AN APPLICATION-SPECIFIC AD HOC QUERY INTERFACE

by

Brian C. Smith and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/106

21 November 1990
(Revised 23 May 1991)

AN APPLICATION-SPECIFIC AD HOC QUERY INTERFACE

by

Brian C. Smith and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/106

21 November 1990
(Revised 23 May 1991)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

AN APPLICATION-SPECIFIC AD HOC QUERY INTERFACE

by

Brian C. Smith and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/106

21 November 1990
(Revised 23 May 1991)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

An Application-Specific Ad Hoc Query Interface*

Brian C. Smith, Lawrence A. Rowe
Computer Science Division-EECS
University of California
Berkeley, CA 94720

Abstract

An application specific database browser is described that allows end-users to express ad hoc queries that include geometric, scientific, and business data. The browser has the desirable properties that it is extensible, can be easily built using existing technology, and uses a paradigm that can be consistently applied across a wide variety of entity-oriented databases.

1. Introduction

An application-specific graphical user interface (GUI) for specifying ad hoc queries can significantly improve end-user database applications. Suppose that you had a database which contained all information about a semiconductor fabrication facility including equipment, utilities, and work in progress (e.g., lots, in-process measurements, etc.) and that a water line burst in the facility. While water flooded onto the lab floor and surrounding area, someone has to find the correct shutoff valve for the water line among the hundreds of valves and dozens of utility lines that feed the facility. After finding the valve, the water line must be traced to determine the equipment connected to the line, so it can be properly shut down before turning off the water. Depending on how long it took to find the valve, determine the equipment to shut down, and do it, the cost of the accident could be substantial.

Disasters such as this indicate the need for an application-specific tool to allow a casual user to perform ad hoc queries that include graphic data. With such a tool, the user could query the database to find the appropriate shutoff valves and determine what equipment must be shut down before closing the valve. An SQL interface could be used to query the database, but it requires a skilled SQL user who is very familiar with the database design. [1] Alternatively, a general-purpose GUI interface for specifying ad hoc queries (e.g., SIMPLIFY from Sun [19], Query-by-Example[23], the Data Access Toolset from Metaphor[4], VQL from Sybase[22], or Databrowse[10], to name a few) could be used, but these solutions all require the user to be familiar with the database design. Natural language interfaces (e.g., [9]) might also provide a solution, but human factors studies indicate that GUI's make better user interfaces than natural language interfaces given the current state of the art in both technologies[3].

A better interface would display a schematic layout of the facility that showed the utility lines and equipment, and allowed the user to find the shutoff valve and equipment connected to the water line by pointing at the water line with a mouse on the display. Figure 1 shows an example of such a schematic layout display which might be part of a larger application that also tracked other relevant facility data.

This paper describes the design and implementation of a simple, extensible system, called CIMTOOL, that allows end-users to specify and run ad hoc queries without having to know SQL or the database design. While the tool described is used in the setting of semiconductor manufacturing, the incremental query construction paradigm and alternative data presentations that it uses can be incorporated into a wide variety of applications. In addition, CIMTOOL was implemented using the PICASSO graphical user interface development environment, and serves as an example of a "real" application using that system. The remainder of the paper describes CIMTOOL and how it was implemented. Section 2 describes the features of CIMTOOL in detail and presents the application data model. Section 3 describes the implementation of the system, and section 4 discusses our experience with both the implementation and the incremental query construction paradigm. The data used in the examples is

*This research was supported by the National Science Foundation (Grant MIP-8715557) and the Semiconductor Research Corporation, Philips/Signetics Corporation, Harris Corporation, Texas Instruments, National Semiconductor, Intel Corporation, Rockwell International, Motorola, Inc., and Siemens Corporation with a matching grant from the State of California's MICRO program.

[1] Of course, you could build predefined queries for anticipated problems, but most problems are unanticipated

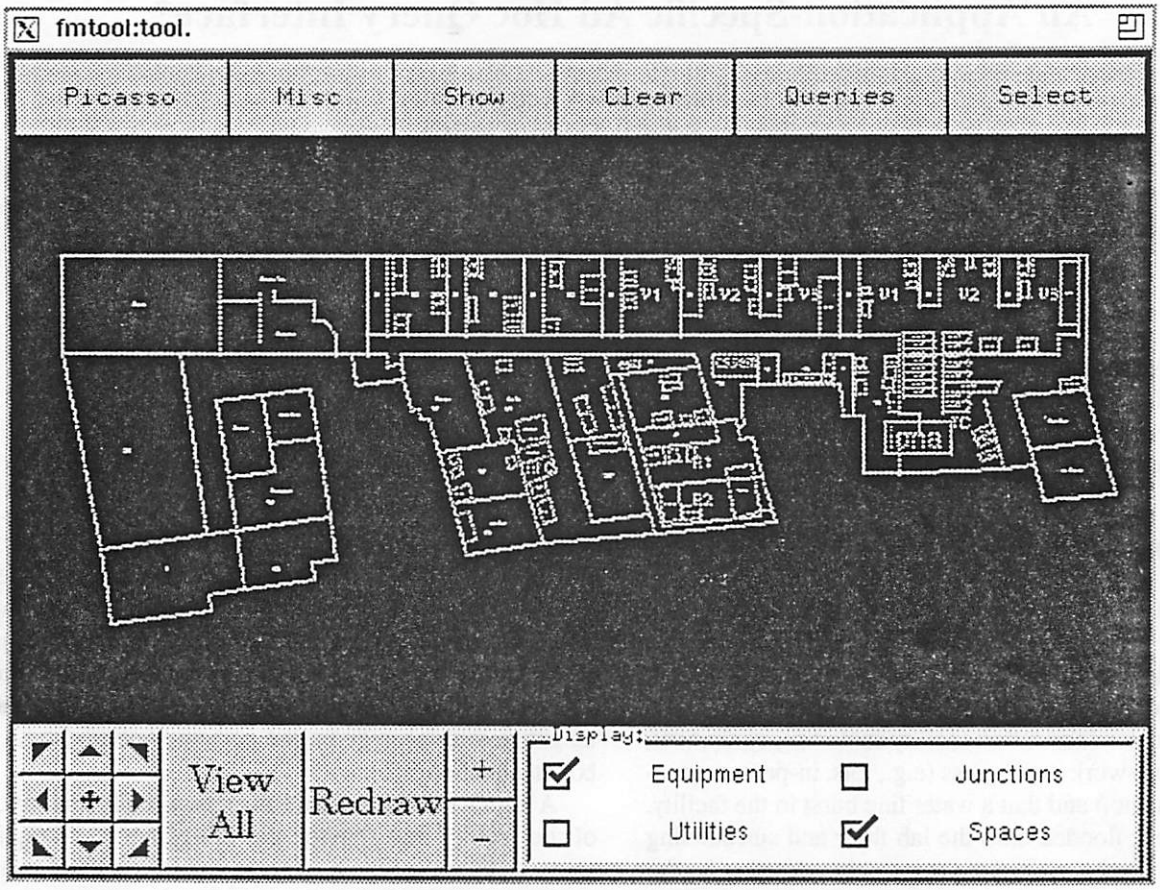

Figure 1: The schematic layout of the facility

stored in a relational database and represents the Berkeley Microfabrication Laboratory (the "Microlab") which is a 10,000 square foot class 100 fabrication facility used for research and education [21].

2. Interface

Figure 2 shows a screendump of the application as it appears on a high resolution bitmap workstation. A floorplan of the facility is displayed in the main window shown in the upper left corner of the figure. The window in the upper right corner is a panel displaying two hierarchical browsers for the equipment in the facility, each of which allow the user to browse a list of equipment. A panel is an auxiliary window of the application which the user may hide at any time. The panel in the lower right corner displays two hierarchical browsers for the utilities in the facility.

Figure 1 is actually an enlarged view of the main window of CIMTOOL. This window displays the floorplan layout of the facility. On a color monitor, entities are displayed in different colors to make it easier for a user to identify and locate different types of entities (e.g., equipment is green, walls are purple, and utilities are red). Various mouse and keyboard inputs invoke operations that allow the user to change the center of the view (i.e., a *pan* operation) and to magnify part of the image (i.e., a *zoom* operation). Other navigation and display options are provided by the buttons below the floorplan (e.g., pan, view all, redraw, and zoom in or out by factors of 2). The checkboxes on the right specify which entities are displayed. For example, a user can toggle the display of junctions at any time by clicking the mouse (i.e., "mousing") on the checkbox labeled "junctions". Turning off the display of entities speeds up the refresh, pan and zoom operations, since there are fewer objects to draw.

Figure 3 shows CIMTOOL's **Equipment Panel**. This panel displays two hierarchical browsers, labeled "UNSELECTED" and "SELECTED", and several buttons. The hierarchical browsers allow the user to view the equipment in the facility. Since the Berkeley Microlab contains over 125 separate pieces of equipment, a single list would be too unwieldy. Consequently, the equipment is listed by *class*, *make*, and *instance*. The class specifies

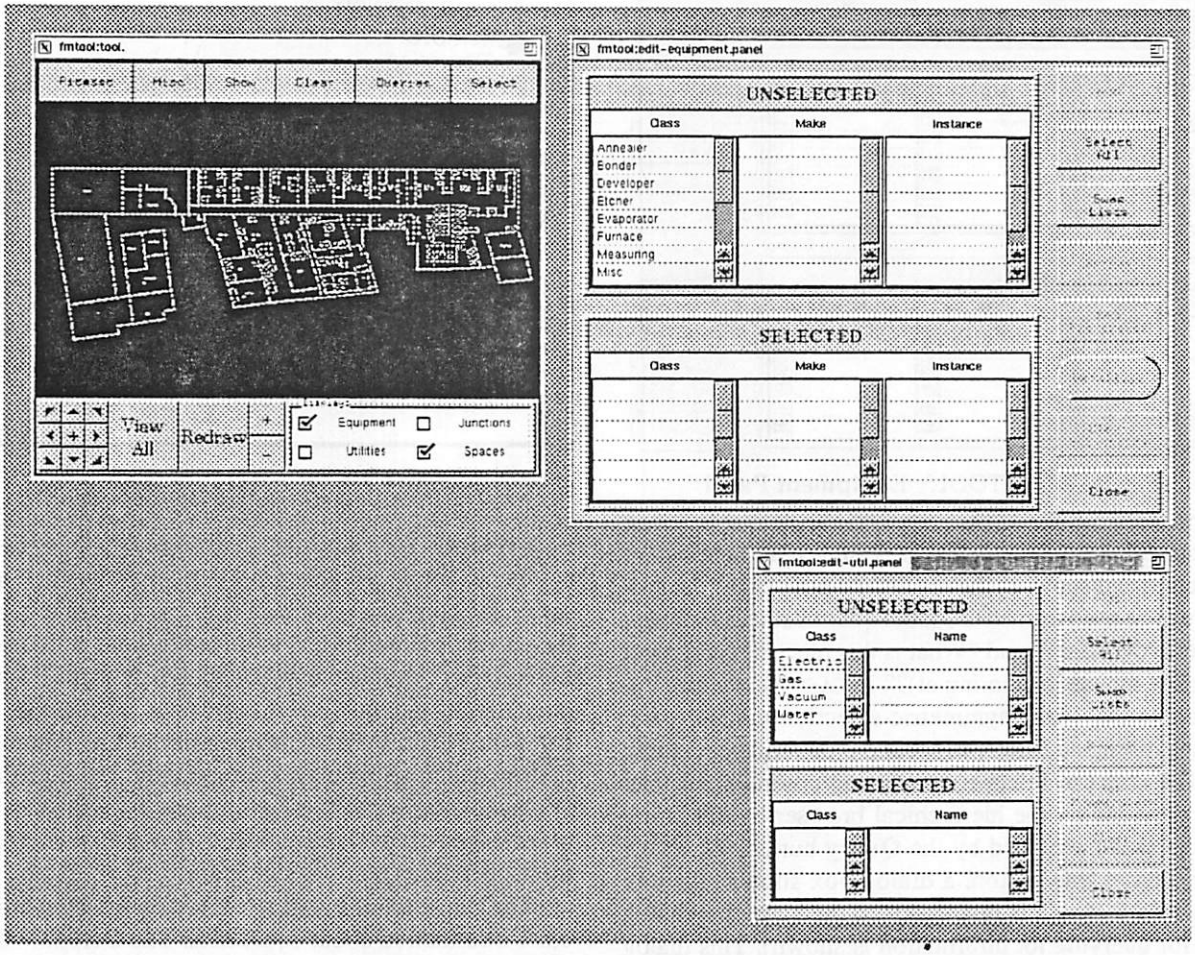

Figure 2: A screendump of CIMTOOL

the type of equipment (e.g., "furnace" or "etcher"). The make identifies the equipment manufacturer (e.g., a "Tylan" furnace or "LAM" etcher). And, the instance specifies a unique identifier assigned to each piece of equipment (e.g., "LAM-1").

To browse the equipment, the user clicks the mouse on the item they want to see in more detail. For example, the user has selected the etcher entry in the class column of the "UNSELECTED" browser with the mouse is the figure, indicating that he wants to see the list of etcher manufacturers.

The user can then browse the etchers of a particular make by selecting the desired manufacturer in the make column or browse another type of equipment by selecting a different class. Other mouse and keyboard inputs allow the user to view more than one class or make of equipment at a given time.

CIMTOOL partitions the equipment in the facility into two disjoint sets: *selected* and *unselected* equipment, which are always displayed in the "SELECTED" and "UNSELECTED" browsers, respectively. At any given time, the selected equipment roughly corresponds to the equipment the user is currently manipulating, and the unselected equipment is all remaining equipment. The user modifies this selected set using either the graphic window or the panel. In the graphic window, equipment may be selected by pointing at a piece of equipment with the mouse or by dragging a box around several equipment icons. Commands are also provided to unselect equipment.

The user can also modify the selected set by clicking the mouse near a particular class, make or instance of equipment in the browser (which highlights the equipment thus selected) and executing the operations provided by the buttons on the right side of the **Equipment Panel**. For example, the **Add** button adds the highlighted equipment in the "UNSELECTED" browser to the selected set. The **Remove** button removes items highlighted equipment in the "SELECTED" browser from the selected set.

The selected entities are highlighted in the main window and listed in the "SELECTED" browser regardless

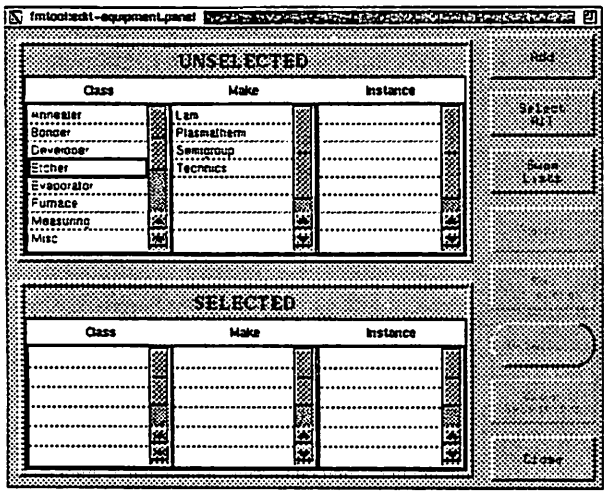

Figure 3: CIMTOOL's **Equipment Panel**

of the selection method. By providing both methods of selection and showing the results in both places, the system can be used to answer spatial queries such as "Where is the LAM ether?" and "What is this piece of equipment (that the mouse is pointing at)?" PICASSO made the implementation of the symmetric selection method easy. This is discussed further in section 3 on implementation.

The user can select entities using criteria other than that provided by the hierarchical browser by executing the operation provided by the **Query** button. When the user presses this button, a dialog box such as the one shown in Figure 4 is displayed (in this case, the dialog box for querying lot information is shown). This dialog box lists all attributes of lots and allows the user to specify values and ranges of the various attributes.

The buttons on the right side of the dialog box specify how the lots that satisfies the query are to be combined with the current set of selected lots. The following set operations that combine the result of the query, called the *query-set*, and the current *selected-set*, are provided:

Add: $result = selectedSet \cup querySet$

Remove: $result = selectedSet - querySet$

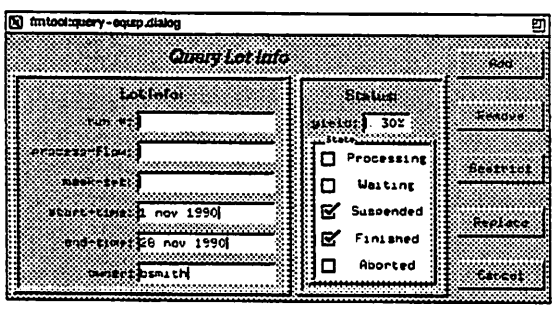

Figure 4: **Query Lot** Dialog Box

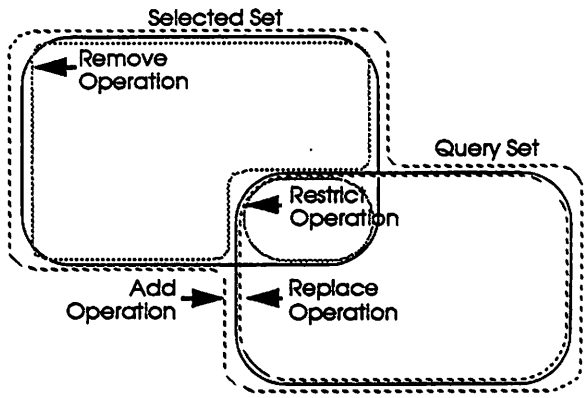

Figure 5: Summary of Joining Operations

Restrict: $result = selectedSet \cap querySet$

Replace: $result = querySet$

For example, to select just the lots that match the query, the **Replace** command is executed. Or, to restrict the currently selected lots to those that match the query, the **Restrict** command is executed. Executing the desired operation by pressing a button hides the dialogs box, updates the "SELECTED" browser in the **Lot Panel**, and updates the main window appropriately. The **Lot Panel** is a panel similar to the **Equipment Panel** which displays selected and unselected lots. Figure 5 summarizes these operations in a Venn diagram.

The **Equipment Panel** also allows the user to browse other information about equipment. The **Detail...** button pops up a menu that allows the user to view maintenance logs for the equipment or to display an image of the equipment. Other equipment-specific functions can be added to this menu such as activating real-time monitoring and control of the equipment, viewing schematics, and accessing an on-line video database that reviews disassembly procedures, theory of operation, and other relevant documentation.

Figure 6 shows an entity-relationship (ER) diagram for the following entities in the database:

equipment	(e.g., furnaces, etchers and microscopes),
utilities	(e.g., gases, electricity and water),
junctions	(e.g., shutoff valves, sprinkler heads and outlets),
spaces	which designate various areas of the facility (e.g., rooms or areas such as the lithography area), and
lots	of wafers.

A panel similar to the **Equipment Panel** is defined for each entity type that includes two hierarchical browsers, one for selected entities and one for unselected entities, and buttons to modify the selected and unselected

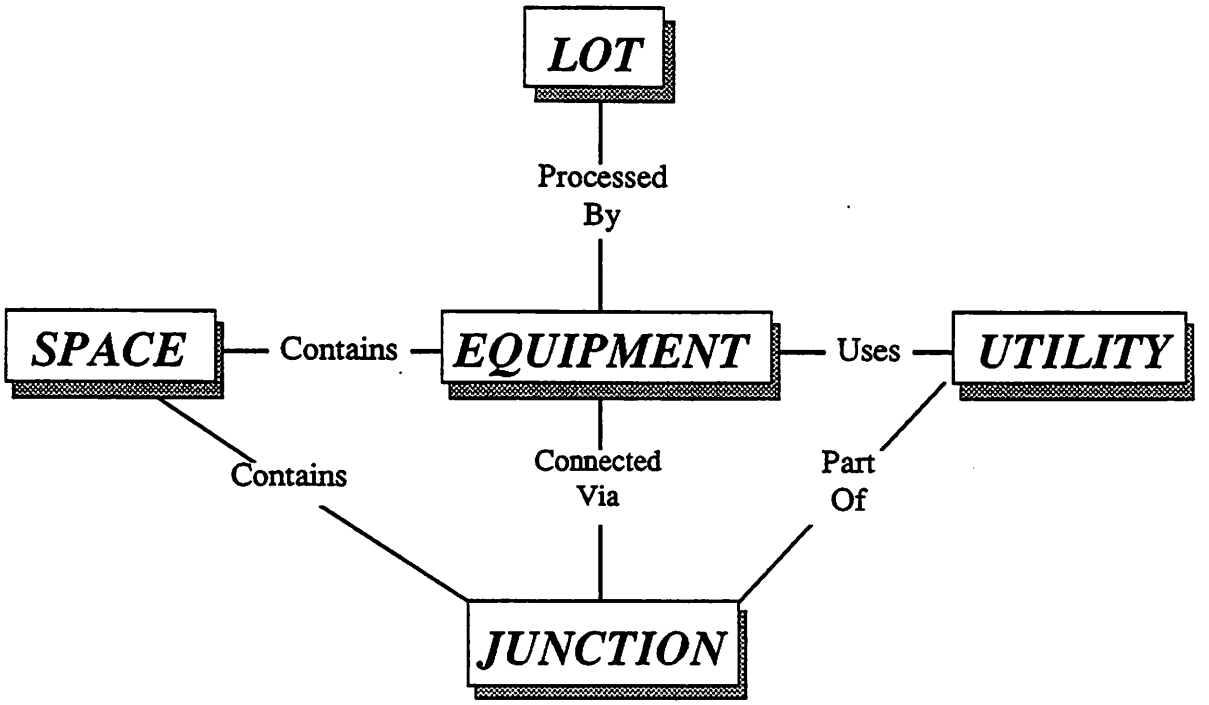

Figure 6: Entity-Relationship Diagram of the CIMTOOL Data Model

sets, query the entities, and examine more detailed data about the entities. The keys in the hierarchical panels are:

equipment (class, make, name)
utility (class, name)
junction (kind)
space (clean-level, name)

The hierarchical browsers in the **Lot Panel** are different than the other panels. Users typically want to see most lot attributes simultaneously and they do not prefer one attribute over another. In this case, the hierarchy is only one level deep and there are no duplicate entries, so the browser simply displays a list of lots. Users normally select lots using criteria specified by a **Query** operation (e.g., "select all lots with yield less than 30%").

The relationships in the ER diagram in figure 6 are summarized in Table 1. Knowing these relationships, the user may formulate queries that use more than one entity. For example, suppose the user wants to list utilities used by the "LAM" etcher. To specify and execute this query, the user clears the selected equipment list and selects the desired etcher(s) either by selecting it in the graphic window or the **Equipment Panel**. The user then executes the **Utility...** operation in the **Select** menu in the main window which displays the dialog box shown in figure 7. The radio buttons on the left side of the dialog box allow the user to specify the desired relationship which can be either "Utilities connected to the Selected Equipment" or "Utilities with Selected Junctions". In this case, the user pressed the button labeled "Utilities connected to Selected Equipment", which indicates that the user wants to select all utilities connected to the equipment in the selected set of equipment (i.e., all utilities connected to the specified etcher(s)). The buttons on the right side of the dialog box allow the utilities that satisfy the query to be combined with the currently selected utilities using the same method as the **Query** dialog box.

A sequence of selection and join operations can be performed to formulate complex queries incrementally. For example, suppose the user wants to inspect lots processed by equipment using either the oxygen or nitrogen

Entity	Relationship	Type
Equipment	Uses Utility	many/many
	Connected by Junction	one/many
	Inside Space	many/many
	Processed Lot	many/many
Utility	Used by Equipment	many/many
	Has Junction	one/many
Space	Contains Equipment	many/many
	Contains Junction	many/many
Junction	Part of Equipment	many/one
	Part of Utility	many/one
	Inside Space	many/many
Lot	Processed by Equipment	many/many

Table 1: Relationships Between Database Entities

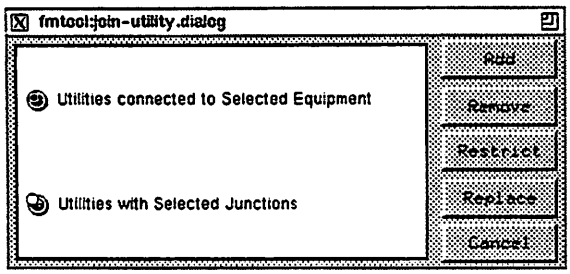

Figure 7: **Query Lot** Dialog Box

gas utilities. The user can execute the query by selecting both oxygen and nitrogen in the **Utility Panel**, selecting the equipment that uses these utilities through the **Select Equipment...** dialog box, and selecting the lots processed by this equipment through the **Select Lots...** dialog box. At each step the user is given feedback, through the panels and graphic window, that shows the result of the currently specified query. For comparison, the equivalent SQL query is:

```
select * from lot
  where lot.rid = recipe_eq_map.rid AND
    recipe_eq_map.eq_id = equip.id AND
    equip.id = eq_junction.eid AND
    eq_junction.uid = utility.id AND
      (utility.name = "oxygen" OR
       utility.name = "nitrogen")
```

Although easier to use than SQL, the ad hoc query facility in CIMTOOL lacks the power of a full-function relational algebra. A relational algebra uses relations as operands, and new relations are produced as the result of an operation [20]. For example, a new relation can be formed which is the result of a join between two distinct relations. In CIMTOOL, relations are equivalent to groups of entities, such as equipment or utilities. Since the types of entities in CIMTOOL are fixed, there is no operation corresponding to a join in the sense of a relational algebra. In other words, there is no way to create a new entity type dynamically. Instead, the operations provided through the **Select** menu are the composition of the equi-join, a selection, and a projection operators in a relational algebra. CIMTOOL's interface provides no analog to the divide operation in a relational algebra. If the power of a full relational calculus were required by an advanced user, a panel could be added through which an arbitrary, ad-hoc query could be entered either through a GUI or by typing an SQL expression.[2]

CIMTOOL's interface is an alternative to a full function ad hoc query interface that simplifies query specification by fixing the entity types, by providing incremental feedback, and by making the common operations easy (e.g., selection in the hierarchical browser). A formal human factors study to compare CIMTOOL with other ad hoc query interfaces has not been done, but our experience indicates that end-users learn the application with a minimal amount of training.

[2] Some GUI query interfaces show the SQL query currently specified by the user. This feature is not currently supported in CIMTOOL, but it could be added easily.

3. Implementation

This section describes the design and implementation of CIMTOOL. CIMTOOL was built using the PICASSO GUI Application Development System [RKS90, SRK90]] which uses the X window system [17] and is implemented in Common Lisp [18] and the Common Lisp Object System (CLOS) [5]. CIMTOOL uses features of PICASSO and CLOS extensively.

CLOS is a multiple inheritance object-oriented programming system for Common Lisp. The CIMTOOL data structures are therefore object-oriented, and based on a design presented in [12]. Since CIMTOOL models a real-world environment (i.e., the fabrication facility) the attributes of the various classes were largely dictated by the properties of the real entities. Furthermore, we wanted to access data already stored in a relational database, so the data structures were designed to allow us to load the data easily from the database. Figure 8 shows the class hierarchy for the CIMTOOL data structure design.

The *facility* class models the facility as a whole. The *equipment* class models a piece of equipment in the facility, such as the "LAM" etcher. The *utility* class models a single utility line in the facility, such as an oxygen line. The geometry and topology of a utility can be thought of as a directed graph. This information is contained in the classes *connection* and *vertex*, which correspond to the edges and nodes of the directed graph, respectively. A *vertex* is a point in the floorplan view of the facility. A *connection* links two vertices by a straight line. The *space* class represents an area in the facility, such as a room. The *walls* attribute of a space contains the geometry of the space, stored as a list of instances of the *walls* class. Some of the walls are not really physical barriers, but merely boundaries for an area in the facility (e.g., the lithography area). These types of walls are represented by instances of the *virtual-wall* class which is a subclass of the *walls* class. Virtual walls are drawn on the display in the same color as walls, but with dotted lines.

Junctions in the facility, such as shutoff valves, sprinklers and wall outlets, are stored as instances of the *junction* class. The *eq-junction* class is a subclass of the *junction* class that represents the junction used to connect a utility to a piece of equipment. Since there are hundreds of junctions in the facility, data common to the types of

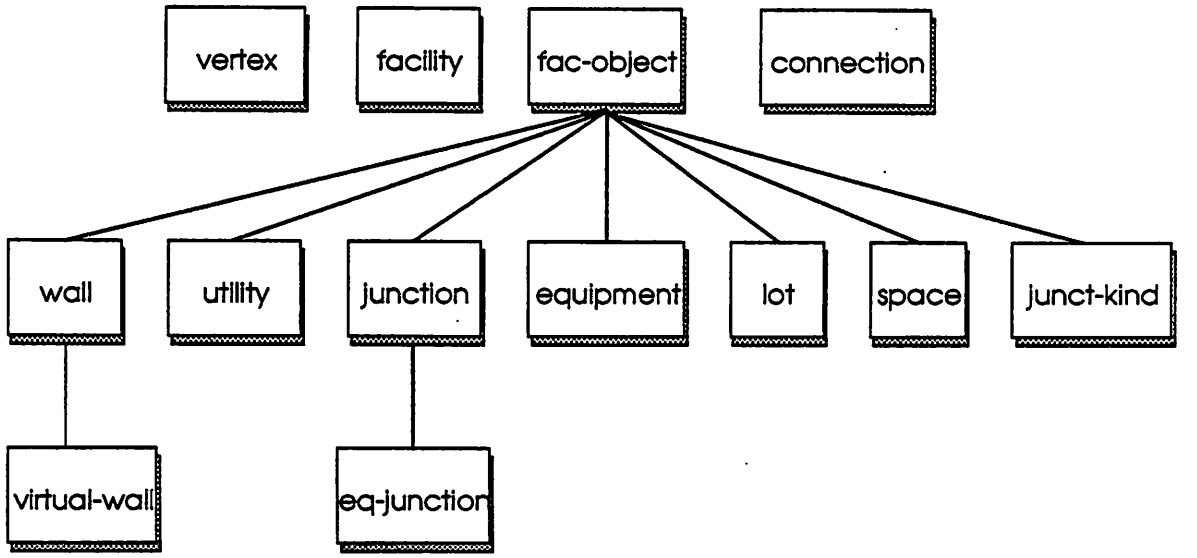

Figure 8:The CIMTOOL Class Hierarchy

junctions within the facility, such as the bitmaps used to draw a type of junction, is factored out and stored in the *junct-kind* class.

In general, most classes are derived from the entity-relationship diagram. Those classes in the figure but not present in the ER diagram are used to model objects with many pieces (e.g., *connection* and *vertex*) or to model objects in which parts were shared (e.g., *walls*). Relationships in the ER diagram are modeled as lists of object identifiers stored in an attribute of the entity.

Just as CLOS provides a natural framework for data structure design, PICASSO provides a natural framework for building GUI's. The PICASSO framework defines five object types: applications, forms, frames, dialog boxes, and panels. An *application* is composed of a collection of frames, dialog boxes, and panels. A *form* contains fields through which data can be displayed, entered, or edited by the user. The graphical display area in the main CIMTOOL window, along with the view controls, is an example of a form. A *frame* specifies the primary application interface. It contains a form and a menu of operations the user can execute. The main CIMTOOL window shown in Figure 1 is actually a frame. A *dialog box* is a modal interface that solicits additional arguments for an operation or user confirmation before executing a possibly dangerous operation (e.g., deleting a file). The **Query Lot** dialog box, shown in Figure 4, is an example of a dialog box; the user is forced to respond before continuing. A *panel* is a nonmodal dialog box that typically presents an alternative view of data in a frame or another panel. The **Equipment Panel** show in figure 3 is an example of a panel.

Applications, forms, frames, dialog boxes and panels are collectively called PICASSO objects (PO's). Frame, panel, and dialog box PO's are analogous to procedures, co-routines, and functions found in conventional programming languages. Each can be passed parameters and declare local variables. A variable which cannot be resolved locally is searched for in its lexical parent. Figure 9 shows the PO's in the CIMTOOL application. In the figure, each PO indicates its lexical parent.

Most CIMTOOL data structures are sets of CLOS objects. In Common Lisp, the natural way to represent a set is as a list. Three lists of equipment are maintained in variables declared in the `main-frame`: `equip-list`, `sel-equip`, and `unsel-equip`. These correspond to all equipment, the selected equipment, and the unselected equipment, respectively. Note that the unselected equipment is just the set difference of the `equip-list` and the `sel-equip`. Similar lists are defined for utilities, lots, junctions and spaces.

Other lists stored in variables defined in the `main-frame` complete the model of the facility. They are summarized in Table 2. Each list is populated with CLOS objects created by loading data from a relational database during the initialization of the tool. The geometric, topological, and reference information is also resolved at this time. For example, the connections and vertices for utilities are combined into polygons stored in main memory, as is the geometric information for spaces, walls, and equipment.

PICASSO provides a wide variety of interface abstractions such as buttons, scroll-bars and menus, collectively called *widgets*. Widgets are also represented by CLOS objects. Two types of widgets were developed specifically for this application: the graphics widget that dis-

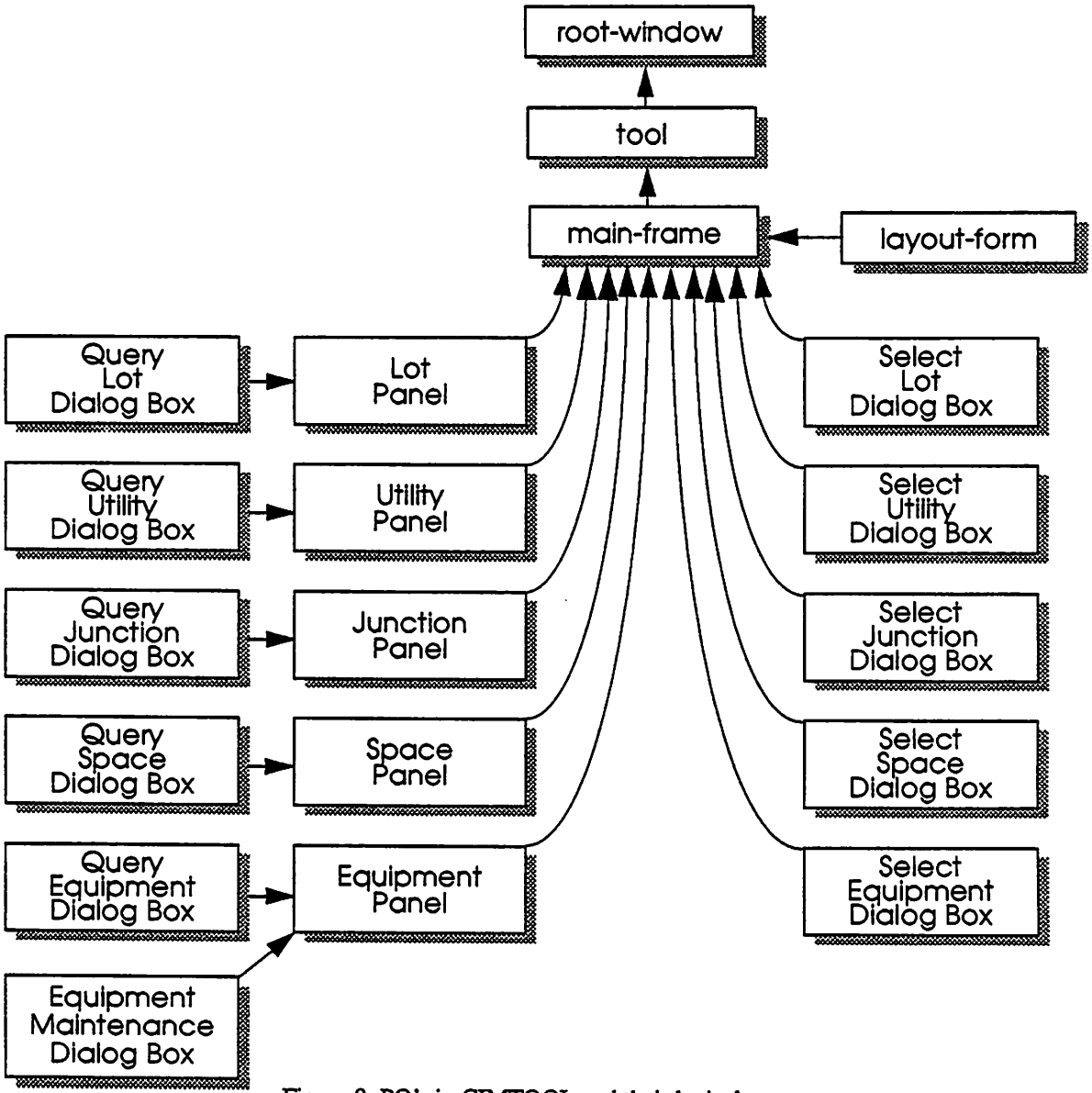

Figure 9: PO's in CIMTOOL and their lexical parents

plays the facility floorplan and the hierarchical browser widget that implements the selected and unselected abstractions in the panels. These widgets use the tool data structures as shown in figure 10 and described in the following paragraphs.

The data displayed in the graphics widget is a tree of graphics shapes (i.e., objects of class *shape*). Two subclasses of the *shape* class are the *polygon* class and the *annotation* class which represent polygons and text. These classes are built into PICASSO. All facility entities that have a graphical representation (i.e., equipment, spaces, junctions and utilities) inherit from the *polygon* class. By inheriting from shapes, facility entities can be displayed directly through the graphics widget.

The graphics widget stores a list specifying the current *selection*. The selection is a list of objects that are currently selected and highlighted in the widget. When the mouse is used to select an item in the graphics wid-

Variable	Element Type	Description
eq-junct	eq-junction	Junction on the equipment
wall-list	wall	Walls referenced by spaces
connect-list	connection	Connection for utility graph
vertex-list	vertex	Vertices used by walls junctions, and connections
junct-kind-list	junct-kind	Junction types in junct-list

Table 2: Lists maintained by CIMTOOL

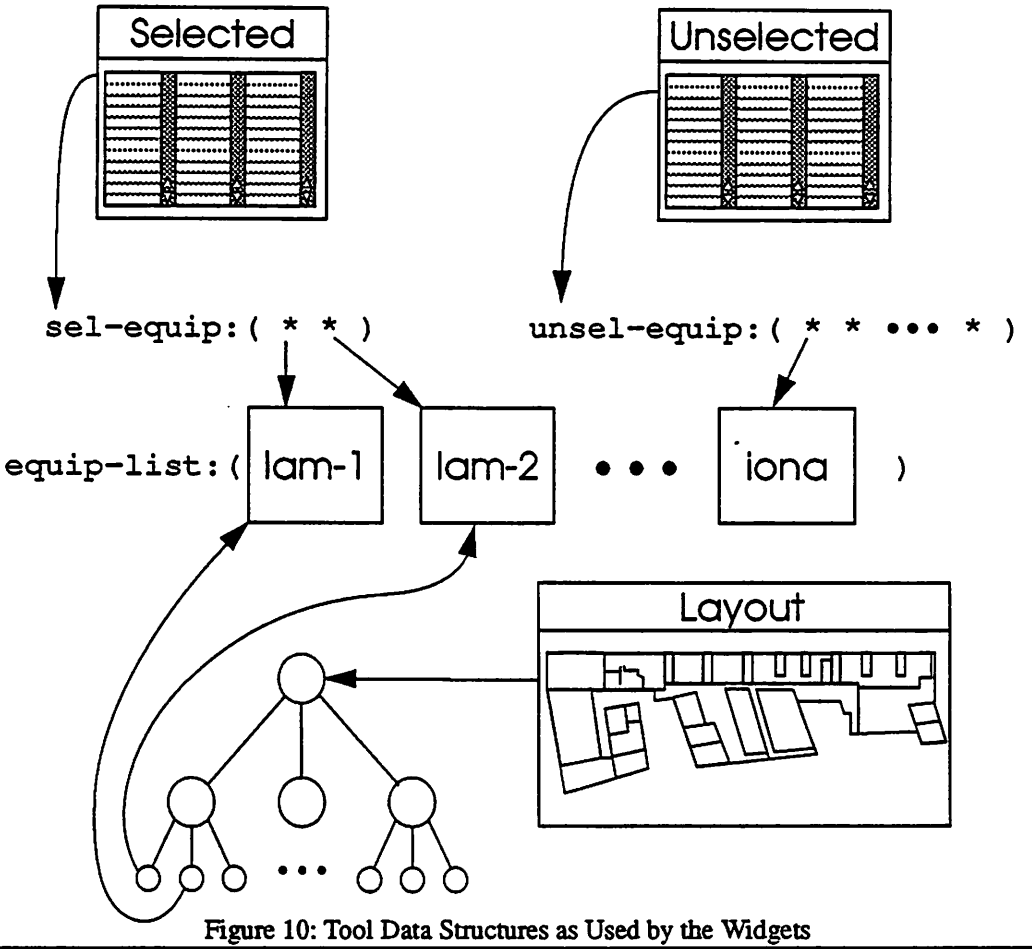

Figure 10: Tool Data Structures as Used by the Widgets

get, the nearest visible item on the screen is added to the selection. If the *selection* of the graphics widget is modified (e.g., by mousing in the graphics window), the application is notified through the binding mechanism described below.

The hierarchical browser widget, or "browser", displays a list of entities. For example, the "SELECTED" browser in the **Equipment Panel** displays the list of equipment stored in the `sel-equip` variable. When an item is selected in a browser, the browser *selection* is modified to include the subset of items in the browser that the user has selected. Like the graphics widget, the browser notifies the application when its selection changes through the binding mechanism described in the next paragraph.

PICASSO provides a constraint mechanism for CLOS object slots and PICASSO variables in the form of *bindings*. A function can be bound to any PICASSO variable or CLOS object slot that will automatically update the variable or slot when the value of the function changes. PICASSO is not the first toolkit to provide constraints. GROW[1], ThingLab[6] and Garnet[8] are among the systems to provide this feature. Although confusing to use at first, constraints greatly simplified the implementation of CIMTOOL.

Bindings are used to maintain consistency between the browsers and the lists they display in CIMTOOL. Figure 11 graphically depicts the constraint network. For example, the value displayed in the "SELECTED" browser in the equipment panel is bound to the value of the `sel-equip` variable of the tool. Thus, whenever the `sel-equip` of the tool changes value, the browser will automatically display the correct value. Similar bindings are created for the other browsers.

Bindings are also used to link the graphics widget to the internal data structures of CIMTOOL. The *selection* of the graphics widget is bound to a list which is the concatenation of the `sel-equip`, `sel-util`, `sel-space`, and `sel-junct` variables of the tool. The `sel-equip` variable is bound to the subset of equipment items in the *selection* of the graphics widget, and the `unsel-equip` is bound to the set difference of the `equip-list` and the `sel-equip`. Similar bindings are created for lots, spaces, utilities and junctions.

Bindings are also used to prevent the user from executing illegal operations. For example, the **Add** opera-

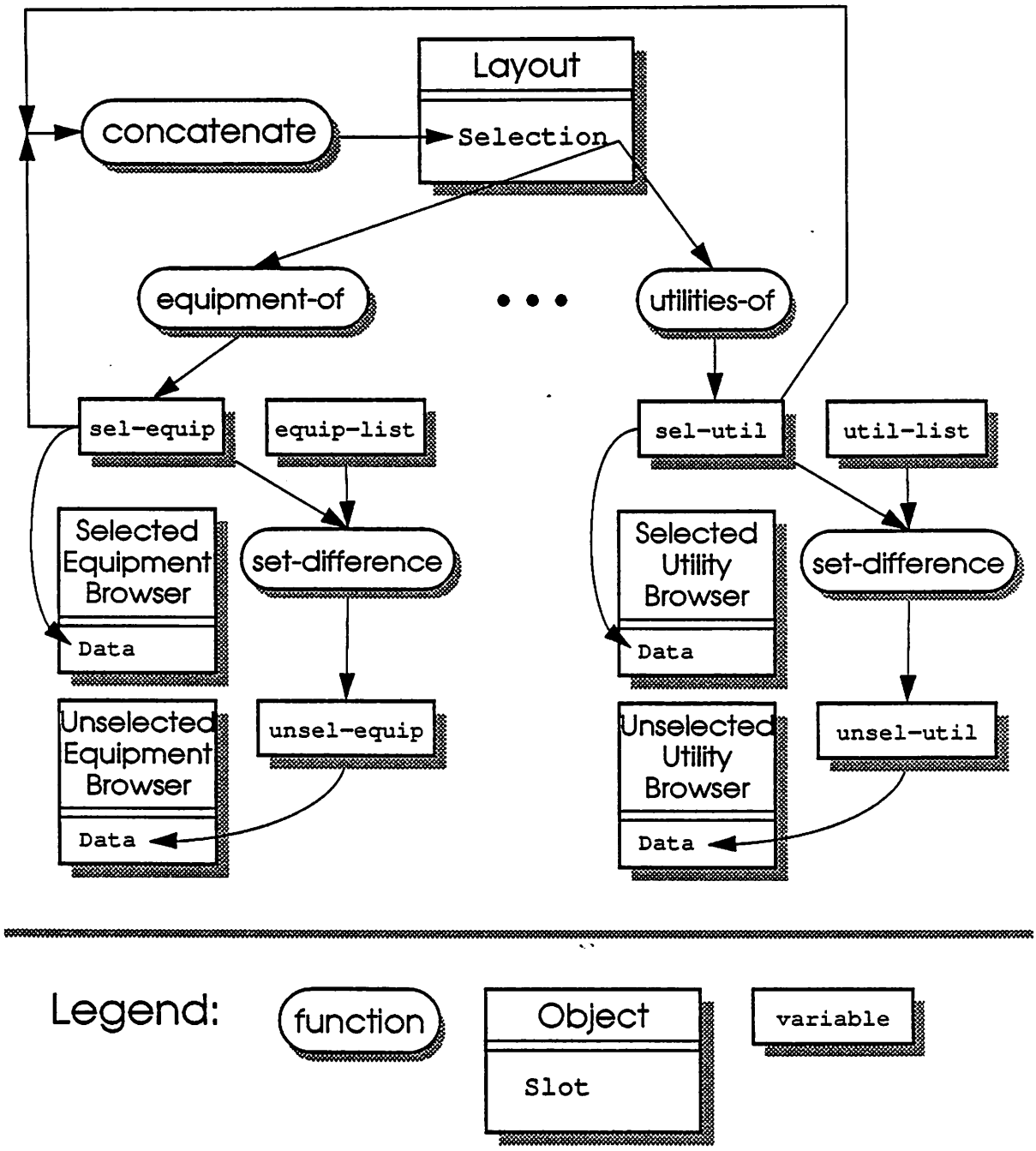

Figure 11: CIMTOOL constraint network

tion in the equipment panel is illegal when the *selection* of the "SELECTED" browser is null (i.e., no equipment has been specified to add), so the **Add** button should be dimmed to reflect this state. This behavior is implemented by binding the *dimmed* slot of the **Add** button object to a function that returns true (i.e., the button should be dimmed) when the *selection* of the browser is null.

CIMTOOL, as described here, was originally written in about a 2 week period, and consists of approximately 1400 lines of interface description code (including comments) and an equal number of lines of code for the application. We have made many changes since then, including support for multimedia data as described in the next section.

4. Discussion

This section discusses the rationale for CIMTOOL's main memory database design, extensions made to CIMTOOL to support multimedia data, and the extensibility of the interface.

4.1 Main Memory Representation

When CIMTOOL is initialized, the data needed to browse the various entity types is loaded into main memory from the database. For example, the class, make, instance, id, and geometric information for browsing equipment is retrieved for all equipment objects in the facility.

We chose to load the data into main memory and run queries locally, rather than execute queries in the database, to minimize interactive response time and to make the implementation feasible with our limited manpower resources. We feared that running queries remotely (i.e., in the database) to retrieve browsing data would be too slow for the short interactive response time that CIMTOOL requires [15]. Also, retrieving the data dynamically would almost always be unnecessary, since most of the information loaded would not have changed since the previous load.

Another option we considered was to use main memory as a cache for the data in the database. Implementing this strategy consistently would require a cache management policy. Since our data manager did not have such a policy built in, we would have to modify other applications that use the database. Because CIMTOOL was intended as a prototype application, we decided not to pursue this option.

Instead, we chose to approximate a solution to the problem of synchronizing the main memory cache and the database. We did this by partitioning the data into two groups, called "static" and "dynamic" data. Static data is data that is unlikely to change over the duration of a session of CIMTOOL usage. An example of static data is the schematic layout of the facility (people don't move 6000 pound machines too often!). Static data is loaded once at CIMTOOL start-up.[3] Dynamic data, on the other hand, is data that is likely to change over the duration of a CIMTOOL session. Equipment maintenance logs are an example of dynamic data. Dynamic data is loaded from the database as it is needed.

This approach works well for CIMTOOL since most of the information used for browsing and selecting entities is static. Furthermore, the static data is fairly small (about 70 KBytes) and can easily fit into workstation memories. Dynamic data that is used for selection is accessed through the **Query** button in the panels. Since this operation queries the database, it always accesses current data, and a small delay seems acceptable for this interaction.

The partitioning provides a good, but not perfect, solution to the problem of synchronizing the main memory database with its counterpart stored in the data manager. An active database would provide a better solution. The ideal tool would be a persistent programming language with a main memory database, as suggested in [13].

4.2 Extensions

CIMTOOL allows the user to access business, scientific and graphical data through a uniform interface. The most recent extension to the application has been to provide access to video data. Our purpose for introducing video was twofold. First, we wanted to prototype a system that could demonstrate the potential of this media to the CIM research community. And second, we wanted to provide the basis of multimedia extensions to PICASSO.

To support video, we needed to introduce several new hardware components to handle the data. Providing a fully digital representation of the video data seemed an overwhelming task, since no commercial products existed at the time we implemented this extension, so we opted for a hybrid analog and digital system.

The hardware consists of a Pioneer 4200 laserdisc player which provides stereo sound and a standard NTSC video signal that can be viewed through a video window interface card that displays the video signal in an X window on a Sun Sparcstation. The player is controlled through an RS-232 interface connected to a serial port on the workstation, as shown in figure 12. ASCII commands to the player allow the application to position the read head at a video frame, to play until a certain frame is reached, and to enable or disable either stereo channel or the video display. The software interface consists of about 300 lines of Common Lisp code.

The 30 minutes of video data on the videodisc consisted of approximately 5 minutes of still images of the equipment in the laboratory, 10 minutes of introductory material on semiconductor manufacturing, and a 15 minute overview describing the history and use of the Berkeley Microlab [7]. In addition, since all audio data was monaural (only the left stereo channel was used), we had about 30 minutes of compact disc quality audio storage available on the right stereo channel. We used this storage to record an audio help system.

The videodisc player associates a number with each frame of video data. Frames are numbered sequentially from the beginning of the disc. Segments of video data

[3] Of course, if the tool were left running continuously on a workstation, we could reload the static data periodically.

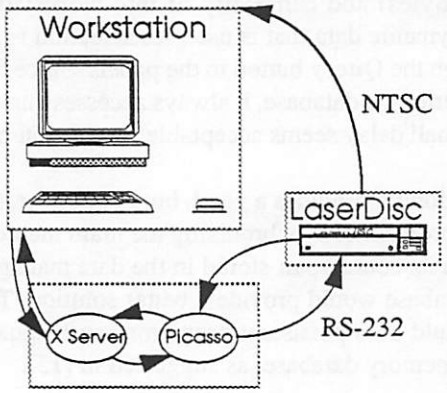

Figure 12: Hardware/software configuration for video extension

are identified and stored for later retrieval. The indexing mechanism consists of a hash table that maps a Lisp expression to a structure giving the first frame number of the segment, the last frame number of the segment, and the state of the audio and video channels. To create the index, we wrote a custom tool in PICASSO.

We modified the user interface to CIMTOOL in three ways to demonstrate this new technology. First, we used a hierarchical browser to provide an interface to the video segments describing the Microlab. This example used monaural video (i.e., one audio channel with video). The data was indexed in a hierarchy, describing the chapters in the overview (e.g., "History", "Equipment", "Lab Control", etc.), and the segments within the chapters (e.g., "Chemical Etching Equipment"). The user can browse the available segments and view them through the **Introduction Dialog** shown in Figure 13.

The second way in which the user interface to CIMTOOL was modified was through the addition of a new item to the popup menu displayed by the **Detail...** button in the **Equipment Panel**. The menu operation **Pictures...** displays still images of the selected piece of equipment so the user can see what a particular piece of equipment looks like. The user interface allows the user to select the next and previous views through the **View Equipment** dialog box shown in Figure 14. This example used single frame video data (i.e., no audio).

Additional operations could be added to this menu to allow the user to access video data describing, for example, disassembly procedures and the theory of operation for the selected piece of equipment. As the volume of video data grows, users will naturally want to browse the available data in an organized, but non-linear fashion. The hierarchical browser is one possible interface to address this problem. Another solution is to use the PICASSO hypermedia system, HIP [2].

The final way in which we modified the CIMTOOL user interface was to add an audio help system. The interface consists of pressing the "Help" key on the keyboard while the mouse is in a button, menu or other interface widget for which the user wants help. Each interface object has an associated audio help track on the disc that is played when help is requested. This example used audio only data.

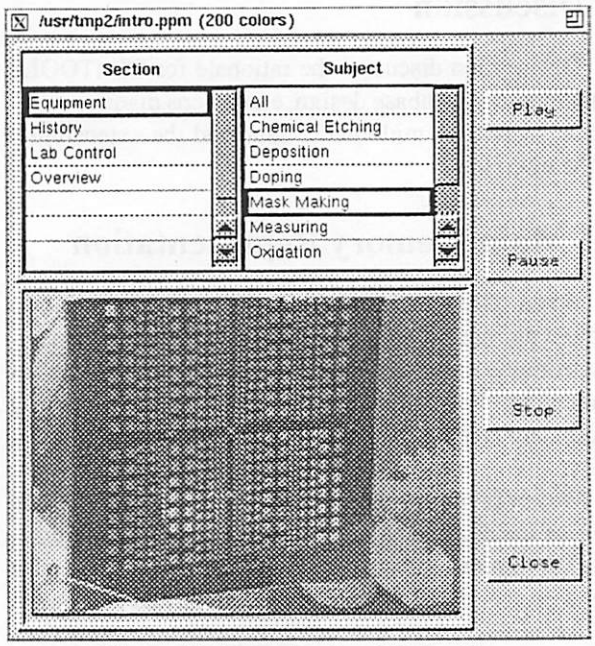

Figure 13: **Introduction** Dialog Box

Assuming that many people are using CIMTOOL or other multimedia PICASSO applications at the same time, a storage and distribution system is required for video data. One solution is to provide laserdisc players at each workstation. There are several drawbacks to this solution:

(1) The data on analog laserdiscs is write once, and production of the discs is an expensive and time consuming process unless a large number of discs are pressed (e.g., more

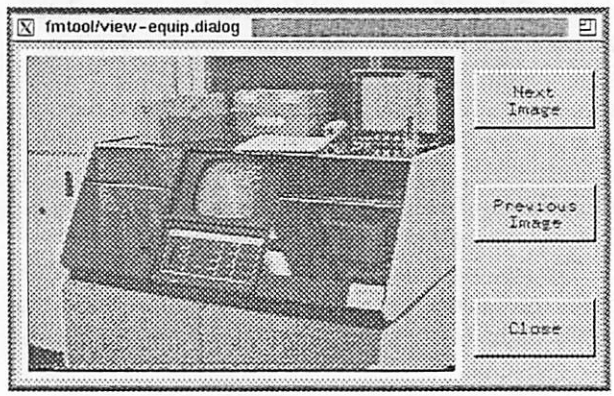

Figure 14: **View Equipment** Dialog Box

than 500).

(2) Updating the data would be expensive, since a new set of discs would have to be produced for each update.

(3) Each disc holds only 30 minutes of data, which is too little for all but the most modest applications. Using more than one disc requires the user to unload and load different discs, similar to floppy discs on a PC.

The best solution is to provide a large, shared, video database, accessible over the same network as the workstation, that stored compressed digital video data. Such a database would have to allow for large amounts of storage and concurrent, real-time access. We are currently developing a prototype video database server.

4.3 Extensibility

One of the best features of the CIMTOOL interface is that it is extensible. When a new entity type is added to the database, three steps are needed to incorporate the new entity into the browser:

(1) A new panel, similar to the **Equipment Panel**, is constructed to display the entity. This includes deciding on the hierarchical ordering in the hierarchical browsers.

(2) A new dialog box, similar to the **Select Utility...** dialog box, is added to support the relationships between the new entity and current entities. The dialog contains radio buttons corresponding to each relationship.

(3) Some of the existing **Select...** dialog boxes must be modified to reflect the new relationships added by the entity.

For example, suppose we wanted to track personnel in the facility. An entity of the new "personnel" type might have the following attributes: name, social security number, status (student or staff), picture, research group name, office location, and a list of equipment they are authorized to operate.

To add this entity to CIMTOOL, we would construct a **Personnel Panel** and a **Select Personnel...** dialog box. The **Personnel Panel** would have two browsers, for "SELECTED" and "UNSELECTED" personnel. The browsers might be ordered by research group and name.

The personnel entity type has a many/many relationship with both the equipment and space entities. There would be two corresponding radio buttons in the **Select Personnel...** dialog box, one entitled "Select Personnel Authorized To Use Selected Equipment" and the other entitled "Select Personnel With Offices Inside Selected Spaces".

Finally, the **Select Equipment...** and **Select Space...** dialog boxes would need to be modified. A radio button entitled "Select Equipment Usable By Selected Personnel" would be added to the **Select Equipment...** dialog box and a radio button entitled "Select Spaces Containing Offices Of Selected Personnel" would be added to the **Select Space...** dialog box. Users would be able to use the modified application as soon as they understood the meanings of the entities and the relationships, since the query construction paradigm would be unchanged.

The ideas contained in the CIMTOOL user interface can be applied to a wide variety of database application, since the components of the interface can be obtained from the entity-relationship diagram using the following mapping:

(1) For each entity, a panel such as the **Equipment Panel** and a **Select...** dialog box such as the **Select Equipment...** dialog box is created. If the entity has graphic attributes, it should appear in the main window as well.

(2) For each relationship pertaining to an entity, a radio button is added to the **Select...** dialog box for that entity.

(3) For entities with many attributes that could be used for selection, a dialog box such as the **Query Lot** dialog box is created. The dialog box should contain fields for all interesting attributes that can be queried.

Reusing the paradigm in different databases would have many advantages. Experience indicates that implementation of new applications would consist of little more than a straightforward surgery of an existing application. By the same argument, existing applications can be easily extended. Alternatively, an application generator could be built to create the applications from the ER diagram. Users would require little or no training to use the new application, since the query construction paradigm and its associated commands would be unchanged. Finally, changes to the underlying database schema would be transparent to the end users, provided the entity-relationship subset used by the application remained unchanged[4].

5. Conclusions

An application specific ad hoc query interface allows end users who are unfamiliar with the design of a particular database to express a useful subclass of queries. The CIMTOOL application illustrates a query specification

[4] Of course, changes in the underlying schema would require some minor mechanical modifications to the application.

model and interface metaphor for one such query interface. The same ideas could be used in other databases and applications. We believe that an application generator could be created to generate applications like CIMTOOL. Such a generator would be similar to the Query-By-Forms application generator Application-By-Forms [11].

Acknowledgments

We want to thank James Hopkins who measured the Microlab by hand and painstakingly created the geometric database used by CIMTOOL. We also want to thank Joe Konstan, Chung Liu, and Steve Seitz who also worked on PICASSO for adding and modifying the abstractions needed to create CIMTOOL.

References

[1] P. S. Barth, "An Object-Oriented Approach to Graphical Interfaces", *ACM Transactions on Graphics*, Vol. 5, No. 2, April 1986, pages 142-172.

[2] B. H. Becker and L. A. Rowe, "HIP: A Hypermedia Extension of the PICASSO Application Framework", to appear in *Proc. NIST Advanced Information Interfaces: Making Data Accessible 1991*.

[3] J. E. Bell and L. A. Rowe, "Case Study of Ad Hoc Query Interfaces to Databases", *Electronics Research Lab. Report M90/78*, SEPT 1990.

[4] Data Access Toolset Manual (1990). Metaphor Computer Systems.

[5] S. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1988.

[6] J. H. Maloney, et. al., "Constraint Technology for User Interface Construction in Thinglab II", *Proc. OOPSLA '89*, New Orleans, LA. October, 1989.

[7] Microfabrication at UC Berkeley (video tape), Electronics REsearch Laboratory, U.C. Berkeley, 1989.

[8] B. A. Myers, et. al. "Comprehensive Support for Graphical, Highly Interactive User Interfaces: The Garnet User Interface Development Environment", *IEEE Computer*, November, 1990.

[9] Natural Language™ Database Retrieval System User Manual, Natural Language Incorporated, Berkeley, CA. 1988.

[10] T.G. Rogers, R. G. G. Cattell, "Entity-Relationship Database User Interfaces", *Readings in Database Systems*, Morgan Kaufmann, San Mateo, CA, pages 359-368.

[11] L. A. Rowe, ""Fill-in-the-Form" Programming", *Proc. 11th Int. Conf. on Very Large Databases*, Stockholm, August, 1985.

[12] L. A. Rowe and C. B. Williams, "An Object-Oriented Database Design for Integrated Circuit Fabrication", *Proc. 1st Int. Conf. on Data and Knowledge Systems for Eng. and Manuf.*, Hartford, CT, Nov 1987.

[13] L. A. Rowe, "Report on the 1989 Software CAD Databases Workshop", *Proc. 11th IFIP Congress*, August 1989.

[14] L. Rowe, J. Konstan, B. Smith, S. Seitz and C. Liu, "The PICASSO Application Framework", *Submitted to UIST '91*. Also available from *Electronics Research Lab. Memorandum M90/18*, Computer Science Division - EECS, U.C. Berkeley, March 1990.

[15] W. R. Rubenstein, R. G. G. Cattell, "Benchmarking Simple Database Operations", *Proc. ACM SIGMOD 1987*, pp 387-394

[16] P. Schank, L. Rowe, J. Konstan, C. Liu, S. Seitz and B. Smith, "PICASSO Reference Manual", *Electronics Research Lab. Memorandum M90/79*, Computer Science Division - EECS, U.C. Berkeley, May 1990.

[17] R. W. Scheifler and J. Gettys, "The X Window System", *ACM Trans. on Graphics* Vol. 5, No. 2 (Apr. 1986).

[18] G. L. Steele, *Common Lisp - The Language*, Digital Press, 1984.

[19] SunSimplify™ 2.0 Reference Manual, Sun Microsystems, Mountain View, CA. 1989.

[20] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, Inc., Rockville, Maryland, 1982.

[21] K. Voros and P. K. Ko, "Evolution of the Microfabrication Facility at Berkeley", *Electronics Research Lab. Memorandum M89/109*, SEPT 1989.

[22] VQL, (1989). Sybase, Inc.

[23] M. M. Zloof, "Query by Example", *National Computer Conference*, 1975, Vol. 44, pages 431-438.