GESTALT: An Expressive Database Programming System[*]

Michael L. Heytens[†]
Rishiyur S. Nikhil[‡]
Massachusetts Institute of Technology

June 23, 1988

Abstract

Many new database applications require computational and data modelling power simply not present in conventional database management systems. Developers are forced to design complex encodings of complex data into a limited set of database types, and to embed DML commands into a host programming language, a notoriously tricky and error-prone enterprise.

In this paper, we describe the design and implementation of GESTALT, a system and methodology for organizing and interfacing to multiple heterogeneous, existing database systems. Application programs are written in a supported programming language (currently C and Lisp) using high-level data and control abstractions native to the language. The system is flexible in that the underlying database systems can easily be replaced/upgraded/augmented without affecting existing application programs.

We also describe our experience with the system: GESTALT has been in daily operational use at MIT for over a year, supporting an information system for CAF, a research facility for the automation of semiconductor fabrication.

1 Introduction

Database management systems have traditionally been used in administrative applications to provide efficient access to large data sets, preserve data integrity and consistency, and to control access. Recently, however, database systems have been applied to diverse new domains such as VLSI computer-aided design and voice and image processing.

A straightforward application of conventional database technology (*e.g.*, current relational systems) to these nontraditional areas has met with limited success. A key problem is the difficulty inherent in expressing complex objects and operations in terms of relations and relational operators. Moreover, conventional systems provide no general mechanism for abstracting over a given set of operations or data, thus all representation and manipulation must be encoded directly in terms of primitive constructs. In many instances, such a relational representation is cumbersome, leading to abstruse application logic.

[*]This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD) through the Office of Naval Research under Contract Number N00014-85-K-0213.

[†]Room 36-667, MIT, 77 Massachusetts Avenue, Cambridge MA 02139, USA; Phone: (617)-253-7811; Arpanet: heytens@caf.mit.edu

[‡]MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA; Phone: (617)-253-0237; Arpanet: nikhil@xx.lcs.mit.edu

To obtain adequate expressivity and abstraction, application developers typically embed query language commands into a host programming language. An embedded interface, however, is notoriously tricky and error-prone. Programmers must contend with both database and programming language concepts, *e.g.*, two mismatched type systems, two error/exception mechanisms, two types of control structures, *etc.* These inherent difficulties make it accessible only to database experts.

A number of research efforts are currently underway aimed at developing more expressive database systems [1,4,7,11,16]. Still, conventional systems are likely to remain the commercially-available state of the art for a number of years. Thus, a natural question is: How can we more effectively organize and interface to today's database systems.

This paper presents one possible organization, developed as part of the integrated circuit computer integrated manufacturing (IC-CIM) effort at the Massachusetts Institute of Technology. Our system, called GESTALT, is currently employed by CAFE (Computer-Aided Fabrication Environment) a sophisticated information system supporting the many facets of IC manufacturing. From a database perspective, CAFE is an especially challenging domain due to the diversity of information it manages, and the many complex programs it supports for the acquisition and manipulation of semiconductor manufacturing data. (For a discussion of IC-CIM data, see [8].)

One of the main motivations behind GESTALT was the need for an application development paradigm that did *not* require database and/or programming language expertise. The implied requirement for simplicity, however, had to be tempered with enough expressive power to permit a conceptually natural representation of the complex structures found in the domain of semiconductor manufacturing.

Another aim of GESTALT was *database independence*, *i.e.*, the model visible to applications was to be independent of the actual underlying database management system or systems. This capability provides a great deal of implementation flexibility as well as an elegant way to support "cross system" queries. The latter (similar to [3,15]) is especially useful in a manufacturing setting, *e.g.*, it can provide applications with a unified view of design, manufacturing, test, and product sales data, even though this information may physically reside at multiple heterogeneous database systems. Without this kind of support, application programmers must query component databases directly (after first mastering all the necessary interfaces!) combining partial results to form the desired answer. This process is complicated, tedious, and error-prone.

Finally, in a design and manufacturing environment, an effective data management system *must* seamlessly encompass a variety of CAD/CAM tools; it is simply not feasible to rewrite data and operations supported by these tools into a single monolithic DBMS. Thus, the GESTALT data model had to support the databases and procedural primitives of such tools.

We begin by presenting an overview of GESTALT in the next section. Section 3 describes the application interface, illustrating the programming model with sample applications. In Section 4 we describe the internal organization of GESTALT, and finally, we conclude with a discussion of the experience gained from using GESTALT in CAFE— a large, complicated system.

2 Overview of GESTALT

In essence, GESTALT is a layer of abstraction which shields application programs from underlying database systems. The application programming paradigm consists of a supported language (currently C and Common Lisp) and a set of abstractions for accessing and manipulating persistent data. An important point is that these abstractions are constructed using mechanisms native to the host language, thus application developers do not have to contend with the programming language–database dichotomy typically found in conventional approaches.

The two supported languages have their strengths and weaknesses. Many new database applications have an artificial intelligence component, for which Lisp is the preferred language. Also, the interactive nature of Lisp is convenient for posing *ad hoc* interpreted queries (analogous to query interpreters in conventional systems). The C interface is available for lower-level tasks or for applications requiring the compactness and speed of compiled C code.

The basic architecture of GESTALT consists of a software system running atop existing heterogeneous databases. GESTALT is not itself a full-fledged DBMS, rather it is a mechanism for logically integrating disparate data managers. This integration enables applications to view data in terms of a unified "global" schema. While GESTALT supports both read-only and update queries, it does not allow dynamic schema modifications. Such updates require changes to the system data dictionary, which must be "hand-coded"[1] by the database administrator (DBA).

Figure 1 illustrates both a conventional application architecture (a) and the organization of applications written in GESTALT (b). In conventional systems, applications are written using an embedded query language interface (*e.g.*, QUEL in C); the resulting program is then transliterated by a pre-processor and finally compiled. Conceptual entities and operations must be encoded and manipulated explicitly in terms of primitives supported by the data model, leading to much complexity in coding applications.

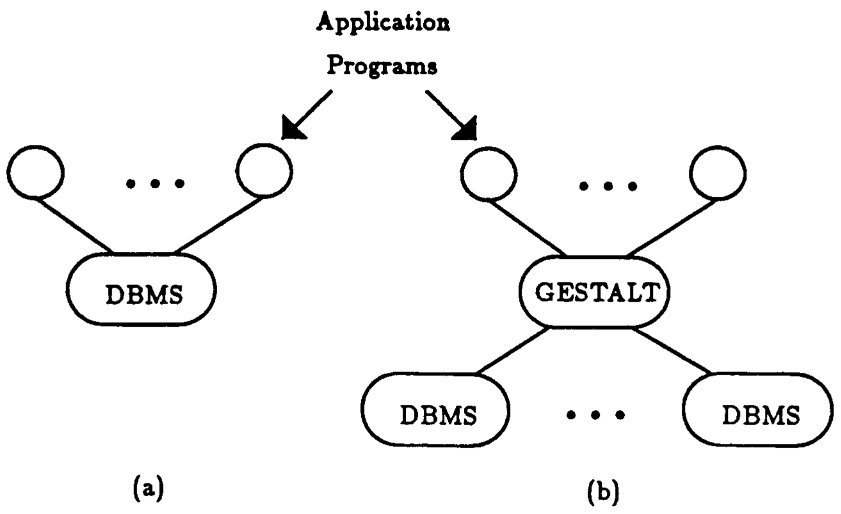

Figure 1: Structure of (a) conventional systems and (b) GESTALT.

In GESTALT, applications are also coded in a host programming language (C or Common Lisp.) Application programmers determine how to query and update persistent data by examining a specification detailing the interface to a collection of *procedural, data,* and *iteration abstractions.* (For a good discussion of the techniques of abstraction and specification, see [10].) The abstractions

[1]A structured schema-interface tool has been implemented which makes updating the data dictionary also straightforward.

which describe persistent data are object-oriented in nature[2], enabling applications to view domain data at a level commensurate with the actual real-world structure.

GESTALT provides a rich set of primitive data types and explicit support for null objects. Without system support of this kind, programmers often resort to nonstandard representations of null values and awkward encodings of, say, temporal and engineering data. In a large project where communication between developers is difficult, this can lead to inconsistencies and incompatibilities between applications.

Many of the procedural abstractions supported by the system are higher-order (*e.g.*, maps and filters). Operations of this sort encourage a clear and elegant style of programming, allowing concise expression of a wider range of computational processes [12,18].

3 The Application Interface

The interface visible to application programs is similar, in many respects, to the functional data model [12,14]. A GESTALT schema consists of a specification of entity or object types, and a collection of operations defined on instances of those types. This information is available to application programmers in two different forms. The first is a textual specification (maintained by the system) which provides an overview of each object type and a description of the effects, modifications, and requirements of the associated operators. The second is the data dictionary; all query facilities supported by GESTALT are available for accessing schema information stored in the system catalogs. The interpreted Lisp interface provides a convenient mechanism for browsing through meta-data of this form.

At the end of this section we present an example application as a concrete illustration of the programming model. We first, however, examine various aspects of the interface in more detail.

3.1 Object Types

All data in a GESTALT system are viewed as abstract types, *i.e.*, each abstract type specifies an object type, a list of named typed slots (which record attributes and relationships) and the set of procedures available for manipulating objects of that type. This is a very powerful tool, enabling the DBA to add new kinds of data objects cleanly, effectively extending the application programmer's virtual machine.

Operations in data abstractions can be divided into the following categories:

selectors — procedures to select an object component
mutators — procedures to update or delete an object
constructors — procedures to create objects
generators — procedures to generate null objects and iterators

A GESTALT system contains both *pre-defined* and *domain-specific* abstract types. The latter are abstractions of objects present in the application domain, while the former encompass all the built-in types, including an extended set of primitive entity types, abstract types for the data dictionary, and list- and tuple-structured types.

[2]In the following sense: Real-world entities are modelled as "objects" of various abstract types, objects can "contain" other objects, and objects can share other contained objects. We do not model inheritance.

It is important to realize that 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. Hiding the representation of built-in types enables the system to perform run-time type checking. It also eliminates the possibility that applications depend on particular encodings of values.

3.1.1 Pre-defined Types

The set of primitive types supported was designed specifically to facilitate modelling the temporal and engineering data commonly found in manufacturing and design domains. The *atomic* members of this set are:

TEXT - variable length array of bytes
INTEGER - fixed point number
FLOAT - floating point number
BOOLEAN - "true" or "false"
DATE - month, day, and year
TIMEOFDAY - hours, minutes, and seconds
TIMEDURATION - elapsed time in hours, minutes, and seconds.

Also pre-defined are several *inexact* and *interval* types. The inexact types record a *value* and an *uncertainty;* interval types consist of an *upper* and *lower* bound. These provide a convenient means of encoding engineering and scientific data, *e.g.,* the thickness of a layer of epitaxial silicon (say 1250 ± 30 Å) can be naturally expressed as an INTEGER_INEXACT with the appropriate value and uncertainty.

To provide applications with a convenient mechanism for grouping related objects, GESTALT provides the structured types LIST and NTUPLE (*n-tuples*). The former is a homogeneous sequence of objects of unspecified length, while the latter is a heterogeneous collection of a fixed number of objects.

GESTALT provides operators to map between pre-defined types and host programming language values. These routines are needed because the representations of even atomic GESTALT types are hidden from application programs. For example, the following Lisp procedure prints the id, crystal orientation, and sheet resistivity of a specified wafer by applying dbl-coerce— an operator which coerces atomic built-in data objects to corresponding Lisp values— to the desired wafer attributes.

```
(defun wafer-print (w output-stream)
  (let ((wd (wafer-waferdesc w)))
       (format output-stream "wafer ~A: <~A>, ~A ohm/square ~%"
               (dbl-coerce (wafer-id w))
               (dbl-coerce (waferdesc-orientation wd))
               (dbl-coerce (waferdesc-sheet-resistivity wd)))))
```

The data-dictionary is represented as a pre-defined set of abstract types, complete with selectors and generators, so that the same computational model can be used to peruse it. These include:

TYPE - built-in and domain-specific types
ATTRIBUTE - object components or slots

OPERATOR - structure for describing operators
OPERATORARG - formal operator arguments

3.1.2 Domain-Specific Types

It is the responsibility of the DBA to augment the data dictionary accordingly with the domain-specific abstract types for a given domain. We have implemented many tools to facilitate this activity.

An example specification is shown in Figure 2. Such specifications are automatically generated by the system from information stored in the data dictionary and are used by application developers. The first three operations (machine_name, machine_available, and machine_labuser) are selectors; they take objects of type MACHINE and return an object corresponding to the desired property or attribute. The next two (modify_machine_available and modify_machine_labuser) provide a mechanism for mutating or modifying objects. Entities of type MACHINE are created by constructor create_machine; this operator takes a name, labuser, available triple and returns a newly-created machine. Finally, generators null_MACHINE and machine_iterator produce a null machine and machine iterator, respectively.

Another example from the CAFE system is the type WAFER, with components: a TEXT id, a wafer description (WAFERDESC, which records dopant, crystal orientation, sheet resistivity, *etc.*), and a BOOLEAN flag indicating whether or not a layer of epitaxial silicon is present.

3.2 Procedural Abstractions

GESTALT's procedural abstractions include generic operators for LISTs and NTUPLEs, as well as common computational processes (such as maps and filters) packaged as higher-order procedures. For example, the following Lisp code prints all wafers from a named lot (a logical set of wafers) with resistivity values below a certain threshold:

```
(map (lambda (w) (wafer-print w t))
     (filter (lot-wafers (lot-with-name name))
             'low-resistivity?))
```

where name identifies the desired lot. Note that map and filter take arbitrary procedures as arguments. Applications can also make use of the procedure mechanism of the host language to create new procedural abstractions.

In addition, a variety of others procedures are supported, *e.g.,* to copy and (extensionally) compare data objects, a simple mark-and-release memory management scheme, *etc.*

3.3 Iterators

GESTALT iterators offer a convenient and space-efficient way to examine instances of a particular type, and are inspired by CLU [10]. Iterators serve the function of "retrieve loops" or "record cursors" found in conventional embedded query languages. For example, the following fragment of C code illustrates how an application might operate on all machines:

Overview

```
A MACHINE is a piece of equipment used in the fabrication of integrated
circuits.  Attributes name, labuser, and available are maintained for each
machine, recording the machine's name, a list of labusers waiting to use the
machine, and the current availability, respectively.  Only the latter two are
mutable.
```

Operations

```
TEXT machine_name(m)
MACHINE m;
   effects Returns machine name.

BOOLEAN machine_available(m)
MACHINE m;
   effects Returns machine availability.

LIST machine_labuser(m)
MACHINE m;
   effects Returns list of labusers waiting to use machine.

MACHINE modify_machine_available(m, available)
MACHINE m; BOOLEAN available;
   modifies m.
   effects Sets availability of m to available.

MACHINE modify_machine_labuser(m, labuser)
MACHINE m; LIST labuser;
   modifies m.
   effects Sets labuser list of m to labuser.

MACHINE create_machine(name, available, labuser)
TEXT name; BOOLEAN available; LIST labuser;
   effects Returns newly created machine.
   requires Name attribute must be unique.

MACHINE null_MACHINE()
   effects Generate null machine.

ITERATOR machine_iterator()
   effects Returns a machine iterator.
```

Figure 2: MACHINE data abstraction.

```
...
while ( BOOLEANtoi( iter_more( machines ))) {
    current_machine = iter_current( machines );
    ... computation involving current_machine ...
}
```

where machines is an iterator, and iter_more and iter_current test for an exhausted iterator and return the head of an iterator, respectively.

3.4 Type and Exception Checking

All GESTALT operators perform dynamic type checking. While static type checking could (in principle) be performed, an unobtrusive implementation would require modifications of the C compiler and Lisp interpreter, which we wished to avoid.

Certain operators also perform null checks at runtime. Generally speaking, operators that examine object components are strict with respect to null objects, whereas operators which do not examine components are nonstrict. For example, a null wafer can be freely included in a list or tuple; a null exception is detected only when an attempt is made to, say, select the id component.

Routines that detect type or null errors generate a run-time warning message and return a null object consistent with their range type. If an attempt is made to coerce a null object to a host language type, then a pre-defined value is returned, after printing a suitable warning message.

In the case of domain-specific abstractions, the DBA is free to augment the run-time checking with additional integrity constraints (or *invariants* in programming language parlance.)

3.5 Object Persistence

In GESTALT, all objects of a user-defined type persist, whereas objects of atomic, list and tuple types are ephemeral (unless they are part of a persistent type). For example, the constructor create_machine creates a persistent machine object, whereas itoINTEGER – which coerces a C integer value to a GESTALT INTEGER– returns a heap-based object whose lifetime is bounded by the duration of the enclosing program.

Associated with each user-defined type is a single persistent extent. The system automatically updates these extents in response to creation and deletion of objects of the appropriate type. Applications can easily examine the contents of an extent *via* an iterator.

3.6 A Sample Application

As an illustration, we present a simple C application. The program, which utilizes the data abstraction of Figure 2, informs the next laboratory user waiting to use a particular piece of processing equipment when he or she is free to do so. The code is shown in Figure 3; for brevity, it assumes (1) there are always waiting users for a machine, and (2) a procedure send_mail_msg, which does the obvious thing.

The application constructs a machine iterator (machines) and then uses it to examine the availability of each machine. If a machine is available, a message is sent to the next labuser, and the availability and waiting list are updated. Finally, the iterator is deallocated.

```
#include "specification.h"

main()
{
    ITERATOR machines = machine_iterator();
    MACHINE current_machine;

    while ( BOOLEANtoi( iter_more( machines )))
    {
        current_machine = iter_current( machines );
            /* grab machine at head of iterator */

        if ( BOOLEANtoi( machine_available( current_machine )))
        {
            LIST labusers = machine_labuser( current_machine );

            send_mail_msg( machine_name( current_machine ), head( labusers ));
                /* inform next user */

            modify_machine_available( current_machine, itoBOOLEAN( 0 ));
            modify_machine_labuser( current_machine, tail( labusers ));
                /* modify availability and labuser list */
        }
    }
    iter_free( machines );
}
```

Figure 3: Sample C application program.

Notice that nowhere is the programmer forced to deal with representational issues; one need only concentrate on the logic of the application at hand.

Note also that one does not have to annotate or flag the database commands in a program (*e.g.*, the ## of embedded QUEL and embedded SQL's EXEC [6].) The programming language and database have been integrated into a single framework, so that the actual encoding of a task closely matches the corresponding abstract computational process, resulting in programs that are perspicuous and easily modifiable.

Finally, since programs have no way of determining where data actually reside, the DBA is free to change the underlying database systems without compromising the correctness of application programs. (They will have to be relinked, however.)

4 The Implementation of GESTALT

Our implementation relies heavily on the proven software engineering principles of abstraction and modularization. First, such techniques constitute a programming methodology which is effective at controlling the complexities inherent in any large programming effort. Second, since the DBA was expected to modify the implementation (*e.g.*, to add a new component database) a clear, well-partitioned internal structure was deemed essential.

In this section we present the internal structure of GESTALT, including a discussion of the steps required to modify the system. We conclude by describing the system configuration currently in use by CAFE.

4.1 System Architecture

The organization of GESTALT (Figure 4) is similar to that of Multibase [15]. The schema architecture consists of a GESTALT global schema, and a GESTALT local schema-local DBMS schema pair for each component system. As in Multibase, the latter insulates the global system from local database details, allowing it to be structured in a clean and extensible manner. All component-specific details are confined to translator modules (one per underlying database) which are responsible for mapping operations on GESTALT local schemas to local database operations.

GESTALT, however, is responsible for translating global requests into operations (in terms of GESTALT local schemas) on one or more of the underlying databases. This translation is performed by the GESTALT evaluator, which is coded in a manner independent of the number and nature of the underlying database systems. All such dependencies are recorded in a dispatch table, enabling the DBA to extend the system in a straightforward way. This table-driven approach is possible because the evaluator assumes a standard interface to each of the underlying systems.

The system is made available to applications in the form of a library of procedures. This library is either linked into compiled application programs (*e.g.*, in C), or made available to an interpreter (*e.g.*, in Lisp).

4.2 Modifying the System

Due to the modular implementation, modifying the system is relatively straightforward. Modifications are required when the underlying database configuration changes or when abstractions are added, deleted, or modified.

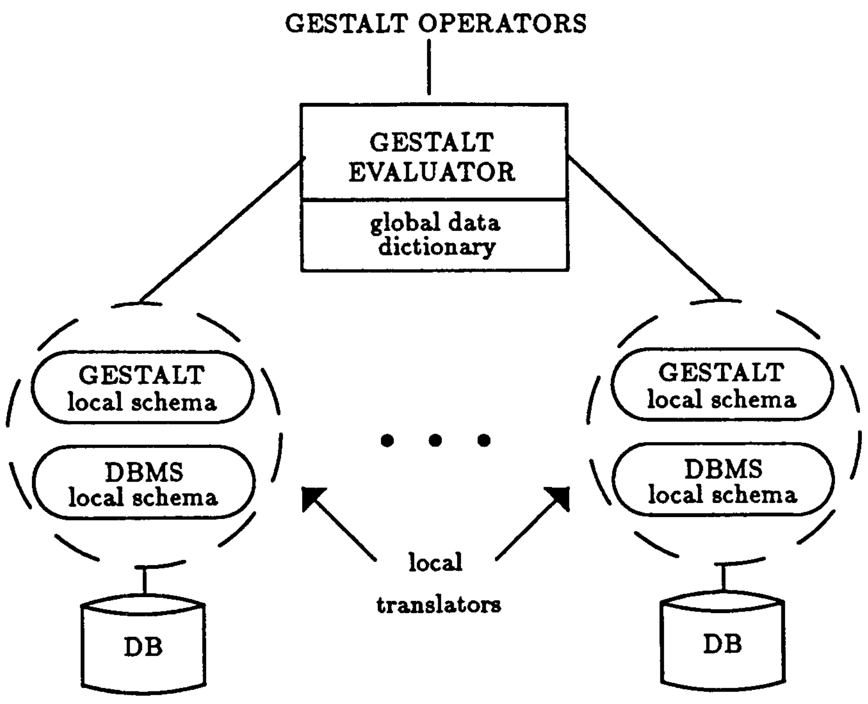

Figure 4: GESTALT system structure.

Adding a database to the system requires updating the dispatch table used by the evaluator, recording the new database in the data dictionary, and incorporating the corresponding implementation module into the system.

Altering the set of procedural abstractions supported by the system simply entails adding, deleting, or modifying the appropriate library routine. When updating data abstractions, however, things are a bit more involved. In the simplest case, modifications of existing data abstractions only involve source-level changes to routines in the corresponding abstraction module (*e.g.*, performing additional integrity constraint checks prior to invoking the evaluator). Modifications that are tantamount to changing the GESTALT schema require adding or deleting abstraction operators, entering new or updated information into the data dictionary and (possibly) changing the schema at one or more of the underlying database systems.

As an illustration, consider the steps required to introduce type LABUSER, which captures the notion of a certified laboratory user. The first step is to add the approriate type, attribute, and operator information to the data dictionary. Secondly, suitable logical structures must be created at one or more of the underlying databases. Finally, a LABUSER specification and implementation module are generated (based on information in the data dictionary) and incorporated into the system.

4.3 Sample Configuration, and Experience

GESTALT has been the database manager of the CAFE system since January, 1987. Since then, approximately 50 programs totalling some 30,000 lines of GESTALT code have been written by several applications programmers. An example is an electronic machine reservation system (EMR) which provides laboratory users with a convenient mechanism for reserving time on processing machines. EMR eliminates paper sign-up sheets from the clean-room, a potential source of contaminating particles. The system is regularly used in the microtechnology laboratories at MIT.

An early database configuration used by CAFE had GESTALT running atop three component databases: University INGRES [17], PRELUDE [13], and a home-brewed system. Each of these databases made an important contribution to the overall system. INGRES provided a reliable, fully-functional data manager. However, due to its lengthy startup time, it was unable to support applications requiring fast, simple data access. To remedy this, PRELUDE— a lightweight ASCII-based DBMS— was incorporated. Finally, the home-brewed system was employed to store the large, variable-length data objects commonly found in the IC manufacturing domain. Storing such objects in either of the other systems would have been difficult or impossible.

The current configuration utilizes only two databases: commercial INGRES from Relational Technology, Inc.[9] and WiSS, the Wisconsin Storage System [5]. Most CAFE data are stored in INGRES; WiSS, because of its support for variable-length data items, handles those objects not easily captured by the relational data model. For example, WiSS is used to store voluminous execution "traces" of program-like process flow descriptions.

Despite the wholesale database changes to move to the current configuration, application programs remained unchanged— they only needed relinking. Thus systems using GESTALT are not locked in to a particular database system; they are free to incorporate newer, more powerful data managers as they become available.

CAFE developers may still write applications using INGRES or WiSS directly, *e.g.,* to take advantage of INGRES' report-writer tools.

5 Conclusions and Future Work

We have described GESTALT, a system which offers an expressive application programming paradigm in which the database and programming language have been integrated into a single framework. The system provides uniform access to existing heterogeneous databases.

While GESTALT offers a number of benefits, there are some difficulties and limitations inherent in its approach. One obvious limitation is the lack of a dynamic data definition capability. Because GESTALT does not have control over the entire programming environment, this is simply not feasible in the system. In general, this problem is very difficult in a multiple heterogeneous database environment (it is analogous to view updating in conventional systems).

Another limitation is the lack of global concurrency control and recovery. Although all interactions with component databases are atomic, the system contains no general mechanism for grouping several GESTALT operations together into a transaction.

GESTALT does no query optimization of its own, thus certain kinds of queries execute inefficiently. For example, tasks implementing the equivalent of a multiway join are not supported well, unless there is a data abstraction that executes it entirely within a component database system.

Our experience to date with the system has been encouraging: CAFE application developers have responded very positively. Despite the wide range of programmer experience (from novices to veteran software engineers) all have commented on how easy the model was to understand, and how quickly they were able to produce sophisticated, working applications.

An immediate need that we plan to address soon is a coherent access control strategy. Currently applications using GESTALT must rely on component database and/or operating system protection mechanisms.

Our longer-term plans involve extending GESTALT so that it is a computationally complete, stand-alone system, incorporating its own persistent objects. We are currently exloring a functional database programming language with an *immutable* or *functional* database [11]. We feel that this combination offers an expressive system for high level applications programming, admits much parallelism (for high performance), and facilitates the management of historical data.

Acknowledgements

We would like to acknowledge the contributions made by Michael McIlrath and Rajeev Jayavant to both the design and implementation of GESTALT. Duane Boning, Paul Penfield, and Donald Troxel also made many helpful suggestions.

References

[1] Albano, A., Cardelli, L., and Orsini, R. *Galileo: A Strongly Typed Interactive Conceptual Language.* Technical Report 83-11271-2, Bell Laboratories, 1983.

[2] Atkinson, M.P., Chisholm, K.J., and Cockshott, W.P. Ps-Algol: An Algol with a Persistent Heap. *SIGPLAN Notices* 17(7):24-31, July, 1981.

[3] Brodie, M.L., Blaustein, B., Dayal, U., Manola, F., Rosenthal, A., CAD/CAM Database Management. *Database Engineering,* Vol. 7, No. 2, IEEE, June 1984.

[4] Carey, M.J., and DeWitt, D.J. Extensible Database Systems. In *Proceedings of the Islamorada Workshop,* February 1985.

[5] Chou, H.T., DeWitt, D.J., Katz, R.H., and Klug, A.C. Design and Implementation of the Wisconsin Storage System, *Software – Practice and Experience,* Vol. 15(10), 943-962, IEEE, October 1985.

[6] Date, C.J. *An Introduction to Database Systems.* Addison Wesley, Reading, Mass., 1986.

[7] Dayal, U., and Smith, J.M. PROBE: A Knowledge-oriented Database Management System. In *Proceedings of the Islamorada Workshop,* February 1985.

[8] Hodges, D.A., and Rowe, L.A. Information management for CIM. In *Proceedings of Advanced Research in VLSI.* Palo Alto, CA, March 1987.

[9] *INGRES Reference Manual,* Version 3.0, VAX/VMS, Relational Technology, Inc., Berkeley, CA, May 1984.

[10] Liskov, B.H. and Guttag, J.V. *Abstraction and Specification in Program Development,* The MIT Press, Cambridge, MA, 1986.

[11] Nikhil, R.S. Functional Databases, Functional Languages. In *Proceedings 1985 Persistence and Data Types Workshop, Appin, Scotland,* August 1985.

[12] Nikhil,R.S. *An Incremental, Strongly-Typed Database Query Language.* PhD thesis, Moore School, University of Pennsylvania, Philadelphia, PA, August 1984.

[13] *PRELUDE Reference Manual,* Release 2.0, VenturCom, Inc., Cambridge, MA 1986.

[14] Shipman, D.W. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems* 6(1):140-173, March 1981.

[15] Smith, J.M., et al., MULTIBASE–Integrating Heterogeneous Distributed Database Systems. In *Proceedings National Computer Conference,* Chicago, May 1981.

[16] Stonebraker, M. and Rowe, L.A. The Design of POSTGRES. In *Proceedings 1986 SIGMOD Conference,* Washington, DC, pp. 340-355, May 1986.

[17] Stonebraker, M., Wong, G., Kreps, P., and Held, G. The Design and Implementation of INGRES, *ACM Transactions on Database Systems* 1(3):189-222 1976.

[18] Turner, D.A. The Semantic Elegance of Applicative Languages. In *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, NH,* pp. 85-92, ACM, October 1981.