note that we do not distinguish between unification and equality constraint.

[Bor79] Alan Borning. *ThingLab - A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, 1979.

[Bor81] Alan Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[Chi85] Uli Chi. Formal specification of user interfaces: a comparison and evaluation of four axiomatic methods. *IEEE Transactions on Software Engineering*, SE–11(8):671–685, August 1985.

[Deu86] Peter L. Deutsch. Panel: user interface frameworks. In *Proceedings of OOPSLA'86 Conference*, Portland, OR, September 1986.

[Dui86] Robert A. Duisberg. *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System.* PhD thesis, University of Washington, 1986.

[Ege86] Raimund K. Ege. *The Filter - A Paradigm for Interfaces.* Technical Report No. CSE-86-011, Oregon Graduate Center, Beaverton, OR, September 1986.

[Ege87] Raimund K. Ege. *Automatic Generation of User Interfaces Using Constraints.* PhD thesis, Oregon Graduate Center, 1987.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison Wesley, Reading, MA, 1984.

[GR83] Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, Reading, Mass., 1983.

[Lel87] Wm Leler. *Constraint Programming Languages.* Addison Wesley, 1987.

[MNG86] David Maier, Peter Nordquist, and Mark Grossman. Displaying database objects. In *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC, April 1986.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proceedings of the 9th Int. Conference on Very Large Data Bases*, Florence, Italy, October 1983.

[Mye83] B. Myers. INCENSE: a system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[SDB86] Reid G. Smith, Rick Dinitz, and Paul Bart. Impulse-86: a substrate for object-oriented interface design. In *Proceedings of OOPSLA'86 Conference*, Portland, OR, September 1986.

[Sut63] I. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, MIT, 1963.

[Van81] C. Van Wyk. *IDEAL User's Manual.* Computing Science Technical Report No. 103, Bell Laboratories, Murray Hill, 1981.

Call for Papers and Participation

International Symposium on Databases in Parallel and Distributed Systems

December 5-7, 1988
Austin, Texas

Sponsored by:
IEEE Computer Society Technical Committee on Data Engineering & ACM Special Interest Group on Computer Architecture (Approval Pending)

In Cooperation With:
IEEE Computer Society Technical Committee on Distributed Processing

Symposium General Chair
Joseph E. Urban
University of Miami

Co-Program Chairs
Sushil Jajodia
NSF
Won Kim
M C C
Avi Silberschatz
UT-Austin

Program Committee
Rakesh Agrawal, *AT&T Bell Labs*
Francois Bancilhon, *INRIA*
John Carlis, *U. of Minnesota*
Doug DeGroot, *TI*
C. Ellis, *Duke U.*
Shinya Fushimi, *Japan*
H. Garcia-Molina, *Princeton U.*
Theo Haerder, *Germany*
Yahiko Kambayashi, *Japan*
Gerald Karam, *Carleton U.*
Michael Kifer, *SUNY-Stony Brook*
Roger King, *U. of Colorado*
Hank Korth, *UT-Austin*
Ravi Krishnamurthy, *MCC*
Duncan Lawrie, *U. of Illinois*
Edward T. Lee, *U. of Miami*
Eliot Moss, *U. of Mass.*
Anil Nigam, *IBM Yorktown Heights*
N. Roussopoulos, *U of MD*
Sunil Sarin, *CCA*
Y. Sagiv, *Hebrew U.*
Jim Smith, *ONR*
Ralph F. Wachter, *ONR*
Ouri Wolfson, *Technion*
Clement Yu, *UI-Chicago*
Stan Zdonik, *Brown U.*

Local Arrangement
Hong-Tai Chou, *MCC*

Publicity
Ahmed K. Elmagarmid, *Penn State*

Finance Chairman
Edward T. Lee, *U. of Miami*

Symposium Objectives

The objective of this symposium is to provide a forum for database researchers and practitioners to increase their awareness of the impacts on data models and database system architecture of parallel and distributed systems and new programming paradigms designed for parallelism. A number of general purpose parallel computers are now commercially available, and to better exploit their capabilities, a number of programming languages are currently being designed based on the logic, functional, and/or object-oriented paradigm. Further, research into homogeneous distributed databases has matured and resulted in a number of recently announced commercial distributed database systems. However, there are still major open research issues in heterogeneous distributed databases; the impacts of the new programming paradigms on data model and database system architecture are not well understood; and considerable research remains to be done to exploit the capabilities of parallel computing systems for database applications.

We invite authors to submit original technical papers describing recent and novel research or engineering developments in all areas relevant to the theme of this symposium. Topics include, but are not limited to,

- Parallelism in data-intensive applications, both traditional (such as Transaction Processing) and non-traditional (such as Knowledge-Based)
- Parallel computer architecture for database applications
- Concurrent programming languages
- Database issues in integrated database technology with the logic, functional, or object oriented paradigm
- Performance, consistency, and architectural aspects of distributed databases

The length of each paper should be limited to 25 double-spaced typed pages (or about 5000 words). Four copies of completed papers should be sent before May 1, 1988 to:
Won Kim, MCC, 3500 West Balcones Center Dr., Austin, TX 78759
(512) 338-3439, kim@mcc.com

Papers due: May 1, 1988
Notification of Acceptance: July 15, 1988
Camera-ready copy due: August 30, 1988

THE COMPUTER SOCIETY

1730 Massachusetts Avenue, N W
Washington, DC 20036-1903

Non-profit Org
U.S. Postage
PAID
Silver Spring, MD
Permit 1398

Dr. David B Lomet
Digital Equipment Corporation
9 Cherry Lane
Westford, MA 01886
USA