FAULTS: AN EQUIPMENT MAINTENANCE AND REPAIR TRACKING SYSTEM USING A RELATIONAL DATABASE

by

David C. Mudie

Memorandum No. UCB/ERL M91/44

23 May 1991

FAULTS: AN EQUIPMENT MAINTENANCE AND REPAIR TRACKING SYSTEM USING A RELATIONAL DATABASE

by

David C. Mudie

Memorandum No. UCB/ERL M91/44

23 May 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

FAULTS: AN EQUIPMENT MAINTENANCE AND REPAIR TRACKING SYSTEM USING A RELATIONAL DATABASE

by

David C. Mudie

Memorandum No. UCB/ERL M91/44

23 May 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

FAULTS: An Equipment Maintenance and Repair Tracking System Using a Relational Database†

David C. Mudie

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

FAULTS combines a forms-based user interface with a facility-wide relational database to record equipment maintenance and repair events as they occur. The semantics of preventive maintenance and repair events are formalized to create clear and unambiguous maintenance reports.

Storing preventive maintenance and equipment failure information in a structured database has several benefits. Accumulated information is automatically indexed to aid diagnosis of failures as they occur. Equipment failure information is available to utility programs for display and statistical analysis to produce summaries such as preventive maintenance intervals, mean time between failures, predicted downtimes, performance trends, etc.

FAULTS is installed in the U.C. Berkeley Microfabrication Facility where it is used to manage more than a hundred pieces of equipment. The **FAULTS** system has provided a significant improvement in the management of preventive maintenance and equipment repair information.

1. Introduction

1.1. What is the problem?

The current practice of equipment maintenance in the semiconductor industry consists of three activities: *equipment qualification*, *preventive maintenance*, and *field repair*. Equipment qualification includes a number of tests that must be completed before the equipment is released to production. Preventive maintenance is conducted at regular intervals and consists of routine inspection and cleaning. Unscheduled field repairs are often necessary during production to resolve equipment malfunctions.

Today, most of the information concerning *equipment qualification* and *preventive maintenance* is collected on paper records or in unstructured text files on a computer. Operators or technicians must refer to the appropriate records while performing maintenance. Information concerning *field repairs*, however, is usually managed in an ad-hoc fashion. Malfunctions are reported orally or recorded in unstructured text reports by the operators. Even when using a computerized paperless system such as COMETS [1] or the Berkeley Laboratory Information

† This research was supported by the National Science Foundation (Grant MIP-9014940) and The Semiconductor Research Corporation, Philips/Signetics Corporation, Harris Corporation, Texas Instruments, National Semiconductor, Intel Corporation, Rockwell International, and Siemens Corporation with a matching grant from the State of California's MICRO program.

System [2], the causes and actions taken to correct malfunctions are lost in long unstructured text files. Since this information has no formal structure, it is difficult to perform even simple analysis to discover and correct common causes of malfunction.

1.2. What is the solution?

This paper presents a new scheme to manage information concerning equipment maintenance and repair activities. The system developed to implement this scheme was designed with four goals in mind. First, the system must help facility management keep track of equipment maintenance events as they occur. Second, the system should help maintenance technicians conduct off-line equipment maintenance by reviewing previous maintenance cases. Third, the system must be usable and maintainable by equipment operators, maintenance technicians and facility managers without need of assistance from expert programmers. Finally, the system should cooperate with automated diagnostic systems by maintaining failure histories of equipment in the facility and by providing a mechanism that allows a diagnostic system to automatically schedule necessary maintenance tasks [3].

The new equipment maintenance system is known as the Berkeley **FAULTS** system. **FAULTS** is a program that allows a user to interact with a relational database [4] through a forms-based interface to record preventive maintenance and field repairs of manufacturing equipment. All information necessary to identify the type and cause of equipment failure, time of occurrence, time of repair, etc. is stored in the relational database. Unlike existing paperless systems, this information is organized in a structured representation so that other application programs can perform facility-wide analysis.

1.3. Related Work

The functionality of **FAULTS** is similar to that provided by Intel's CEPT system [5], a comprehensive Equipment Performance Management tool built on top of WORKSTREAM [6], a commercial CAM platform. The CEPT system coordinates WORKSTREAM's paperless WIP management system with its own database of equipment knowledge to support maintenance scheduling, analysis of equipment failures, and process monitoring for Statistical Process Control.

1.4. Outline of Paper

The remainder of this paper describes the design and implementation of **FAULTS**. Section 2 provides an example scenario demonstrating how the **FAULTS** system meets the first three goals identified above. Section 3 describes the design of the **FAULTS** software system, including the data structures used to model maintenance and repair activity, and briefly discusses how the implementation of these structures meets the fourth design goal of the system. Section 4 describes the success of **FAULTS** in a working environment, and identifies some possible extensions to the system. Section 5 concludes the paper with a discussion of results.

2. An Application Example

This section presents an example of how the **FAULTS** system is used during the lifetime of a typical equipment repair event. In the Berkeley Microfabrication Facility, the lifetime of a repair event (commonly referred to as a *problem*) is marked by three steps: *reporting, diagnosis,* and *clearing*. Additionally, once an event is recorded, it may be *analyzed* by a diagnostic program or report generator.

2.1. Problem Reporting

During an LPCVD (low pressure chemical vapor deposition) processing step, Bob (the operator) observes that the display terminal of the Tylan furnace controller ("TYCOM") on the LPCVD furnace does not respond to his commands. Bob tries the "reset" command he knows, but the TYCOM terminal still does not respond. Bob switches over to an ASCII computer terminal next to the malfunctioning controller, and invokes the **FAULTS** equipment maintenance system to report the problem.

Figure 1 shows the first screen of the **FAULTS** system. This screen is called the *Equipment Status Board.* Like other **FAULTS** screens, it is composed of an information display occupying most of the screen and a list of available operations on the bottom line. The Equipment Status Board lists problems reported for all equipment in the facility. The Status Board currently shows a problem on the LAM plasma etcher, but Bob's TYCOM problem has not yet been reported. Bob executes the "New" command to report a new problem.

Bob is prompted for the name of the equipment he is working on ("tylan16") and then is asked to describe the malfunction he is reporting. **FAULTS** presents a menu of problems known to occur on the LPCVD furnace tube, and Bob selects the entry most descriptive of his problem: a "terminal" problem within the group of "tycom" errors. Figure 2 shows the screen during this

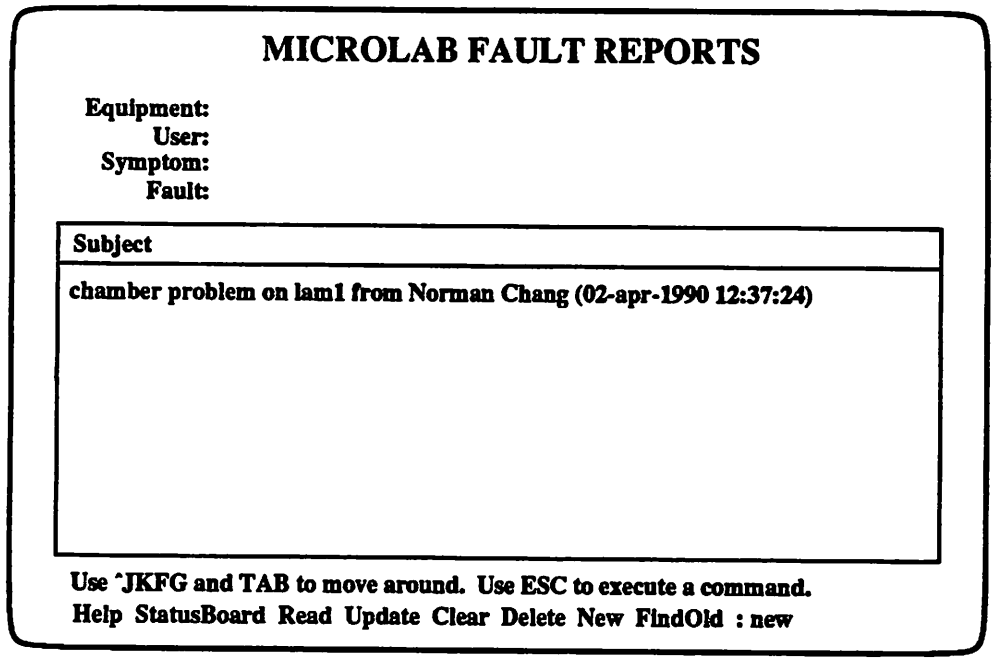

Fig. 1. The Equipment Status Board. Users can browse current problems, initiate a new report, or examine old reports.

selection process. Escape commands are available if none of the menu items seem correct or if Bob decides to cancel the report.

When Bob has identified the observed symptoms of his problem, the screen changes to that shown in Figure 3. This screen confirms the information Bob entered, such as the name of the malfunctioning equipment and the date the report is being filed. A short summary line is automatically built from known information to help technicians identify this report at a glance. Bob is prompted to type in free-form text comments so he can report on the particular circumstances of this failure or describe symptoms not found in the previous menus.

When Bob is done entering comments, the report is released as a "pending" problem. FAULTS adds the new report to the Equipment Status Board, and automatically notifies the responsible technician and supervisor via electronic mail. Bob can also send "carbon copies" of the report to himself or other operators of the equipment.

2.2. Problem Diagnosis

Kate, the technician recently assigned to tylan16, receives electronic mail describing Bob's problem on the machine. Kate is unfamiliar with the TYCOM control system, so she invokes the FAULTS system on her workstation and executes a command to find previous occurrences

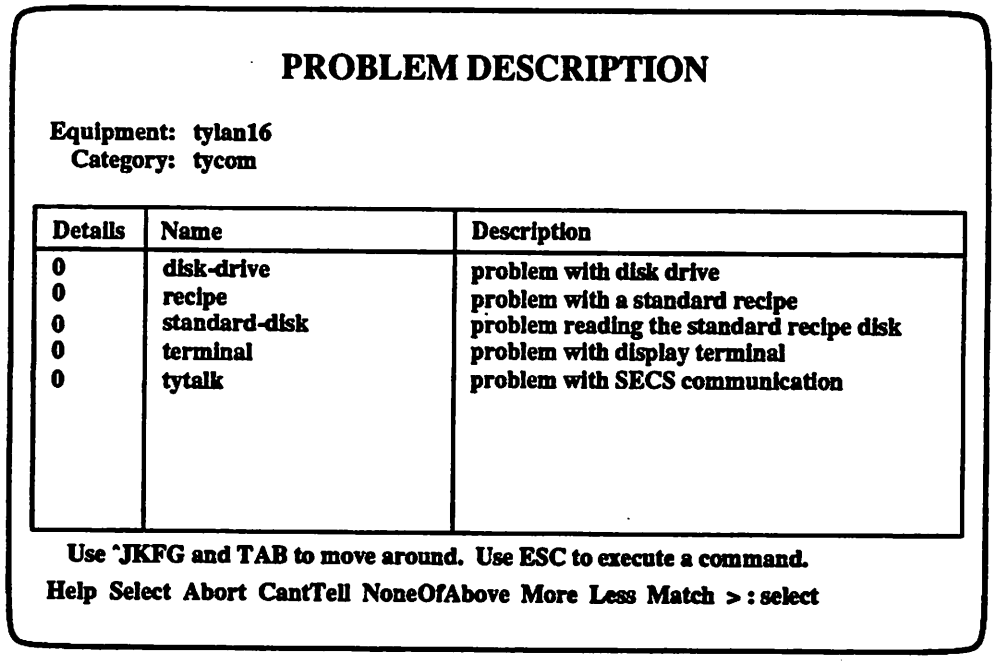

Fig. 2. Identifying problem symptoms. An operator observes a TYCOM malfunction (the control screen has no response) and selects the **terminal** symptom from the Tylan furnace menu.

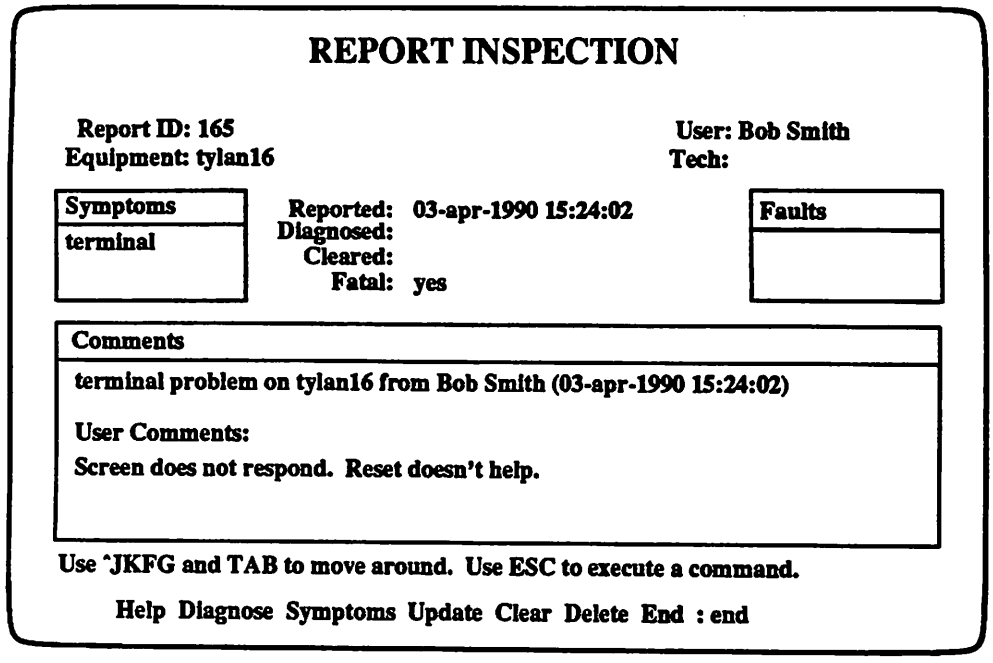

Fig. 3. Completing a problem report. The report can be checked and modified by the operator before it is released to the Status Board.

of similar failures. The system locates several reports that may be relevant, as shown in Figure 4.

Kate browses the contents of the old reports, as shown in Figure 5, and learns that on a previous occasion the TYCOM display screen had locked up because of a faulty printed circuit board in the controller. Armed with this knowledge, Kate inspects the furnace control system and discovers that one of the circuit boards is indeed defective. She orders a replacement board from the equipment manufacturer and is told that the new board will take four days to arrive.

Still within the **FAULTS** system, Kate records her *diagnosis* of the problem by choosing from a menu of faults known to occur on the Tylan furnace tubes, as Bob chose from a menu of symptoms. This process is shown in Figure 6. Kate also enters a brief note stating that the furnace will remain down for another four days while waiting for the replacement part. The revised report is posted to the Equipment Status Board so operators will be aware that the furnace is out of commission for the rest of the week.

2.3. Problem Clearing

A week later, the replacement board arrives. Kate fixes the controller and verifies that the furnace tube is operating normally. She enters the **FAULTS** system and executes a command to *clear* the problem report. **FAULTS** prompts Kate to confirm her diagnosis of the faulty controller board, then asks her to type in more text comments describing the repair procedure and

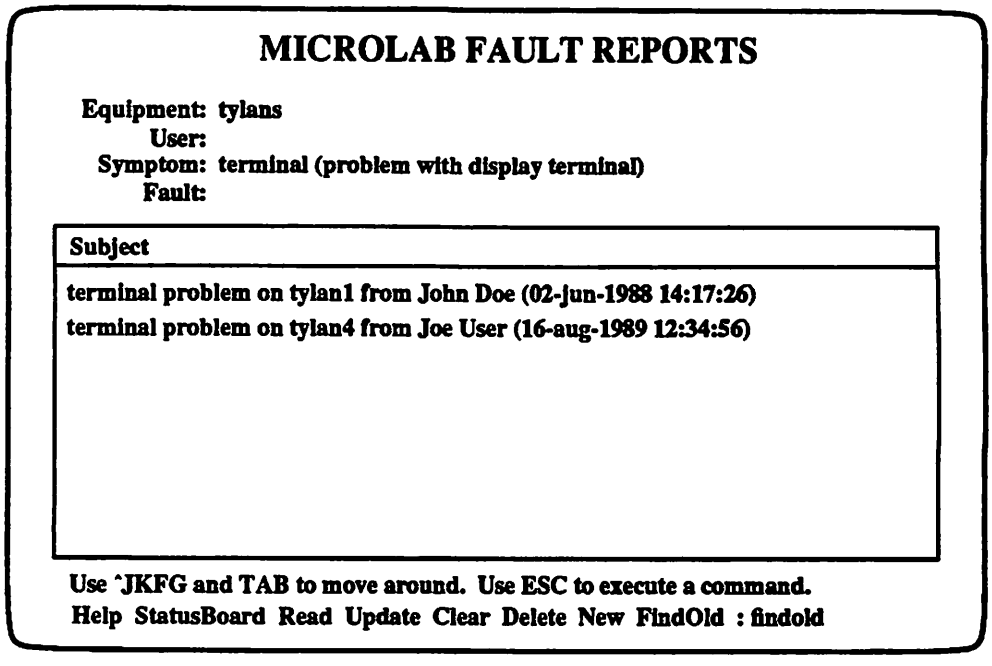

Fig. 4. Locating old reports. **FAULTS** queries the database for reports concerning the specified symptom.

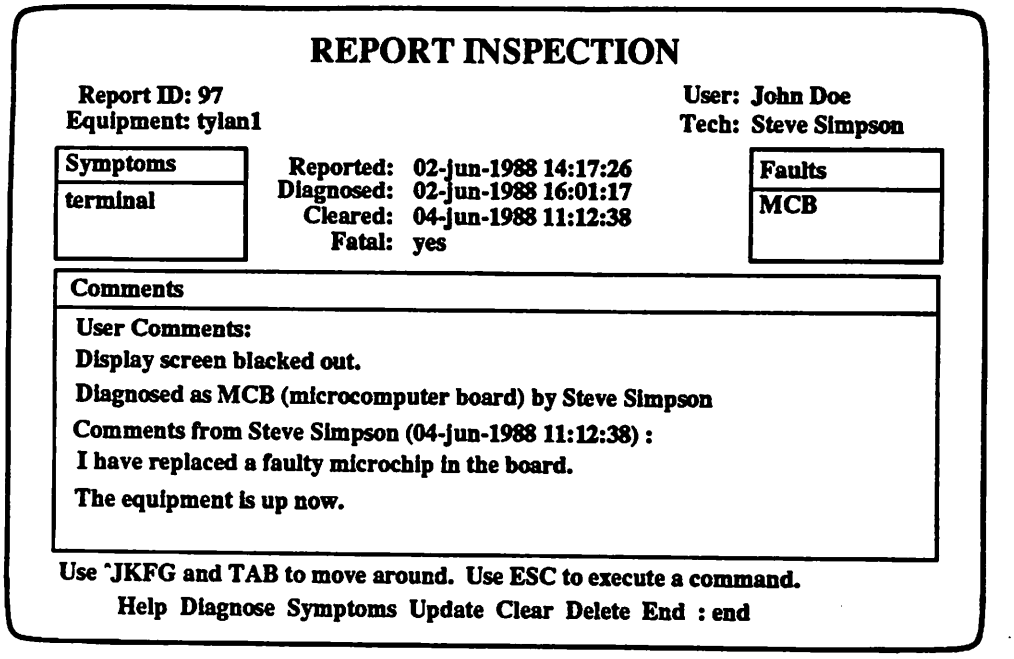

Fig. 5. Browsing a problem report. This screens displays all the information concerning a particular maintenance event. The text comments may be scrolled to read further.

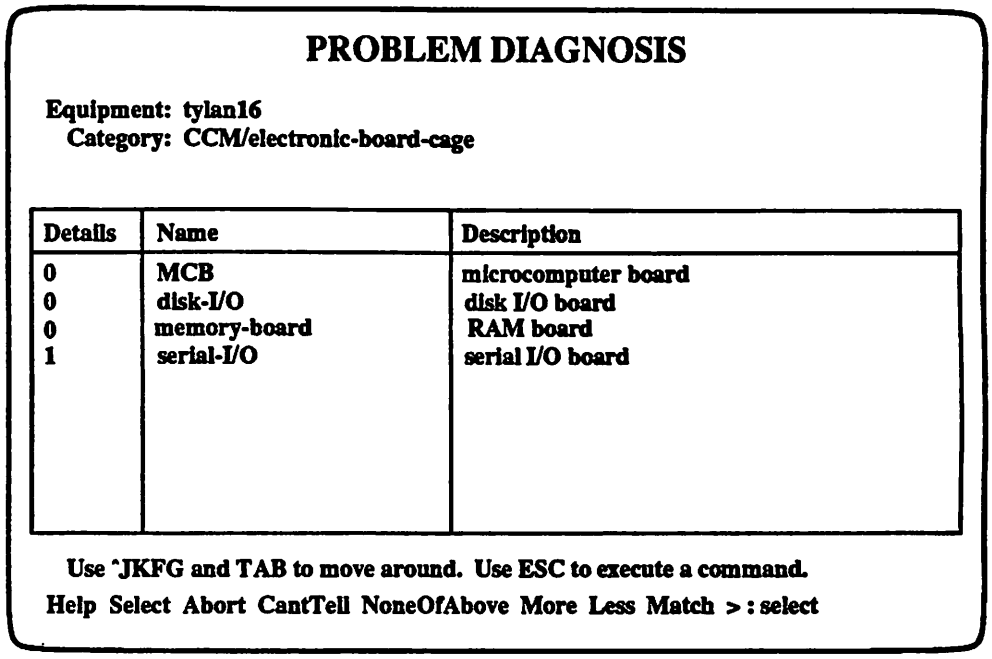

Fig. 6. Identifying equipment faults. **Electronic-board-cage** (defective electronic board cage) was selected under the **CCM** (central control module) category. Finally, the technician chooses **MCB** (defective microcomputer board) from the four choices under **electronic-board-cage.**

any peculiar circumstances of the repair. Kate notes that the replacement board took longer than promised to arrive. When her comments are complete, **FAULTS** removes the report from the Status Board and sends out electronic mail announcing that the furnace is back in operation.

This example demonstrates the day-to-day use of **FAULTS** to manage current equipment problems and to maintain a history of previous equipment failures. **FAULTS** also provides a set of command screens to update the contents of each fault and symptom menu. This allows qualified technicians to create new problem categories when the need arises, without support from a database expert. Each command screen in **FAULTS** has an accompanying ***Help*** screen describing the command options available. (As an example, the help screen for the Equipment Status Board is shown in Figure 7.) These on-line screens help novice users learn to operate the system without a bulky manual.

2.4. Failure Analysis and Downtime Statistics

Information accumulated in the **FAULTS** database is available to independent analysis programs as well as to human operators. One important function of these programs is to summarize equipment maintenance information. For example, facility managers often ask "How much equipment downtime occurred in the last three months?", "How much money did we spend on furnace maintenance last year?", and so on. Technicians frequently ask questions such as "What was the failure last time this symptom was observed?" or "How often does the TYCOM

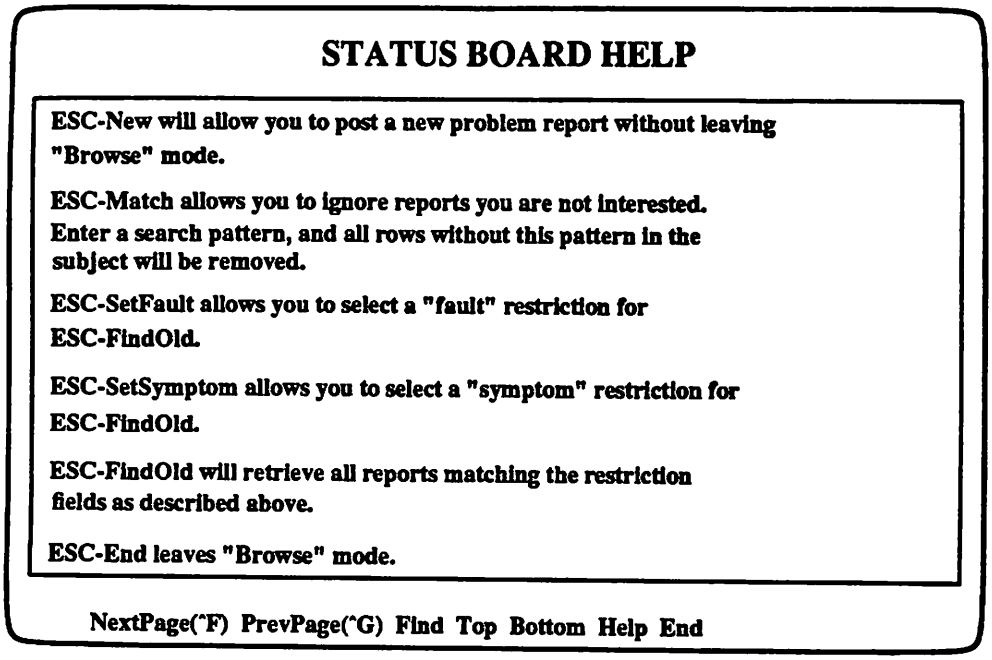

Fig. 7. Example of a **Help** screen. Such a screen can be called up from any of the **FAULTS** command screens.

controller fail on this furnace?" Since the information associated with each equipment maintenance event is stored in a well-defined relational format, analysis programs and report generators can query the **FAULTS** database to produce answers to these questions.

Simple plotting of various statistics such as equipment downtimes and failure frequencies (a "Pareto" chart) give concise visual summaries of the cleanroom operation. As an example, Figure 8 summarizes the failures that occurred on our Tylan furnaces during the last year.

3. Design and Implementation of FAULTS

This section discusses the design goals of **FAULTS**, then outlines the implementation that achieves these goals.

3.1. Design goals

The **FAULTS** system was initially conceived as a tool to gather data for automated equipment diagnosis, but early on it became apparent that the data collected would be valuable to a much wider set of applications. We identified four major design goals that must be met to to support a wide range of application programs while making the process of data collection easy for users. First, **FAULTS** should serve as a useful tool to track equipment failure and maintenance activities on the floor of the facility, so that managers and technicians can schedule repair activities more effectively. Second, to faciliate problem diagnosis, **FAULTS** should provide a

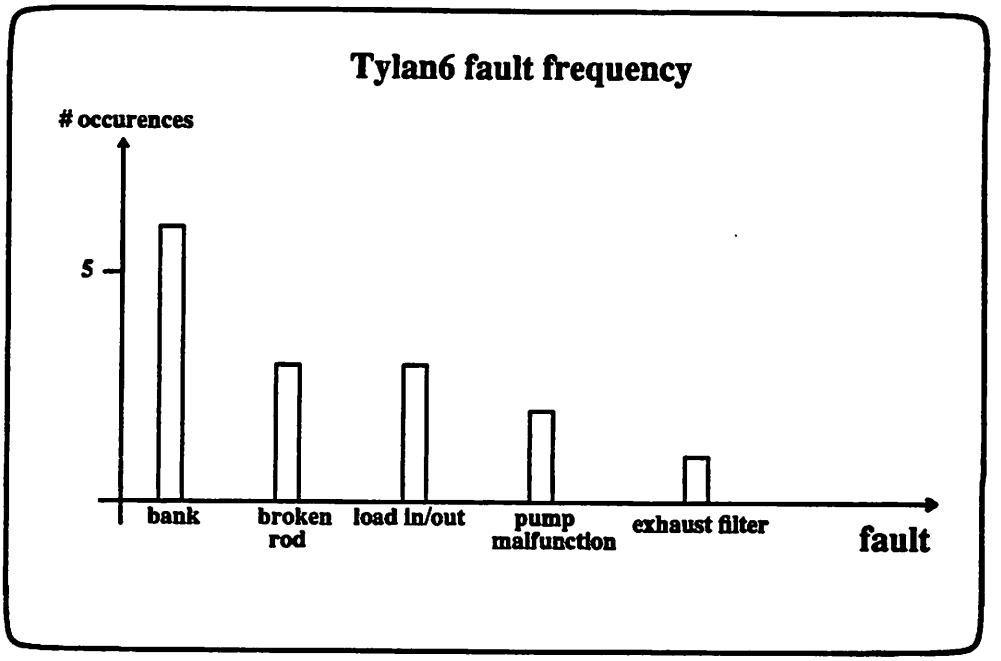

Fig. 8. A histogram showing fault frequencies on Tylan furnaces over the last year.

simple and efficient interface to allow retrieval and analysis of past equipment failures. Third, **FAULTS** must be easy to learn and easy to use, and the system must not require supervision from a database expert. This goal puts equipment experts in direct control of the equipment knowledge base and reduces demand on the computer support staff. Finally, **FAULTS** should provide a platform for development of other CIM packages that require access to the database of equipment history. A consistent application interface will allow many modules of an integrated CIM system to take advantage of a single data set built and maintained by **FAULTS.**

3.2. Overview of components

Figure 9 presents a simple diagram of the way data is shared between applications in the BLIS system, in which numerous programs access a single shared database. Use of a shared database eliminates the problems of maintaining integrity and synchronization between several copies of the same data. In addition, sharing makes data from different applications available in a central location for easy cross-referencing. While the database is presented as a logical whole, it can be physically distributed across a network of data servers for storage efficiency. Distributed database management tools make the details of this implementation transparent to application programs. [7]

Rather than "hardwiring" database query code directly into numerous application programs within the **FAULTS** system, database manipulation code is confined to a single library of reusable routines called **faultlib**. The **faultlib** package is used by several database applications

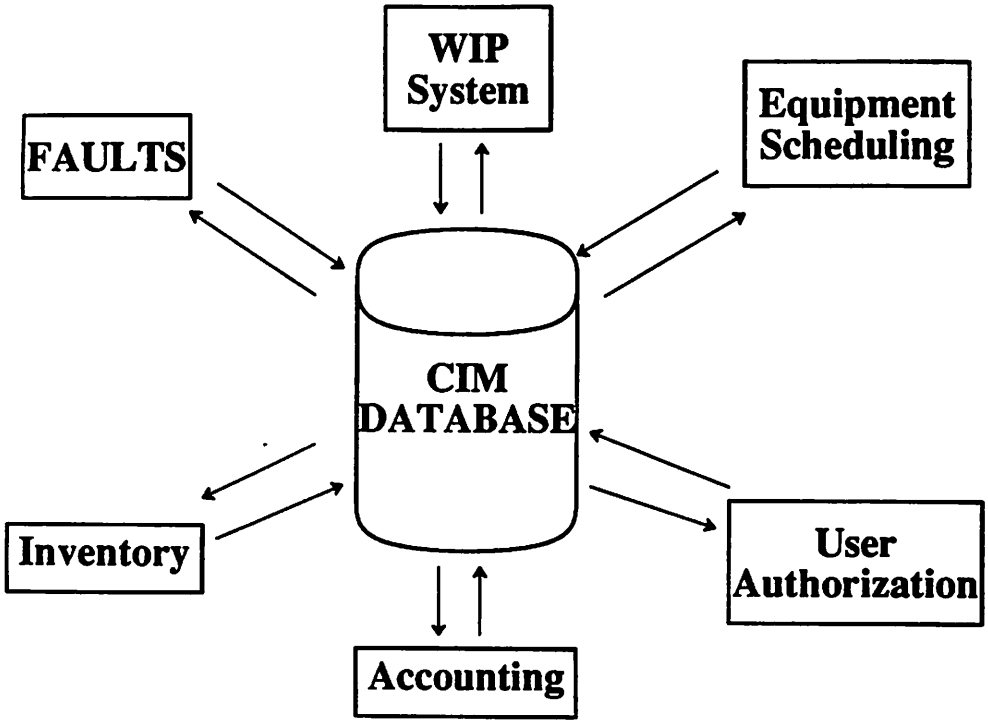

Fig. 9. BLIS database model.

(including the front-end user interface of **FAULTS** itself) to manipulate the **FAULTS** database without knowledge of the data model's relational implementation. Figure 10 diagrams this approach. For run-time performance efficiency, some **FAULTS** applications make special-purpose queries directly to the database.

3.3. The FAULTS Data Model

The "Report Inspection" screen (Figure 5 in the previous scenario) gives some indication of the amount of information associated with each equipment maintenance event that occurs in a microfabrication facility. Details such as the date of the problem occurrence, the operator and technician involved, and some description of the maintenance required must be captured and stored for future analysis. To support efficient storage and retrieval of this information, a record-keeping system must coerce the information surrounding each event into a clear and unambiguous format. We derive this format by defining the semantics of an equipment maintenance event and identifying the information that must be captured. In this section, we describe the data abstractions used by **FAULTS** to represent an equipment maintenance event.

For simplicity of design and management, an object-oriented data model was used to build the **FAULTS** database. The data model consists of five major data types and the relationships between objects of these types. These five data types are: *labusers, resources, comments,*

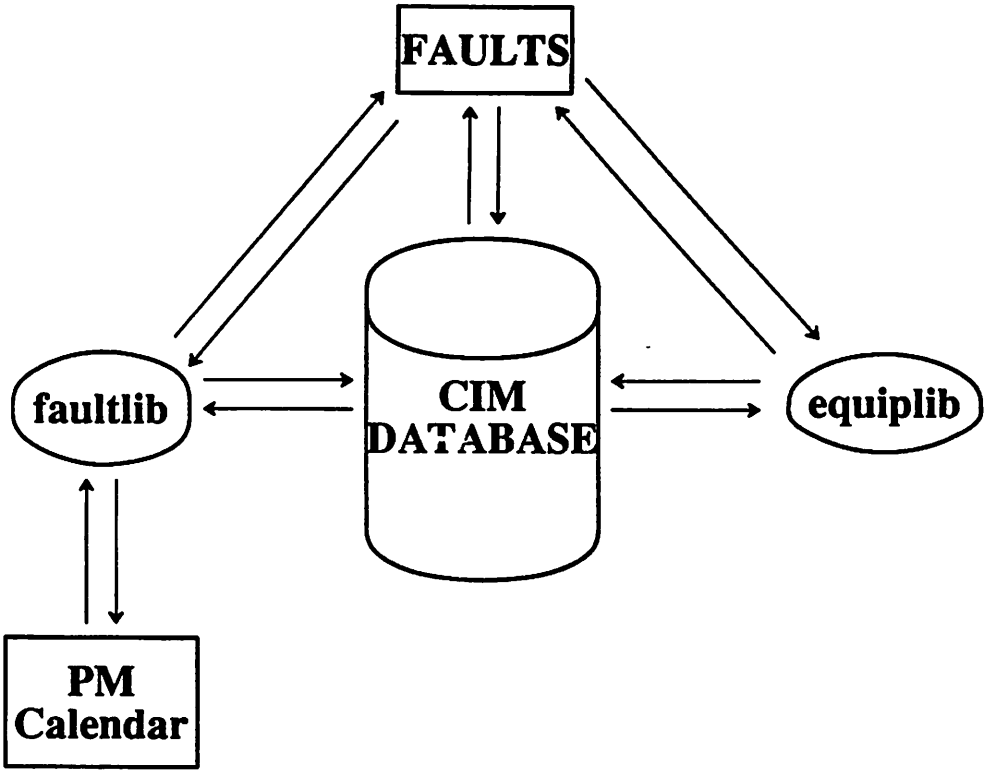

Fig. 10. Structural components of **FAULTS**

reports, and *faultsymps*. The first four items are straight-forward representations of the information associated with real-world objects. The fifth data type, *faultsymps*, models diverse equipment failure modes. *faultsymps* have no real-world counterpart. The *labusers* and *resources* data types contain many attributes used by accounting and control programs but not by **FAULTS**. These unused attributes are omitted from this description of the **FAULTS** data model for clarity. In this section, we present each of the five data types in turn and discuss the relationships that exist between them. Figure 11 shows an Entity–Relationship diagram of the **FAULTS** data model.

Labusers

A *labuser* is a person involved with the Berkeley Microlab. Administrative staff, technicians, and equipment operators are all *labusers*. Two attributes are stored about each labuser:

- *name*: First and last name of the user (e.g., "Bob Smith").
- *labuser ID*: A unique number to identify this labuser.

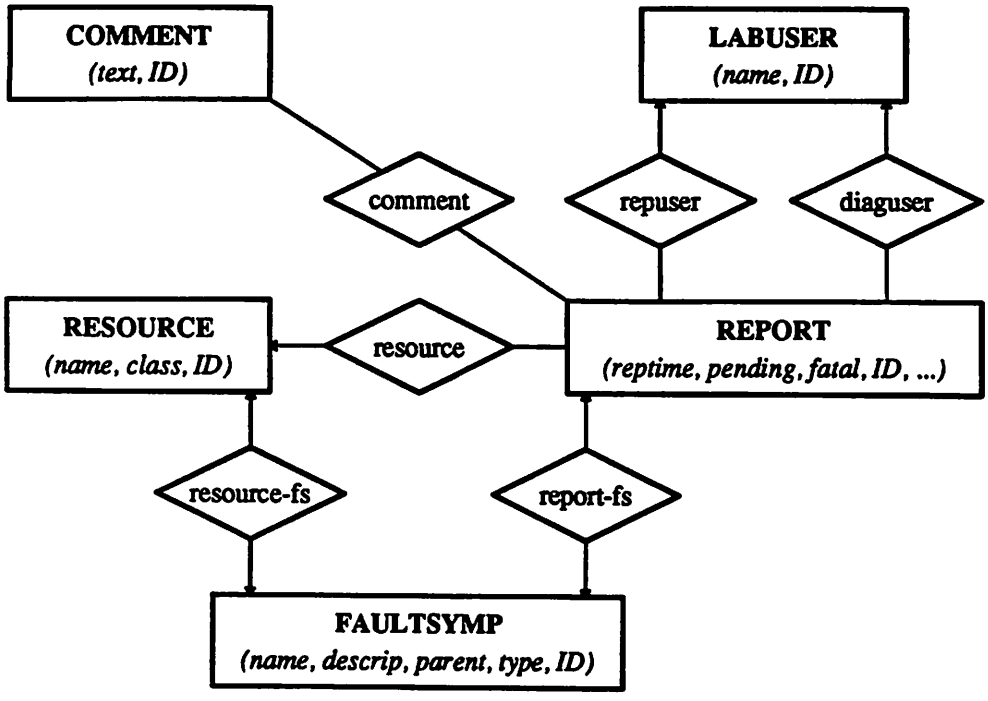

Fig. 11. Entity–Relationship Diagram of the **FAULTS** database.

Boxes indicate database objects
Italics indicate attributes of an object
Diamonds indicate relationships between objects
An arrowhead follows a "many-to-one" relationship

Resources

A *resource* is an item that is used in the microfabrication process. In general, *resources* include chemicals stocked by the facility, tweezers used to handle wafers, and many other items. **FAULTS** is only concerned with resources classified as *equipment*. Three attributes are stored about each resource:

- *name*: A brief name identifying the resource (e.g., "tylan16").
- *class*: The general group this resource belongs to (e.g., "equipment").
- *resource ID*: A unique number to identify this resource.

Comments

A *comment* is a block of ASCII text of arbitrary length. *Comments* are used to store unformatted text entries for later retrieval. Two attributes are stored about each comment:

- *text*: The contents of this text block.

- *comment ID*: A unique number to identify this comment.

Reports

Each *report* represents a single equipment maintenance event. A *report* object stores all the data unique to this event, such as the date of occurrence and current status. The report object also makes reference to the *labusers* and *resources* that may be associated with more than one such event. The following attributes are stored about each report:

- *reptime*: The date this event was entered into the system.
- *diagtime*: The date this event was diagnosed by a technician.
- *cleartime*: The date this event was cleared from the Status Board.
- *pending*: True if this event is still on the Status Board.
- *fatal*: True if this event prohibits operation of the equipment until cleared.
- *repuser*: A reference to the *labuser* who entered this event.
- *diaguser*: A reference to the *labuser* who diagnosed this event.
- *resource*: A reference to the *resource* this event occurred on.
- *comment*: A reference to the *comment* storing text for this event.
- *report ID*: A unique number to identify this report.

Faultsymps

To cross-reference similarities between equipment maintenance events, **FAULTS** requires a clear and unambiguous method to describe the nature of each such event. We use a hierarchical structure of *fault* and *symptom* categories, with an entry for each situation that can occur.

The qualitative information that describes a maintenance event is divided into two conceptual groups: *faults* and *symptoms*. A *symptom* is defined as an occurrence that may be observed by an operator or diagnosis module when an equipment malfunctions. An operator might observe the *symptom* that the motorized wafer boat has gotten stuck during a furnace run. A *fault* is defined as the underlying cause behind an equipment malfunction. *Faults* are first identified by a maintenance technician's diagnosis, then verified by the person who actually repairs the equipment. As an example, the "wafer boat stuck" symptom might have been caused by a "boat controller" fault.

To avoid ambiguity of terms, operators and technicians must agree upon a standard terminology for each of the faults and symptoms that can occur on equipment. **FAULTS** enforces this standard by identifying each fault or symptom with a single descriptive keyword. Users then select from a menu of known keywords, rather than describing the problem in their own language. Each keyword is paired with a long description of the fault or symptom to aid user recognition.

The faults and symptoms for a specific piece of equipment are logically grouped into hierarchical categories to simplify the search for a particular entry. In order to avoid duplication of information, hierarchies can be extended upward to include clusters of similar equipment. These hierarchical categories are developed by consulting with an expert technician for each piece of equipment. A technician with sufficient privileges can modify the hierarchy of categories without leaving **FAULTS** if he discovers a problem category that was not foreseen by the expert. An example of these hierarchical structures is shown in Figure 12.

When specialized equipment of the same type are grouped together in a cluster, the corresponding specialized faults and symptoms are earmarked for that particular equipment

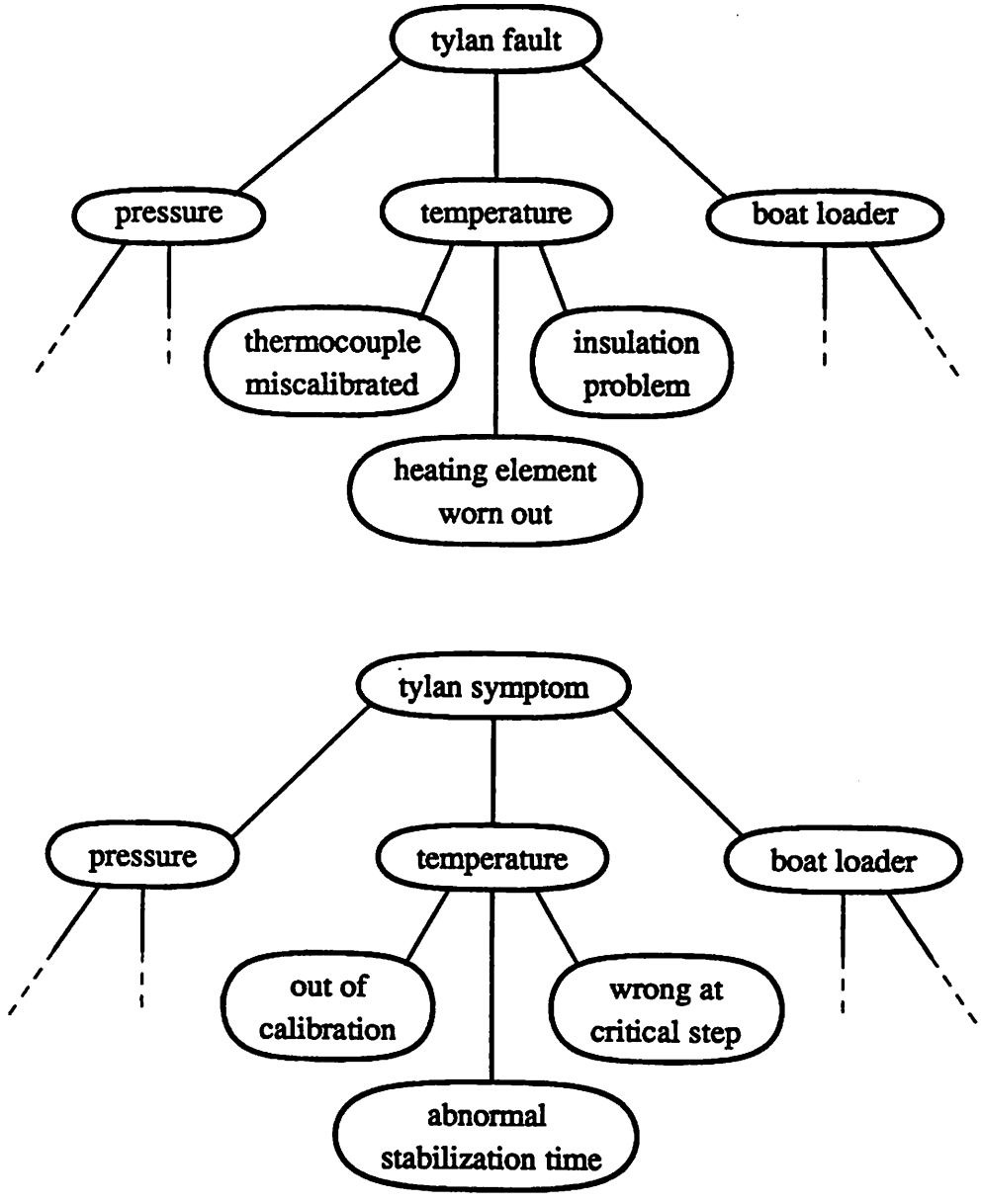

Fig. 12. An example of *fault* and *symptom* hierarchies.

while others remain generic throughout the cluster. As an example, hierarchical categories are created for a furnace cluster that contains 16 individual reactors. Among these, there are 10 different kinds of reactors that use different sets of gases over various temperature and pressure ranges. **FAULTS** will only display the fault and symptom entries appropriate to the reactor in use.

Faults and symptoms share many of the same properties, so they are abstracted into a single data type called a *faultsymp*. The following attributes are stored about a *faultsymp*:

- *name*: The keyword name of this faultsymp.
- *description*: The long description of this faultsymp.
- *type*: The type of this faultsymp, either "fault" or "symptom".
- *parent*: A reference to this faultsymp's superior in the hierarchy of categories.
- *faultsymp ID*: A unique number to identify this faultsymp.

Additionally, **FAULTS** recognizes a many-to-many relationship *resource-fs* that associates specialized *faultsymps* with the *equipment resources* on which they occur, and a many-to-many relationship *report-fs* that associates *faultsymps* with the *reports* they describe. While the data model can associate many symptoms and many faults with each problem report, the **FAULTS** user interface currently enforces a *single-fault assumption*: at most one fault can be associated with each report. This restriction simplifies the data set for analysis by automated diagnostic agents. If symptoms from several faults are reported together, a command is available to split the symptoms between two or more reports, each with a single fault.

3.4. The FAULTS Procedure Library

The data types defined above are stored in a commercial relational database (INGRES [8]). This database contains a table to represent each of the five data types, along with numerous index and cross–reference tables to define how data objects are associated with each other. To insulate programs from the details of the database implementation, we created a library of interface procedures called **faultlib**. This library is used to achieve **FAULTS'** fourth major goal of cooperation with other programs. This section describes the functional interface provided by the **faultlib** library.

The **faultlib** library may be linked with ESQL/C application programs. Table 1 summarizes the functions provided; use of these functions is detailed in Appendix A. **Faultlib** consists of approximately 2800 lines of **C** and **SQL**.

3.5. User interface

The user interface to **FAULTS** is implemented in ABF [9]. This interface is a collection of command screens, or *frames*, each with a form to display some data and a menu of operations available. The form in each frame gathers information from the user and displays the result of database queries. The menu of operations allows the user to invoke **faultlib** procedures on appropriate data objects. For example, to clear a report on equipment "tylan16," the user enters the "Equipment Status Board" screen and moves the cursor onto the problem summary line for tylan16. She then executes the "Clear" operation to remove the problem from the Status Board. If the report has been properly diagnosed, the **FAULTS** user interface will call the `unpend()` operation in **faultlib** to remove it from the list of pending problems. Each frame has an accompanying "Help" frame to describe the operations available, and an "End" or "Abort" operation to leave the screen. Frames can be nested; in other words, executing an operation in one frame may lead you into another to gather the appropriate data.

The user interface to **FAULTS** currently is composed of twelve frames and ten high-level procedures that group together common sequences of **faultlib** operations. The interface code

Managing faultsymps	
`newfs`	Create a new entry in fs tree
`delfs`	Delete fs from tree
`delhiddenname`	Unhide fs on named equipment or group
`addhiddenname`	Hide fs on named equipment or group
`faultroot`	Locate root of fs tree for named equipment
`fspathname`	Write path name for fs into path
`loadfstree`	Load fs & descendants into core
`walkfstree`	Traverse a tree of fs loaded by loadfstree
Managing reports	
`newreport`	Create a new report
`delreport`	Delete report and associations
`clonereport`	Duplicate a report and associations
`assocfs`	Associate fs with report
`unassocfs`	Delete any association of fs with report
`unassocall`	Delete all associations with this report
`makesubject`	Generate a subject line for specified report
`pendreport`	Mark report as "pending"
`unpendreport`	Mark report as "cleared"
`fatalize`	Mark report as "fatal"
`unfatalize`	Mark report as "non-fatal"
Managing comments	
`newcomment`	Create a new comment
`delcomment`	Delete a comment
`clonecomment`	Duplicate a comment
`setsubject`	Set the first line of a comment
`opencomment`	Open a comment for appending
`appendline`	Append text to open comment
`closecomment`	Closes open comment
Managing equipment status	
`writereport`	Write a problem report to concerned parties
`writestatus`	Update status board for named equipment

Table 1. Functions provided in **faultlib.**

consists of approximately 2200 lines of **OSL**, the fourth-generation language used with ABF, and 700 lines of **C** and **SQL** procedures.

3.6. Data Analysis Applications

Diagnostic systems use **faultlib** to initiate problem reports and query equipment history without specific knowledge of the database structures. A prototype diagnostic system has been developed for the LAM plasma etcher, to monitor equipment performance via a SECS

communication link and notify technicians when anomalous behavior is observed. [10] [11] Other applications that use the **faultlib** library include "pmstat," an automatic maintenance calendar to report "maintenance due" problems at regular intervals, and "faultrpt," a report generator to compile statistics of fault frequency, equipment uptime, and mean time to failure, as defined by the SEMI Guideline on Equipment Reliability [12]. (As an example, Figure 8 above shows the frequency of faults occuring on a Tylan furnance tube.) These **faultlib** applications range in size from less than one hundred to over one thousand lines of C and SQL code.

4. Experience and Future Extensions

This section discusses our experience running **FAULTS** in a working fabrication facility and presents some ideas for future development.

4.1. User Feedback

When **FAULTS** was first installed in the U.C. Berkeley MicroLab, it met with mixed reactions from users and staff. "It's too slow" was the most common complaint, followed closely by "it's too cumbersome to use." During the first few months of use, we improved the performance of the system considerably by modifying the data model implementation to more efficiently handle a few common queries. We gained further improvement by storing free-form text blocks directly in the UNIX filesystem rather than in a relational data table. This change reduced contention for the database and allowed the text to be retrieved more efficiently.

While users were pleased with how quickly they were able to learn to use **FAULTS**, several of the equipment operators felt that it was too much work to fill out the structured reports, and they would have preferred to retain the free-form input style used by the previous system. This feeling is exacerbated by the fact that these operators receive little direct benefit from the extra work of using **FAULTS**. The big payoff is for equipment technicians and lab managers, who can predict downtimes more accurately and search the equipment history more efficiently. Previously, technicians had to check the Equipment Status Board, a Preventive Maintenance Status Board, and their own electronic mailboxes to check on the condition of equipment in the facility. **FAULTS** improved report management and integration with the maintenance calendar allows the condition of all equipment in the facility to be summarized in a single unified Status Board.

4.2. Data Collected

During six months of operation, **FAULTS** has collected 1000 reports describing 500 kinds of failures on 100 pieces of equipment. Since only five problem types per equipment have occurred in this period, the equipment knowledge base is still rather sketchy. To make **FAULTS** immediately useful for application prototyping and development of diagnostic tools, we "seeded" the knowledge base with a full set of reports describing the average failures of a LAM plasma etcher over a ten year period. This false history is used as a test-bed for **FAULTS** application programs, which will be able to work accurately on other equipment as the database history grows.

4.3. Managing the Knowledge Base

For the most part, the data screens and command options provided by **FAULTS** are easily understood and readily used by staff members to create, edit, and destroy problem reports in the database. Unfortunately, managing the fault and symptom trees has proven a more difficult task.

While the command interface itself is no more difficult to use, the abstract data structure is less readily understood and the operations needed to edit the data are more complicated. Even when users have grasped the tree structure underlying the fault and symptom categories, it is not obvious when and how nodes or subtrees can be moved or deleted. We tried to compensate for these difficulties with extensive "Help" documentation and command procedures that do extensive error-checking before committing a change to the database. Most users are still hesitant to edit the fault and symptom trees before being tutored by an experienced user.

4.4. Extensions

Clearly, we need a graphical interface to the **FAULTS** system. As X11 display terminals become more common in the facility, we would like to extend the **FAULTS** user interface to take advantage of mouse input and bitmapped graphic capabilities. With a mouse, the keystrokes used for command selection and tedious cursor positioning could be replaced with simple point-and-click operations. Bitmapped graphics could be used to more effectively display the reports generated by **FAULTS**, and a graphic direct-manipulation interface to the faultsymp management routines should alleviate the comprehension difficulties discussed above.

We would also like to add more intelligence to **FAULTS'** report retrieval and problem diagnosis functions. We are currently designing a case-based matching algorithm that will allow **FAULTS** to automatically group similar reports together, and to calculate a measure of "closeness" between two reports for use during problem diagnosis.

5. Conclusions

The Berkeley **FAULTS** system is fully implemented and used by equipment operators, lab technicians, and administrative staff to record and analyze equipment maintenance activities efficiently and effectively. Application programs use the **FAULTS** database for preventive maintenance scheduling and automated malfunction diagnosis. **FAULTS** has been running as a standard component of the Berkeley facility management software for the last six months.

Acknowledgements

I would like to thank Norman Chang, Gary May, and and the staff of the U.C. Berkeley MicroLab for donating valuable knowledge, time, and patience to this project. I would also like to thank my research advisors Lawrence Rowe, Costas Spanos, and David Hodges for their support and guidance.

6. Appendix A: Faultlib Functions

The **faultlib** library may be linked into an application program to supply the following C functions:

Managing faultsymps

```
int newfs(int type, int parent, char *name, char *desc)
```
Create a new entry in fs tree.
Returns 0 on failure, new id on success.

```
int delfs(int fs)
```
Delete `fs` from tree. May leave dangling associations

or orphaned children.
Returns 0 on failure, `fs` on success.

`int delhiddenname(int fs, char *name)`
Unhide `fs` on named equipment or group. `Name` may be a regular expression to match several equipment at once.
Returns count of rows deleted.

`int addhiddenname(int fs, char *name)`
Hide `fs` on named equipment or group. `Name` may be a regular expression to match several equipment at once.
Returns count of rows added.

`int faultroot(char *eqname, char *type)`
Locate root of fault or symptom tree for named equipment. `Type` should be "fault" or "symptom".
Returns 0 on failure, id of root fs on success.

`int fspathname(int fs, char *path)`
Write path in fs tree from root to `fs` into `path`.
Returns 0 on failure, `fs` on success.

`int loadfstree(int fs, int resource)`
Load `fs` & descendants into core. If `resource` is non-zero, fs's hidden on that resource are ignored.
Returns count of fs loaded.

```
int walkfstree(int fs, int (*previsit)(int), (*invisit)(int),
    (*postvisit)(int))
```
A general purpose routine to traverse a tree of fs loaded by `loadfstree`, rooted at the `fs`. If non-NULL, callback functions `previsit()`, `invisit()`, and `postvisit()` are applied before visiting each node's children, after visiting each child, and after visiting all of a node's children. ID of the current fs node is passed as parameter to the callbacks. If a callback returns non-zero, `walkfstree()` aborts traversal and returns this status.
`walkfstree()` returns status code from callback, or 0 on successful traversal.

Managing reports

```
int newreport(int report, int resource, char *reptime, int repuser,
    int comment)
```
Create a new report in the database using given resource ID, report time, and reporting user. If `report` or `comment` parameters are passed non-zero, these IDs will be assigned

to the new report. If zero, new IDs will be generated.
Returns 0 on failure, new report ID on success.

`int delreport(int report)`
Delete report, associated comment, & fs associations from database.
Returns 0 on failure, `report` on success.

`int clonereport(int report)`
A new report is created from the original, inheriting these
attributes: resource, reptime, repuser, and fatal.
The original report's comment is copied (not shared),
and both comments are annotated to reflect the cloning.
fs associations are not inherited.
Returns 0 on failure, new report ID on success.

`int assocfs(int report, int fs)`
Associate `fs` with `report`.
Returns 0 on failure, `report` on success.

`int unassocfs(int report, int fs)`
Delete any association of `fs` with `report`.
Returns 0 on failure, `report` on success.

`int unassocall(int report)`
Delete all associations with this report.
Returns 0 on failure, `report` on success.

`int makesubject(int report, char *text)`
Generates a subject line for specified report,
based on equipment, user, symptoms, and report date
of report. Subject returned in `text`.
Returns 0 on failure, 1 on success.

`int pendreport(int report)`
Mark report as "pending" (currently on status board).
Returns 0 on failure, `report` on success.

`int unpendreport(int report)`
Mark report as "cleared" from status board.
Returns 0 on failure, `report` on success.

`int fatalize(int report)`
Mark report as "fatal": equipment should not
be operated until problem cleared.
Returns 0 on failure, `report` on success.

`int unfatalize(int report)`

Mark report as "non-fatal": equipment can be operated despite problem.
Returns 0 on failure, `report` on success.

Managing comments

`int newcomment()`
Creates a new comment.
Returns 0 on failure, new comment ID on success.

`int delcomment(int comment)`
Deletes a comment from the database.
Returns 0 on failure, `comment` on success.

`int clonecomment(int comment)`
Copies all lines of specified comment.
Returns 0 on failure, new comment ID on success.

`int setsubject(int comment, char *text)`
Set the first line ("subject") of a comment to `text`.
Returns 0 on failure, `comment` on success.

`int opencomment(int comment)`
Open a comment for appending. Only one comment may be open at a time.
Returns 0 on failure, `comment` on success.

`int appendline(char *text)`
Append `text` to currently open comment.
Returns 0 on failure, 1 on success.

`int closecomment()`
Closes currently open comment.
Returns 0 on failure, ID of closed comment on success.

Managing equipment status

`int writereport(int report, char *mode, char *cc)`
Write a problem report to concerned parties.
`mode` should be "new", "update", or "clear", reflecting current problem status. `Cc` should be `NULL` or a list of users to receive "courtesy copies" of the problem report.
Returns number of lines written.

`int writestatus(int resource)`
Update status board for pending reports on this equipment.

If any reports are marked fatal, the equipment is locked;
otherwise, any equipment lock is removed.
Returns number of lines written.

Example of Use

The following fragment of C code shows how the **faultlib** procedures can be used to manipulate the database. In this example, the automatic maintenance calendar is generating a "maintenance due" report for a piece of equipment.

```
/*
* Post a maintenance report for given equipment.  Memo describes
* the maintenance to be performed.
*
* Returns new report ID on success, 0 on failure.
*/

int PostMaintenance(char *eqname, char *memo)
{
    int resource;
    int report;
    int symptom;
    int comment;
    int userid;
    char buf[BUFSIZ];

    /*
    * Fetch attributes for new report.  MAINT_SYMP is a prefined symptom
    * ID used to report maintenance events on all equipment.  Then
    * create new report, using pre-allocated comment ID.
    */
    resource = getequipid(eqname);
    symptom = MAINT_SYMP;
    userid = getuid();
    comment = newcomment();
    report = newreport(0, resource, "now", userid, comment);
    if (report <= 0) {
     printf("ERROR: Can't create maintenance report for %s0, eqname);
     return(0);
    }

    /*
    * Indicate this is a maintenance report.
    */
    assocfs(report, symptom);

    /*
```

```
    * Write out an informative report body.  Create a subject based on
    * report attributes, and include the memo to show what maintenance
    * is due.
    */
    makesubject(report, buf);
    setsubject(comment, buf);
    opencomment(comment);
    appendline(buf);
    appendline("");
    appendline(memo);
    appendline("");
    appendline("This problem was reported automatically.");
    appendline("");
    closecomment();

    /*
    * Mark report to appear on Status Board.
    */

    pendreport(report);

    /*
    * Notify concerned parties of new problem and update Status Board.
    */

    writereport(report, "new", NULL);
    writestatus(resource);

    printf("Maintenance due on %s.\n", eqname);
    printf("Posted report #%d to Status Board.\n", report);

    return(report);

}
```

References

[1] *COMETS Reference Manual*, Consilium Inc., 1990.

[2] L. A. Rowe and C. B. Williams, *An Object-Oriented Database Design for Integrated Circuit Fabrication*, U.C. Berkeley Electronics Research Laboratory Memorandum M87/43, 1987.

[3] N. H. Chang, C.J. Spanos, *Continuous Equipment Diagnosis Using Evidence Integration: An LPCVD Application"*, IEEE Transactions on Semiconductor Manufacturing, February 1991.

[4] H.E. Korth, A. Silberschatz, *Database System Concepts*, McGraw-Hill Inc, 1986.

[5] P. Rampalli, A. Ramesh, N. Shah, *CEPT - A Computer Aided Manufacturing Application for Managing Equipment Reliability, Availability, and Maintainability in the Semiconductor Industry*, Proceedings of the Ninth IEEE/CHMT International Electronic Manufacturing Technology Symposium, 1990.

[6] *WORKSTREAM Reference Manual*, Consilium Inc, 1990.

[7] *INGRES/NET User's and Administrator's Guide*, Relational Technology Inc, 1989.

[8] *INGRES/SQL Reference Manual*, Relational Technology Inc, 1989.

[9] *INGRES ABF/4GL Reference Manual*, Relational Technology Inc, 1989.

[10] G. S. May, C. J. Spanos, *Automated Malfunction Diagnosis of a Plasma Etcher*, International Semiconductor Manufacturing Science Symposium, May 1991.

[11] G. S. May, *Automated Malfunction Diagnosis of Integrated Circuit Manufacturing Equipment*, U.C. Berkeley Electronics Research Laboratory Memorandum M91/33, 1991.

[12] *Book of SEMI Standards, Volume 2: Equipment Automation*, Semiconductor Equipment and Materials Institute Inc, 1987.