SYSTEM FACILITIES FOR CAD DATABASES

by

Charles M. Eastman

DRC-15-5-79

September 1979

Institute of Building Sciences
Carnegie-Melion University
Pittsburgh, PA 15213

SYSTEM FACILITIES FOR CAD DATABASES

by Charles M. Eastman
Carnegie-Mellon University
Pittsburgh, PA. 15213

1. Introduction

Design is the specification of man-made artifacts. Among the reasons one designs is to both *predict the expected performance of the artifact,* to make sure these are responsive to apriori criteria, and to *identify the set of actions that will efficiently lead to realizing the artifact,* eg. fabrication planning. Many specifications are used during design, as intermediate products leading to the final specification.

Until recently, design has been a handicraft. The generation, selection and representation of intermediate and final specifications have been done manually. Starting about 25 years ago, computer-aided design (CAD) has developed as an alternative to handcrafted efforts. During most of this period, CAD has meant the mechanization of specific design tasks, eg. analysis of the behavior of some form of structure or drawing a perspective of some assembly, as discrete application programs. Recently, integration of these separate programs has been pursued in several design fields, involving structuring in a machine readable database information regarding the product being designed. The product, for example a car, chemical plant or building, is represented in an evolving integrated model that is a replacement to the many disparate collections of information traditionally used, i.e. drawings, engineering calculations, written specifications, etc. The benefits resulting from use of design databases include reduced costs of preparing data for application programs, especially analyses, reduced cost of producing the final specification, improved consistency management afforded by the machine readable model and opportunities for further automation.

Design databases have many of the same functional requirements as management oriented databases. They must provide effective file management and means of associative access, as well as controls for concurrent use. Both types of database must provide a common interface to a number of applications. Both types need to be able to generate reports - both tabular and graphic - and to provide backup in case of system failures. They both must provide security against unauthorized access. The *unique* system requirements of design databases pertain not to these general capabilities, but from the need to organize information and process it in a manner that supports design use and decision making.

In this paper, I attempt to lay out the special needs of design databases, as compared to the facilities provided in conventional database systems now available. Special attention is

given to the needed system level functions that we have come to believe best respond to the process of design, as it is now practiced. Our response to these needs is also presented, as we have approached them in the implementation of GLIDE and GLIDE2. The ideas have been developed and implemented in GLIDE and GLIDE2 through the combined efforts of a team of students and staff that has been working on design databases since 1974[1]. This paper is one of a series by the author's group on information processing in design (see also [Akin, 1979], [Eastman, 1979] and [Lafue, 1979;1979a]).

2. Nature of design

Design has been romanticized in our culture, to an unprecedented degree. In many people's minds, it is associated with an almost mystical creation of "something new" (see for an example [Koestler,l970]). But any design is ultimately the composition of existing entities, though the result may be objects with entirely new properties. Different approaches to design, then, can be ordered according to the degree that they parameterize existing objects. Design involving object descriptions whose parameterization is at a disaggregated level and where the interaction effects have not been thoroughly explored can thus lead to unanticipated classes of solutions.[2] It is on this class of design that I wish to focus.

Most design problems are decomposed into functional subsystems, where the subsystems are defined by convention. (For an interesting effort to formally decompose a design problem into its appropriate subsystems, see [Alexander,1964]). A subsystem is a *abstraction* of the total object. The abstraction may be useful because models exist of the subsystem's behavior along with standards for its performance or because it is compatible with models of other subsystems [Eastman,1979]. Subsystems typically can be based on any of several technologies, each having its own models of performance. Thus no one set of models is sufficient for evaluating subsystems whose technology has not been fixed. Subsystems are hierarchical. At most levels in the hierarchy, the choice exists (at least theoretically) of selecting an existing subsystem or designing one's own, recursively. The open-endedness of design can easily explode when the hierarchical definition of subsystems becomes deep.

The sequential definition of subsystems can follow many different orders. In current practice, various orders are selected, in response to different problem contexts and goals. The advantage provided by a particular decision sequence is that if subsystem A is defined

---

*The participants in this work include Mark Birnbaum, David Baker, Max Henrion, Gil la a Lafue, Robert Thornton and Kevin Wailar.

⁻For a food review and torn* ntw contributions to tha scientific literature on design, see [Akin,1979].

before B, then sensitivity of the design of B to changes on A can be explored iteratively. The reverse, however, is not possible.

Constraints on the sequence of dccisionmaking are the dependencies of variables on each other. Most design decisions require input values that are the result of other decisions. Such a relation is called a *functional dependency* [Bernstein, 1976]. Cyclic dependencies correspond to simultaneous equations. In practice, functional dependencies restrict decision sequences only marginally. Experienced engineers and designers rely on normative data for early estimates of subsystem performance, allowing them to make decisions in almost any order desired. Iteration allows later replacement of the normative data with a detail design.

Because of the number of dependencies involved, decisions are tentative and are open to change with new evidence. New evidence commonly takes the form of revised estimates of performance or resource consumption resulting from the detailing of other subsystems. At the same time, the cost of iterative changes grows quickly as new decisions are based on the variables that one desires to change.

Most subsystems affect multiple performances or otherwise interact by consuming common resources such as available space, dollars, allowed rates of failure, weight, electrical service, efficiency, maintenance resources, etc. (For an important characterization of subsystem interactions, see [Freeman and Newell, 1971]). Performance evaluation consists of determining the value of a dependency for which a criterion has been specified.

Many representations are used in design. Any representation is useful to the degree that it either helps to maintain specified dependencies or facilitates evaluating them. Thus multi-functional objectives result in multiple representations.

Eventually, subsystems are defined whose performances can be predicted with satisfactory certainty and in enough detail to guarantee the artifacts construction. This constitutes a "complete" design. Currently, specifications for the design are generated in parallel with the design itself, so that specification preparation is not considered a distinct task.

All of this requires great mental involvement. This involvement has the important side-effect of providing many cognitive cues to the designer. They are useful for retrieving from experience possibly relevant uses or contextual conditions not defined in the initial task. The ability to enrich design objectives in this way is one of the means by which design is an artistic activity. Extending the set of objectives is an important source of innovation in design.

This *very* cursory description of the design process is obviously sketchy, but adequate to identify the issues I wish to take up shortly. The above description may be condensed into

the following propositions regarding design:

1. artifacts to be designed are decomposed into subsystems. Alternative decompositions result in different subsystems. Choice of subsystems is an important design decision.

2. because of the many dependencies and criteria that must be satisfied, designers should retain control of the decision making sequence, including the ability to iterate and vary the order of decisions during different iterations.

3. designers should be able to use normative data as a surrogate for detail designing for any of the subsystems available to them.

4. many representations are needed to effectively evaluate a design's multiple performances.

5. information flow to and from designers should be of high density, supporting full engagement of their intellectual capabilities on the design problem.

3. Current limitations of database systems

Current database systems do not respond very well to several of these requirements[1]. Most of their shortcomings derive from three sources: the static structure of database systems, the speed of their access mechanisms and their lack of tools for managing integrity.

3.1. Their static structure

Conventional databases distinguish and separate the *data definition language* (DDL) from the *data manipulation language* (DML). The DDL supports declarations of data types such as records, variables and constants. It also includes various inter-record accessing structures, such as ISAM, inverted file or linked lists. The database structure as defined in the DDL, including al! permanent record declarations and accessing structures between them, is called its *schema**. As in most languages, alteration of the schema forces re-compilation of the database program. The DML, on the other hand, incorporates sort and merge or set operations, as well as access operators for the various structures offered in the DDL. Thus it provides tools for accessing the database in application programs. These facilities are separated so as to respond to different personnel responsibilities, as described below.

---

[1] Two sets of standards have been proposed for database systems, fenersted by the CODASYL Data Bata Taak Group [1971] and fh» ANSI-SPARC committees [Jardina, 1977). Thaaa etendarda ara the baaia for moat of the feneret assertions offarad rafardinf existing database practices.

[*] The CODASYL Report propoaes a einfle schema. The ANSI standard proposes three different types of schemes, based on different ebstractions of the database.

Applications are developed in general purpose languages, especially COBOL and FORTRAN, and linked with the database system. Adding a new application to an existing database involves relinking the system and re-loading the stored data. Thus they are added to conventional database programs only occasionally. In addition, if the application requires extension of the schema, then the database must be recompiled.

The users of database systems typically are distinguished as one of three types. Responsibility for the schema organization and for the loading of data into data structures are those of a *database administrator.* This responsibility is distinct from the *application* programmer, who develops programs that use but do not alter the database structure (though record instances certainly may be created or destroyed). Thus the DDL is the tool of the database administrator and the DML the tool of the application programmer. *Users* apply the application programs and/or read data in the database.

For a design database, the costs associated with schema modification and extension of applications are great. It is often impossible to anticipate at the outset all subsystems needed or analyses required and their associated subschemas. A fixed schema also determines the major order of decisionmaking, another conflict with the earlier stated propositions. Thus a much more dymanic schema definition facility is needed.

3.2. Limited speed of access mechanisms

An attribute of most of the components in a design, that affects both resources and a variety of performances, is their *shape.* Shape also provides information for high bandwidth interaction, eg. graphically. Until recently, the modeling of shape information has focussed on surfaces [Barnhill and Rcisenfeld, 1974]. However, the last five years has seen development of a *theory of solid shape integrity.* These new results allow shapes to be defined and sculpted as complete structures, without user concern for the geometric or topological components defining them [Baer, Eastman and Henrion, 1979]. Geometric modeling of both surfaces and solid shapes requires variable numbers of face, edge and vertex entities and the numbers change as the shape is manipulated.

This dynamism produces a second problem with traditional databases. Most commercial database systems rely on fixed format records, where each logical record corresponds to a physical record on secondary storage. As a shape often has hundreds of geometric entities defining it and a drawing has hundreds of separate shapes, computing a drawing currently may require thousands of separate record accesses. If each access is a random one, with latencies typical of current devices, the result is that geometric modeling of shapes or surfaces in a database is a slow and expensive proposition. In contrast, interactive design

requires that a drawing be displayable in seconds. Needed is either a hardware technology that greatly reduces latency times for random accesses on mass storage or else variable length data structures or the ability to store arbitrary sets of records in adjacent physical locations for consecutive access[1].

3.3. Lack of tools to aid integrity management

A third problem is that available database systems provide very limited facilities for managing integrity in large data structures. While semantic integrity is an important issue in all database problems, it is especially important in design. Current practice already relies on multiple representations so as to both maintain and evaluate particular relationships. Keeping these multiple representations consistent, however, is a significant problem.

Integrity maintenance in databases is wholely the responsibility of procedures that manipulate data. Thus integrity is the responsibility of (all) application programmers. With extensible and dynamic schemas, such an approach becomes almost impossible. Some means to centralize or modularize the responsibility of integrity management is called for. While the CODASYL and ANSI standards include procedural attributes, few systems have implemented this feature. Thus no support is provided for the procedural representation of data, an important tool for integrity management. One line of effort to improve integrity is *relational* databases [Codd,1970]. They decompose a database into small sets of variable types (called Relations) that in all uses have the same semantic relationship. Relational databases can simplify that task of writing "correct" operations.

4. C-MU work on design databases

The initial work on design databases at Carnegie-Mellon University resulted in a program called BDS (Building Description System). It was a program for defining and composing large numbers of polyhedra. Its control structure was a hierarchy of menus [Eastman, Lividini and Stoker, 1975]. Later, we embedded the geometric modeling facilities developed in BDS into a general programming language, in a configuration thought directly useful for developing design applications. The new system was called GLIDE (for Graphical Language for interactive DEsign) and became operational in 1977 [Eastman and Henrion,1977]. Recently, we were asked to develop a portable, production version of GLIDE. In this undertaking, we chose to re-think many of our earlier assumptions. The result is GLIDE2, now being implemented [Eastman and Thornton,!979]. Below, I attempt to relate the evolution of our

[1] Relational databases can potentially reduce this problem if they support a major restructuring of physical records when applying the "join" operation. CODASYL databases that support repeating groups also alleviate this problem.

thinking about the system features needed for design databases, as they have been embedded in GLIDE and GLIDE2.

4.1. The Structure of GLIDE

The general conception of a design database can be characterized as shown in Figure 1. The information describing a particular design effort is organized as a *project database.* It holds part and assembly information, data needed for engineering analyses and other information unique to the current project. The project database may refer to *project independent data,* that describes supporting information determined exogenously from any particular project. Typical contents of project independent data might be material properties and standard part catalogs. Multiple users access and extend the project database, each class of user interacting through a unique *subschema* or *view* of the database. From the project database, data is extracted and passed to a number of integrated or stand-alone application programs. An interface to independently implemented applications would consist of a *mapping* program that computes needed dependent data and formats it with other stored information to generate the proper input stream for the application, in character or possibly binary form. A number of reports also are generated, including intermediate and final drawings, specifications, and production information.

Within this general framework, the process of design requires a method for adaptive extension of the model imbedded in the project database schema. As subsystem technologies are defined, the data for defining instances of the technology, applications supporting their definition and programs for evaluating them, must be added to the database.

One approach to adaptive schema development is to rely on procedures capable of dynamically modifying the database schema. These procedures should be capable of a variety of actions: adding or modifying existing data in the DB; extracting and computing data and formatting it for input to another application, or extending the schema format to depict new data types, variables or constants. The language for defining these procedures needs to embody the functions of both the DDL and DML of traditional database systems. The implied goal is to use these procedures to automate the traditional functions of the DB administrator. In this new organization, his responsibilities are not to determine the physical and logical organization of data, but to deal with procedural matters, such as to decide on what application programs should be used and their sequence of application.

In addition to the capability for dynamically evolving a schema, a computing environment for integrated CAD should include a number of other needed functions, including record facilities and operations for geometric modeling, graphical input and display functions and a

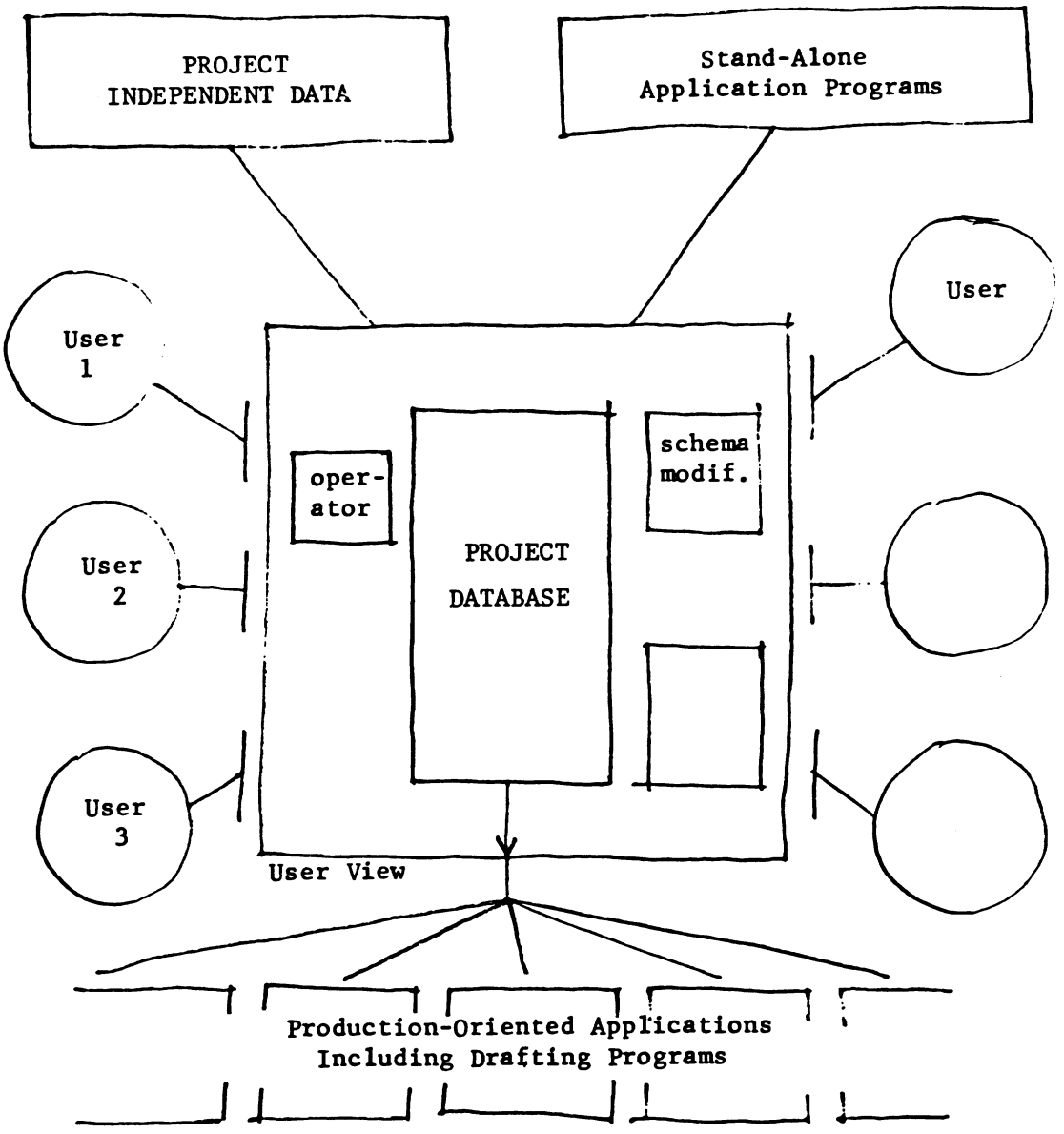

Figure 1: Schematic diagram of a design database system.

user-oriented command processor. Our primary research goals, then, were to provide a single well-structured environment for both developing integrated design applications, capable of modifying the database to suit the application, and for executing those applications in the context of the project database.

These notions were integrated with those of structured programming to produce GLIDE. GLIDE is a block structured and semi-interpretive language. Its compiler generates code in an intermediate execution language (called SLIP), as it is entered line by line. At the outermost block level, each complete statement also is immediately executed. Thus control statements embedding one or more blocks are executed when the full tree of executable code has been generated. Compilation is by recursive descent.

Because of the desire for adaptive extension of allowed applications, procedures are pre-compiled in SLIP and separately stored, then loaded and linked *when they are invoked.* This segments the code in large applications. It also allows separately compiled applications to be added to the database without relinking or reloading of data. Each procedure is stored as a record on secondary memory, that is automatically accessed and linked when it is called. Recursion, of course, is supported. Pre-compiling with run-time linking requires the user to explicitly define those entities that are to be imported into a procedure's execution environment. The name and type of external objects, including the parameters of procedures and their types, are defined in an "external" declaration.

GLIDE provides as simple types real, integer, text (variable length) and boolean. These may be used to define either variables or user-defined *attributes* (corresponding to a type declaration). Attributes may be used only to define the components of a Form Record, the general purpose record type. For defining relations between records, GLIDE provides a pointer type, called an Item, that may be assigned references to record instances. Both variables and attributes may be of type Item. In addition, a Set Record is provided, which is a variable length unordered list of Items. Through these variable types and structures, GLIDE supports the schema organization recommended by the CODASYL standard.

For geometric modeling, GLIDE provides a Topology record type, for storing the adjacency structure of faces, edges and vertices and the Polyhedron record type, for storing the geometry of these entities. Both are variable length, to support shape changes while still providing fast access. GLIDE includes the Euler operators for defining Topologies, [Eastman and Weiler, 1979] and the spatial set operators for sculpting and defining complex shapes. It also includes a Location attribute for placing objects in Cartesian space and a MOVE operator for relocating and rotating them. Also included are orthographic and perspective display transformations, simple alphanumeric 1-0 and cursor input, plus translations between screen

and project coordinates for graphic 1-0. These operations can be used to define or manipulate shapes with planar or approximated simple curved surfaces [Birnbaum, et al, 1978].

GLIDE recognizes two modes of declarations. These correspond to the static and dynamic modes in Pascal. Local (static) declarations are managed via a stack and exist for only limited periods of time. Global declarations are allocated to database records and exist permanently. At the end of a session, the global symbol table is saved along with the records and record directory for use in later sessions.

Global declarations are given a local identifier during compilation that is used in all compiler generated code. Global declaration entries are made only at execution time, however, and thus can be executed conditionally or from within procedures. Record instance creation is of course a run-time event.

In general, global entries are managed with a reference count. Undeclare operations eliminate the name of a variable, which is one of its references. When the reference count drops to zero, a record is deleted automatically.

Because declarations and instantations are run-time events, the compiler cannot check these and run-time checking must be extensive. Attribute identifiers and their type are stored in record instances with their values and searched at run-time. (This search time is inconsequential in comparison to record 1-0 costs).

In GLIDE types are weak. The dynamic adding of record attributes results in the compile-time definition of a type having no information about record organization. Thus a type corresponds only to the structure used, eg. Form, Polyhedron, Topology, Set, and to the simple scalar types.

GLIDE is written in Bliss for the TOPS-10 operating system. It occupies 58K words of core plus a working area that may be sized by the user [Eastman and Henrion, 1979].

4.2. Experience With GLIDE 1

GLIDE is in use at six institutions. It has been used in a variety of applications, including architecture, civil engineering and machine design. In October, 1978, we were given the opportunity to develop a portable production version for the U. S. Army Corps of Engineers, based on the performance of its prototype version. This provided the opportunity to re-consider earlier decisions and alter the language where appropriate. In our experience with GLIDE, three types of shortcomings suggested basic problems:

1. while the goal of an adaptive program and schema structure was achieved, this adaptiveness introduced new problems. Global attribute types applied across all applications; an attribute name could be used only once. Thus an application programmer had to know all attribute names and some of the associated semantics in order to develop an application without duplicate definitions. Application development and especially schema extension required comprehensive knowledge of the total database schema, or more accurately, all possible schcmas in which the application might be applied. As a result, combinatorial issues allowed only modest forms of extensibility. Instead of automating the role of database administrator, we seemed instead to have required *ewery* application programmer to become a database administrator.

2. in GLIDE, compilation and invocation time checking of records is weak; an argument is checked as a simple type, or as a record structure type, but no checks are made that the components of a record match what the program expects. Any command was subject to failing half-way through, leading to half completed updating and thus logical inconsistencies and integrity failures. Combined with only simple forms of exception handling, the result was a fragile system that could easily mangle a database, making it ill-structured for the desired applications.

3. in predefining the record format for polyhedra, trade-offs were made between fast execution and the desire to make very complex polyhedra. Also, surface types and attributes for faces, edges and vertices had to be fixed. Any fixed set of choices for the record format used in geometric modeling turned out to be inadequate for some application. Thus a more flexible means to add or modify geometric entities seemed desirable.

These shortcomings, plus the requirement for portability, became the motivations for modifying GLIDE. In addition, certain logical inadequacies in syntax and semantics were also addressed.

4.3. The Extended Objectives of GLIDE2

In order to respond to these issues, several significant changes were made in the design of GLIDE2. The easiest one to respond to was the limitation on the data structures supporting geometric modeling. Whereas we had assumed that it was possible to define at the system level a general structure for geometric modeling, this turned out to be premature. In GLIDE2 we eliminated these record types and instead provide adequate structures for developing different geometric models, to be offered as system records and procedures. This flexibility is achieved at a slight cost in processing speed.

The first two problems required a more detailed rethinking of the system design. It was clear that schema extensions, without strong aids for managing integrity, lead to inevitable run-time errors. There are two general approaches to improving integrity: First, better compile-time type checking can reduce the number of run-time errors encountered. Second,

better rules and procedures could improve the structure of integrity relations and thus simplify the kinds of integrity issues programmers have to deal with. Strong compile-time type checking, however, stylistically is not normally considered consistent with an adaptive system structure. Strong compilation type checking typically results in a rigid static organization of data and process. The compiled types define the semantics accessible by programs. New programs may use existing types but new or altered types require recompilation of the program.

In the earlier version of GLIDE, now called GLIDE 1, the application programmer was required to have global knowledge of the database context in which each application would be used, in order to manage the integrity relations between the extensions associated with the application and its context. This greatly limited the form of extensions that were possible. More general forms of extensibility seem to require that only (local) knowledge of the schema context should have to be known.

Local extensibility, as we understand it, involves the following four issues:

1. local naming of permanent data and processes. This implies a segmented name space, plus either compile-time equivalences and/or multiple pointer references for accessing shared data.

2. the ability to define new records and the- access paths between them and existing records during execution.

3. managing the global integrity of the database, including the correctness of data in the extensions vis-a-vis known permanent data as well as with other extensions.

4. adding new applications to operate on the extended data schema during execution.

The database extensions must be well structured, especially if integrity is to be guaranteed. The kind of structure we envision, that supports a combination of extensions, is shown in Figure 2. The database schema is organized as a "canonical model" of the object being designed. Extensions are appended to the canonical model or to other extensions. What they are appended to is determined by what existing information they depend on, eg. their functional dependencies. However, the functional dependencies may be in both directions, ie. an update of data in the extension may require an update in what it is appended to, and vice versa. Integrity of the extension and canonical model must be managed when updates are initiated in either location.

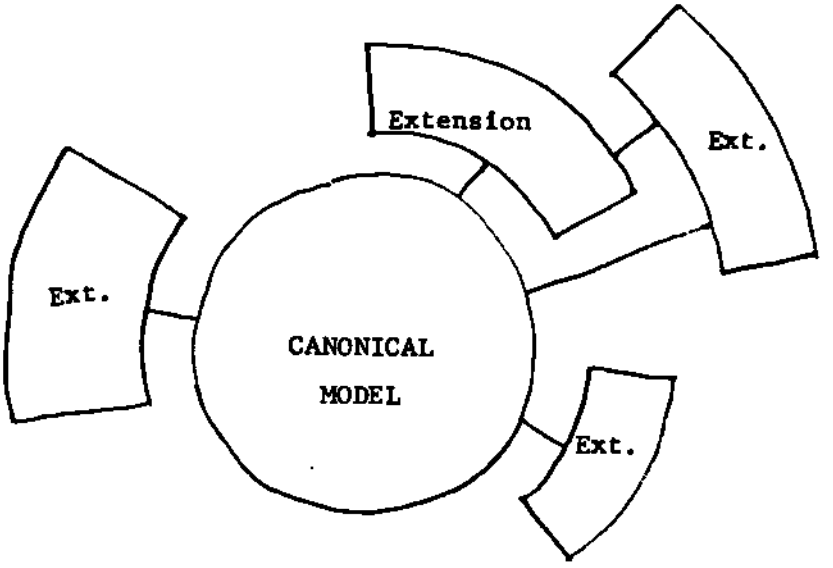

Figure 2: Modular extension of a database schema.

4.4. The Structure of GLIDE2

These functional needs suggested significant changes from the previous version of GLIDE, especially in the area of type definition. The needs also identified functional capabilities not adequately dealt with in existing language designs. In order to focus our efforts on these new issues and not divert effort to re-inventing other better understood language features, we chose to make GLIDE2 an extension of an existing language.

An obvious choice was Pascal, which was previously agreed to be the new implementation language. Pascal is strongly typed, incorporates a powerful set of structuring mechanisms, incorporates in its heap a run-time form of data allocation and it is relatively portable [Jensen and Wirth,1974]. In addition, it is amenable to the basic modifications needed to make a database language. The heap in standard Pascal can be replaced with a space management system, allocating to secondary storage. This space manager can also be made to manage routines compiled in P-code, so that a program is segmented and automatically swapped in and out of primary memory. The symbol table for the dynamic space manager (now the permanent database) can be stored at the end of a user session along with the record directory of the space manager for automatic re-start in multiple session use. Also, the outer block level can be made interpretive, so that compilation and execution follows each complete outer block statement.

The other features we desired, however, were more difficult, especially because they have

several possible implementations, each with differing side effects. Below, 1 outline some of the issues considered and the strategies followed for: segmented namespace, schema extensions and conversion, and integrity management. These features are still under review and some may be modified. We drew upon the experience of other recent language design efforts, especially Euclid [Lampson,1977], Telos [Travis, et al,1977], ALPHARD [Shaw, Wulf and London,1978] and the many papers reviewing the features of Pascal. For other extensions and details of the design of GL1DE2, see [Eastman and Thornton, 1979}

4.4.1. Segmented name space

Local naming of variables involves two issues: multiple use of common names, and accessing common data with differing names.

In most languages, (including Pascal), local use of non-unique names is allowed for stack allocated items. Uniqueness is required only at the block level. Sharing stack allocated variables is sometimes provided by an equivalence declaration (Wulf, et al.,1971) but is not a great importance for our objectives. Thus local naming is an issue only for permanent items defined in the database. Sharing of dynamically allocated data items and accessing them using different names is provided by the Pascal pointer. Non-unique names, however, implies some form of structure to the global namespace that is not provided in Pascal.

Our method for segmenting the namespace was to add a new construct, called a Frame, in which all permanent declarations reside and which is the context for all actions. Frames are organized in a tree; multiple Frames may be declared within another one, but each has only one "parent" Frame. One Frame, called the Root, has no parent. Only one Frame is the locus (or context) of action at any moment and is called the *current Frame.* Execution of any process assumes that all objects it refers to without Frame qualifiers are within the current Frame. Thus Frames may have declared in them any of the objects supported by GLIDE, in any order. (Deletion is a more complex issue, see Eastman and Thornton (1979), pp. 23-25). Names within a Frame must be unique. Objects outside the current Frame must be identified relative to it. A path through the Frame hierarchy may be specified that starts in the current Frame and ends with the simple Identifier for the desired object. This construct is called a *Pathname* and is defined with the following syntax*:

---

[1] The meta syntax *vd hmrrn* it that proposed by Wfrth [1977].

Pathname = [FrameSpecifler "V1 Identifier.

FrameSpecIf ier = Frameldent If ier { "V Frameldent If ler }$

Frameldentifier = Identifier "ROOT" | "I" | { "V I }

Two special Framenames are provided: ROOT for the root Frame and S for the parent of the Frame now accessed. A Pathname through a Frame hierarchy is specified by listing the Frame names that must be traversed, each separated by "\". In addition to the normal set of actions, we add the ability to move the locus of action from one Frame to another, using the commands ENTER (Frame) and LEAVE. (S and ROOT do not alter which is the current Frame.) Other operations *are* provided for creating and destroying Frames.

It is assumed that different designers will have their own Frames in which to operate. Some data will be unique to one Frame; other data will be shared by several, using pointer variables (with possibly varying names) for their common access. Applications can rely on local names within the context provided by a Frame.

4.4.2. Schema extensions

Schema extension involves, in addition to defining new records and instances of them, some way to access the new records from existing records, or vice versa. In a strongly typed environment, these access paths must be defined by types that are extensions to the definition of the existing types. Three classes of type extension are possible:

- using an existing type *to define a new type.* The new type can be called a *complex type* and a type used in defining the complex type a *base type.* Access to all the fields of some complex type should be possible with no possibility of run-time errors.

- a more powerful extension is *to extend a type and all the variables and components it defines by another type.* The extension adds new fields to the existing variables. Again, strong typing should prevail.

- the most general type extension is *to an individual variable.* That is, a variable is extended to be the union of its current type and some new base type.

Simply allowing new components to be added to existing types (and possibly all variables of that type) loosens type definitions and results in the possibility of run-time errors. Thus, in parallel to the notion of type extension, means must be provided for accessing the extensions, without the possibility of run-time errors, eg. by providing unambiguous determination whether or not the extended components are present in a particular variable.

This implied to us not only that a type extension results in an item being composed of multiple types, but that the components of only one type are accessible at a time. Conversion from one type to another must be handled explicitly.

In GLIDE2, type conversion is provided by a special form of discriminating case statement, called a *Typecase*. The Typecase has the following syntax:

```
TypeCaseStatement = "TYPECASE" SelectorIdent ":=" Expression
        "OF" TypeCase { ";" TypeCase } [ "OTHERWISE" statement
        { ";" statement } ] "END".

TypeCase = [TypeCaseLabel ":" Statement].

TypeCaseLabel = TypeIdentifier.

SelectorIdent = Identifier.
```

The SelectorIdent can be a variable of any type. The statements internal to the Typecase statement are labelled with type identifiers, called TypeCaseLabels. In the execution of the Typecase statement, each of the TypeCaseLabels is compared, in order, with the allowed type conversions of the SelectorIdent. For each match, the corresponding statement is executed. Within each executed statement, the SelectorIdent has as its type the type matching the TypeCaseLabel.

The Typecase statement isolates the conversion of a dynamic variable of one type to another type, allowing the extended fields to be accessed. The conversion is executed only if it is legal. The result is that programs not accessing an extended type need not be changed in any way. A schema extension however, defines new types that can be accessed by the applications that require them, using the Typecase for the conversion.

Since the third type of extension, of individual variables by a new base type, seems inevitably to result in run-time checks on the type of each data object and because its possible applications were not well understood, it has not been incorporated in GLIDE2. The first two have. They allow an application programmer to define extensions to the types available to a CAD system and to the components of existing records. Type extension of existing variables requires recompilation of the extended data objects but not the code that accesses them.[1]

---

[1]The current syntax provides a compile-time definition for defining new types as an extension of existing ones. It also provides a run-time procedure for extending existing types and all declared objects of that type or which use it as a component. The need for both compile-time and run-time constructs is being questioned, and under review.

4.4.3. Integrity management

Integrity of the extended model must be managed in two ways, corresponding to the two ways that updates may come about. Updates to the extended part of the schema can rely on operators provided within a structured programming Module that defines the extension, eg. the Form construct in Alphard [Flon,1975; Wulf, London and Shaw,1976). Such a structure type has been added to those provided by Pascal, in part because of its general value in supporting structured programming. A Module may include a number of fixed components, including *types* and *procedures.* It is an isolatable construct that has no Knowledge about objects outside of itself (unless they are explicitly Imported) and likewise the names internal to the Module are not available to be accessed from outside of it unless they are explicitly Exported. This provides a hiding mechanism that allows the programmer a narrow and well-defined interface between the outside world and the information internal to the Module. Types declared within a Module may not be Exported, though those outside may be Imported. Imported objects are bound to their corresponding objects for the scope of execution. Objects of type Module can be created in the same manner as other data types, eg. by declaring them or by dynamic creation.

For the needs of CAD, the definition of a Module may incorporate a schema extension, in terms of the new type definitions internal to it and in terms of extensions to existing types, the procedures that use the extended data and the operators allowed on the extended data. Instantiation of the Module creates the schema extension, as data of new types related to extensions of existing types. Because the application programmer knows to what part of the already extant schema the extension is related to, the Module operators can maintain *both* the internal integrity of the extension and the integrity resulting from modifications to the extension and requiring updates or checking of the extant schema.

Updates to the canonical model either directly or indirectly by the Module operators, however, must respond to the effects of many uncoordinated extensions. An update to it may have many effects, not known at the time the canonical model was defined. Needed is some way to add modularly to the semantic definition of the extant schema those relations it has to the extensions. These definitions must be additive in their effect.

Such integrity management goals were first explored in languages for artificial intelligence (AI), such as the Planner family of languages [Hewitt,1971]. They provided the concept of a "demon", a procedure that could be executed after any action to manage local semantic relations. An alternative tool for semantic integrity was developed in both AI and database research in the form of *integrity constraints.* An integrity constraint is (1) an assertion regarding the database that must be true for its data to be semantically meaningful. In order

to apply such an assertion, other information is required, including: (2) when the assertion is to apply and (3) the action to be taken when it does not. These three components are integrated into an operation called an *integrity check.* An integrity check is executed just prior to or after access to the data object it is bound to. It executes in one of three conditions: on a Read, a Write or Delete. An integrity check is bound to one of three possible semantic units, eg. by their simple types (or in Relational databases, to domains), or by their structured types (corresponding to Relations in the Relational model of data) or by the variables themselves (Relational tuples) see McLeod and Hammer,[1976]. Integrity checks differ from demons by changing the locus of control from a central mechanism that executed all integrity procedures after every action, to execution of a procedure only with a particular data access. The latter form could be structured to execute an integrity assertion only when one of the arguments of the predicate was altered.

We have implemented a limited form of integrity check mechanism and call them *System Invoked Routines* (SIR). In our implementation, the choice was in how to best associate procedures to a data item. The logical choice was to bind SIRs to types, both simple and structured ones. However, the need to add integrity procedures to an existing schema meant they would be added as type extensions. This implied however that Typecase conversions would have to be defined initially in all code accessing variables that might eventually have associated SIRs. The Typecase would have to have within it the possible extensions to each data item in the canonical model. These were exactly the kinds of responsibilities and required foresight we wanted to remove from the application programmer. Alternatively, explicit type conversions could be abandoned. But this would reduce the strength of typing, which to us was not an acceptable alternative.

A third alternative is to treat a SIR's association with data objects as strictly a run-time event. They would behave similarly to values and be "assigned" to a variable. Multiple SIRs could be assigned to a variable, however, and when the variable is accessed, the SIRs are executed in the order of their assignment. Explicit de-assignment is required to remove them from a variable.

A SIR is invoked just prior to accessing the object it is assigned to. An implicit system parameter defines the class of access, eg. READ, WRITE, UNDECLARE. The kinds of access correspond to entry points within the SIR, so that a unique block is associated with each. Only the block corresponding to the kind of access is executed.

In the context in which they have been introduced here, SIRs are used to update an extension to the schema, when the canonical model part of the database, or any other part that the extension relates to, has been updated. Each schema extension binds SIRs to

variables that if changed, result in changes to the extension. Each extension can add the needed SIRs to the canonical model to guarantee its own integrity without knowing about other possible extensions.

An important question is how to manage integrity constraint processing. In many cases, immediate propagation is not desired, especially when generating trial alternatives. In other cases, significant pre-processing is sometimes justified. Lafue is exploring alternative means of managing constraint processing, particularly delayed propagation [Lafue,!979} The proper management of SIRs so as to guarantee effective processing is an important research area, not yet adequately explored.

SIRs can be used in many other ways, beyond integrity management between schema extensions. They can be used for maintaining security of data, when the USERID is a system variable. They can be used in place of Module procedures, if this need is ever required.

The Module combined with SIRs offer one means for tractably maintaining integrity in a large adaptively extended system without re-compilation and without global knowledge of its structure[1].

5. Conclusion

A major cost in the development of large programs today is the integration of many stand-alone segments into one integrated package. (Indeed, this has been a major cost in implementing the two GLIDEs.) However, the ability to adaptively extend a program and database, without recompiling, offers a new form of program development that may prove attractive, we suspect, far beyond the area of computer-aided design. To date, only the two models, of compilation and interpretation, have been available. The ability of a compiler to aid in determining correctness and the potential for optimization have made compilation the logical choice for most system development. GLIDE2 suggests that these virtues may be combined with an incremental form of system development similar to interpretation that could significantly reduce the costs of producing large systems.

Note: The work described here was supported in parts by the National Science Foundation, grant number MCS-76-19072 and by the U. S. Army Corps of Engineers, contract number DACA88-78-R-0014. Helpful comments on earlier drafts were received from Gilles Lafue, Robert Thornton and Kevin Weiler. Any inaccuracies in describing our joint work, however, are the author's alone.

[1] Bindtnf of SIR* to variables is not completely satisfactory •specially from » conceptual viewpoint, but provides the best combination of features we have been able to realize thus far.

6. References

Akin, Omer, "Models of architectural knowledge: an information processing model of design", unpublished Ph.D. thesis, Department of Architecture, Carnegie-Mellon University, Pittsburgh, PA. 1979.

Alexander, C. NOTES ON THE SYNTHESIS OF FORM, Harvard University Press, 1964.

Barnhill, R. and R. Reisenfield, COMPUTER AIDED GEOMETRIC DESIGN, Academic Press, N. Y. 1974.

Baer, A., C. Eastman and M. Henrion, "A survey of geometric modeling", COMPUTER AIDED DESIGN, (1979), in process; also Institute of Physical Planning Research Report No. 66, Carnegie-Mellon University, March, 1977.

Bernstein, P. "Normalization and functional dependencies in the relational database model" Ph.D. Thesis, University of Toronto, Canada, 1975.

Birnbaum, M. Eastman, C, Lafue, G.Jhornton, R., and Weiler, K. GLIDE REFERENCES MANUAL, Second Edition, Institute of Physical Planning, Carnegie-Mellon University, October, 1978.

Bobrow, D. and B. Raphael, "New programming languages for artificial intelligence research", COMPUTING SURVEYS, 6:3 (Sept. 1974), pp. 155 - 174.

CODASYL, Database Task Group, April 1971 Report, ACM, New York, 1971.

Codd,E.F. "A relational model of data for large shared data banks", CACM. 13:6, 1970 pp.377-387.

Eastman, C. "Information and databases in design: a survey of uses and issues in building design", Proc. DESIGN RESEARCH SOCIETY CONFERENCE, Bristol, England, Sept. 1979.

Eastman, CM. and M. Henrion, "GLIDE: A system for implementing design databases", PArC International Conference on the Application of Computers to Architecture and Urban Planning, Berlin, May 1979.

Eastman, C. and K. Weiler, "Geometric modeling using the euler operators", Proc. FISRT-ANNUAL CONFERENCE ON COMP. GRAPHICS IN CAD/CAM SYSTEMS, D. Gossard ed. MIT Press, Cambridge, Mass, 1979.

Eastman, C. and R. Thornton, "A report on the GLIDE2 language definition", Institute of Physical Planning Report, 22 March, 1979, Carnegie-Mellon University, Pittsburgh, Pa.

Eastman,. C. and M. Henrion, "GLIDE: Language for design information systems", PROCEEDINGS ACM S1GGRAPH CONFERENCE 1977, San Jose, CA. ACM, New York.

Eastman, C, J. Lividini and D. Stoker, "A database for designing large physical systems" 1975 National Computer Conference, AFIPS Press, New Jersey 1975, pp. 603-611.

Flon, L. "Program design with abstract data types", Department of Computer Science Research Report, Carnegie-Mellon University, June 1975.

Freeman, P. and A. Newell, "A model of functional reasoning and deisgn", Proc. SECOND INTERNATIONAL CONF. ON ARTIFICIAL INTELLIGENCE, British Computer Society, London, 1971.

Hewitt, C. "Procedural Embedding of Knowledge in Planner" PROC. 1JCAI2, London, September 1971.

Jardine, D. THE ANSI/SPARC DBMS MODEL, North-Holland Press, N.Y. 1977.

Jensen, K. and N. Wirth, PASCAL USER MANUAL AND REPORT, Springer-Verlag, New York, 1974.

Koestler, A. THE ACT OF CREATION, Macmillan Co. N.Y. 1967.

Lafue, G. "An Approach to Automatic Maintenance of Semantic Integrity in Large Design Databases" 1979 NATIONAL COMPUTER CONFERENCE, AFIPS Press, June 1979, New York.

Lafue, G. "Integrating Language and Database for CAD Applications" CAD Journal, Vol. II, No. 3, May 1979a.

Lampson, B. W., J. J. Horning, R. L. London, et al. "Report on the Programming Language Euclid" SIGGRAPH Notices, February 1977.

McLeod, Dennis "High level expression of semantic integrity specifications in a relational database system" MIT/LCS/TR-165, Laboratory for Computer Science, MIT, Cambridge Mass., September 1976.

Travis, L. and M. Honda, et al. "Design Rationale for TELOS, A PASCAL-based AI Language" SIGPLAN Notices 12:8 (August 1977).

Wirth, N. "What can we do about unnecessary diversity of notation for syntactic definitions" CACM 20:11 (1977).

Wulf, W., D. Russell and A. N. Habermann, "BLISS: A system for systems programming" CACM.14:12 (1971).

Wulf, W. A., R. L London and M. Shaw, "Abstraction and verification in ALPHARD: introduction to language and methodology" Carnegie-Mellon University, Department of Computer Science Research Report, June 14, 1976.