Gestalt-class: A Persistent, Multi-user CLOS Application Environment

Michael B. McIlrath, Michael L. Heytens, and Thomas J. Lohman

Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology, Cambridge MA 02139

Author contact: mbm@mit.edu (617 253 4183)

Abstract

We describe Gestalt-class, a CLOS-based application environment supporting persistence, multiple inheritance, and relational querying capabilities. Gestalt-class has been in on-line use for several years in a multi-user computer integrated manufacturing (CIM) system supporting several integrated circuit fabrication laboratories at MIT and elsewhere. Gestalt-class has also been used in stand-alone computer aided design (CAD) applications and, more recently, in the development of infrastructure for distributed, collaborative research and design. We explain the implementation and evolution of Gestalt-class and discuss our experience with its use in application and schema development, support, and maintenance.

1. Introduction

As part of a project on computer aided integrated circuit fabrication at MIT, we designed and implemented a dynamic, object-oriented, persistent programming environment using a relational database as the principal backing store. The environment is implemented in CLOS and operates atop Gestalt, an independent abstraction layer providing an integrated procedural interface to multiple heterogeneous data storage systems[1]. This organization provides applications with transparent access to persistent objects, which are described and manipulated solely via CLOS methods. Application programmers utilize the rich object modeling and generic functions of CLOS in the integrated environment of Common Lisp to aid in program development.

We begin with an overview of the Gestalt system, followed by a description of the Gestalt object model, implementation, and mapping from the object model to the relational model.
Next, we discuss the implementation of the interface in CLOS and give some examples of its use. We conclude with a report on the status of the work.

2. Gestalt

The basic architecture of Gestalt consists of a library, implemented in C, running atop existing databases or storage systems. Gestalt is not itself a database; rather, it is a mechanism for logically integrating data storage systems. Using Gestalt, applications are written in a host programming language (originally C; later Common Lisp, via a foreign function interface).

The main motivation behind the design and implementation of Gestalt was the need for an application development environment that did not require database programming expertise. By unifying different database system interfaces under one common interface, application programmers are able to avoid learning various database manipulation languages and can concentrate on the design of the applications themselves. In addition, there was a desire for database independence, i.e. for the model visible to applications to be independent of the actual underlying system data models. Gestalt's main architectural goal was to encapsulate various data manipulation interfaces and data models, including the relational model, under one common data model and procedural programming interface. This provides a great deal of implementation flexibility as well as an elegant way to support "cross system" queries. The latter is especially useful in CAD/CIM; it can provide applications with a unified view of design, manufacturing, and simulation data, even though this information may physically reside within multiple heterogeneous data storage systems.

Gestalt in essence has a global schema multidatabase architecture and thus allows the underlying data storage systems to maintain a level of autonomy. If necessary, data can be manipulated by an application directly through the storage system interface (e.g. SQL), independent of Gestalt and its translation module for that particular storage system. Any actively running Gestalt programs will not see any such data updates until they explicitly access the underlying data via the Gestalt interface. (In practice, we use direct storage system interfaces only rarely, for various maintenance operations.)

2.1 Gestalt Data Model

The Gestalt data model specifies, in a language-independent manner, the mechanisms by which data are described and manipulated. In an actual implementation, these descriptive and manipulative mechanisms are expressed in terms of constructs native to the application programming language (e.g., C or Common Lisp) being used. All data are captured via typed objects and values, where objects have identity, and values do not. Objects and values may contain named, typed attributes. Gestalt supports an extended set of pre-defined value types, including the usual scalars such as strings, integers, and booleans, as well more complicated types for recording interval, inexact, and temporal data. Value types are not persistent unless they are contained within a user defined type. Even though a particular type may be built-in, its behavior is still specified only in terms of the operators defined on instances of that type, so that the user sees no semantic difference between built-in and user-defined types; both are accessed by the application programmer through their defined interfaces.

Crucially, a Gestalt schema is self-describing: the "meta-data" defining Gestalt types are Gestalt data themselves, of predefined object types *dbtype* and *dbattribute***.** Schema maintenance and modification is itself a Gestalt application task. Application developers may create new object types and test them by using the enhanced schema in their programs. This ability to create and modify new object types has provided for especially rapid prototyping and development of applications.

Gestalt did not originally support subtyping or inheritance. The need for “full object orientation” in applications and their persistent data became apparent early on, however, and a model and implementation compatible with CLOS was chosen, both because of the generality of the CLOS object model and the use of CLOS in the development of an extensible representation language for fabrication processes[2].

In addition to attribute names and types, the specification of an object type includes additional attributes which define certain integrity constraints or behavior that instances of the type must satisfy. These attributes are recorded in the corresponding *dbtype* and *dbattribute* schema objects. *Dbtype* contains the attributes *db-storage-system*, identifying the storage system holding this type; *deletable;* and *supertypes*. *Dbattribute* supports the attributes *one-to-one* (*vs.* list-valued); *unique* (can two objects of the same type exist with equal values of this attribute); *can-be-null*; *invertible* (should Gestalt provide an operator for fetching objects using the value of this attribute as a key?); *active* (is this attribute’s value set explicitly, or is it computed as needed by examining the object’s relationship with other objects?); *prefetched* (should the value of this attribute be brought into memory whenever the object is fetched?); and *mutable.* Active attributes correspond to “instance variables” or “slots” in other object oriented programming environments; Gestalt models both active and “passive” (computed) attribute access uniformly.

Gestalt operators include *selectors*, *mutators*, *constructors*, and *iterators* (used to perform set-oriented queries using a predicate). All Gestalt operators perform dynamic type checking. Routines that detect run-time errors raise an exception flag and return a null object consistent with their range type.

2.2 Gestalt implementation

All Gestalt operations are implemented via a generic database evaluator, *dbeval.* To implement an operation, the evaluator interacts with the underlying storage system through a translation module specified in a operation dispatch table, where each storage system has its own translation module. The evaluator is independent of the specifics of the underlying data storage systems. The translation module receives requests expressed in terms of Gestalt operators, and accesses the appropriate storage system through its own specific interface. The results of the operation are then formatted and returned to the evaluator. Because of this design, no queries are hardwired into the Gestalt system, and all arguments are evaluated at runtime. Arguments to *dbeval* include the operator type and the data; Gestalt objects are passed and returned via handles uniquely identifying the object by the logical pair (*type-id, entity-id*), where *type-id* identifies the type and *entity-id* is unique within a type.

Multiple storage systems are typically integrated in Gestalt installations; the particular systems used depend on the needs of the intended applications and their data and performance requirements. One data storage system is always present: the *schema-db* storage system, a private, fast access store which holds the meta-data (objects of type *dbtype* and *dbattribute*).

2.3 Mapping to relational model

A relational database is not required by Gestalt. However, the most heavily used Gestalt installations have used a relational database as the storage system for most of the application data types, due to the ability of modern relational systems to handle large volumes of data and the relative efficiency of implementation of set-oriented queries.

Each column of a relational table represents an (active) attribute of the relation while each row captures the data associated with those attributes. The columns within a relation must all be of a built-in or atomic type, i.e. a type which is known to the relational database. Gestalt represents each object class with two primary relational tables and zero or more secondary relations. The two primary relations are the *base* and *eid.* The base table is used to store all single-valued attributes within the object. The *eid* relation is used to store the next available entity id for this object type. Secondary or associative relations are used to store data associated with multi-valued object attributes. Each attribute within an object type that is list-valued (not *one-to-one*) has an associated relational table.

Each object instance has a single entry within the base table. Each base table contains an internal column, named *eid*, which stores the entity id associated with this object instance. This Gestalt-internal column is used to uniquely identify an entry within the base relation (i.e., it is the primary key). Each object instance may have multiple entries within the associative relations associated with that object type. Each entry within an associative relation contains the entity id of the object instance which is stored in the base table, its value, and a sequence number. The entity id can then be used to *join* the base table with the associative table in order to fetch the object's attribute values. The sequence number is used to order the attribute values associated with that object instance.

The relational model has no concept of inheritance or sub-typing. Inheritance is implemented entirely at the Gestalt level, by copying inherited attributes. Subtypes are treated just like other types when mapped onto the relational model (i.e. they have their own base tables, eid tables, and associative tables).

2.4 Transaction/Concurrency Support

Gestalt is more than just a persistent storage layer; it allows multiple applications to access and manipulate data concurrently and maintains the integrity of Gestalt objects. All primitive Gestalt database operations (selectors, mutators, etc.) use transactions when interacting with the underlying data storage system. For example, when creating an object within the Ingres relational database system, several entries may be made to existing relational tables. In order to assure that these are treated as one logical operation, they are packaged into a multi-statement Ingres transaction.

Gestalt does not have true user-defined application-level transactions. Instead, a simple locking mechanism allows concurrent applications to obtain and release locks on Gestalt objects.

3. CLOS interface

Gestalt was originally designed for use with a static (C) application programming interface; type-specific procedure declarations are automatically generated from the schema definitions. The advantages of exposing the generic evaluator, *dbeval*, were soon apparent, particularly in being able to write generic inspector and browser programs and debugging tools; it was first used through Common Lisp via a foreign-function interface (without CLOS). It is the *dbeval* interface that is used in the CLOS implementation.

3.1 Gestalt-object classes and the gestalt-class metaclass

Gestalt objects appear to CLOS application programmers as CLOS objects. For each Gestalt object type, there is a corresponding CLOS class whose metaclass is *gestalt-class* (a subclass of *standard-class*), and whose *instance-slots* are the active attributes. These classes also inherit from *gestalt-object*, an abstract class used to dispatch certain generic functions but principally to store the Gestalt object handle (*entity-id*, *type-id)* pair as an instance variable. The purpose of the *gestalt-class* metaclass is to override the *standard-class* accessors; *gestalt-class* accessors invoke *dbeval* (via a foreign-function or remote procedure call interface) to perform the appropriate Gestalt *select* or *mutate* operation. Both primitive data and objects are translated transparently between lisp and Gestalt; when object references are returned from the persistent store through Gestalt (e.g., as a result of a slot access), CLOS objects of the appropriate *gestalt-object* subclass are created. Active attribute values are saved in the *instance-slots*, both when written by the CLOS application and when returned from a Gestalt access; the CLOS objects thus act as both proxy objects and application caches for the Gestalt objects.

The *gestalt-class* metaclass is also used to specialize *make-instance*; after the standard initialization protocol is run, slot-values from the newly initialized object instance are used as arguments to the Gestalt constructor (again, via *dbeval*). Slots are typed according to the corresponding *dbattribute*, and type-checked (actually, the type-checking is an option but we normally leave it turned on). Of course, type errors would be caught by the Gestalt layer but we have found it much easier for debugging to have these errors caught at the CLOS level.

Instance identity (*eq*-ness) of objects of class *gestalt-object* is preserved by maintaining a hash-table of instances in each *gestalt-class* class object, using the *entity-id* as the hash key. Whenever an object reference is returned by Gestalt (i.e., in response to a slot-access, object creation, or set-oriented query), the appropriate hash table is checked. If the *entity-id* is already present, the corresponding instance is updated and returned, rather than a new one being created. Hence, within the same program image, CLOS instances representing the same Gestalt object are always *eq*.

Programs can also create non-persistent “prototype” instances by defining classes that inherit from a *gestalt-object* class but specify *standard-class* as the metaclass instead. For example, a large, complex design object may need to be fully assembled in order to be checked against design rules before being committed to the database. By defining a transient class that is fully type-compatible with the persistent one, the object need not be stored only to be deleted again if one of the design rule checks fails. (We have not provided such classes automatically as they have been required for only a limited number of application types.)

3.2 Queries

Methods that invoke *dbeval* to perform inverse (e.g., *person-with-name*) and passive attribute queries are generated by an extended initialization protocol for *gestalt-class*. Set-oriented queries are assembled dynamically and may include a pattern specification used to construct a Gestalt iterator predicate.

3.3 Schema evolution

As with Gestalt types, the *dbtype* and *dbattribute* types are available as CLOS class objects. We do not hide them from the CLOS programmer, nor have we attempted to provide automatic creation of *dbtype* objects from (or as a result of) instantiation of *gestalt-classes.* Instead, CLOS programmers create new schema types using *make-instance* on the *dbtype* class. The function *generate-class-definitions-from-schema* creates the appropriate CLOS class definitions, given the name of an existing *dbtype*. It also generates the necessary definitions for the associated inverse and passive query methods. We normally run this function over all types in the database when creating a new system image. However, dynamically loaded code may call *generate-class-definitions-from-schema* in order to obtain class definitions for types created or modified since the last system build. Schema changes do not mandate recompiling or reloading any files.

4. Examples

Before one can create a type within a Gestalt database, the attributes that are to be associated with that type must be created. The following code example creates an attribute called *facility,* assuming that a *dbtype* FACILITY already exists:

```
(make-dbattribute :name “facility”
                  :dbtype (dbtype-with-name “FACILITY”)
                  :onetoone t :unique t
                  :canbenull nil :invertible nil
                  :active t :mutable t :prefetched t)
```

The following code fragment creates the *dbtype* MACHINE, assuming *dbattribute-list* is the list of necessary attributes:

```
(make-dbtype :name “MACHINE” :overview “Lab Equipment”
             :domainspecific t
             :dbattributes dbattribute-list)
```

When a new type is created, all necessary Ingres relations are also created. In Figure 1, we have the MACHINE type along with the Ingres relations that are created in order to hold the actual data. There are three interesting points to make about the mapping example. First, all atomic and composite types are stored “in-line”, i.e. within the base relation. Composite types (e.g. TIME, which is made up of a DATE and a TIMEOFDAY) are broken down into their atomic parts before they are physically stored within a base relation record. Second, all one-to-one references to objects of a user-defined type are stored within a base relation record as an (entity id, type id) pair. The type id is also stored because it makes the support of subtyping easier. If an attribute is defined to be of a certain type then it is possible for objects of that type, as well as objects of any of its subtypes, to be stored within that slot. Third, all attributes that are lists are stored in an associative relation. Here, the *operators* attribute is a list of type LABUSER, and its related data are stored in the Operators_Associative relation. Each MACHINE instance will have zero or more entries in this relation.

Once a new *dbtype* has been created, a call to the function *generate-class-from-schema* creates the necessary CLOS classes and methods in order to create and manipulate instances of the new type. Using our example MACHINE type, we can create new instances of type MACHINE. Since the *last_maintenance* attribute can have a null value, it need not specified upon the creation of the new MACHINE instances.

```
(make-machine :name “machine1” :facility fab1
              :operators labuser-list1)
```

Once the instances of MACHINE have been created, it is now possible to access and to manipulate the data. All database access (including the previously mentioned object creation statements) goes through three layers: the Gestalt CLOS interface layer, the Gestalt *dbeval* layer, and the relational data manipulation layer.

The following examples illustrate the transformation of CLOS level calls to relational query statements. All relational queries are generated dynamically (e.g. using dynamic SQL). Given the evaluator operation code, the correct query statement is generated, using information which is stored with the type and attribute definitions. Relational table names and column names are fetched from this meta-data store and used to create the query at runtime. This gives the system added flexibility and means that no code need be changed if a table or column name were to change.

The first is a simple update of *my-machine*’s *last_maintenance* attribute. The last two examples show how one can query the database to either fetch a list valued attribute or fetch a list of objects which match a given criterion.

```
(setf (machine-last_maintenance my-machine) new-time)
```

effectively calls the “evaluator”:

```
dbeval(MUTATE, “last_maintenance”, my-machine, new-time);
```

This tells *dbeval* to update the *last_maintenance* attribute with the value *new-time* for the object: *my-machine*. This, in turn, generates the following underlying SQL relational query statement (refer to Figure 1 for the names of the relational tables):

```
UPDATE Machine_Base SET Date = :DateVariable,
                        Timeofday = :TimeofDayVariable
                    WHERE EntityId = :machineEntityId
```

Because TIME is a composite type, it must first be broken down into its atomic parts in order to be stored. Those parts are represented by the DateVariable and the TimeofdayVariable.

The following selects all MACHINE objects with name *machinename* and belonging to facility *fab1*:

```
(setof ‘machine
   :where `(and (= facility ,fab1) (= name ,machinename)))
```

When this query is passed to *dbeval*, the *where* clause is assembled into a Gestalt *predicate* object. Predicate objects represent arbitrary boolean expressions that are used as query constraints.

```
dbeval(ITERATOR_GENERATE, “MACHINE”, predicate);
```

where “MACHINE” is the type name of the type to be queried, and *predicate* is an object representing the *where* conditions. This generates the following relational query:

```
SELECT EntityId, Date, Timeofday FROM Machine_Base
   WHERE Name = :MachineName AND
         FacilityId = :fab1EntityId AND
         FacilityTypeId = :facilityTypeId
```

Here, we have supplied the machine name, and the *entity-id* and *type-id* of the FACILITY object, and are getting back the remaining (prefetched) attributes for all MACHINEs which match the *where* clause. Gestalt creates the proper MACHINE objects and an *iterator* object to obtain them. The CLOS interface layer uses the iterator to create a list of *gestalt-class* object instances. Note that attributes of type TIME are values, not objects with identity, and hence persist only for the duration of their associated MACHINE objects.

The method invocation

```
(machine-operators my-machine)
```

results in the effective equivalent of

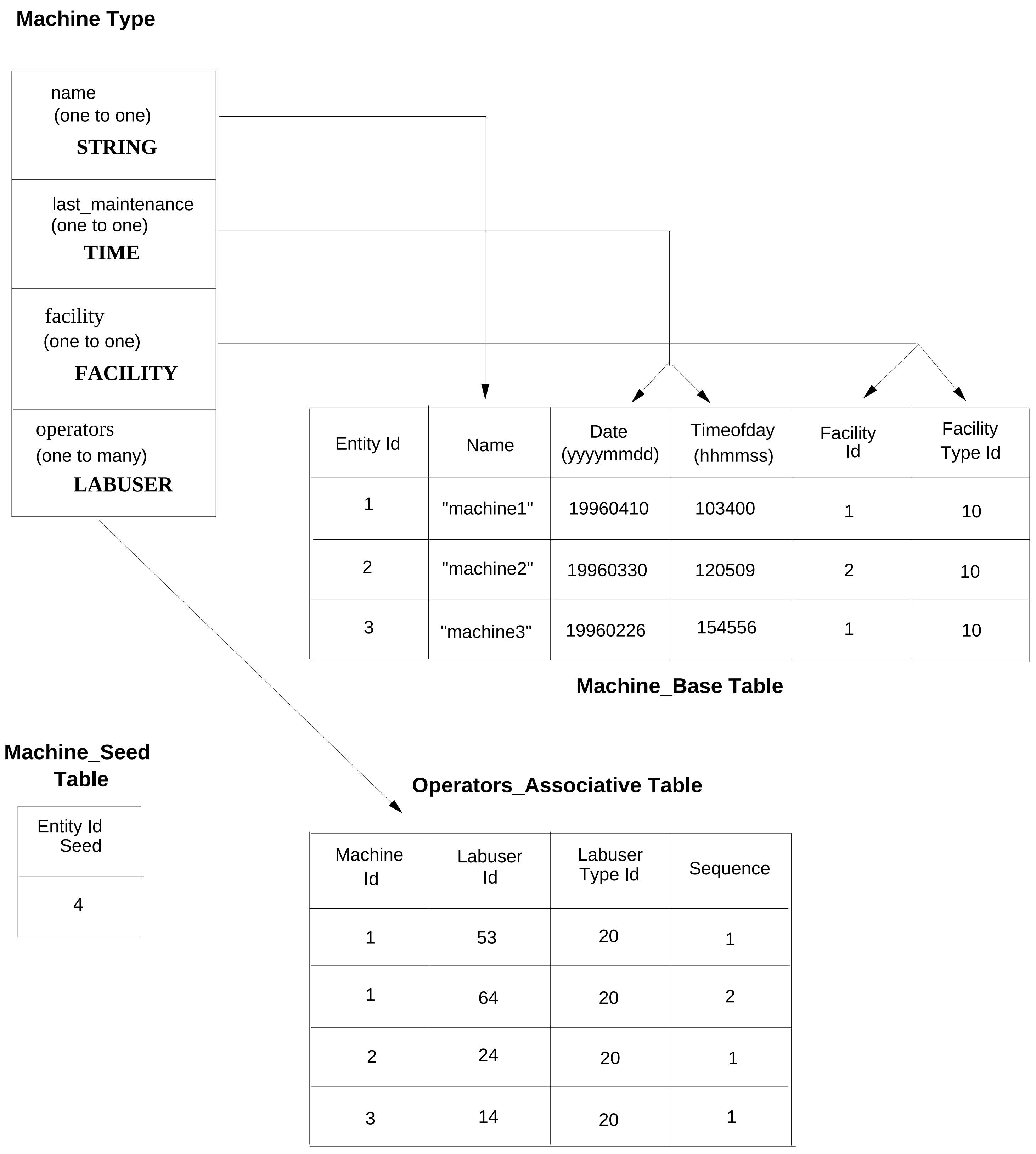
```
dbeval(SELECT, “operators”, my-machine);
```

producing the following SQL query:

```
SELECT t1.name, t2.sequence
   FROM Labuser_Base t1,Operators_Associative t2
   WHERE t2.MachineId = :machineEntityId AND
         t2.LabuserTypeId = :labuserTypeId AND
         t1.EntityId = t2.LabuserId
   ORDER BY t2.sequence ASC
```

Here, we are assuming that the LABUSER type contains one prefetched attribute, *name*. The query will return to Gestalt a list of names that were found to match the *where* clause. This list of names will then be used to create Gestalt LABUSER objects that are then passed up to the CLOS layer. If LABUSER had subtypes, then additional queries would be required to fetch any objects belonging to those subtypes.

Figure 1



Machine Type

name (one to one) **STRING**
last_maintenance (one to one) **TIME**
facility (one to one) **FACILITY**
operators (one to many) **LABUSER**

Entity Id	Name	Date (yyyymmdd)	Timeofday (hhmmss)	Facility Id	Facility Type Id
1	"machine1"	19960410	103400	1	10
2	"machine2"	19960330	120509	2	10
3	"machine3"	19960226	154556	1	10

Machine_Base Table

Machine_Seed Table

Entity Id Seed
4

Operators_Associative Table

Machine Id	Labuser Id	Labuser Type Id	Sequence
1	53	20	1
1	64	20	2
2	24	20	1
3	14	20	1

5. Use and experience

The CLOS application environment described has been used principally in CAFE[3], a computer integrated manufacturing system for integrated circuit semiconductor fabrication that has been in use for several years in research and development laboratories at the main campus at MIT, at the MIT Lincoln Laboratory, and at Case Western Reserve University. Gestalt-class has also been used in related computer aided design applications; e.g., [4].

The current CAFE implementation runs on SUNOS platforms using an implementation of Kyoto Common Lisp (KCL) with enhancements and additions by William Schelter, Rick Harris, Rajeev Jayavant, and the authors, and the PCL implementation of CLOS, using the INGRES relational database as the principal storage system for application data. A large number of students and staff at different institutions have contributed various applications to CAFE; the vast majority were neither database nor lisp experts. We have added, removed, and upgraded various databases and storage systems to the CAFE system, transparently to Gestalt-class and its applications.

The CAFE database and Gestalt-class have been used recently in the development of a remotely accessible repository for semiconductor fabrication processes[5]. New schema objects to model catalogs and libraries of manufacturing processes were added to the on-line, running, CAFE system at MIT, connected to our existing laboratory processes, and used in a demonstration (at Stanford) of a web-based process editor, all without the need to do any recompiles, off-line “schema reloads”, or “system builds.”

Acknowledgments

We thank Gregor Kiczales for getting us started in connecting a persistent storage system in PCL. We are especially thankful to all the programmers who have used Gestalt-class, particularly Duane Boning and Donald Troxel for their many suggestions. Various enhancements and utilities were contributed by Will Martinez, Michael Ruf, and Jeff Thompson. We thank our reviewers, Janet Coleman and Geoffrey Margrave, for their comments. This work was supported by DARPA under contracts MDA-972-88-K-0008 and N00174-93-K-0035.

References

1. Heytens, M. L. and R. S. Nikhil, “GESTALT: an expressive database programming system”, ACM SIGMOD Record, vol. 18, no. 1, March, 1989.
2. McIlrath, M. B. and D. S. Boning, “Integrating semiconductor process design and manufacture using a unified process flow representation”, Proc. Second Intl. Conf. on CIM, Troy, NY, May, 1990 (IEEE Comp. Society)
3. McIlrath, M. B., D. E. Troxel, M. L. Heytens, P. Penfield, Jr., D. S. Boning, and R. Jayavant, “CAFE -- the MIT computer aided fabrication environment”, IEEE Transactions on Components, Hybrids, and Manufacturing Technology, vol. 15, no. 2, 353-360, May, 1992.
4. Boning, D. S., M. L. Heytens, and A. Wong, “The intertool profile interchange format: an object-oriented approach,” IEEE Transactions on CAD., vol. 10, 1150-1156, Sep. 1991.
5. McIlrath, M. and D. Boning, “Process repository”, 1995 Symposium on Hierarchical Technology CAD---Process, Device, and Circuits, Stanford University, Palo Alto, CA, Aug. 10, 1995.