Remote Fabrication of Integrated Circuits

Software Support for the M.I.T. Computer Aided Fabrication Environment

by

Jimmy Y. Kwon

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

September 1995

Signature of Author .... [illegible]
Department of Electrical Engineering and Computer Science, August 24, 1995

Certified by .... [illegible]
Donald E. Troxel
Professor of Electrical Engineering
Thesis Supervisor

Accepted by .... [illegible]
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students

Remote Fabrication of Integrated Circuits

Software Support for the M.I.T. Computer Aided Fabrication Environment

by

Jimmy Y. Kwon

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Abstract

The Computer Aided Fabrication Environment (CAFE) is a software system developed at MIT for use in all phases of integrated circuit fabrication. While still undergoing development and enhancements, CAFE provides day-to-day support to research and production facilities at MIT, with both standard and flexible product lines.

One of the limitations of the present CAFE system is that it is not a fully open system. An open system is one that is designed to accommodate components from various software applications and can be viewed from three perspectives: portability, integration, and interoperability. This document is concerned with the idea of interoperability between different Computer Integrated Manufacturing (CIM) systems, and describes the extension to CAFE's architecture to support what is called Remote Fabrication.

With the goal of general interoperability between different CIM systems in mind, the Virtual Object Manager is developed. The Virtual Object Manager provides a framework for three key modules to support interoperability: the Object Management Module, the Export/Import Module, and the Remote Message Passing Module. Each of these modules are discussed and implemented for the CAFE platform.

Building upon the interoperability tools in the Virtual Object Manger, application layer support is then developed for the specific task of remote fabrication in CAFE. The application modules and additional software tools are described, and together form the CAFE Remote Fabrication System, a prototype system providing remote fabrication capabilities. An example of a remote fabrication session is described, stepping the reader through CAFE's remote processing paradigm and showing how the various tools are used. In the form of a tutorial, the example starts with installing a process flow into the CAFE database and creating a lot of wafers. The operate-machine/next-operation processing cycle is then done using both local and remote machines, showing the interoperability features at work. When the processing cycle completes a traveller file is generated, summarizing the entire history of processing.

While this document describes and implements a remote fabrication system specifically for CAFE, it is hoped that this document brings into focus some of the issues involved in general interoperability between different CIM systems, and provides a useful framework for future work.

Thesis Supervisor:	Donald E. Troxel
Title:	Professor of Electrical Engineering

Acknowledgments

I would like to thank Greg Fischer for his help in answering many of my questions regarding the CAFE system. He has endured my questioning on all aspects of CAFE, including describing the various application modules, their purpose, and their gory details; helping me understand the uses of the objects in the CAFE database; and providing general Lisp language help, pointers, and tips. His feedback was friendly, quick, practical, and extremely valuable.

Tom Lohman provided extensive help in dealing with and understanding the Ingres database. Without his help, some of the key objects and ideas described in this document would never have surfaced.

I'd like to thank Mike McIlrath for being the Lisp and GESTALT interface guru. He has been very constructive in helping me learn the details of Lisp and CLOS, the Common Lisp Object System. His wisdom, experience, and sense of humor have stimulated many interesting theories which led the way to many of the ideas presented in this document.

John Carney's work on message passing, in particular his socket routines, was crucial to the overall development of CAFE's interoperability tools. He also provided many helpful tips on the remote fabrication project during my numerous conversations with him.

I'd like to thank William Moyne, James Kao, Myron Freeman, and Francis Doughty for providing a pleasant and friendly environment to work in.

Finally, I'd like to thank Professor Troxel for making all this work possible.

This work has been made possible by support from the Advanced Research Projects Agency (ARPA) under Contract N00174-93-C-0035.

Table of Contents

List of Figures

Part 1

Introduction to Fabrication of Integrated Circuits Using CAFE

1 CAFE: Computer Aided Fabrication Environment

CAFE, or the Computer Aided Fabrication Environment, is a software system developed at MIT for use in all phases of integrated circuit fabrication. The system attempts to integrate applications and tools for process design and development, optimization, device modeling and simulation, documentation, technology CAD (Computer Aided Design), factory and resource planning. quality control, and the manufacturing of integrated circuit wafers. While still undergoing continual development and enhancements, CAFE presently provides day-to-day support to research and production facilities at MIT with both standard and flexible product lines [1].

1.1 CAFE's Architectural Framework

The CAFE architecture supports a wide variety of software modules, including development tools and applications. The architectural framework can be divided into three major layers: the Infrastructure Layer, the Data and Tool Integration Layer, and the Application Layer. Figure 1 shows a view of the layers and their components.

The *Infrastructure Layer* consists of low-level routines and utilities that interface directly to the operating system. CAFE runs on top of UNIX, and most of the applications at this level are written in C. The database schema is based on GESTALT, an object-oriented data model. GESTALT provides a layer of abstraction by mapping user-defined objects onto existing database systems (i.e., a relational database management system), sparing application programs the underlying details of the particular database. CAFE applications access persistent data through a common database interface layer which provides well defined programmatic interfaces. At present, the supported programmatic interfaces include C (typically for lower-level applications) and Common Lisp (for higher-level applications). The Infrastructure Layer also contains a common menu toolkit to provide for a user interface among CAFE applications.

The *Data and Tool Integration Layer* attempts to provide for a consistent model for all CAFE application modules. The CAFE schema is a high-level object-oriented conceptual model of integrated circuit fabrication activities, realized through the GESTALT modeling and interface layer. The representation of the manufacturing process provides for a unified representation for all CAFE applications and include the PFR – the Process Flow Representation, an extensible framework for knowledge about process steps – and the PIF – the Profile Interchange Format, a uniform representation of wafers. Programmatic interfaces to the CAFE schema are used by application modules to consistently manipulate these high-level representations. The CAFE menu system provides a common user interface to all CAFE applications.

The *Application Layer* consists of programs and modules that the user directly interacts with. These include applications that perform process design and simulation, fabrication utilities such as machine operation and data collection facilities, data analysis programs such as statistical process controllers, and resource scheduling and maintenance programs.

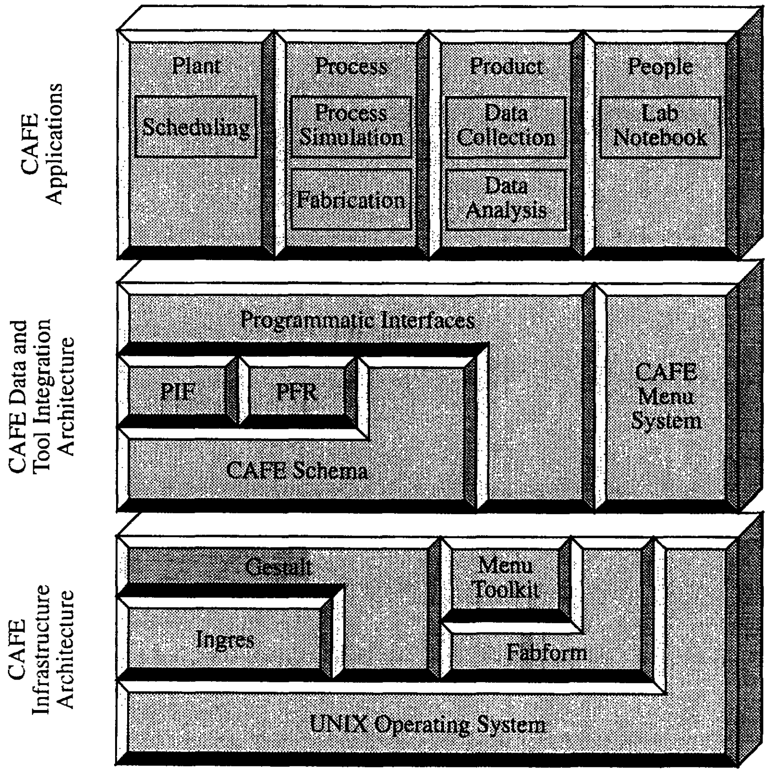

Figure 1. CAFE Architectural Framework

1.2 Remote Fabrication and Interoperability

This document is concerned with adding new features to the Data and Tool Integration Layer and the Application Layer to support what is called *Remote Fabrication.* The CAFE architectural framework currently is not a fully open system. An open system is one that is designed to accommodate components from various applications and can be viewed from three distinct perspectives: portability, integration, and interoperability [29]. Portability is the aspect of a system that allows it to be used in various environments. Integration is the aspect of a system that is concerned with the consistency of the various human-machine interfaces between an individual and all the hardware and software components in the system. Interoperability is concerned with the ability of components of the system to exchange information with other components. In a broader framework, interoperability is refers to the ability of components of one system to effectively exchange information with components of another distinct system.

While the existing CAFE system has a fairly high degree of integration of software components (with varying degrees of portability), it is markedly inadequate in terms of interoperability. The processing paradigm in the present CAFE system actively supports local fabrication activities. These are activities done in MIT's own research and production facilities for MIT researchers, staff, and technology developers. If these activities could be wholly contained locally (within MIT), then there wouldn't be much of a problem. Unfortunately, limiting the playground is not the way for new technology researchers and developers, and seeking and utilizing outside facilities is becoming more common. There are several reasons why this trend is developing. One reason is that although MIT has a useful set of fabrication equipment, it is impractical (space-wise and cost-wise) to have everything. IC fabrication machines are typically high precision equipment that are very costly to purchase and maintain. A designer may wish to use such cutting-edge/high-technology equipment which MIT doesn't have. How does the designer document such use in the process flow representation, and how does the designer assemble the necessary information to use such equipment? Machines also inevitably break down at times and repair times may be very long. By routing the affected process steps to use outside machines that are in operation, the delays can be reduced. Also, if, perchance in the future, MIT's facility becomes overworked with excessive fabrication activities, some of the processing steps may be routed to external facilities more capable of handling large inventories to alleviate the situation. How does one handle the massive information and data to be kept tracked of and communicated? By making a system interoperable and able to communicate this information automatically with other systems, the process designers can focus more on the fabrication process rather than fighting information overload.

The current CAFE system limits the fabrication activities to those done locally at MIT. Even though there is nothing stopping the designer in manually performing remote fabrication activities, he must unfortunately keep track of and communicate all the necessary information by himself. Such "hand-written" data, although computer-readable (i.e., it may be stored as a computer text file), is not readily computer-manipulable [2] and therefore has limited value for other CAFE applications that normally use the data. This document addresses this limitation in the CAFE system by building a framework for interoperability.

To make the existing CAFE system a fully interoperable system is a large and difficult problem. There are many issues and problems about the current CAFE system that need to be resolved. There is, however, a broader issue on how to design a generic interoperable system. One issue is that CAFE is only one system out of several different CIM (Computer Integrated Manufacturing) systems. There are many different types of database engines, schema models, and languages. Consider the simple graph shown in Figure 2. If everybody used one database engine, model, and language, then interoperable systems would be *homogeneous systems* [21] with a distributed database system (DDB), where information is shared among the systems and their actual location is dynamically handled by the entire system, or with a federated database system (FDB), where each system is autonomous and keeps full control on the data which are locally stored. Information sharing becomes a given without any incompatibilities. However, everybody doesn't use the same database engine, model, and language. There are several different database engines, each with their own strengths, and several different database models and languages. With the same database engine there can be different database models and languages, highlighting the need for *language and model interoperability.* There could also exist the same database model and language over different engines, highlighting the need for *system level interoperability.* Finally, the most generic case is *general interoperability,* where the database engines and the models and languages are different.

	same DB model and language	different DB model and language
same DB engine	homogeneous systems (DDB, FDB)	language interoperability
different DB engine	system-level interoperability	general interoperability

Figure 2. Interoperability between Databases

This document discusses issues for general interoperability in CAFE, but builds support more closely related to homogeneous system interoperability. The constrained problem is making one CAFE system interoperable with another identical, but distinct, CAFE system, in the application of fabrication. The most immediate reason for this is to support remote fabrication of integrated circuits between MIT and Lincoln Laboratory, both of which have IC fabrication facilities and both of which are using the CAFE system. The remote fabrication system described in this document builds a prototype framework that could be used as a starting point for development in general interoperability in CAFE.

1.3 CAFE Fabrication Process Paradigm

A major use of the CAFE system is to facilitate the design of process flows for fabricating integrated circuits. A process flow is a sequence of instructions and steps that describe what to do to a semiconductor wafer. An example of a process flow is where one starts with a bunch of silicon wafers, perform a stress relief oxide step to minimize the stress effects of nitride deposition and processing, and then perform a liquid-phase chemical vapor deposition (LPCVD) step of silicon nitride on the wafers. The designer can describe these process steps by using existing pre-installed operation sets (or "opsets" for short) or creating new ones by detailing exact instructions and the machine equipment needed. These custom process flows are formatted text files and are usually designed with the help of a flow editor. Once the process flow file is designed it can be installed into the CAFE database and checked for errors such as faulty or illegal sequences of machine steps (i.e., steps that may violate the integrity of the machine or wafer such as forgetting to perform a cleaning/purification step or an inspection step, steps that might be meaningless or redundant, or steps that may even prove harmful and dangerous). A summary of the process flow representation design cycle is shown in Figure 3.

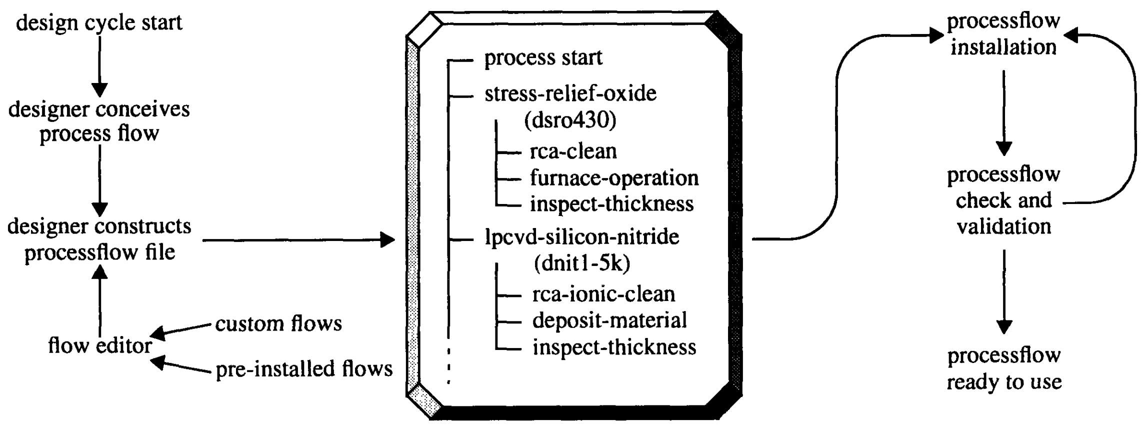

Figure 3. PFR Design Cycle

After the process flow has been validated it can be used as a template to start creating the actual wafers by scheduling the machine operations. Resources in the facility are then put to use by having laboratory operators execute the machine operations on the appropriate equipment in the order described in the process flow. At various points in the fabrication process the necessary data can be collected and documented in the CAFE database. This information can then be used for applications such as machine performance history reports, wafer process simulation, or as general documentation about the process.

The processing paradigm used in the CAFE system is shown in Figure 4. After the designer writes the PFR and successfully installs it into the CAFE database, the designer then assembles a set of silicon wafers that will be processed upon. This assemblage of wafers is called a *lot*. Once the lot has been created, the designer starts the process cycle on the lot. The process flow contains a sequence of instructions on what to do to the lot of wafers. These instructions will ultimately involve operating machines or pieces of equipment on the lot. The lot of wafers progress through each machine processing step sequentially until the end of the process flow. When the processing of the lot is done, the designer might wish to review the history of the entire process flow by generating a *traveller*. The traveller is typically a human readable file that summarizes all the machine steps, equipment settings, readings and measurements, and other notes and comments made during the process flow cycle.

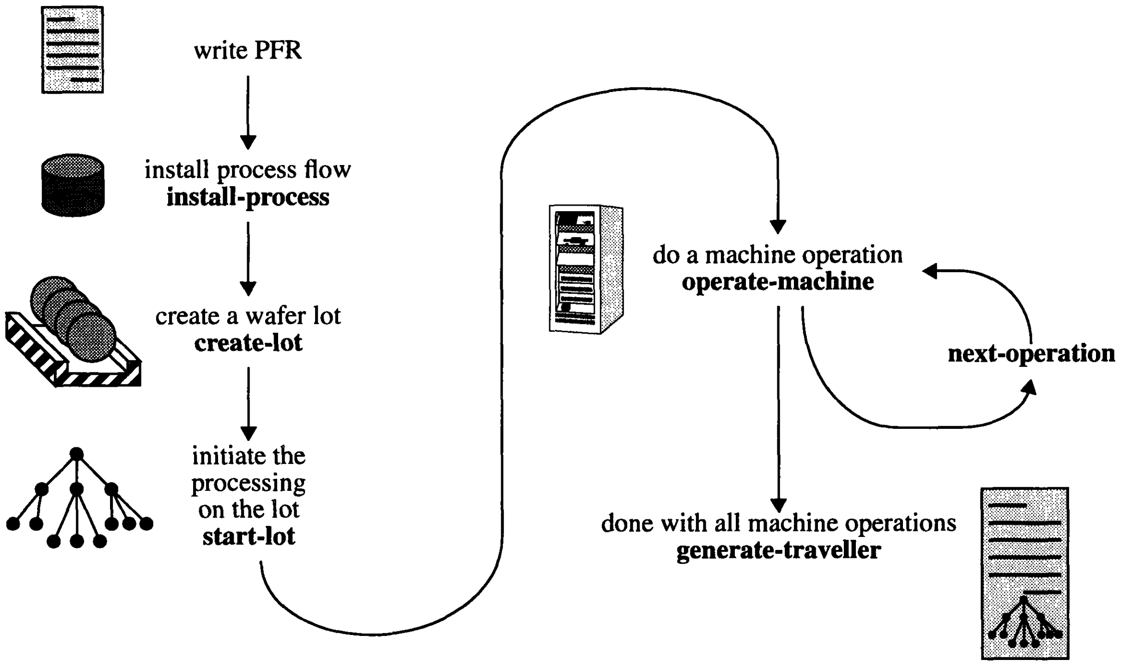

Figure 4. CAFE Processing Paradigm

1.4 CAFE Remote Fabrication Process Model

One of the shortcomings of the processing paradigm in CAFE is the lack of support for handling machine operations for machines that don't exist in the facility, i.e., the operate-machine/next-operation cycle in Figure 4 is strictly for local machines. To perform a machine operation for a lot at some remote facility, the designer must break the CAFE processing paradigm by leaving his CAFE session and manually do the operate-machine/next-operation cycle for himself.

A better solution is shown in Figure 5. The CAFE processing paradigm has been extended to support remote machine operations. The extension to the process model is shown within the dotted box. For remote processing to take place, the appropriate data must be extracted and assembled into a suitable form for transmission. This encoded data is then transferred by some communication channel to the remote site, which then receives the encoded data, decodes it, then reassembles the original data. The remote site may then perform its own operate-machine/next-operation cycle on the lot until the remote processing is done. New data is then assembled, encoded, transmitted, then decoded and reassembled in the original site. The normal operate-machine/next-operation cycle can then be resumed until the processing on the lot is completed.

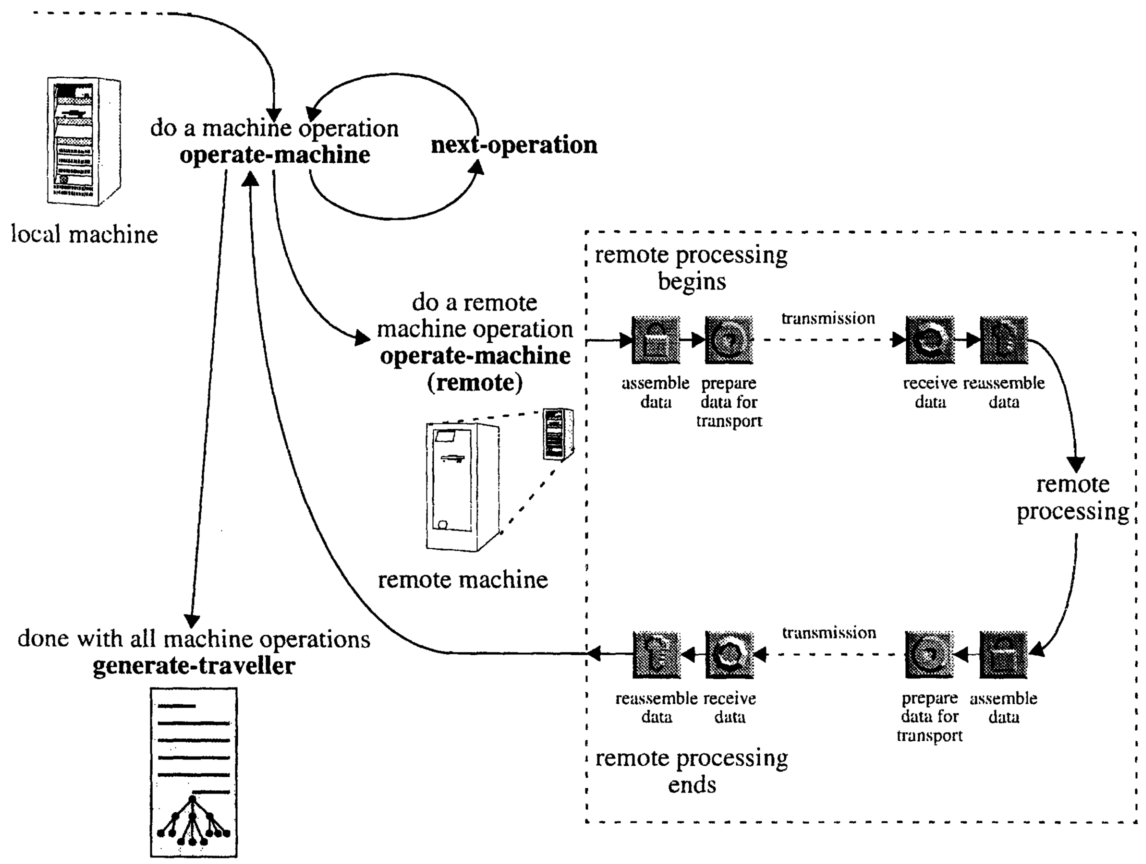

Figure 5. CAFE Extended Processing Paradigm

This document will discuss the different mechanisms involved in the dotted box in Figure 5. Part 2 will describe the software tools and infrastructure that will enable the CAFE system to do remote processing and fabrication. Part 3 will build upon these tools and describe the Application Layer modules that implement remote processing. It will also provide a walk-thru of an example remote fabrication experiment between two fictitious facilities, which will demonstrate how all the software tools and modules are combined to provide support for remote fabrication.

Part 2

Remote Fabrication Software Toolkit

2 Brief Introduction to the CAFE Database

The database schema in CAFE is based on GESTALT, an object-oriented extensible data model [1]. GESTALT provides an abstraction layer by automagically mapping user-defined objects onto the underlying database engine and providing a programming interface to a host language to access and manipulate the persistent data. The Remote Fabrication software toolkit attempts to avoid dealing with programming the database directly and potentially suffering from database language dichotomies in the event a different database engine is used, so GESTALT is used as the interface layer to the database. The development language platform used is primarily Common Lisp due to its flexibility and ease for rapid prototyping of software, and the reader is assumed to have some familiarity with the language and its object-oriented extension, the Common Lisp Object System (CLOS).

2.1 Objects, Classes, and Persistence

Most of the information in the CAFE database is contained in the form of objects. An object is an organized collection of data, and there are established ways of creating and generating these objects, and selecting and changing an object's components. A simple example of an object is a number, like 42, or a string "Hello World." A more complex example of an object is the TIME object, which contains two strings containing the date and the time of day. The example below shows a function being used to generate the TIME object and a variable which is bound to a LABUSER object. The function **describe** is used to browse the contents of the objects. Note that the LABUSER object with the name "merklin" contains a slot called "facility." This slot contains a list of two other objects, which are a different type of object called FACILITY. Objects in CAFE can, and quite often do, point to other objects.

```
CAFE>(current-time)
#<TIME 45621120>

CAFE>(describe *)

#<TIME 45621120> is an instance of class #<Gestalt-Class TIME 33042440>:
 The following slots have :INSTANCE allocation:
 %ENTITY      10987144
 DATE         "07/12/95"
 TIMEOFDAY    "15:07:49"

CAFE>merklin
#<LABUSER merklin 40222340>

CAFE>(describe *)

#<LABUSER merklin 40222340> is an instance of class #<Gestalt-Class LABUSER
33035560>:
 The following slots have :INSTANCE allocation:
 %ENTITY     9465864
 NAME        "merklin"
 FACILITY    (#<FACILITY ICL 45621240>
              #<FACILITY TRL 45621220>)
 ADVICE      "(:active :t)"
```

Figure 6 shows the hierarchy of some of the classes in Lisp and CAFE. Two interesting classes are the **standard-object** class and the **gestalt-object** class. Classes that are defined by the user in Lisp become subclasses of the **standard-object** class. Most of the objects in the CAFE database are a subclass of the **gestalt-object** class. These objects typically are *persistent*, meaning that when they are created they are stored permanently in the CAFE database and can be used by other applications that access the database. It also means that when the current Lisp session ends, the object will continue to exist and can be retrieved in a later session. Note that **gestalt-object** is a subclass of **standard-object** because in essence it is a user-defined class, defined by the GESTALT interface.

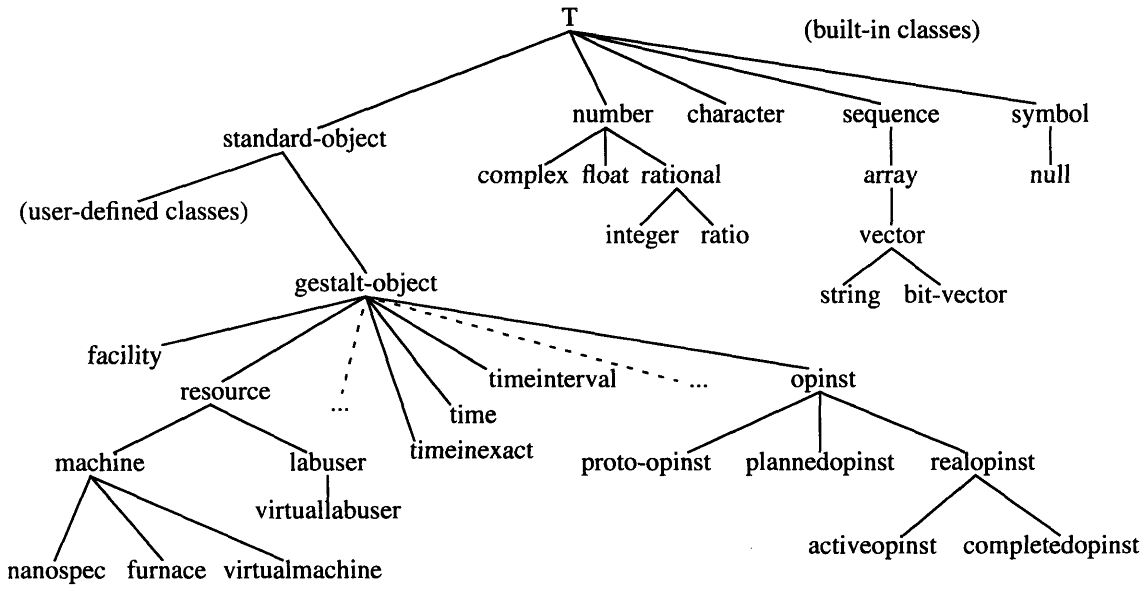

Figure 6. CAFE Class Hierarchy

Because persistent objects must remain known in the CAFE database, there should be a way to uniquely identify these objects so that different applications that use the database can access the same object. One way to identify an object, and the way that is used in this document, is to assign an *object key* to the object when it is first created. An object key is a list of two elements: the object's class name and an entity ID number, i.e., (<object-class-name> <entity-ID-number>). Each time a new instance of a class is made, the object is given a unique entity ID number. The mechanism for assigning these numbers is handled by the GESTALT interface and the underlying database. The object key provides a one-to-one mapping to the object. Given a persistent object, one can retrieve its unique object key, and given an object key one can retrieve the object. For now this only applies to persistent objects. Non-persistent objects don't have a unique entity ID and therefore cannot be retrieved in this way. Later on, this restriction will be relaxed a little bit. The session below shows an example of retrieving the object key for the LABUSER object called **merklin**, and retrieving the object by using its object key.

```
CAFE>merklin
#<LABUSER merklin 40222340>

CAFE>(get-object-key *)
(LABUSER 555)

CAFE>(get-object '(labuser 555))
#<LABUSER merklin 40222340>
```

2.2 Definitions of some CAFE Database Objects

Much of the information associated with the fabrication process exist as objects in the CAFE database. Most of the objects are named to correspond to the actual physical entities that they refer, keeping the conceptual meaning the same to the user and programmer.

wafertype — A wafertype object represents the type of wafer that is used for fabrication. Information that characterizes the wafertype include whether the silicon was grown epitaxially, the substrate dopant used (i.e. boron, arsenic, phosphorus) and the substrate resistivity, the wafer orientation (Miller index), and various other characteristics.

wafer — A wafer object represents its associated physical entity. The wafer object contains the wafertype of the wafer as well as the same laser ID code on the physical wafer.

waferset — A waferset object is merely an organized grouping of wafers, and may contain a unique name that identifies the set.

lot — A lot object is a grouping of wafersets. Machine operations are conceptually done on lots instead of individual wafers, even though the actual machine may process the wafers serially. A particular inspection equipment may only be able to inspect wafers one at a time. The lot also contains information about its creator (the designer) and its creation date, its current status in relation to its process cycle and its past history, and other related information. Given the lot object the programmer can determine exactly what has been done and what to do next.

facility — The facility object documents information about the fabrication facility of the same name, including its physical location and description of its use.

machine — A machine object is associated with the physical machine of the same name. It includes a description of its use, the facility it currently resides in, a data communication protocol with the actual machine (if the machine is connected to a network or local computer system), scheduling information, and a list of operators who are qualified to operate the machine.

labuser — Human operators are also mirrored in the database as labuser objects. This is useful in resource planning and management, since humans as well as machine and equipment are needed to run the fab.

processflow — A processflow object is part of a tree of processflow objects. Together, the tree documents all the information the process designer has documented to specify a fabrication process. The processflow then becomes a template for other designers to use.

task — Task objects are similar to processflow objects. When a processflow tree has been installed and is ready to be used for processing, a copy of the processflow tree is made using task objects. The constructed task tree then becomes an instantiation of the processflow. This is done to allow the flexibility of altering the processing steps or inserting additional steps in case something goes wrong, or if the designer wishes to explore alternate avenues of fabrication.

opset — Short for operation set, an opset is the same as a task object. Conceptually, while a task specifies a single atomic machine operation, an opset consists of a group of these tasks that together form a cohesive operation. This hierarchical arrangement helps designer think in more higher-level operations. An example is a Generic Diffusion Deposition Opset, shown in Figure 7, which consists of the atomic machine operations: RCA cleaning step, the diffusion deposition set, and an inspection step.

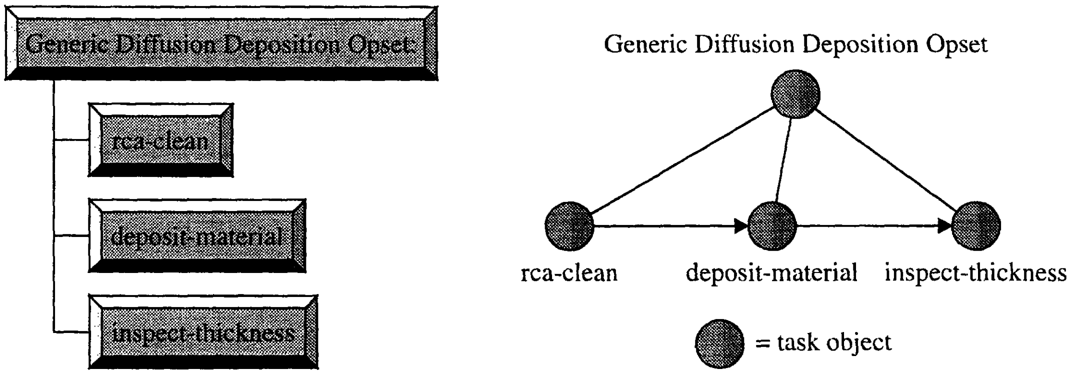

Figure 7. Generic Diffusion Deposition Opset

The top node is a task object, which may be thought of as an opset task object. Note that these opset tasks are instantiated versions of the associated opset flows. The bottom level tasks are called leaf tasks and contain the atomic machine operations.

opinst — Short for operation instance, opinsts are the objects that actually carry the run-time data specified by the designer and collected by the labuser or machine operator during actual fabrication. All the leaf tasks in a task tree have an associated opinst object because the leaf tasks are the tasks that contain the actual machine operations. The opinst object specifies the actual machine being used, the appropriate settings on the machine (as specified by the designer), readings that need to be collected during the processing, and the time interval of the entire machine step. There are several classes of opinsts, including the proto-opinst, activeopinst, and completedopinst. These will be explained later.

traveller — A traveller is the name given to the history of processing on a given lot. The traveller report typically contains information on the individual machine operations and their readings and settings. The report may be formatted in a variety of styles to fit the viewer, such as grouping the machine steps into opsets.

intervals — There are several places in the CAFE database where a single number cannot adequately specify a desired characteristic. In these cases, interval objects are used to specify a range of values (i.e. 1.0 — 5.5). Depending on the particular application, these ranges may specify a probabilistic distribution such as a Gaussian or equiprobability, or may simply denote a range as in a time interval.

3 Database Interoperability Tools

This section describes the architecture and tools that extend CAFE's existing system framework to support database interoperability. Collectively these tools are part of the *Virtual Object Manager* (VOM for short). The Virtual Object Manager is responsible for handling and keeping track of information traveling between its own local system and other remote systems. The VOM resides in CAFE's Application and Tool Layer and provides a layer of abstraction for CAFE applications.

To understand the fundamental spirit of the VOM and its pieces, let's consider an example of the communicating philosophers (viciously taken from [30]). This examples addresses some of the issues associated with establishing effective communications between two philosophers from diverse cultures. For example, suppose an Asian philosopher and a European philosopher wish to discuss some aspect about which each has thought. In Figure 8(a), we represent the exchange of pure thought by a dotted line between two clouds. Rules for effective exchange of ideas may call for an agreed set of assumptions and definitions between the two philosophers — a protocol for the communication of ideas. Unfortunately, the figure doesn't represent a feasible dialog since the two philosophers think in different languages and culture, and are physically located on distinct continents.

Figure 8(b) extends the exchange of ideas between the philosophers with the idea that each philosopher can translate thoughts into an expressive language, represented by the elliptic bubble. If the philosophers were in the same room, it would be possible to translate one philosopher's spoken words into the language understood by the other. The protocol for communication might now include rules of conduct (i.e., while one philosopher is speaking the other is not, with a willingness of both to stop talking if both begin to speak at the same time). It is still necessary, however, to transmit the spoken words (either before or after the translation) from the site of the speaking philosopher to the site of the listening philosopher.

In Figure 8(c), the physical transmission of spoken words is done by telegram. The protocol for effective communication might now address issues related to the means of transmitting the contents of the telegram from the sending site to the receiving site. Cloud-to-cloud communication is now done via bubble-to-bubble communication; that is, cloud-to-cloud interoperation is implemented by translating a sender cloud into a sender bubble, translating and transferring bubbles, and translating the receiving bubble to the receiving cloud.

Figure 9 shows the framework of tools existing in the VOM. The major blocks in the VOM are the export/import module, the object management module, and the remote message passing module. In the spirit of the communicating philosophers, the object management module represents some of the house-keeping functions needed in each of the philosophers and resides in the thought clouds of each philosopher (each philosopher needs to be aware of the existence of the other!). The export/import module provides the mechanism for translation between the pure thought cloud of a philosopher and his message bubble. The remote message passing module provides for the means to send and receive these messages to and from different sites. The object management and export/import modules will be discussed in this section, while the remote message passing module will be discussed in the section 4. This is done because the object management and export/import modules are operationally more closely coupled than the remote message passing module. Figure 10 shows a more detailed look of the modules in the VOM to help in the ensuing discussion.

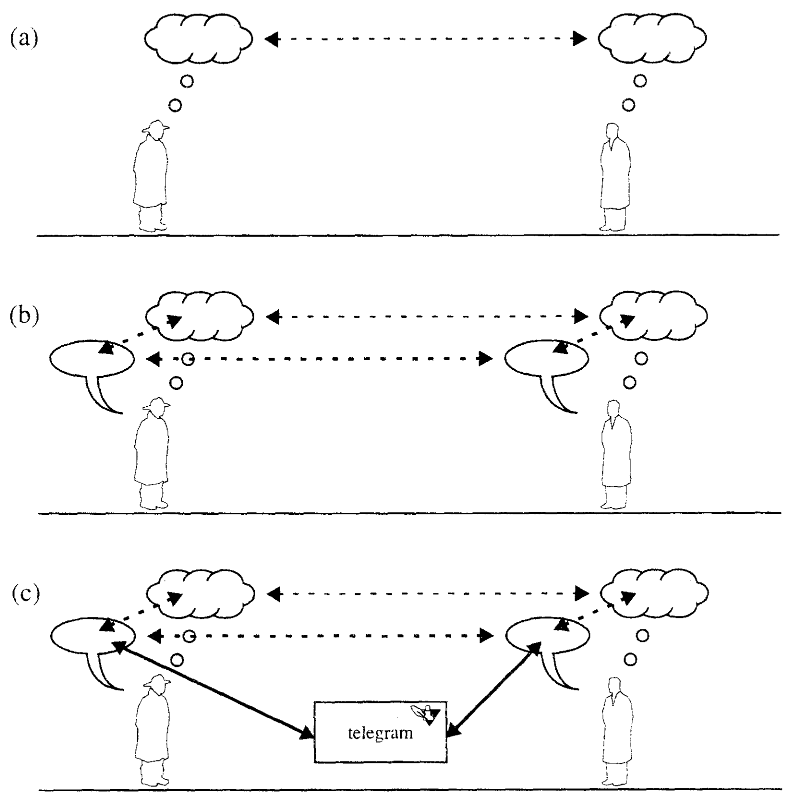

Figure 8. The Communicating Philosophers

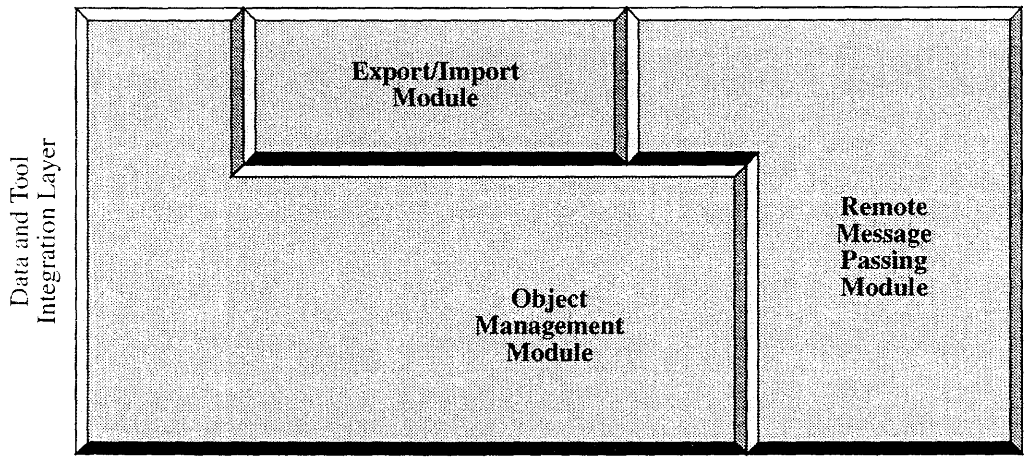
Data and Tool Integration Layer
Export/Import Module
Object Management Module
Remote Message Passing Module

Figure 9. CAFE Virtual Object Manager Framework

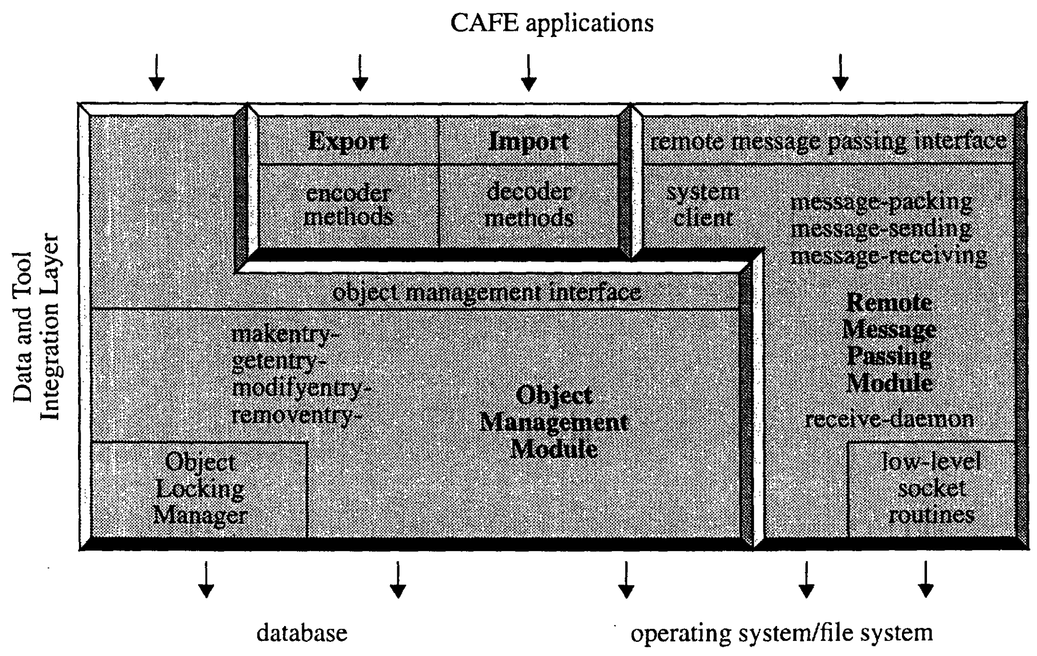

Figure 10. CAFE Virtual Object Manager Internal Architecture

3.1 Object Management Module

One of the most important task of the VOM is to keep track of what information has been communicated to other databases. Since in CAFE information is primarily contained in database objects, the task includes maintaining a record of what objects have been sent to and from a remote database. The principal record-keeping vehicle is the *object association list*. To understand why an object association list is needed, consider an example where one database called DB1 has an object that it wishes to communicate to another database called DB2. Under the constraints and assumptions described in Part 1, suppose DB1 and DB2 are similar in that both databases support the notion of object keys, even though the underlying implementations may be different. The object that DB1 wishes to send is something called a foo-record for the new $\overline{\varphi}_u$ product. The foo-record has a particular DB1 object key (foo-record 1745). DB2 also contains many foo-records of its own, independent of DB1. Suppose by some process the information stored in (foo-record 1745) from DB1 was copied over to DB2. Because the implementations of the two databases are independent, DB2 creates a new foo-record with an exact copy of the information and assigns it an object key (foo-record 33973). The same foo-record is now known to both DB1 and DB2.

Suppose DB2 does some analysis and updates the information stored in the foo-record. It makes the change in its own copy of the foo-record and now wishes to propagate the change back to DB1. With the communication issues aside, how does DB2 tell to DB1 which foo-record it is modifying. DB2 can't simply say that it is updating the records in the object with object key (foo-record 33973) because that object key doesn't exist in DB1 (or it may be a completely different record). If DB2 simply sends a copy of its (foo-record 33973) back to DB1, then DB1 will think it is new information and create a new foo-record with an object-key (foo-record 1746). If DB1 somehow knew that its own (foo-record 1745) is the same as DB2's (foo-record 33973), then if DB2 sent the new information in (foo-record 33973) back to DB1, then DB1 should be smart enough to realize the translation of object keys and modify (foo-record 1745) instead of creating a new object (foo-record 1746) or unintentionally modifying its own (foo-record 33973) (which will in general be different from DB2's (foo-record 33973), including being nonexistent).

One way to maintain this object association is to have a repository list with an entry that explains:

my own (foo-record 1745) is associated with DB2's (foo-record 33973)

This puts the burden of maintaining the proper object translation on DB1. If DB2 decides to reoptimize its database (rehashing its tables, balancing B-trees), the record with object key (foo-record 33973) might later be assigned another object key like (foo-record 391). The object association entry in DB1 then becomes obsolete and either DB2 will have to send a message to DB1 saying that the association has changed, or DB1 must actively poll DB2 and query about such changes. A better solution would be to have more symmetry by making DB1 and DB2 maintain their own object association lists. Under this scheme the entries might first appear like:

(in DB1)   my own (foo-record 1745) is associated with DB2's (foo-record 33973)

(in DB2)   my own (foo-record 33973) is associated with DB1's (foo-record 1745)

This scheme still does not avoid the problem when internal object keys change. If DB2's object with object key (foo-record 33973) gets remapped due to a database optimizer, both association entries in DB2 and DB1 must be updated. If we completely remove knowledge of the other database's particular object key and assign a *neutral* object key to the foo-record, then the association entries can look like:

(in DB1)   my own (foo-record 1745) is associated with (foo-record 1) when talking to DB2

(in DB2)   my own (foo-record 33973) is associated with (foo-record 1) when talking to DB1

The neutral object key (foo-record 1) is an arbitrarily assigned key (in this case, the same class name **foo-record** with an arbitrary entity ID number **1**). Using this object key abstraction layer, each database can do whatever optimizations or object reordering necessary and update its own object association list. The neutral object key (foo-record 1) remains the same throughout the entire lifetime of the foo-record. Figure 11 shows a graphical view of the foo-record associations between the two databases.

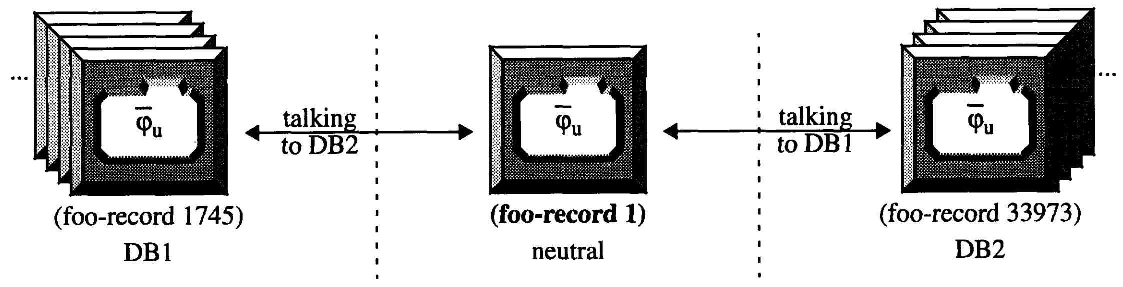

Figure 11. Object Association Example between DB1 and DB2

3.1.1 Object Association List in CAFE

In CAFE we can make an object association list by creating a global variable called ***object-assoc-list*** which will contain the association entries. The ***object-assoc-list*** is a list of domain entries, where a domain is the name used to identify the remote database. Each domain entry contains a list of object association entries between the local database and the domain. For the example above, the ***object-assoc-list*** that will exist in DB1 will look like:

```
CAFE>*object-assoc-list*
(((:TO-DOMAIN "DB2")
  ((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1))))
```

If another object association entry for (foo-record 1745) is added for a different remote domain, the object association list might look like:

```
CAFE>*object-assoc-list*
(((:TO-DOMAIN "DB2")
  ((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1)))
 ((:TO-DOMAIN "another-DB")
  ((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 7))))
```

The format for the object association entry is:

(<local db object key> IS-ASSOCIATED-WITH <neutral object key>)

To make an entry into the ***object-assoc-list***, we use the constructor function **makentry-object-assoc-list**, which takes as required arguments the local database's object key, the designated neutral object key, and the domain name. The function returns the association entry made if it was successful:

```
CAFE>(makentry-object-assoc-list '(foo-record 1745) '(foo-record 1) "DB2")
((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1))
```

If you don't remember the specifications of the argument list, in most cases you can get a brief documentation string by using **help** or the Lisp's **documentation** facility:

```
CAFE>(help 'makentry-object-assoc-list)
-----------------------------------------------------------------------------
MAKENTRY-OBJECT-ASSOC-LIST                                         [Function]
Args: (obj-key encoded-key domain)
Adds an object association entry into the object association
list. The format for an entry is:
     (<local database object> is-associated-with <target database object>)
-----------------------------------------------------------------------------
```

To retrieve a particular association entry in a domain, we use the accessor function **getentry-object-assoc-list**, which takes as arguments an object key and the domain name:

```
CAFE>(getentry-object-assoc-list '(foo-record 1745) "DB2")
((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1))
```

The object key may either be the local database's object key or the neutral object key. If the neutral object key is used, then we need to add the keyworded argument **:inverse t**:

```
CAFE>(getentry-object-assoc-list '(foo-record 1) "DB2" :inverse t)
((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1))
```

To remove a particular association entry, the destructor function **removentry-object-assoc-list** is used. If the entry was able to be deleted, a symbol value of **REMOVED** is returned. If not, then another symbol value would be returned summarizing the reason for failure.

```
CAFE>(removentry-object-assoc-list '(foo-record 1745) "another-DB")
REMOVED
```

Note that although the ***object-assoc-list*** is human-readable, all accesses to it should be done through the advertised protocol functions describe above. In this way the user of the ***object-assoc-list*** is free from writing the lower level code, and the developer retains the freedom and flexibility to change the underlying mechanism. This means that even though throughout this document we stress human readability over time/space efficiency, we should never be interpreting the contents of the ***object-assoc-list*** directly. When an object association entry is retrieved, we use the function **object-assoc-key** to extract the local database object key, and the function **object-assoc-encoded-key**[1] to extract the assigned neutral object key:

```
CAFE>(getentry-object-assoc-list '(foo-record 1745) "DB2")
((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1))

CAFE>(object-assoc-key *)
(FOO-RECORD 1745)

CAFE>(object-assoc-encoded-key **)
(FOO-RECORD 1)
```

There are also protocol functions for creating object keys and accessing the object class name and the entity ID numbers, but these won't be discussed here.

3.1.2 Virtual Object Association List in CAFE

The *virtual* object association list is functionally identical to the object association list, and in certain respects is unnecessary in the VOM. While the object association list is used to hold association entries from the host database to a remote database, the virtual object association list is used to hold association entries of objects that are initially from another database, but now exist locally. This is one way of identifying where an object was originally created and who the true owner is. Using the example of foo-records, if the creator of the foo-record was DB1, then in DB1 the object association entry goes in ***object-assoc-list*** and in DB2 the object association entry goes in ***virtual-object-assoc-list***. Figure 12 shows the different types of associations between DB1 and DB2.

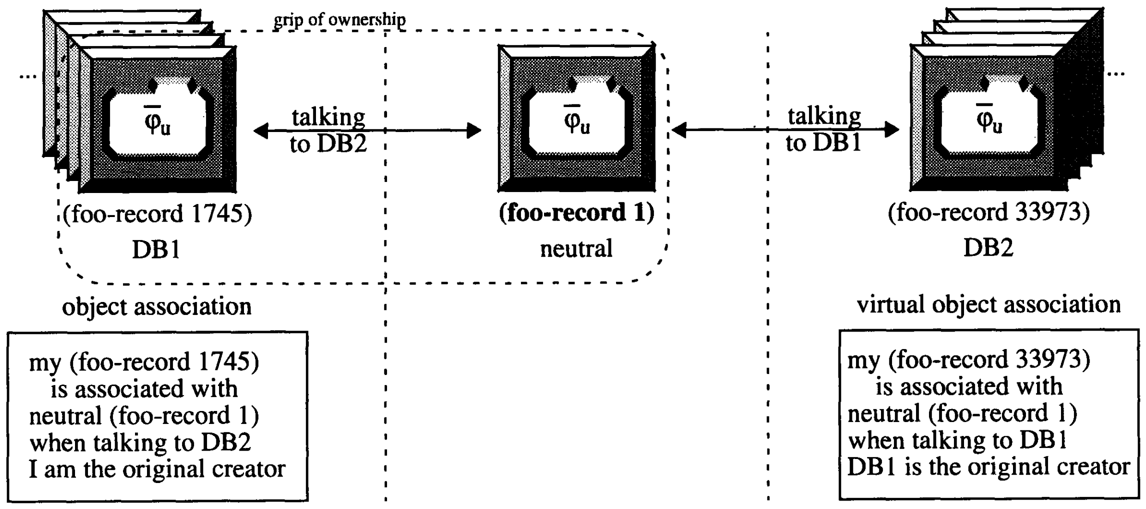
grip of ownership
talking to DB2
talking to DB1
$\bar{\varphi}_u$
(foo-record 1745)
DB1
(foo-record 1)
neutral
(foo-record 33973)
DB2
object association
virtual object association
my (foo-record 1745) is associated with neutral (foo-record 1) when talking to DB2 I am the original creator
my (foo-record 33973) is associated with neutral (foo-record 1) when talking to DB1 DB1 is the original creator

Figure 12. Revised Object Association Example between DB1 and DB2

The accessor functions for ***virtual-object-assoc-list*** are of the same style as ***object-assoc-list***. In database DB2, we can construct an entry by:

```
DB2>(makentry-virtual-object-assoc-list '(foo-record 33973) '(foo-record 1)
"DB1")
((FOO-RECORD 33973) IS-ASSOCIATED-WITH (FOO-RECORD 1))
```

1. In a sense the neutral object key is an encoded key for a particular object. It might have been better to call this function **object-assoc-neutral-key** for consistency, but it's too late.

```
DB2>*virtual-object-assoc-list*
(((:FROM-DOMAIN "DB1")
  ((FOO-RECORD 33973) IS-ASSOCIATED-WITH (FOO-RECORD 1))))
```

In database DB1, the object association entry is maintained in ***object-assoc-list***:

```
DB1>*object-assoc-list*
(((:TO-DOMAIN "DB2")
  ((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1))))
```

3.1.3 Object Database and Virtual Object Database Lists

The object and virtual object databases are repositories for ancillary information about object and virtual object associations, respectively. Examples might include the time an object association entry was made, the last modification time, status of the object (including access privileges, a measure of the object's stability), and object recovery information. The object and virtual object databases are lists that are similar in structure to the object and virtual object association lists — the database lists are divided into domain entries, each of which is a list of entries that identify an object key and whatever complementary information. Using the foo-record example, the object database might look like:

```
DB1>*object-database*
(((:TO-DOMAIN "DB2")
  ((FOO-RECORD 1745)
   (:IS-ASSOCIATED-WITH (FOO-RECORD 1))
   (:TIMESTAMP "Mon Aug  7 11:50:09 EST 1995")
   (:OBJECT-OWNER "Gym Z. Quirk"))))
```

More will be said later about the contents of the object database (contained in the global variable ***object-database***) and virtual object database (contained in the global variable ***virtual-object-database***).

3.1.4 get-encoded-object-eid and the *neutral-object-entity-id-assoc-list*

There is the question: How do we assign *neutral* object keys to objects that are to be communicated between databases? The method function **get-encoded-object-eid** handles this assignment of unique neutral object keys. The function takes as arguments an object and the target domain that the object will be communicated to. The domain argument is needed because the neutral object key (foo-record 1) will in general be associated with a different foo-record object:

```
CAFE>*object-assoc-list*
(((:TO-DOMAIN "DB2")
  ((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1)))
 ((:TO-DOMAIN "another-DB")
  ((FOO-RECORD 8245) IS-ASSOCIATED-WITH (FOO-RECORD 1))
  ((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 7))))
```

In this example, the foo-record object with key (foo-record 1745) is associated with the neutral object key (foo-record 1) when talking to domain DB2. However, a different foo-record object, one with a key (foo-record 8245), is associated with the neutral object key (foo-record 1) when talking to domain named "another-DB." In fact, (foo-record 1745) is associated with a different neutral object key, namely (foo-record 7), between the local database and "another-DB."

The assignment of unique neutral object keys is an interesting problem. Neutral object keys imply an agreement between two different databases that when they talk about their own object associated with the neutral object key, the objects are meant to be the same. Suppose both databases, DB1 and DB2, are actively talking to each other and transferring all sorts of objects. If DB1 decides to transfer an object and assign it a neutral object key, DB2 must not use that same neutral object key for any other object except the one presently being communicated, so DB2 must update its state to prevent using the same neutral object key. What happens when DB1 and DB2 want to communicate their own respective objects to the other *at the same time*? Constantly communicating the state of neutral object keys may take up significant bandwidth and slow things down quite a bit.

One solution is to give a parity status to each of the databases. Upon some initial agreement, one database will acquire even parity and the other will acquire odd parity. The database with even parity will forevermore use even-numbered entity ID's when constructing neutral object keys, and the database with odd parity will use odd-numbered entity ID's when constructing neutral object keys. For example, suppose DB1 initially starts the information transaction with DB2 by sending a foo-record. By looking at its object association lists (***object-assoc-list*** and ***virtual-object-assoc-list***), DB1 realizes that it has never sent anything to DB2 so it bestows itself with odd parity and assigns a neutral object key of (foo-record 1) to the foo-record and sends the information on its way:

```
DB1>*object-assoc-list*
(((:TO-DOMAIN "DB2")
  ((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1))))
```

Some time later, DB2 receives the new foo-record, incorporates it into its own database, and updates its object associations:

```
DB2>*virtual-object-assoc-list*
(((:FROM-DOMAIN "DB1")
  ((FOO-RECORD 33973) IS-ASSOCIATED-WITH (FOO-RECORD 1))))
```

Now DB2 wishes to send two of its own foo-records to DB1. Because DB2 has never communicated to DB1, DB2 checks its object association lists before assigning a neutral object key. DB2 discovers that DB1 has already sent objects to itself and has proclaimed odd parity because of the neutral object key (foo-record 1). DB2 then assigns itself even parity and takes the next two even-numbered entity ID's for its neutral object keys:

```
DB2>*object-assoc-list*
(((:TO-DOMAIN "DB1")
  ((FOO-RECORD 42866) IS-ASSOCIATED-WITH (FOO-RECORD 2))
  ((FOO-RECORD 42867) IS-ASSOCIATED-WITH (FOO-RECORD 4))))
```

In the meantime, DB1 decides to send a few more foo-records to DB2 before receiving back a reply. Because DB1 has odd parity, it uses the next two odd-numbered entity ID's:

```
DB1>*object-assoc-list*
(((:TO-DOMAIN "DB2")
  ((FOO-RECORD 1745) IS-ASSOCIATED-WITH (FOO-RECORD 1))
  ((FOO-RECORD 1888) IS-ASSOCIATED-WITH (FOO-RECORD 3))
  ((FOO-RECORD 1889) IS-ASSOCIATED-WITH (FOO-RECORD 5))))
```

Once the initial agreement has been made, DB1 and DB2 can continue to assign unique neutral entity ID's to foo-records without worrying about entity ID conflicts. This goes for other classes of object as well. If DB2 wishes to send another object of type data-bar to DB1, it will use even-numbered entity ID's because of its even parity status:

```
DB2>*object-assoc-list*
(((:TO-DOMAIN "DB1")
  ((FOO-RECORD 42866) IS-ASSOCIATED-WITH (FOO-RECORD 2))
  ((FOO-RECORD 42867) IS-ASSOCIATED-WITH (FOO-RECORD 4))
  ((DATA-BAR 20164) IS-ASSOCIATED-WITH (DATA-BAR 2))
  ((DATA-BAR 20165) IS-ASSOCIATED-WITH (DATA-BAR 4))))
```

Given the context clues found in ***object-assoc-list*** and ***virtual-object-assoc-list***, the function **get-encoded-object-eid** can determine what the database's parity is and record that knowledge in an internal table stored in the ***neutral-object-entity-id-assoc-list***. The list maintains the next neutral entity ID numbers that can be used given an object and the particular target domain. The information contained in this list is automatically maintained and updated by **get-encoded-object-eid** and is transparent to the user.

3.2 Object Locking Manager

While the above discussion has focussed on the sharing of objects and the VOM's task of transparently handling and maintaining various aspects of object associations between databases, also important is the issue of concurrency control, i.e., who has read/write access to the information contained in a given object. In a self-contained database system such as a relational database management system (RDBMS), issues related to concurrency control and transaction management, as well as space management, cache and file management, and recovery are all done and enforced typically in the lower portions of the system. Extending these protocols to multiple databases, which may be very different in many respects, is a somewhat difficult problem, especially when these databases do not integrate very well with each other and communication between them may be slow and costly. The Object Locking Manager is a module whose purpose is to handle concurrency control between such databases. Objects transferred from one database to another must be *locked* in one database (i.e., given read-only status or even no-access status) before the other database can make any changes to such objects. Without an object locking protocol, both databases could make changes to a shared object and end up with an object containing corrupt information (consider the case where two independent programs write data simultaneously to the same file).

While an important module, the Object Locking Manager is not implemented. However, for the purposes of the Remote Fabrication prototype system for CAFE, we can get away with this deficiency. Much of the information in the CAFE database that are needed to support remote fabrication are contained in static objects. For example, the processing instructions needed to perform machine operations are basically static information. The real information-carrying objects are encapsulated in opinsts, and even these are essentially static. Machine settings, instrument readings, and other data are recorded into the computer and initially stored in temporary objects called activeopinsts. When the particular machine operation is complete, a new object called a completeopinst is created and the information stored in an activeopinst is copied to it. The original activeopinst object is destroyed, and the next machine operation is performed by creating another activeopinst object. Under the present processing paradigm in CAFE, past machine operations are never revisited (in the same processing session), and so the information in the completedopinst never changes. Because completedopinsts are always created and never reused, all new information worth sharing between one CAFE database and another is contained in new objects. While this may be exploiting a CAFE-specific characteristic, it was necessary due to time and resource limitations. There are some objects in CAFE that don't lend themselves nicely to this solution and would benefit from an object locking protocol.

3.3 Export Tool: The encode-object method

So far we have talked about the object management functions of the VOM using object association lists and object keys. In this subsection and the next, we talk about the mechanisms of translating the information stored in one database into a form that can be incorporated into another database. The mechanism for *exporting* information, that is, the mechanism for translating information from the CAFE database to a transportable form, is embodied in a set of methods under the generic function **encode-object**. The mechanism for *importing* information, that is, the mechanism for translating information from the transportable form and incorporating it into the database, is embodied in a set of methods under the generic function **decode-object**. This subsection deals with the **encode-object** methods and discusses one particular implementation for use in CAFE.

Recalling the example of the communicating philosophers, the export and import tools implement the translation of information from the database (the "thought cloud") into a more transportable form (the "bubble"). The word *transportable* implies two things: the notion that something is carryable (*–portable*), and that it can be carried across to some distant location (*trans–*). Objects in the CAFE database are stored as a collection of bits in computer memory or disk, and the exact details of how the data is structured and stored is dependent on various factors such as the operating system platform, the database engine, and the architecture of the computer system and its peripherals (i.e., memory, disk). To simply binary-copy an object to another system is a mistake without knowing the such details. In fact, because the database the object is being copied to can be completely different in implementation from the original host database, binary-copying is not an option and a common language needs to be developed that can be understood by both databases.

Unfortunately, because of the diversity of database engines, models, and languages, and the diversity of CIM systems, it is unclear what this common language should be. There is considerable research opportunity in determining a set of assumptions and definitions, syntax, and semantics for a language that can be used to describe integrated circuit processing and fabrication, one that is useful enough to gain wide industry support and is flexible and extensible enough to adapt to future needs. However, this research is beyond the scope of this document. Instead of defining such a universal language, we opt for a simpler solution that allows two different CAFE systems to exchange objects. The **encode-object** method to be described below give one possible implementation of exporting objects between CAFE databases (PFR-based). The **encode-object** method can be easily extended to allow different implementations, i.e., exporting information from a CAFE system into a form such as PROCIS (Process Implementation Service), or into a form understandable by a CIM system that uses BPFL (Berkeley Process Flow Language [6]).

The following subsections will describe the CAFE **encode-object** method. The basic syntax of the generic function **encode-object** method is:

(encode-object <obj> <to-target> <to-domain>)

where:

<obj> is the CAFE Lisp object to be encoded.

<to-target> is target language identifier. This argument is used to specify the particular implementation of the export mechanism to be used. For this document, the only implementation is the CAFE method, so this argument will always appear as **'cafe**.

<to-domain> is the name of the destination domain that the object is being encoded to.

The **encode-object** method primarily dispatches on its <obj> argument. That is, the particular **encode-object** function used in a given situation depends on its *applicability* to the object. There are many references on "method applicability" and generic function programming in Lisp [10]. Figure 13 shows the class hierarchy in CAFE and summarizes the **encode-object** method applicable when given an object of a particular class.

While only the basic mechanisms of **encode-object** is described below, the actual implementation is quite complex in order to allow flexibility in dealing with the CAFE database. The brave reader may wish to consult the source codes to see the "nuts and bolts" of the current implementation.

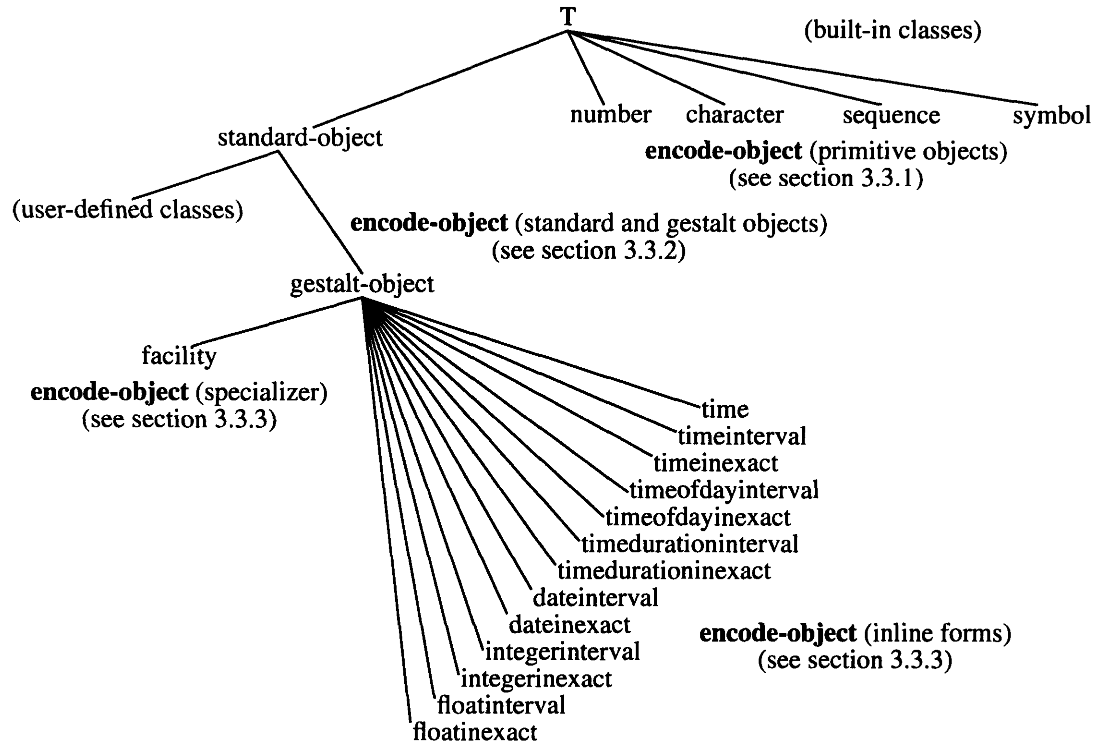

Figure 13. **encode-object** Method Summary

3.3.1 Encoding primitive objects: symbols, strings, numbers

In Lisp, primitive objects include those under the class of symbols, strings, and numbers. Because these objects are largely understood by most computer systems, they are "encoded" into themselves. Here are some examples of the some primitive objects being encoded from the "hoth" domain (running CAFE) to the domain named "wonderland."

```
CAFE>(encode-object 643 'cafe "wonderland")
encoding object 643
done encoding object 643
643

CAFE>(encode-object "BPSG Deposition" 'cafe "wonderland")
encoding object "BPSG Deposition"
done encoding object "BPSG Deposition"
"BPSG Deposition"

CAFE>(encode-object 'status-symbol 'cafe "wonderland")
encoding object STATUS-SYMBOL
done encoding object STATUS-SYMBOL
STATUS-SYMBOL
```

3.3.2 Encoding persistent objects

The real heart of **encode-object** is the encoding of persistent objects in CAFE's database. A simple example will help introduce some of the concepts. Here, we wish to encode a wafertype object called **merklin-sugar-wafer** to the "wonderland" domain:

```
CAFE>(wafertype-with-name "merklin-sugarwafer")
#<WAFERTYPE merklin-sugarwafer 40222460>

CAFE>(setf fwafertype *)
#<WAFERTYPE merklin-sugarwafer 40222460>

CAFE>(describe *)

#<WAFERTYPE merklin-sugarwafer 40222460> is an instance of class #<Gestalt-Class
WAFERTYPE 32000060>:
 The following slots have :INSTANCE allocation:
 %ENTITY                  12476128
 NAME                     "merklin-sugarwafer"
 EPITAXY                  NIL
 SUBSTRATE_DOPANT         "As"
 ORIENTATION              "100"
 SUBSTRATE_RESISTIVITY    #<FLOATINTERVAL 55464720>
 EPI_DOPANT               NIL
 EPI_THICKNESS            NIL
 EPI_RESISTIVITY          NIL

CAFE>(encode-object fwafertype 'cafe "wonderland")
encoding object #<WAFERTYPE merklin-sugarwafer 40222460>
encoding object "merklin-sugarwafer"
done encoding object "merklin-sugarwafer"
encoding object NIL
done encoding object NIL
encoding object "As"
done encoding object "As"
encoding object "100"
done encoding object "100"
encoding object #<FLOATINTERVAL 55464720>
encoding object 10.0S0
done encoding object 10.0S0
encoding object 20.0S0
done encoding object 20.0S0
done encoding object #<FLOATINTERVAL 55464720>    db key: NIL
    return key: (:FLOATINTERVAL (:LOWER 10.0S0) (:UPPER 20.0S0))
encoding object NIL
done encoding object NIL
encoding object NIL
done encoding object NIL
encoding object NIL
done encoding object NIL
done encoding object #<WAFERTYPE merklin-sugarwafer 40222460>
    db key: (WAFERTYPE 27)    return key: (:POINTER WAFERTYPE 1)
(:POINTER WAFERTYPE 1)
```

In the above example, the object **merklin-sugarwafer** is a CAFE **wafertype** object containing the slots identified by name, epitaxy, substrate_dopant, orientation, substrate_resistivity, epi_dopant, epi_thickness, and epi_resistivity, each with a slot value. Judging from the output of **encode-object**, the method starts with the wafertype object and then proceeds to encode the objects in each of the slots, starting with the **name** slot, which contains the string object "merklin-sugarwafer." That is, invoking **encode-object** on **merklin-sugarwafer** recursively invoked **encode-object** again on the string. Because "merklin-sugarwafer" is a primitive object, the result of encoding is the string itself, and **encode-object** then moves on to encoding the next slot called **epitaxy**. Note that the slot called **substrate_resistivity** contains a **floatinterval** object. This is another object that also contains slots of its own:

```
CAFE>(wafertype-substrate_resistivity fwafertype)
#<FLOATINTERVAL 55464720>

CAFE>(describe *)

#<FLOATINTERVAL 55464720> is an instance of class #<Gestalt-Class FLOATINTERVAL
33170160>:
 The following slots have :INSTANCE allocation:
 %ENTITY     12476424
 LOWER       10.0S0
 UPPER       20.0S0
```

Because this object is not a primitive object, **encode-object** recursively invokes itself on **floatinterval** and encodes its slots, which are named **lower** and **upper**. The interesting thing is the return value after encoding the floatinterval, `(:FLOATINTERVAL (:LOWER 10.0S0)(:UPPER 20.0S0))`. Objects of type **floatinterval** are encoded in a special way using specializer methods, and the return values are known as *inline forms.* These **encode-object** specializer methods are discussed later on.

The **encode-object** method continues recursively invoking itself on each of the slot objects until they are all done. The final output after encoding **merklin-sugarwafer** is worth investigating. The db key shows the object's object key as given by the GESTALT database interface. The return key and the final result of **encode-object** show a value of (:POINTER WAFERTYPE 1). This is an encoded form of the wafertype object that is used by this particular implementation of the export tool (i.e., the language syntax of the CAFE **encode-object** method). It consists of a keyword ':POINTER followed by the neutral object key, in this case (WAFERTYPE 1). So the wafertype object has been assigned a neutral object key. Where is the encoded form of the object? The output variable ***encoded-list*** contains the answer:

```
CAFE>*encoded-list*
(((WAFERTYPE 1) (:TIMESTAMP "Wed Jul 26 16:17:04 EST 1995")
  (:SLOTS (NAME "merklin-sugarwafer") (EPITAXY NIL)
          (SUBSTRATE_DOPANT "As") (ORIENTATION "100")
          (SUBSTRATE_RESISTIVITY
              (:FLOATINTERVAL (:LOWER 10.0S0) (:UPPER 20.0S0)))
          (EPI_DOPANT NIL) (EPI_THICKNESS NIL) (EPI_RESISTIVITY NIL))))
```

The ***encoded-list*** is a list consisting of encoded object entries. The above example contains a list of only one entry. The entry consists of the neutral object key, (WAFERTYPE 1), followed by two sections. One section is marked by the keyword :TIMESTAMP, which contains the time the object was encoded. The other section is marked by the keyword :SLOTS, which contains a list of slot names followed by their respective values. Note the inline form under the slot called substrate_resistivity. The ***encoded-list*** provides a form of **merklin-sugarwafer** that is potentially readable by the "wonderland" database. It is in a form that can easily be stored into a text file and transported (by a network, floppy disk, etc.). Assuming that the "wonderland" database knows about the structure of wafertype objects and how to create new ones[2], this ***encoded-list*** captures all the information necessary to reconstruct the object and creating the proper object association.

Let's consider another example. In this case the waferset object **frobozz4** is to be encoded. The waferset **frobozz4** is a waferset containing two wafer objects, "frobozz4,1" and "frobozz4,2."

2. An implicit assumption on the CAFE **encode-object** method is that both databases know the class definition of the objects being transferred. However, because class definitions are also objects in Lisp, they can be encoded via **encode-object** and reconstructed, allowing the receiving database to know about the class structure. But this is a rather twisted way of dealing with unknown object classes and in general one should be careful dealing with these "meta-objects."

```
CAFE>frobozz4
#<WAFERSET frobozz4 55465600>

CAFE>(describe *)

#<WAFERSET frobozz4 55465600> is an instance of class #<Gestalt-Class WAFERSET
36015220>:
 The following slots have :INSTANCE allocation:
 %ENTITY    12244920
 NAME       "frobozz4"
 TIME       #<TIMEINTERVAL 45622600>
 WAFER      (#<WAFER 45622160>
             #<WAFER 45622140>)
 PARENT     NIL
 ADVICE     NIL

CAFE>(waferset-wafer frobozz4)
(#<WAFER 45622160>
 #<WAFER 45622140>)

CAFE>(mapcar #'describe *)

#<WAFER 45622160> is an instance of class #<Gestalt-Class WAFER 36014640>:
 The following slots have :INSTANCE allocation:
 %ENTITY        12438392
 ID             "frobozz4,1"
 TYPE           #<WAFERTYPE merklin-sugarwafer 40222460>
 LASERID        "W1"
 TIME_BROKEN    NIL
#<WAFER 45622140> is an instance of class #<Gestalt-Class WAFER 36014640>:
 The following slots have :INSTANCE allocation:
 %ENTITY        12438808
 ID             "frobozz4,2"
 TYPE           #<WAFERTYPE merklin-sugarwafer 40222460>
 LASERID        "W2"
 TIME_BROKEN    NIL
(NIL NIL)
```

Doing an **encode-object** on **frobozz4** results in recursively invoking **encode-object** on each of its slots before completing the encoding process on **frobozz4**. When the encoding process proceeds into the wafer objects, note that the wafer's type slot is the **merklin-sugarwafer** object which has previously been encoded. Since it has already been encoded and assigned a unique neutral object key, there is no need to do it again. The **encode-object** method is smart enough to detect this case and prevent the re-encoding of **merklin-sugarwafer**. For example, by invoking **encode-object** again on **merklin-sugarwafer** results in:

```
CAFE>(encode-object fwafertype 'cafe "wonderland")
encoding object #<WAFERTYPE merklin-sugarwafer 45621000>
done encoding object #<WAFERTYPE merklin-sugarwafer 45621000>
    db key: (WAFERTYPE 27)    return key: (:POINTER WAFERTYPE 1)
(:POINTER WAFERTYPE 1)
```

The result of ***encoded-list*** is shown below (note that the entry for **merklin-sugarwafer** was carried over from its first encoding):

```
CAFE>*encoded-list*
(((WAFERTYPE 1) (:TIMESTAMP "Wed Jul 26 16:17:04 EST 1995")
  (:SLOTS (NAME "merklin-sugarwafer") (EPITAXY NIL)
          (SUBSTRATE_DOPANT "As") (ORIENTATION "100")
          (SUBSTRATE_RESISTIVITY
              (:FLOATINTERVAL (:LOWER 10.0S0) (:UPPER 20.0S0)))
          (EPI_DOPANT NIL) (EPI_THICKNESS NIL) (EPI_RESISTIVITY NIL)))
 ((WAFER 1) (:TIMESTAMP "Wed Jul 26 17:33:11 EST 1995")
  (:SLOTS (ID "frobozz4,1") (TYPE (:POINTER WAFERTYPE 1))
          (LASERID "W1") (TIME_BROKEN NIL)))
 ((WAFER 3) (:TIMESTAMP "Wed Jul 26 17:33:12 EST 1995")
  (:SLOTS (ID "frobozz4,2") (TYPE (:POINTER WAFERTYPE 1))
          (LASERID "W2") (TIME_BROKEN NIL)))
 ((WAFERSET 1) (:TIMESTAMP "Wed Jul 26 17:33:12 EST 1995")
  (:SLOTS (NAME "frobozz4")
          (TIME (:TIMEINTERVAL
                    (:LOWER (:TIME (:DATE "07/07/95")
                                   (:TIMEOFDAY "11:40:40")))
                    (:UPPER NIL)))
          (WAFER (:LIST (:POINTER WAFER 1) (:POINTER WAFER 3)))
          (PARENT NIL) (ADVICE NIL))))
```

In the encoded form of the waferset **frobozz4**, the wafer slot contains another type of syntax. This syntax is the LIST syntax, identified by the keyword :LIST and followed by the elements of the list. This list contains two elements which happen to be :POINTER forms. These are pointers to other encoded entries in ***encoded-list***, namely the entries for neutral object keys (wafer 1) and (wafer 3). Another example of encoding lists of objects is:

```
CAFE>(encode-object (list 1 2 3 'a 'b 'c) 'cafe "wonderland")
encoding object (1 2 3 A B C)
encoding object 1
done encoding object 1
encoding object 2
done encoding object 2
encoding object 3
done encoding object 3
encoding object A
done encoding object A
encoding object B
done encoding object B
encoding object C
done encoding object C
done encoding object (1 2 3 A B C)
(:LIST 1 2 3 A B C)
```

Note that the encoding of the waferset, wafer, and wafertype objects above made use of neutral object keys, and so have object associations. The **encode-object** method automatically updates the ***object-assoc-list*** for object associations and the ***object-database*** for supplementary information:

```
CAFE>*object-assoc-list*
(((:TO-DOMAIN "wonderland")
  ((WAFERTYPE 27) IS-ASSOCIATED-WITH (WAFERTYPE 1))
  ((WAFERSET 3881) IS-ASSOCIATED-WITH (WAFERSET 1))
  ((WAFER 9731) IS-ASSOCIATED-WITH (WAFER 1))
  ((WAFER 9732) IS-ASSOCIATED-WITH (WAFER 3))))
```

The exact details of how **encode-object** works can be found in its source file. The reader is encouraged to explore its language syntax and to make extensions should the need arise.

3.3.3 Specialized encoding of particular objects

The encoding examples above showed that while many objects in the CAFE database can be handled by the recursive nature of **encode-object**, there are a few types of objects that bypass this generalized encoding. The primitive objects like numbers and strings are one such class. There are also objects like the floatinterval and timeinterval that, while they could be encoded using the above mechanism, are handled in a different way for efficiency reasons, or because of some unique behavior of the object.

The floatinterval and timeinterval encoded forms are known as *inline* forms. Inline forms are used for certain objects because these objects do not need the full power of **encode-object** and so are more efficiently encoded into a more compact form. The objects in CAFE that are encoded into an inline form are: **time**, **timeinterval**, **timeinexact**, **timeofdayinterval**, **timeofdayinexact**, **timedurationinterval**, **timeduration-inexact**, **dateinterval**, **dateinexact**, **integerinterval**, **integerinexact**, **floatinterval**, and **floatinexact**. Many of these objects share certain properties (intervals and inexact quantities).

There is one CAFE object that is encoded using specialized behavior — the **facility** object. An example of a facility object is the object associated with MIT's Integrated Circuits Lab (ICL):

```
CAFE>(facility-with-name "ICL")
#<FACILITY ICL 45621240>

CAFE>(describe *)

#<FACILITY ICL 45621240> is an instance of class #<Gestalt-Class FACILITY
36202340>:
 The following slots have :INSTANCE allocation:
 %ENTITY          6211400
 NAME             "ICL"
 LOCATION         "Building 39 (2nd and part of 4th floors)"
 DESCRIPTION      "Integrated Circuits Fabrication Lab"
 TIMEAVAILABLE    (#<TIMEAVAILABLE 45621040>
                   ....)
```

The list under the timeavailable slot has been truncated because of its extensive number of elements. The **encode-object** method for **facility** objects behaves normally as the generalized **encode-object** method except that the timeavailable slot is not recursed into and instead encoded as a null list. This was done because of pragmatic reasons: the timeavailable slot typically contained a large number of **timeavailable** objects that had no real use in the current CAFE processing model, and was a significant overhead in the encoding process. Because there was no real need to include such information in the description of the facility object, the encoding of the timeavailable slot was skipped. If at a later time such information becomes important this specializer method can be removed. From the rules of method applicability, **facility** objects will then be encoded using the generalized recursive **encode-object** function.

3.4 Import Tool: The decode-object methods

This subsection deals with the **decode-object** methods, the mechanism for *importing* information from an encoded form into a database. An important point is that for a given encoder there needs to be an associated decoder. In general, the encoder and decoder will be specific to the CIM system's architecture as well as the specifications of the encoded language, since each system may have different database engines, models, and language.

Figure 14 shows some possibilities of encoder/decoder configurations between two sites. In Figure 14(a), both sites A and B utilize the same encoded form 1, so they both use an encoder that exports information to this form and a decoder that imports information from this form. Since the sites have different databases, they each have their own specific implementation of the encoder and decoder.

Figure 14(b) shows another possibility for information exchange. In this case, site A communicates with site B using encoded form 1, while site B communicates with site A using a different encoded form 2. This is entirely acceptable as long as both sites are willing to deal with two different encoded forms for transferring object information.

For this particular document, we can get away with the configuration shown in Figure 14(c). Both sites A and B are using the same CAFE system with the same database engine, model, and language, and operating on equivalent platforms. The **decode-object** method described here is the associated decoder for the encoder discussed in the previous subsection.

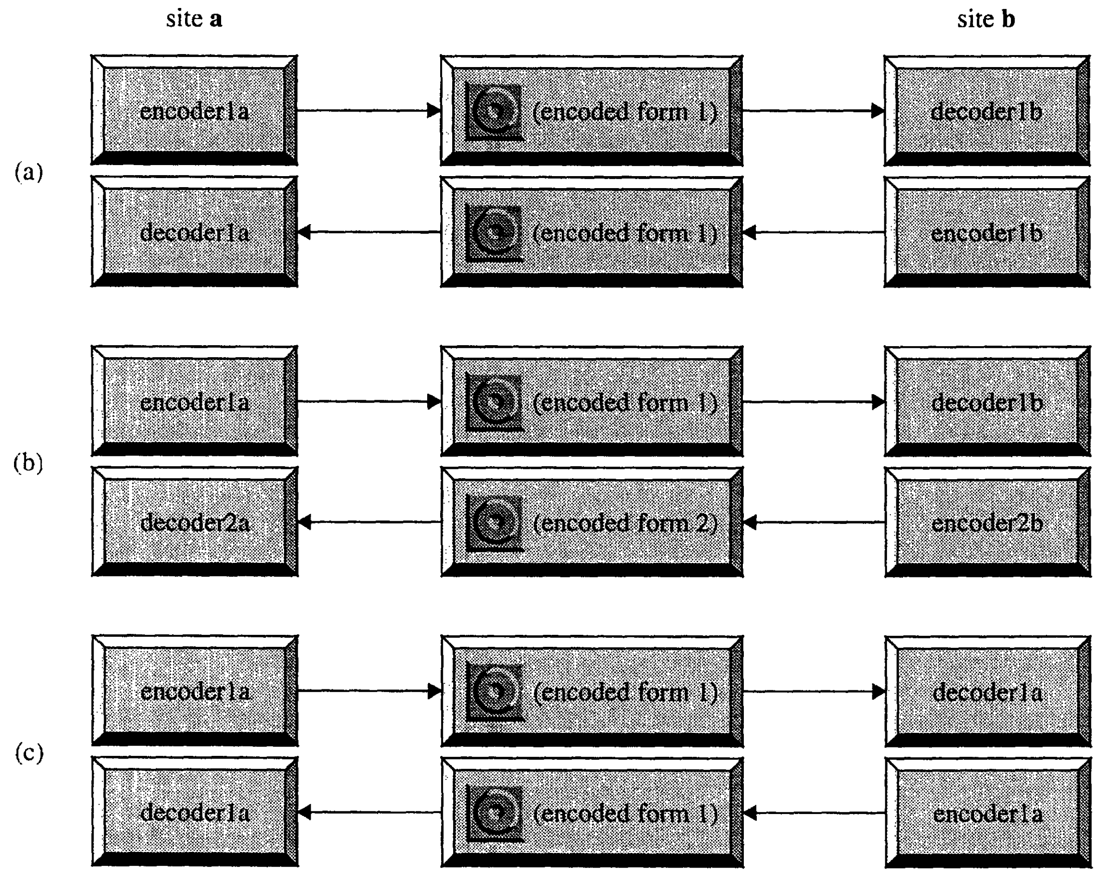

Figure 14. Possible Encoder/Decoder Configurations

It turns out that the decoder implemented for the CAFE system is significantly more complex than the encoder. There are more issues to resolve when creating objects in a database using **decode-object** than in converting existing objects into an encoded form using **encode-object**. These issues include introducing new virtual objects into a database (and the relative security of incorporating foreign information), modifying existing objects, and reusing equivalent objects. Some objects undergo renaming such as the **wafertype**, **waferset**, **wafer**, **lot**, while some objects undergo a class transformation such as **labuser** and **virtuallabuser** objects. There are also special issues related to **facility**, **machine**, and **virtualmachine** objects. These specialized decoder methods are used to handle CAFE's remote fabrication model and will be better understood in Part 3.

For example, when a **labuser** object is being incorporated into another CAFE database, a special **decode-**

object method is used to first check whether or not the lab user already exists. This is common since a person may log into several different servers and use the same user name on each of them. If the same **labuser** does indeed exist in the database receiving the encoded **labuser**, the existing **labuser** object is used instead of the incorporating a new one. If the **labuser** doesn't exist, it gets incorporated as a **virtuallabuser** object. The **virtuallabuser** class is equivalent to the **labuser** class but has slightly different schema properties.

Another example is translation between **machine** objects and **virtualmachine** objects. In remote fabrication the designer needs access to machines that don't exist in his local facility. This access is through **virtualmachine** objects in the CAFE database. However, at the remote facility these virtual machines are really local to its own site and so are accessed by **machine** objects. The **decode-object** specializer method handles this translation.

The reader is encouraged to experiment with the CAFE **decode-object** methods and to examine the mechanisms of the database object constructor and mutator/modifier methods. There are also additional features not described here, including the automatic remote query capability of the decoder when a decode operation cannot be completed.

3.4.1 Decoding primitive encoded objects

Decoding the primitive encoded objects are simple, since the encoded forms evaluate to themselves. Here are some examples of the domain "wonderland" decoding the objects sent by domain "hoth":

```
CAFE>(decode-object 643 'cafe "hoth")
decoding object 643 from target CAFE, domain hoth
done decoding object 643  value: 643
643

CAFE>(decode-object "BPSG Deposition" 'cafe "hoth")
decoding object "BPSG Deposition" from target CAFE, domain hoth
done decoding object "BPSG Deposition"  value: "BPSG Deposition"
"BPSG Deposition"

CAFE>(decode-object 'status-symbol 'cafe "hoth")
decoding object STATUS-SYMBOL from target CAFE, domain hoth
done decoding object STATUS-SYMBOL  value: STATUS-SYMBOL
STATUS-SYMBOL
```

3.4.2 Decoding persistent encoded objects

Decoding the encoded forms of persistent objects is also relatively straightforward. Let's assume that the encoded information was somehow transferred to this database (in "wonderland") and the result was placed into the ***encoded-list***. **decode-object** uses the information contained in this variable to reconstruct the objects. Suppose we wanted to reconstruct the **merklin-sugarwafer** object. The syntax of **decode-object** is similar to **encode-object**, except instead of using an <obj> argument we use the :POINTER form on the neutral object key (wafertype 1)[3].

```
CAFE>(decode-object '(:pointer wafertype 1) 'cafe "hoth")
decoding object (:POINTER WAFERTYPE 1) from target CAFE, domain hoth
encoded object type: :POINTER
decode-object-listform (:POINTER WAFERTYPE 1) -------- start
decode-object-listform: (WAFERTYPE 1) found in encoded list
decoding object ((WAFERTYPE 1)
                 (:TIMESTAMP "Wed Jul 26 16:17:04 EST 1995")
                 (:SLOTS (NAME "merklin-sugarwafer") (EPITAXY NIL)
```

3. The output of **decode-object** is rather long even for this simple example, so this is the only example that will contain full output.

```
                         (SUBSTRATE_DOPANT "As") (ORIENTATION "100")
                         (SUBSTRATE_RESISTIVITY
                             (:FLOATINTERVAL (:LOWER 10.0S0)
                                 (:UPPER 20.0S0)))
                         (EPI_DOPANT NIL) (EPI_THICKNESS NIL)
                         (EPI_RESISTIVITY NIL))) from target CAFE, domain hoth
encoded object type: (WAFERTYPE 1)
(decode-object-introduce-new-virtual-db-object)
decoding object (:SLOTS (NAME "merklin-sugarwafer") (EPITAXY NIL)
                        (SUBSTRATE_DOPANT "As") (ORIENTATION "100")
                        (SUBSTRATE_RESISTIVITY
                            (:FLOATINTERVAL (:LOWER 10.0S0)
                                (:UPPER 20.0S0)))
                        (EPI_DOPANT NIL) (EPI_THICKNESS NIL)
                        (EPI_RESISTIVITY NIL)) from target CAFE, domain hoth
encoded object type: :SLOTS
decode-object-listform (:SLOTS (NAME "merklin-sugarwafer")
                               (EPITAXY NIL) (SUBSTRATE_DOPANT "As")
                               (ORIENTATION "100")
                               (SUBSTRATE_RESISTIVITY
                                   (:FLOATINTERVAL (:LOWER 10.0S0)
                                    (:UPPER 20.0S0)))
                               (EPI_DOPANT NIL) (EPI_THICKNESS NIL)
                               (EPI_RESISTIVITY NIL)) -------- start
decoding object "merklin-sugarwafer" from target CAFE, domain hoth
done decoding object "merklin-sugarwafer"  value: "merklin-sugarwafer"
decoding object NIL from target CAFE, domain hoth
done decoding object NIL  value: NIL
decoding object "As" from target CAFE, domain hoth
done decoding object "As"  value: "As"
decoding object "100" from target CAFE, domain hoth
done decoding object "100"  value: "100"
decoding object (:FLOATINTERVAL (:LOWER 10.0S0) (:UPPER 20.0S0))
 from target CAFE, domain hoth
encoded object type: :FLOATINTERVAL
decode-object-listform (:FLOATINTERVAL (:LOWER 10.0S0) (:UPPER 20.0S0)) --------
start
decoding object 10.0S0 from target CAFE, domain hoth
done decoding object 10.0S0  value: 10.0S0
decoding object 20.0S0 from target CAFE, domain hoth
done decoding object 20.0S0  value: 20.0S0
decode-object-listform (:FLOATINTERVAL (:LOWER 10.0S0) (:UPPER 20.0S0)) --------
end
done decoding object (:FLOATINTERVAL (:LOWER 10.0S0) (:UPPER 20.0S0))  value:
#<FLOATINTERVAL 46702600>
decoding object NIL from target CAFE, domain hoth
done decoding object NIL  value: NIL
decoding object NIL from target CAFE, domain hoth
done decoding object NIL  value: NIL
decoding object NIL from target CAFE, domain hoth
done decoding object NIL  value: NIL
decode-object-listform (:SLOTS (NAME "merklin-sugarwafer")
                               (EPITAXY NIL) (SUBSTRATE_DOPANT "As")
                               (ORIENTATION "100")
                               (SUBSTRATE_RESISTIVITY
                                   (:FLOATINTERVAL (:LOWER 10.0S0)
                                    (:UPPER 20.0S0)))
                               (EPI_DOPANT NIL) (EPI_THICKNESS NIL)
                               (EPI_RESISTIVITY NIL)) -------- end
```

```
done decoding object (:SLOTS (NAME "merklin-sugarwafer") (EPITAXY NIL)
                             (SUBSTRATE_DOPANT "As")
                             (ORIENTATION "100")
                             (SUBSTRATE_RESISTIVITY
                                 (:FLOATINTERVAL (:LOWER 10.0S0)
                                     (:UPPER 20.0S0)))
                             (EPI_DOPANT NIL) (EPI_THICKNESS NIL)
                             (EPI_RESISTIVITY NIL))
  value: ((NAME "merklin-sugarwafer") (EPITAXY NIL)
          (SUBSTRATE_DOPANT "As")
          (ORIENTATION "100")
          (SUBSTRATE_RESISTIVITY #<FLOATINTERVAL 46702600>)
          (EPI_DOPANT NIL) (EPI_THICKNESS NIL)
          (EPI_RESISTIVITY NIL))
(decode-object-construct-object WAFERTYPE specializer)
**constructing remote wafertype object**
slotized-ds: (:NAME "merklin-sugarwafer" :EPITAXY NIL :SUBSTRATE_DOPANT
              "As" :ORIENTATION "100" :SUBSTRATE_RESISTIVITY
              #<FLOATINTERVAL 46702600> :EPI_DOPANT NIL
              :EPI_THICKNESS NIL :EPI_RESISTIVITY NIL)
new slotized-ds: (:NAME "hoth:merklin-sug" :EPITAXY NIL
                  :SUBSTRATE_DOPANT "As" :ORIENTATION "100"
                  :SUBSTRATE_RESISTIVITY
                  #<FLOATINTERVAL 46702600> :EPI_DOPANT NIL
                  :EPI_THICKNESS NIL :EPI_RESISTIVITY NIL)
done decoding object ((WAFERTYPE 1)
                      (:TIMESTAMP "Wed Jul 26 16:17:04 EST 1995")
                      (:SLOTS (NAME "merklin-sugarwafer") (EPITAXY NIL)
                              (SUBSTRATE_DOPANT "As")
                              (ORIENTATION "100")
                              (SUBSTRATE_RESISTIVITY
                                  (:FLOATINTERVAL (:LOWER 10.0S0)
                                      (:UPPER 20.0S0)))
                              (EPI_DOPANT NIL) (EPI_THICKNESS NIL)
                              (EPI_RESISTIVITY NIL)))  value: #<WAFERTYPE
hoth:merklin-sug 40222500>
decode-object-listform (:POINTER WAFERTYPE 1) -------- end
done decoding object (:POINTER WAFERTYPE 1)  value: #<WAFERTYPE hoth:merklin-sug
40222500>
#<WAFERTYPE hoth:merklin-sug 40222500>
```

Basically, **decode-object** finds the entry for (wafertype 1) in the ***encoded-list*** and recursively invokes itself on the encoded slot list by retrieving the :SLOT entry for **merklin-sugarwafer**. After decoding the values for the slots, **decode-object** finally creates a new object and returns a wafertype object curiously labelled as **hoth:merklin-sug**. This wafertype contains the same information as **merklin-sugarwafer** except that it has been renamed. Not all objects undergo a renaming procedure, and those objects that do are done using specialized decoder methods. The main point to be gathered here is that **decode-object** works in a similar way to **encode-object** by recursively invoking itself on slots. Note that **decode-object** also automatically updates its object association lists. Because the wafertype object originated at the "hoth" domain, the object association entry is placed in ***virtual-object-assoc-list***:

```
CAFE>*virtual-object-assoc-list*
(((:FROM-DOMAIN "hoth")
  ((WAFERTYPE 63) IS-ASSOCIATED-WITH (WAFERTYPE 1))))
```

Observe that here in "wonderland" the wafertype object has an object key of (wafertype 63) while in "hoth" the object key was (wafertype 27). In both domains, the neutral object key for the wafertype object is (wafertype 1).

3.4.3 Decoding specialized encoded objects

Just as there are specialized encoded forms from specialized **encode-object** methods, there are also the associated specialized **decode-object** methods. The inline forms (such as **time**, **timeinterval**, and **timeinexact**) encoded with a specific method each have a specific decoder method to properly reconstruct the original object. The example below shows the decoding of an inline **time** object:

```
CAFE>(decode-object '(:TIME (:DATE "08/07/95")
                            (:TIMEOFDAY "15:40:20"))
      'cafe "hoth")
decoding object (:TIME (:DATE "08/07/95") (:TIMEOFDAY "15:40:20")) from target
CAFE, domain hoth
encoded object type: :TIME
decode-object-listform (:TIME (:DATE "08/07/95")
                              (:TIMEOFDAY "15:40:20")) -------- start
decoding object "08/07/95" from target CAFE, domain hoth
done decoding object "08/07/95"  value: "08/07/95"
decoding object "15:40:20" from target CAFE, domain hoth
done decoding object "15:40:20"  value: "15:40:20"
decode-object-listform (:TIME (:DATE "08/07/95")
                              (:TIMEOFDAY "15:40:20")) -------- end
done decoding object (:TIME (:DATE "08/07/95") (:TIMEOFDAY "15:40:20"))
  value: #<TIME 46702560>
#<TIME 46702560>
```

Also, like the specialized **encode-object** method for **facility** objects, there is a specialized **decode-object** method that will create the **facility** object even when the timeavailable slot is unspecified.

4 Remote Message Passing Tools

In section 3 we divided the VOM into three major modules: the object management module, the export/import module, and the remote message passing module. The object management module and the export/import module are two of the major modules that were discussed in that section. The third important module in the VOM is the remote message passing module. The remote message passing module consists of a suite of tools that provide the mechanism for sending and receiving messages to and from different sites.

There are a variety of mechanisms that can be used to implement the remote message passing module. Information can be written to a file and transmitted by electronic mail or floppy disk. The particular implementation described in this section transfers data through an ftp-like (file transfer protocol) connection via TCP/IP (transmission control protocol/internet protocol) sockets. The architecture of this particular message passing implementation is shown in Figure 15. There are three general layers: the low-level layer, the mid-level layer, and the high-level layer. The low-level layer contains system routines written in C, borrowing from the work done in [7] and [8]. The low-level layer also contains Lisp wrapper functions to the system routines and acts as the low-level Lisp interface. The mid-level layer contains the message passing tools useful for CAFE applications. The high-level interface contains some message passing functions that might serve as applications that use the message passing interface, or as additional tools for other CAFE applications to build upon.

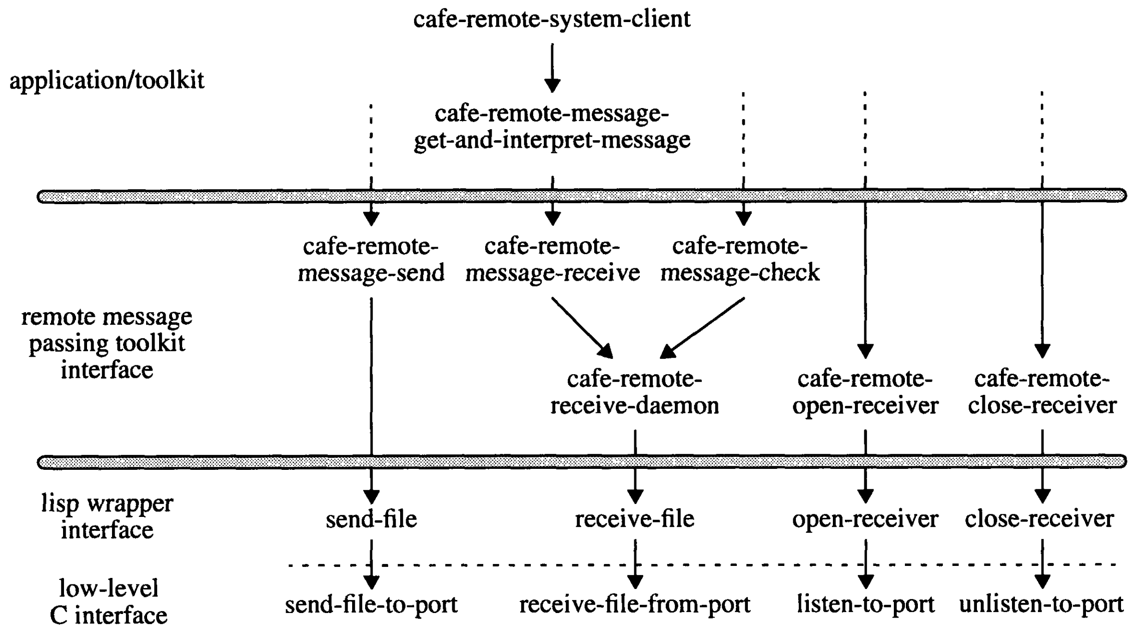
cafe-remote-system-client
application/toolkit
cafe-remote-message-get-and-interpret-message
cafe-remote-message-send
cafe-remote-message-receive
cafe-remote-message-check
remote message passing toolkit interface
cafe-remote-receive-daemon
cafe-remote-open-receiver
cafe-remote-close-receiver
lisp wrapper interface
send-file
receive-file
open-receiver
close-receiver
low-level C interface
send-file-to-port
receive-file-from-port
listen-to-port
unlisten-to-port

Figure 15. Remote Message Passing Module Architecture

4.1 The Low-Level Message Passing Interface

The low-level system routines are written in C and interface to the operating system TCP/IP socket libraries. The four executables that provide the bare bones support are: **send-file-to-port**, **receive-file-from-port**, **listen-to-port**, and **unlisten-to-port**. The commands **listen-to-port** starts a process that opens a TCP/IP port and "listens" for messages to arrive. If the port receives a valid message (according to the message passing protocols set by **listen-to-port**) then the message can be retrieved by **receive-file-from-port**. Messages can be sent to a particular host through a port by using **send-file-to-port**. Finally, **unlisten-to-port** is used to close the port connection and return the resource to the operating system.

The low-level Lisp interface consists of Lisp wrapper functions that mirror the system routines. These functions are **send-file**, **receive-file**, **open-receiver**, and **close-receiver**, respectively, and have similar argument syntaxes as the associated executables. The mid-level message passing routines use these Lisp functions to gain access to the system routines.

4.2 CAFE Message Passing Tools: the Mid-Level Interface

The mid-level interface consists of routines that are used by applications in the CAFE Application Layer. It consists of two sublayers. In the lower sublayer are generic open and close receiver functions, open and close sender functions, and a message-receiving daemon that acts as a server for receiving and buffering incoming messages. In the upper sublayer are the message sending and receiving methods and a function to check for new messages.

In general, the communications protocol can be described as initially opening up a channel for the communication to take place, sending a message through the open communications channel, and then closing the channel when communications have terminated. At the receiving end, the communications protocol involves opening up the channel, checking if a message exists and if it does, receiving the message, and then after receiving the message closing the channel. For example, when a person wishes to talk to someone else via telephone, the person removes the telephone speaker and microphone assembly off a hook switch, waits for a dial tone and then inputs a telephone number. At the other end of the communications channel, the telephone rings to signal that someone wishes to deliver a message. The person then removes the telephone off the hook and delivers an introductory response. Once both parties have given their greetings, active communication may take place. When both parties are done talking they give their concluding remarks and return the telephone to the hook switch, terminating the connection. In this example the acts of activating the telephone, dialing numbers, and speaking a greeting phrase may be designated as opening the connection. The active communications between the parties may be designated as sending and receiving messages. The termination of the conversation and the returning of the telephone onto the telephone hook switch may be designated as the closing of the connection. In the CAFE system, the communication is done using the internet. The opening and closing procedures involve setting up TCP/IP ports or sockets.

4.2.1 Opening and Closing the sender and receiver

The functions for opening and closing a receiver port are **cafe-remote-open-receiver** and **cafe-remote-close-receiver**, respectively. The function **cafe-remote-open-receiver** takes a single argument, the port number. When invoked, **cafe-remote-open-receiver** opens the specified port for receiving by spawning a process that listens for messages on the port. It also registers the open receiving port into a global variable called ***active-cafe-ports***, the active cafe ports list that contains status information about all the open ports that are being used for CAFE. Once the port is open for receiving, messages may be transmitted into CAFE through the specified port.

The function **cafe-remote-close-receiver** takes a single port number argument. When invoked, **cafe-remote-close-receiver** closes the specified port by running a process that does the internal details of closing a TCP/IP socket and then destroying the listening process spawned by **cafe-remote-open-receiver**. It also unregisters the port from ***active-cafe-ports***.

In the present implementation of the remote message passing module, there are no explicit CAFE functions for opening and closing a connection when sending information, i.e., there are no functions **cafe-remote-open-sender** and **cafe-remote-close-sender**. The mechanism for opening and closing a port during a send operation is taken care of by the low-level interface.

4.2.2 The CAFE Remote Message Daemon

The CAFE Remote Message Daemon is a function that acts as a server for receiving messages and buffering them. In light of the example of the telephone conversation, the Message Daemon might be an answering machine that stores incoming messages and organizes them for later retrieval. Likewise, the CAFE Remote Message Daemon awaits incoming messages on ports that are open for receiving (registered in ***active-cafe-ports***) and if it detects a message it checks to see if the message is a valid CAFE message. Valid messages are those that have a certain format so that CAFE may be able to distinguish them from incoming garbage or messages that are intended for some other application not in CAFE. By using the Message Daemon, CAFE applications don't need to explicitly open up receiving ports, check for valid messages, and buffer them: they simply ask the Message Daemon for messages. An example of running the Message Daemon is:

```
CAFE>(cafe-remote-open-receiver :port 8186)
receiver port 8186 successfully opened.
OPENED

CAFE>(cafe-remote-open-receiver :port 8187)
receiver port 8187 successfully opened.
OPENED

CAFE>(cafe-remote-receive-daemon)
cafe-remote-receive-daemon initiated Fri Aug  4 10:19:04 EST 1995
press <return> to exit
```

The ***active-cafe-ports*** contains status information of each port such as when a receiving port was opened and a list of channels that have messages awaiting.

```
CAFE>*active-cafe-ports*
((8187 (:INITIATED "Fri Aug  4 10:18:49 EST 1995")
  (:FILECOUNTS ("dev" (:LASTFILECOUNT 3))
      ("frobozz-dev" (:LASTFILECOUNT 0))
      ("frobozz2" (:LASTFILECOUNT 1)) ("frobozz3" (:LASTFILECOUNT 2))
      ("frobozz4" (:LASTFILECOUNT 0)) ("moomoo" (:LASTFILECOUNT 13))
      ("system" (:LASTFILECOUNT 0)) ("test-machine" (:LASTFILECOUNT 0))
      ("wpmoyne2" (:LASTFILECOUNT 0))))
 (8186 (:INITIATED "Fri Aug  4 10:18:36 EST 1995")
  (:FILECOUNTS ("NIL" (:LASTFILECOUNT 1))
      ("channel1" (:LASTFILECOUNT 205)) ("channel2" (:LASTFILECOUNT 3))
      ("dev" (:LASTFILECOUNT 2)) ("frobozz-dev" (:LASTFILECOUNT 0))
      ("frobozz2" (:LASTFILECOUNT 0)) ("frobozz4" (:LASTFILECOUNT 0))
      ("moomoo" (:LASTFILECOUNT 2)) ("parking-lot" (:LASTFILECOUNT 0))
      ("scrap" (:LASTFILECOUNT 0))
      ("superduper-secretpooper-channel" (:LASTFILECOUNT 0))
      ("system" (:LASTFILECOUNT 0)) ("unknown" (:LASTFILECOUNT 2))
      ("wpmoyne2" (:LASTFILECOUNT 0)) ("testing" (:LASTFILECOUNT 1)))))
```

4.2.3 Sending and Receiving messages

The message sending function is **cafe-remote-message-send** and takes as arguments the remote message (which can be any form that can be evaluated by the Lisp evaluator, i.e., strings, lists, symbols, numbers), the destination TCP/IP host name (such as "garcon.mit.edu" or "18.62.0.242"), the port on the destination host to send the message through, and a channel name designation. Because a CAFE receiver port can receive messages from a variety of other hosts, the channel name serves to help determine which message came from a particular host and to better sort the incoming messages. An example of sending a simple greeting message is shown below:

```
CAFE>(cafe-remote-message-send "Hello There!" 'cafe "garcon.mit.edu"
                               :port 8186 :channel "testing")
:REMOTE-MESSAGE "testing"
Message sent.
"Hello There!"SENT
```

A short time later, the CAFE Remote Message Receive Daemon on a far away host listening in on port 8186 receives the message, validates that the message was intended for CAFE, and then buffers the message in an appropriate place:

```
CAFE>(cafe-remote-receive-daemon)
cafe-remote-receive-daemon initiated Fri Aug  4 10:19:04 EST 1995
press <return> to exit
receive on port 8186, channel "testing": /homes/merklin/src/lisp/remote/remote-
messages/8186/testing/incoming-file-0001
```

When an application wishes to check for any messages on the "testing" channel, it can use **cafe-remote-message-check**:

```
CAFE>(cafe-remote-message-check 'cafe :port 8186 :channel "testing")
T
```

Since a message is indeed waiting, the application can then grab the contents of the message:

```
CAFE>(cafe-remote-message-receive 'cafe :port 8186 :channel "testing")
"Hello There!"
```

The :port keyword argument is not necessary under normal circumstances. Each CAFE system may be given a default CAFE receiving port number stored in ***default-receive-port***.

4.2.4 Message Packaging routines

While the remote message passing routines above may be used to send a variety of information, a standardized way of formatting the message is needed for remote fabrication applications. One possible format is for the message to have a header containing routing information followed by a body containing the message "payload." A standardized header can be generated using **cafe-remote-message-standard-header**:

```
CAFE>(cafe-remote-message-standard-header :from-channel "testing" :comments
"This is an example header")
(:HEADER (:FROM (:DBDOMAIN "garcon1") (:DOMAIN "garcon.mit.edu")
                (:CHANNEL "testing") (:PORT 8186)
                (:HEADER-PACKAGED-ON "Fri Aug  4 10:46:15 EST 1995")
                (:HEADER-PACKAGED-BY "merklin"))
         (:SEND-REPLY-TO (:DBDOMAIN "garcon1")
             (:DOMAIN "garcon.mit.edu") (:CHANNEL "testing")
             (:PORT 8186))
         (:COMMENTS "This is an example header"))
```

Note that this header is different from that generated and used by **cafe-remote-message-send** and **cafe-remote-message-receive**, which automatically have their own formats and attach their own headers. In fact, as the message travels through the various layers in the OSI (Open Systems Interconnect) network model, additional headers and message formattings are added. OSI is a seven-layer model in which the top layer (the Application Layer) is the application software itself. This top layer uses the above header information. The lower six layers implement the network mechanisms (from top to bottom: Presentation Layer, Session Layer, Transport Layer, Network Layer, Data Link Layer, and the Physical Layer).

A typical message used by the remote fabrication system in CAFE might look like below. Here the user receives a reply from a remote facility that the processing status of one of his lots is on the operational step for LPCVD polysilicon deposition:

```
((:HEADER (:FROM (:DBDOMAIN "garcon2") (:DOMAIN "garcon.mit.edu")
                 (:CHANNEL "system") (:PORT 8186)
                 (:HEADER-PACKAGED-ON "Mon Aug  7 15:07:48 EST 1995")
                 (:HEADER-PACKAGED-BY "merklin")
                 (:COMMENTS "sample message"))
          (:SEND-REPLY-TO (:DBDOMAIN "garcon2")
              (:DOMAIN "garcon.mit.edu") (:CHANNEL "system")
              (:PORT 8187) (:COMMENTS "..."))
          (:COMMENTS
              "this message is an automatic remote response generated by the
               system client"))
 (:MESSAGE-REPLIES
     ((PROCESSING-STATUS (LOT 3))
      "lot sample-lot is currently on task LPCVD-POLYSILICON")))
```

4.3 The High-Level Interface

The high-level interface makes use of the message sending, receiving, and packaging routines for the task of doing remote fabrication. The two examples below describe one way how applications can automatically construct messages, send them to the appropriate domains, and interpret incoming messages.

The example functions make use of a host table address list contained in the global variable ***host-table-address-list***. The host table address list contains entries on all the domains known to the CAFE system that can perform remote processing of integrated circuits. Each entry contains the domain's name, its full host name and advertised receiving port, plus user information which may contain a brief description of the facility.

4.3.1 An Example Remote Message Interpreter

The function **cafe-remote-message-get-and-interpret-message** can be used to receive an incoming CAFE message and automatically parse its contents. Messages for remote fabrication applications have six distinct sections that can be interpreted by **cafe-remote-message-get-and-interpret-message**: the header section, the database section, the events section, the requests section, the replies section, and the error section.

The header section is marked by a list beginning with the keyword **:header**. The format of the header is the same as that generated by **cafe-remote-message-standard-header**. The function **cafe-remote-message-get-and-interpret-message** uses the information contained in the header to determine where the message came from and where to send messages in case a reply is needed.

The database section is marked by the keyword **:database**. This section houses the encoded object information generated by **encode-object** described in Section 3. The database section is essentially a copy of ***encoded-list***.

The events section is marked by the keyword **:message-events**. This section is used to contain messages related to events that have occurred in a facility, such as a critical machine going down for repairs or updates on a particular process flow.

The requests section begins with the keyword **:message-requests**. This section is used by CAFE applications that wish to query a remote facility for some information, such as the processing status of a particular lot. The replies section, marked by **:message-replies**, is used to give return messages to such requests.

The error section, headed by **:message-error**, is used to transmit information of a critical nature. Remote messages with this section may be given higher priority in interpretation than other remote messages.

4.3.2 The CAFE Remote Message System Client

The CAFE Remote Message System Client is a separate process that listens for messages on a specific channel called "system." This System Client takes these system messages, interprets their contents (using the mechanism from **cafe-remote-message-get-and-interpret-message**), and automatically generates appropriate replies, requests, events, and errors. The example below shows a sample session of the System Client on domain "garcon2" (on garcon.mit.edu with receiving port 8187) receiving a message sent from a remote CAFE system (with a domain name of "garcon1" on garcon.mit.edu with a receiving port 8186) querying about the processing status for the lot named "frobozz." The System Client parses the message request and constructs a return reply message back to "garcon1."

```
CAFE>(cafe-remote-system-client)
cafe-remote-system-client initiated Fri Aug  4 10:42:51 EST 1995
press <return> to exit
Fri Aug  4 15:34:31 EST 1995 ==== start system message ====
Fri Aug  4 15:34:31 EST 1995 ==== start header parsing ====
  from dbdomain garcon1 domain garcon.mit.edu  channel frobozz5  port 8186
  header packaged on Fri Aug  4 15:34:24 EST 1995 by merklin
  send to dbdomain garcon1 domain garcon.mit.edu  channel frobozz5  port 8186
header comments: "remote query generated by operate-machine (remote)"
Fri Aug  4 15:34:31 EST 1995 ====  end  header parsing ====
Fri Aug  4 15:34:31 EST 1995 ==== start requests parsing ====
Fri Aug  4 15:34:31 EST 1995 ==>
           (CAFE-REMOTE-SYSTEM-CLIENT-REQUEST-PROCESSING-STATUS
            (LOT 5) :DOMAIN "garcon1" :CHANNEL "frobozz5")
locking /homes/merklin/src/lisp/remote/garcon2/virtual-object-assoc-
list...locked.
unlocking /homes/merklin/src/lisp/remote/garcon2/virtual-object-assoc-
list...unlocked.
```

```
locking /homes/merklin/src/lisp/remote/garcon2/object-assoc-list...locked.
unlocking /homes/merklin/src/lisp/remote/garcon2/object-assoc-list...unlocked.
  result: (:RVAL "lot frobozz5 is currently on task NIL")
:REMOTE-MESSAGE "frobozz5"
((:HEADER (:FROM (:DBDOMAIN "garcon2") (:DOMAIN "garcon.mit.edu")
                 (:CHANNEL "system") (:PORT 8186)
                 (:HEADER-PACKAGED-ON "Fri Aug  4 15:34:31 EST 1995")
                 (:HEADER-PACKAGED-BY "merklin")
                 (:COMMENTS "simple remote query example"))
          (:SEND-REPLY-TO (:DBDOMAIN "garcon2")
              (:DOMAIN "garcon.mit.edu") (:CHANNEL "system")
              (:PORT 8187) (:COMMENTS "the return address for this query"))
          (:COMMENTS
              "this message is an automatic remote response generated by the
system client"))
 (:MESSAGE-REPLIES
     ((PROCESSING-STATUS (LOT 5))
Message sent.
      "lot frobozz5 is currently on task Chocolate Chip Deposition")))
Fri Aug  4 15:34:32 EST 1995 ====  end  requests parsing ====
Fri Aug  4 15:34:32 EST 1995 ====  end  system message ====
```

Part 3

Remote Fabrication Application Support

5 The CAFE Remote Fabrication System

It is an old saying that a tool unused is a useless tool. This section will build upon all the features from the remote fabrication toolkit and describe the CAFE-specific applications that enable remote processing of wafers. In the first subsection, the CAFE database is revisited and additional software tools are described to provide CAFE-specific functionality. The second and third subsections describe the application software modules that provide remote fabrication support to the user, namely the **operate-machine** methods for remote processing and **start-remote-lot**.

5.1 Special Objects and Tools: The CAFE Database Revisited

In Part 2 we described the functions and mechanisms of the database interoperability tools and how they operate on objects in the CAFE database. This subsection describes a small set of these objects that have special meaning to the CAFE Remote Fabrication System, and some of the specialized tools that operate on these objects.

5.1.1 Notion of Remote Facilities

In the CAFE database, **facility** objects are associated with the physical IC fabrication facilities of the same name. For example, MIT has an IC fab called ICL, the Integrated Circuits Lab. In the CAFE database, this facility has an associated facility object with the name "ICL":

```
CAFE>(facility-with-name "ICL")
#<FACILITY ICL 45611200>
```

```
CAFE>(describe *)

#<FACILITY ICL 45611200> is an instance of class #<Gestalt-Class FACILITY
36202340>:
 The following slots have :INSTANCE allocation:
 %ENTITY          10981560
 NAME             "ICL"
 LOCATION         "Building 39 (2nd and part of 4th floors)"
 DESCRIPTION      "Integrated Circuits Fabrication Lab"
 TIMEAVAILABLE    (#<TIMEAVAILABLE 45611000>
                   ...)
```

The **facility** object serves as a way to register the physical facility in the CAFE database. This way, software applications may store and retrieve relevant data about a particular facility such as its location (i.e. in MIT building 39), a human-readable description of the fab, and other information.

The **facility** object also offers the flexibility of registering a physical fab even when that fab doesn't really exist locally. For example, the facility at Lincoln Laboratories is a fabrication site that is some distance away from MIT. However, we can register this facility into MIT's CAFE database by making a facility object associated with Lincoln Laboratories. CAFE applications can then use the object and the information stored in it just like other facility objects. However, for remote processing to work, there must be something that designates the facility object associated with Lincoln Labs as a remote facility and not a local facility at MIT.

One way to distinguish a facility object as referring to a remote facility is to attach a tag to the object that identifies it as a remote facility. This can easily be done by adding a simple keyword to the location slot in the facility object. In the example below, the facility "RCF" is a facility in a far away place from us:

```
CAFE>(facility-with-name "garcon2:RCF")
#<FACILITY garcon2:RCF 45612540>

CAFE>(describe *)

#<FACILITY garcon2:RCF 45612540> is an instance of class #<Gestalt-Class
FACILITY 36202340>:
 The following slots have :INSTANCE allocation:
 %ENTITY          12911528
 NAME             "garcon2:RCF"
 LOCATION         "somewhere over the rainbow :remote"
 DESCRIPTION      "Remote Cookie Factory - yum yum"
 TIMEAVAILABLE    NIL
```

A query function can then be used to test for the case when a facility is a remote one or a local one:

```
CAFE>(facility-remote? **)
T
```

Note that at the remote end, the facility called "RCF" is local to itself and so its facility object does not contain the **:remote** keyword.

5.1.2 Machines and Virtual Machines

In the similar spirit with remote facilities, machine objects in CAFE that exist at remote sites need to be somehow designated as being remote. One simple way is to attach a **:remote** keyword somewhere in the machine object's description and then write custom query functions to test whether or not a machine is local or remote. However, in CAFE machine objects have special uses in CAFE's Fabrication Tools. One tool, **operate-machine**, is a generic function consisting of methods that dispatch directly on these machine objects. If we can extend **operate-machine** to handle these remote machines differently from the existing **operate-machine** methods, we can preserve the program behavior on normal machine objects and reuse the existing code rather than restructuring the entire mechanism for **operate-machine**.

The way to extend **operate-machine** is to construct a new machine object class, the **virtualmachine** class, that would be used for machine objects that are located remotely. An example of such an object is the depositor machine that exists at facility "RCF":

```
#<VIRTUALMACHINE depositor in garcon2:RCF 53701600> is an instance of class
#<Gestalt-Class VIRTUALMACHINE 33035600>:
 The following slots have :INSTANCE allocation:
 %ENTITY                12969904
 NAME                   "depositor"
 DESCRIPTION            "auto-generated by machines-with-name"
 FACILITY               #<FACILITY garcon2:RCF 45612540>
 DATACOMM               NIL
 SETTINGSCLASS          NIL
 READINGSCLASS          NIL
 SCHEDULE_OPERATIONS    NIL
 SCHEDULE_INFO          NIL
 OPERATORS              NIL
 ADVICE                 ":virtual :remote"
```

The **virtualmachine** class is a subclass of the **machine** class, and inherits all its slots and schema properties. Because the "depositor" machine is a **virtualmachine** object, an **operate-machine** method can be written to handle the "depositor" in a different manner than a local machine. This new behavior for **virtualmachine** objects will be described later on.

Due to this additional class and new behavior associated with it, a few CAFE functions had to be extended. The function **machines-with-name** is used to retrieve a list of machines that match a naming criterion. The naming criterion consists of a string containing the facility name and the machine name, with the wildcard * being used to retrieve more than one machine:

```
CAFE>(machines-with-name "ICL:developer")
(#<MACHINE developer in ICL 61527640>)
```

In the remote fabrication system, the naming criterion has been extended to allow the retrieval of remote machines. The naming criterion may now consist of the remote database's domain name, its facility name, and the machine name:

```
CAFE>(machines-with-name "garcon2:RCF:developer")
garcon2:RCF:developer = garcon.mit.edu:8187:RCF:developer
searching for facility RCF...facility found.
searching for machine developer...

find-host-facility-machine: machine not found
  Does the machine developer exist at garcon2:RCF?  (Y or N) y
constructing virtualmachine garcon2:RCF:developer...done.
machine found.
(#<VIRTUALMACHINE developer in garcon2:RCF 61527360>)
```

In this example, **machines-with-name** first translates the domain name into the associated host name and receiver port (retrieved from ***host-table-address-list***). While the remote facility "garcon2:RCF" is registered in the database, the particular machine is not. At this point, **machines-with-name** asks the user whether or not the machine exists at the remote facility, and if the user answers "Y" a new **virtualmachine** object is created. A better solution would be a mechanism that would automatically query the remote host about the existence of the machine in question and using the underlying VOM to handle the transfer of information. In any case, **machines-with-name** returns a list of local or remote machines it finds that match the extended criterion. Note that the machine in facility ICL and the machine in the remote facility RCF both have the same name "developer." While machine names are unique within a single facility, they do not have to unique between different facilities.

5.1.3 Labusers and Virtual Labusers

Lab users and human operators are also registered in the CAFE database using **labuser** objects. When the information about a remote lab user is needed, i.e., a human operator in the remote facility "garcon2:RCF," the lab user registers as a **virtuallabuser** object in the local database. The **virtuallabuser** class is a subclass of **labuser** and inherits all its slots. The schema properties of **virtuallabuser** are slightly different from **labuser**, though, for pragmatic reasons that allow greater flexibility in development. Special functions and routines may be written to take advantage of the local/remote distinction of a lab user.

5.2 Operate-Machine Methods

The primary CAFE application the user interacts with during the processing and fabrication of wafers and integrated circuits is **operate-machine**. At each step in the fabrication process, the lot of wafers undergo some sort of processing by a particular machine or equipment in the fab. Based on this particular machine, **operate-machine** automatically retrieves and initializes all the necessary data for the process step, including the appropriate machine settings and special processing instructions. **Operate-machine** also provides a user interface for a person to enter in specific machine readings and comments, storing them in appropriate **opinst** objects. When the current task is finished **operate-machine** internally updates the task tree and status of the lot and notifies other applications of the changes.

For processing tasks that need to be done remotely using machines in remote facilities, **operate-machine** must have new behavior. This behavior includes retrieving and assembling the appropriate information, locating where the remote facility is and reporting the information to the user so that the lot of wafers can be physically transported there, exporting objects and sending the data to the remote site, and providing a mechanism to query the status and progress of the lot when remote processing is underway. Figure 16 shows the three different **operate-machine** methods that exist and when each is used, depending on the **machine** object being dispatched on. The **operate-machine** method for **virtualmachine** objects is used when a remote processing step has been reached.

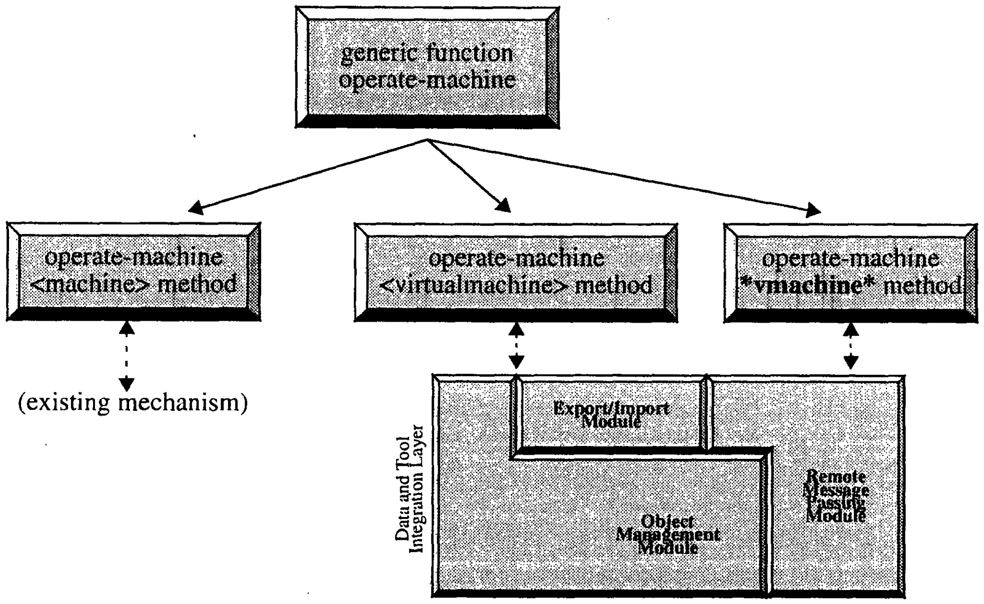
generic function operate-machine
operate-machine <machine> method
operate-machine <virtualmachine> method
operate-machine *vmachine* method
(existing mechanism)
Data and Tool Integration Layer
Export/Import Module
Object Management Module
Remote Message Passing Module

Figure 16. The Various **operate-machine** Methods

The third **operate-machine** method dispatches on a special machine called ***vmachine***. This ***vmachine*** machine object is a special machine that is used in the last task of a remote opset to signal that processing is done and the lot and any new data is to be sent back to the original system. The mechanism of how **operate-machine** works in relation to the processing tasks will be better understood in section 6, where a remote fabrication experiment will be carried out and described.

5.3 Start-Remote-Lot Function

While so far we have been focussing on the local side of things such as how **operate-machine** will prepare the appropriate data and send it off to the remote site, we have neglected the detail of how the remote facility will actually start the processing once the lot has been physically received. In a similar paradigm of initiating the processing flow on a lot created locally in CAFE using **start-lot**, the operator uses the new software module **start-remote-lot** to import and create a lot that was originally created on another CAFE system. After executing **start-remote-lot**, the operator can resume the typical **operate-machine/next-operation** cycle on the received lot.

6 A Remote Fabrication Example using CAFE

This section will describe a simple example of remote fabrication between two CAFE systems, "garcon1" and "garcon2." The example will walk-through various pieces of software modules in CAFE and its new remote fabrication extension, and can be used as a tutorial of some of CAFE's features.

The two CAFE systems, "garcon1" and "garcon2," are both development software systems at MIT. Although in reality they reside on the same computer and thus share executable code, each CAFE is linked to a distinct database, each containing different information from the other. This database separation between the two CAFE systems provides a good development environment to create the prototype remote fabrication system in CAFE.

The process flow tree used in this example describes a developmental recipe for making the next generation cookie. Because the recipe is still in its early development stage and uses radically new baking technology, it's identity is kept secret and will be referred in this example as the "Cookie-1" recipe. The reader is forewarned that any attempt of espionage and maliciously divulging top-secret information to rival cookie makers will be met with unimaginable repercussions. With this thought in mind, the process flow tree titled "MERKLIN-COOKIE-DEV" is shown in Figure 17 (alternatively, the tree may also be referred to as the task tree). The process flow tree consists of three main operation sets, or opsets: SubProcess 1, SubProcess 2, and SubProcess 3. Each opset contains a few leaf nodes. These leaf nodes contain the actual atomic machine steps that a cookie lot will undergo processing.

The first and last opsets are done in the local facility at "garcon1." The local facility is called "TCL," short for "The Cookie Laboratory." The leaf nodes under SubProcess 1 and SubProcess 3 specify the processing using the local machines in "TCL" and are shown next to the leaf nodes in Figure 17.

The second opset, labelled SubProcess 2, are done in a remote site at "garcon2." The remote facility there is called "RCF," short for "Remote Cookie Factory," and the machine names are labelled appropriately to reflect their location.

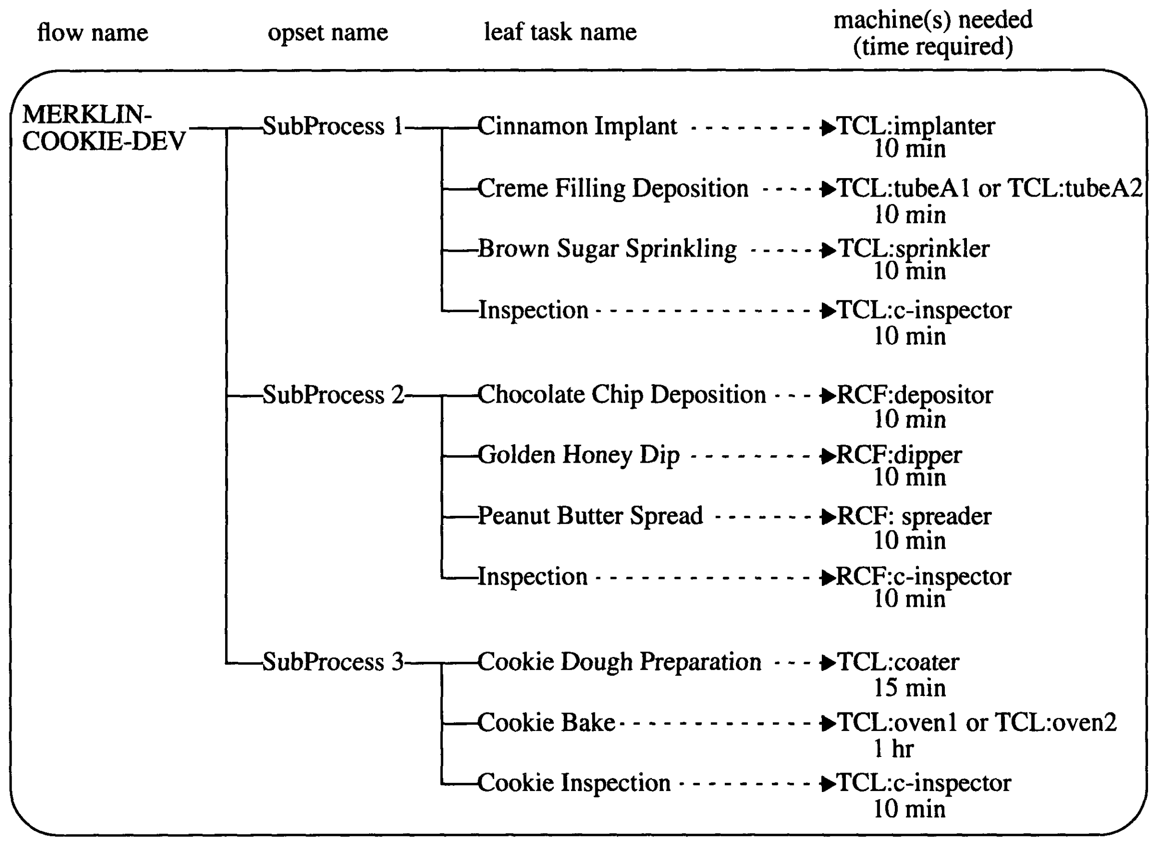

Figure 17. Processflow/Task Tree for MERKLIN-COOKIE-DEV

6.1 Cookie Fabrication in Facility "TCL" at Domain "garcon1"

For now, let's assume that a designer wrote the original process flow and installed it into the CAFE database at "garcon1." To start creating some development cookies, we run a CAFE session on a terminal and after an opening banner screen we arrive at the main menu:

```
User: merklin       Project:                            Facility:
Lot:                  Flow:                              Machine:

    Fabrication Tools...     Process and data collection tools.
    Lot Management...        Create and start a new lot, etc.
    Scheduling...            Scheduling machine/facility use.
    Reports...               Access data on machines, lots, labusers, etc.
    Process Flow...          Process Flow Representation Management.
    Notebook...              Laboratory Notebook and Projects Menu.
    General Purpose...       Run a shell to access UNIX commands.
    Mail and Messages...     Communicate with other people.
    Labstaff Tools...        CAFE Management Tools.
    CAFE System...           CAFE System Tools.
    Help...                  On-line manuals and documentation.
    Quit or Sign Out of Lab  Exit this program.

Menu Choice:

Use control-Z or control-X control-Z to hide your wand.
Please remember to sign out of the lab.
```

6.1.1 Creating and Starting the Cookie Lot

The first thing to do is to create a new lot for the development cookies by selecting **Lot Management**. The screen changes to show the following menu:

```
User: merklin       Project:                          Facility:
Lot:                   Flow:                           Machine:

    Create Lot              Create a new lot.
    Start Lot               Append process flow to lot and/or create lot tasks.
    Start Remote Lot        Install data for a remote lot.
    Abandon Lot             Delete a lot which has not been processed.
    Modify Lot Status       Set current lot's status.
    Modify Lot Priority     Set current lot's priority.
    Create Waferset         Create a new waferset for a lot.
    Broken Wafer            Specify that a wafer is broken.
    Quit or Sign Out of Lab Exit this program.
    <SPACE>                 Return to previous menu.

Menu Choice:
```

We then choose **Create Lot** to create a new lot. The selection will then pull up some screens requesting information about the characteristics of the lot. We enter the following information:

```
Creator of the Lot: merklin
Responsible User for the Lot: merklin
Lot Name: cookie1
Process: MERKLIN-COOKIE-DEV
No. of wafers: 2
Wafer ID starting value: 1

        Wafer Laserid            Wafer Type
1    cookie-1               merklin-sugarwafer
2    cookie-2               merklin-sugarwafer
```

For this simple example, we'll only create two development prototype cookies. The cookies initially start out on specially-made sugar wafers. After a few more screens CAFE will create the appropriate objects for the lot named **cookie1**.

```
Creating waferset cookie1-1 for lot cookie1...
Are you finished creating wafersets for this user/lot? (y / n) y
Creating a default waferset cookie1 containing all wafers for lot cookie1
Creating lot cookie1 for user merklin on 08/20/95 with process MERKLIN-COOKIE-DEV
```

When we return to the Lot Menu, we let CAFE know that we want to start using **cookie1** by entering into the lot name in the lot field:

```
User: merklin       Project:                          Facility:
Lot: cookie1           Flow: MERKLIN-COOKIE-DEV        Machine:
```

Next we initiate the processing cycle by selecting **Start Lot**. CAFE then creates the task tree for the lot and generates the necessary objects required for the **operate-machine** cycle.

```
Checking wafersets...OK
MERKLIN-COOKIE-DEV ...
SubProcess 1 ...
```

```
Cinnamon Implant ...
Creme Filling Deposition ...
Brown Sugar Sprinkling ...
Inspection ...
SubProcess 2 ...
Chocolate Chip Deposition ...
Golden Honey Dip ...
Peanut Butter Spread ...
Inspection ...
SubProcess 3 ...
Cookie Dough Preparation ...
Cookie Bake ...
Cookie Inspection ...
Updating task pointers...
Creating traveller report from scratch - Please wait...
Creating past report...
Creating present report...
Creating future report...
```

6.1.2 Operate-Machine/Next Operation Cycle

When **Start Lot** completes, we are now ready to start making some cookies. After suiting up for the clean room in the "TCL" facility and acquiring the necessary items (such as the sugar wafers, containers, chemicals and ingredients, etc.), we can start the first machine operation by selecting **Fabrication Tools** from the Main Menu. The screen changes to the following menu:

```
User: merklin       Project:                           Facility:
Lot: cookie1           Flow: MERKLIN-COOKIE-DEV         Machine:

    Operate Machine          Perform an operation on the current machine.
    Batch Operations         Perform a machine operation on multiple lots.
    Next Operation           Show next operation for current lot.
    Ready Operations         Show queue of operations ready for machine(s).
    Append Comments          Append comments to a lot's completed operation(s).
    Machine Log              Show logs for current machine.
    Generate Traveller       Lot report at opset level.
    Select Facility          Set current facility for logging operations, etc.
    Select Machine           Set current machine for logging operations, etc.
    Select Lot               Set current lot for logging operations, etc.
    Collect Data             Automated data collection from selected machines.
    Quit or Sign Out of Lab  Exit this program.
    <SPACE>                  Return to previous menu.

Menu Choice:
```

While we could use **Operate Machine** directly and provide the exact machine to use next on the lot, it is much easier to use **Next Operation** and let CAFE help us choose a machine. Selecting **Next Operation** gives a screen asking for the desired machine to use for the next operation:

```
                         Next Operation
                         --------------

Next machine operation for cookie1 is
Cinnamon Implant

which can be performed on these machines:
TCL:implanter.
```

```
The time required is 10 Minutes.


Enter desired machine here-->TCL:implanter

CTRL-X CTRL-C to save.
CTRL-X CTRL-Q to abort.
```

Since TCL:implanter is the only machine that can do the first task, "Cinnamon Implant," we select it. The data screen then shows where we can view processing instructions and enter log information:

```
MACHINE: TCL implanter       DATE: 08/20/95 14:34:08     USER: merklin
PROCESS: NIL                 WAFERS: Lot cookie1: 1 2
Start 08/20/95 14:36:47
Settings
         INGREDIENT                    cinnamon
End   08/20/95 14:47:00
Comments:
```

On the machine TCL:implanter, we need to set the machine settings to use cinnamon for the ingredient specification. After the processing is complete, we can continue on to the next operation by the same method as the first task:

```
                          Next Operation
                          --------------

Next machine operation for cookie1 is
Creme Filling Deposition

which can be performed on these machines:
TCL:tubeA1 TCL:tubeA2.

The time required is 10 Minutes.


Enter desired machine here-->TCL:tubeA1

CTRL-X CTRL-C to save.
CTRL-X CTRL-Q to abort.
```

Here we are given a choice of two tube machines. In the case shown above, TCL:tubeA1 was selected, but any tube will do. In much the same manner as before, we arrive at the data screen. This time, the operator of the tube machine accidently burned himself while processing the cookie lot. A comment was added and a suggestion was made to help prevent future mishaps:

```
MACHINE: TCL tubeA1          DATE: 08/20/95 14:49:37     USER: merklin
PROCESS: NIL                 WAFERS: Lot cookie1: 1 2
Start 08/20/95 14:49:55
Settings
         INGREDIENTS                   vanilla creme
End   08/20/95 14:59:55
Comments:
the operator scorched himself on the tube... we'll need to add some safety
precautions.
```

The next two machine operations are relatively straightforward. The last task is the inspection step, where the operator is required to measure the thickness of the creme layer.

```
MACHINE: TCL sprinkler       DATE: 08/20/95 15:10:04     USER: merklin
PROCESS: NIL                 WAFERS: Lot cookie1: 1 2
Start 08/20/95 15:11:24
Settings
         INGREDIENTS                  fine brown sugar
End   08/20/95 15:21:04
Comments:


MACHINE: TCL c-inspector     DATE: 08/20/95 15:29:17     USER: merklin
PROCESS: NIL                 WAFERS: Lot cookie1: 1 2
Start 08/20/95 15:29:30
Readings
         CREME_LAYER_THICKNESS        5.5 cm
End   08/20/95 15:31:18
Comments:
```

6.1.3 Remote Operation: Start

The first opset, SubProcess 1, is complete. The next opset contains four machine operations that are to done remotely at "RCF" in the domain "garcon2." The CAFE remote fabrication system detects this and becomes active. The next operation greets us with some information concerning the remote facility:

```
operate-machine for remote processing...
machine: #<VIRTUALMACHINE depositor in garcon2:RCF 55233220> (VIRTUALMACHINE 19)
...
Remote Facility information: garcon2 (garcon.mit.edu:8187)
MIT CAFE garcon2 development database/Merklin's db playland
50 Vassar St., 36-295
Cambridge, MA 02139

remote-task-structure:
(#<TASK MERKLIN-COOKIE-DEV 63422540>
 (REMOTE-OPERATION-START
     #<TASK SubProcess 2 63422400> REMOTE-OPERATION-END))
lot: #<LOT cookie1 63422620>
activeopinst: #<ACTIVEOPINST 63421240>
  Ready to encode objects?  (Y or N) y
...(encoding session)
```

The next few lines show some information about what objects are going to be encoded and prepared to be transferred to "garcon2." The CAFE objects to be transferred are the **lot** object for **cookie1**, a task tree (and associated flow) that contains the necessary remote processing instructions to be done at facility "RCF," and an **activeopinst** which contains some information such as the time when we started the encoding process and sent the message. The task tree that is sent is shown in Figure 18. From the figure, we can see that only the tasks that need to be done at facility "RCF" are included. The other local tasks are not necessary for "garcon2" to know, and since this **cookie1** is a secret prototype it is better to save network bandwidth by transmitting short messages that contain information on a strictly need-to-know basis.

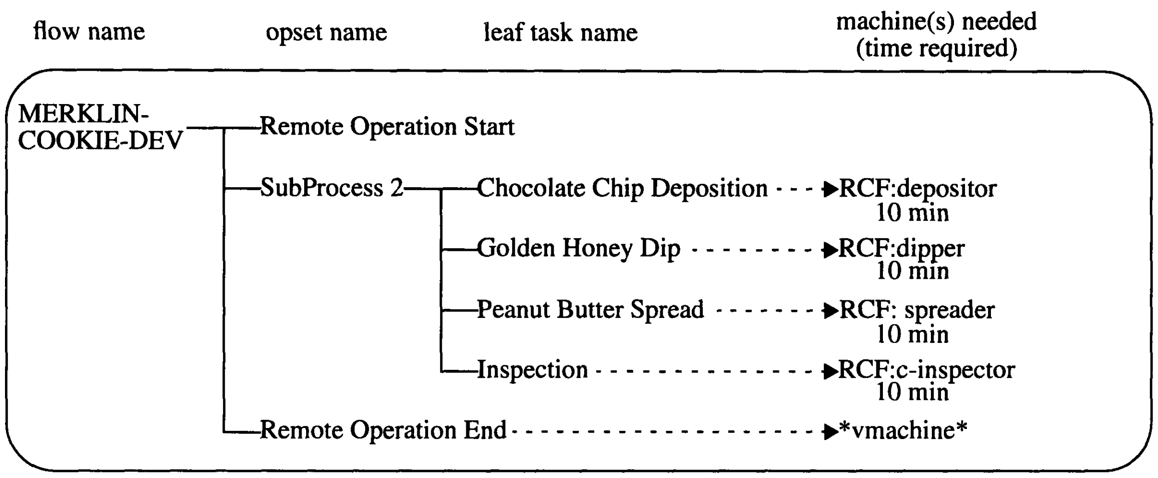

Figure 18. Processflow/Task Tree that is Exported to the Remote Site

At the prompt, we can simply say "yes" to encode the objects because we know what we are doing. The expert tool then processes and encoded the appropriate objects and processing instructions. After some time the encoding process completes and we are asked if we are ready to send the information to "garcon2," in which we answer positively.

```
toplevel encoded reference pointers:
toptask: (POINTER TASK 65)
subtask pointer list: ((POINTER TASK 69) (POINTER TASK 67)
                       (POINTER TASK 71) (POINTER TASK 73)
                       (POINTER TASK 75))
current activeopinst pointer: (POINTER ACTIVEOPINST 11)
lot: (POINTER LOT 9)
  Ready to send remote message to dbdomain garcon2 (garcon.mit.edu:8187) on
channel cookie1?  (Y or N) y
saving backup remote message to "/tmp/CAFE_garcon2_cookie1_or1_a703"...done.
...(message sending session)
Message sent.
```

Once the information has been successfully transmitted, we should send the physical cookie lot to the remote site at the address given in the above screen so that processing can really take place.

6.2 Cookie Fabrication in Facility "RCF" at Domain "garcon2"

Meanwhile at the remote domain "garcon2," the CAFE remote message receiving daemon detects an incoming message and stores it. The message is from "garcon1" and contains information about processing **cookie1**.

```
CAFE>(cafe-remote-open-receiver :port 8187)
receiver port 8187 successfully opened.
OPENED

CAFE>(cafe-remote-receive-daemon)
cafe-remote-receive-daemon initiated Sun Aug 20 11:31:41 EST 1995
press <return> to exit
receive on port 8187, channel "cookie1": /homes/merklin/src/lisp/remote/remote-
messages/8187/cookie1/incoming-file-0001
```

6.2.1 Installing the Remote Lot

Assuming that Mr. Q receives the physical cookie lot from facility “TCL” some time later, he can then proceed to continue the processing cycle for **cookie1**. He runs his own CAFE session and selects the Lot Management menu:

```
User: mrq            Project:                                Facility:
Lot:                    Flow:                                 Machine:

    Create Lot               Create a new lot.
    Start Lot                Append process flow to lot and/or create lot tasks.
    Start Remote Lot         Install data for a remote lot.
    Abandon Lot              Delete a lot which has not been processed.
    Modify Lot Status        Set current lot's status.
    Modify Lot Priority      Set current lot's priority.
    Create Waferset          Create a new waferset for a lot.
    Broken Wafer             Specify that a wafer is broken.
    Quit or Sign Out of Lab  Exit this program.
    <SPACE>                  Return to previous menu.

Menu Choice:
```

Since the lot was originally created in another CAFE system, Mr. Q chooses **Start Remote Lot** to retrieve the remote message:

```
**** Start Remote Lot ****

current installed channels:
  cookie1
  dev
  frobozz-dev
  frobozz2
  frobozz3
  frobozz4
  frobozz5
  moomoo
  system
  test-machine
  wpmoyne2
Enter the name of the remote lot (<enter> to abort): cookie1
```

A listing of remote lots is shown, and Mr. Q selects **cookie1**. CAFE searches for information concerning the remote lot and finds a message waiting:

```
message(s) found...
Interpret next message?  (Y or N) y
```

Mr. Q is also knows what he is doing, so he tells CAFE to go ahead and interpret the message.

```
Sun Aug 20 16:30:08 EST 1995 ==== start message ====
Sun Aug 20 16:30:08 EST 1995 ==== start header parsing ====
  from dbdomain garcon1 domain garcon.mit.edu  channel cookie1  port 8186
  header packaged on Sun Aug 20 16:27:19 EST 1995 by merklin
  send to dbdomain garcon1 domain garcon.mit.edu  channel cookie1  port 8186
header comments: "generated by operate-machine (remote)"
Sun Aug 20 16:30:08 EST 1995 ====  end  header parsing ====
saving backup remote message to "/tmp/CAFE_garcon1_cookie1_crmm_a0710"...done.
...(interpretation session)
```

After the header is parsed, the message is interpreted and the import tool is invoked to decode and incorporate the processing information for **cookie1** into garcon2's CAFE database. Various objects are generated and custom updated (a.k.a. *hacked*) according to the specific requirements of the system.

```
constructing remote operation start/end tasks...done.
updating task tree...tnaaaa done.
rehacking remote lot garcon1:cookie1...done.
...
updating active opinst...done.
Retrieving traveller report cache...
Creating traveller report from scratch - Please wait...
Creating past report...
Creating present report...
Creating future report...
Generating traveller report - Please wait...
...(object update session)
```

6.2.2 Operate-Machine/Next-Operation Cycle at the Remote Facility

Once **Start-Remote-Lot** is done installing the data, Mr. Q then proceeds to set his lot field to the newly created remote lot under the lot name **garcon1:cookie1**. The lot name change helps avoid naming conflicts, since there might be an existing lot **cookie1** at facility "RCF."

```
User: mrq           Project:                          Facility:
Lot: garcon1:cookie1   Flow: MERKLIN-COOKIE-DEV        Machine:
```

Mr. Q then goes to the Fabrication Menu by selecting **Fabrication Menu** from his CAFE's Main Menu and then does **Next-Operation**. The next operation step for **garcon1:cookie1** is the Chocolate Chip Deposition task, which uses the depositor machine in facility "RCF:"

```
                              Next Operation
                              --------------

Next machine operation for garcon1:cookie1 is
Chocolate Chip Deposition

which can be performed on these machines:
RCF:depositor.

The time required is 10 Minutes.


Enter desired machine here-->RCF:depositor

CTRL-X CTRL-C to save.
CTRL-X CTRL-Q to abort.
```

Note that the machine is called RCF:depositor and not garcon2:RCF:depositor. Mr. Q is performing machine operations in facility "RCF" so the machine is local.

The processing cycle continues in much the same manner as it was done at facility "TCL" in our domain "garcon1." The next four data screens for each task are shown below:

```
MACHINE: RCF depositor       DATE: 08/20/95 16:18:00     USER: mrq
PROCESS: NIL                 WAFERS: Lot garcon1:cookie1: 1 2
Start 08/20/95 16:20:24
Settings
         INGREDIENTS                   chocolate chip
         TYPE                          chunky
End   08/20/95 16:48:28
Comments:
 (:encoding-started-on "Sun Aug 20 16:20:24 EST 1995") (:decoding-completed-on "
Sun Aug 20 16:33:46 EST 1995")


MACHINE: RCF dipper          DATE: 08/20/95 16:51:22     USER: mrq
PROCESS: NIL                 WAFERS: Lot garcon1:cookie1: 1 2
Start 08/20/95 16:51:26
Settings
         INGREDIENTS                   honey
End   08/20/95 17:01:27
Comments:


MACHINE: RCF spreader        DATE: 08/20/95 17:12:13     USER: mrq
PROCESS: NIL                 WAFERS: Lot garcon1:cookie1: 1 2
Start 08/20/95 17:12:17
Settings
         INGREDIENTS                   peanut-butter
         GRADIENT                      parallel planar
End   08/20/95 17:25:17
Comments:


MACHINE: RCF c-inspector     DATE: 08/20/95 17:53:25     USER: mrq
PROCESS: NIL                 WAFERS: Lot garcon1:cookie1: 1 2
Start 08/20/95 17:53:31
Readings
         TOTAL_THICKNESS               5.9 cm
End   08/20/95 17:55:35
Comments:
not as good as expected... we'll have to talk more about modifying the process
to enhance the thickness.
```

6.2.3 Remote Query from Facility "TCL" at Domain "garcon1"

Back at our facility "TCL" in domain "garcon1," we're getting anxious about the processing status of our cookie lot. We can rerun **Next Operation** and perform a remote query to facility "RCF" at domain "garcon2" about the lot's status:

```
operate-machine for remote processing...
machine: #<VIRTUALMACHINE depositor in garcon2:RCF 55233220> (VIRTUALMACHINE
The current remote task is active.
The lot cookie1 is at site garcon2:RCF
Select an option:
  1 - remote query about lot's status
  2 - check for remote message replies/events
  3 - resend backup remote message
  4 - do nothing/exit
Enter your selection: 1
```

```
:REMOTE-QUERY
((:HEADER (:FROM (:DBDOMAIN "garcon1") (:DOMAIN "garcon.mit.edu")
                 (:CHANNEL "cookie1") (:PORT 8186)
                 (:HEADER-PACKAGED-ON "Sun Aug 20 17:11:00 EST 1995")
                 (:HEADER-PACKAGED-BY "merklin"))
          (:SEND-REPLY-TO (:DBDOMAIN "garcon1")
              (:DOMAIN "garcon.mit.edu") (:CHANNEL "cookie1")
              (:PORT 8186))
          (:COMMENTS
              "remote query generated by operate-machine (remote)"))
 (:MESSAGE-REQUESTS (PROCESSING-STATUS (LOT 9))))
Message sent.
```

Once the query message is sent, the receive daemon at domain "garcon2" receives the message:

```
receive on port 8187, channel "system": /homes/merklin/src/lisp/remote/remote-
messages/8187/system/incoming-file-0001
```

The message is a system message, and the remote system client at domain "garcon2" intercepts the message, interprets it, and automatically sends a return reply:

```
CAFE>(cafe-remote-system-client)
cafe-remote-system-client initiated Sun Aug 20 13:31:34 EST 1995
press <return> to exit
Sun Aug 20 17:11:04 EST 1995 ==== start system message ====
Sun Aug 20 17:11:04 EST 1995 ==== start header parsing ====
  from dbdomain garcon1 domain garcon.mit.edu  channel cookie1  port 8186
  header packaged on Sun Aug 20 17:11:00 EST 1995 by sysop
  send to dbdomain garcon1 domain garcon.mit.edu  channel cookie1  port 8186
header comments: "remote query generated by operate-machine (remote)"
Sun Aug 20 17:11:04 EST 1995 ====  end  header parsing ====
Sun Aug 20 17:11:04 EST 1995 ==== start requests parsing ====
Sun Aug 20 17:11:04 EST 1995 ==>
 (CAFE-REMOTE-SYSTEM-CLIENT-REQUEST-PROCESSING-STATUS
                                   (LOT 9) :DOMAIN "garcon1" :CHANNEL
                                   "cookie1")
  result: (:RVAL "lot cookie1 is currently on task Golden Honey Dip")
:REMOTE-MESSAGE "cookie1"
((:HEADER (:FROM (:DBDOMAIN "garcon2") (:DOMAIN "garcon.mit.edu")
                 (:CHANNEL "system") (:PORT 8186)
                 (:HEADER-PACKAGED-ON "Sun Aug 20 17:11:08 EST 1995")
                 (:HEADER-PACKAGED-BY "sysop")
                 (:COMMENTS "yippee kay-yea"))
          (:SEND-REPLY-TO (:DBDOMAIN "garcon2")
              (:DOMAIN "garcon.mit.edu") (:CHANNEL "system")
              (:PORT 8187) (:COMMENTS "you better..."))
          (:COMMENTS
              "this message is an automatic remote response generated by the
system client"))
 (:MESSAGE-REPLIES
     ((PROCESSING-STATUS (LOT 9))
      "lot cookie1 is currently on task Golden Honey Dip")))
Message sent.
Sun Aug 20 17:11:09 EST 1995 ====  end  requests parsing ====
Sun Aug 20 17:11:09 EST 1995 ====  end  system message ====
```

Switching back again to our facility "TCL" in the domain "garcon1," our own remote receive daemon captures the return reply:

```
CAFE>(cafe-remote-open-receiver :port 8186)
receiver port 8186 successfully opened.
OPENED

CAFE>(cafe-remote-receive-daemon)
cafe-remote-receive-daemon initiated Sun Aug 20 13:31:41 EST 1995
press <return> to exit
receive on port 8186, channel "cookiel": /homes/merklin/src/lisp/remote/remote-
messages/8186/cookiel/incoming-file-0001
```

We can then select option 2 to check and interpret the remote message reply:

```
The current remote task is active.
The lot cookiel is at site garcon2:RCF
Select an option:
  1 - remote query about lot's status
  2 - check for remote message replies/events
  3 - resend backup remote message
  4 - do nothing/exit
Enter your selection:   2
message(s) found...
interpret next message?  (Y or N) y
Sun Aug 20 17:23:08 EST 1995 ==== start message ====
Sun Aug 20 17:23:08 EST 1995 ==== start header parsing ====
  from dbdomain garcon2 domain garcon.mit.edu  channel system  port 8186
  header packaged on Sun Aug 20 17:11:08 EST 1995 by sysop
  comments: "yippee kay-yea"
  send to dbdomain garcon2 domain garcon.mit.edu  channel system  port 8187
  comments: "you better..."
header comments: "this message is an automatic remote response generated by the
system client"
Sun Aug 20 17:23:08 EST 1995 ====  end  header parsing ====

saving backup remote message to "/tmp/CAFE_garcon2_cookiel_crmm_a0703"...done.
Sun Aug 20 17:23:08 EST 1995 ==== start replies parsing ====
  original request: (PROCESSING-STATUS (LOT 9))
  return value: "lot cookiel is currently on task Golden Honey Dip"
Sun Aug 20 17:23:08 EST 1995 ====  end  replies parsing ====
Sun Aug 20 17:23:08 EST 1995 ====  end  message ====
```

6.2.4 Sending the Data Collected back to Facility "TCL" at Domain "garcon1"

After the last inspection step, a special operation step is taken to reassemble the data collected during the remote processing of **garcon1:cookie1**, export the data, and send the information back to the original facility, "TCL" at domain "garcon1." CAFE first asks Mr. Q whether to start the encoding process on the new data stored in the **completedopinsts**, at which Mr. Q answers affirmatively:

```
Next machine operation for garconl:cookiel is
MERKLIN-COOKIE-DEV:remote-operation-end
which can be performed on these machines:
(vmachine)
Do you wish to operate machine? (y/n) y

operate-machine for remote processing termination...
machine: #<MACHINE vmachine in vfacility 46703220> (MACHINE 448)
```

```
completed opinsts: (#<COMPLETEDOPINST 62163420>
                    #<COMPLETEDOPINST 62163100>
                    #<COMPLETEDOPINST 57410100>
                    #<COMPLETEDOPINST 62162640>)
  Ready to encode objects?  (Y or N) y
...(encode object session)
```

After the objects have been encoded and a return message constructed, Mr. Q then proceeds to send the information back to "garcon1."

```
  Ready to send remote message to dbdomain garcon1 (garcon.mit.edu:8186) on
channel cookie1?  (Y or N) y
saving backup remote message to "/tmp/CAFE_garcon1_cookie1_orlt_a710"...done.
...(message sending session)
Message sent.
```

Mr. Q also packages the processed lot **garcon1:cookie1** and sends it back to facility "TCL" at the domain "garcon1." As far Mr. Q is concerned, he has finished the remote processing of the lot at facility "RCF."

6.3 Cookie Fabrication in Facility "TCL" at Domain "garcon1"

Soon after Mr. Q has processed the cookie lot and sent the return information, the remote message receive daemon at "garcon1" receives the information and stores the message:

```
receive on port 8186, channel "cookie1": /homes/merklin/src/lisp/remote/remote-
messages/8186/cookie1/incoming-file-0002
```

6.3.1 Remote Operation: End

We can now run **Next-Operation**, retrieve the message, interpret the message for new processing information, and incorporate the new data back into out database:

```
The current remote task is active.
The lot cookie1 is at site garcon2:RCF
Select an option:
  1 - remote query about lot's status
  2 - check for remote message replies/events
  3 - resend backup remote message
  4 - do nothing/exit
Enter your selection: 2
message(s) found...
interpret next message?  (Y or N) y
Sun Aug 20 17:28:18 EST 1995 ==== start message ====
Sun Aug 20 17:28:18 EST 1995 ==== start header parsing ====
  from dbdomain garcon2 domain garcon.mit.edu  channel cookie1  port 8186
  header packaged on Sun Aug 20 17:05:24 EST 1995 by mrq
  send to dbdomain garcon2 domain garcon.mit.edu  channel cookie1  port 8187
header comments: "generated by operate-machine (remote termination)"
Sun Aug 20 17:28:18 EST 1995 ====  end  header parsing ====

saving backup remote message to "/tmp/CAFE_garcon2_cookie1_crmm_b0703"...done.
...(message interpretation and object incorporation session)
done.
  result: OK
```

6.3.2 Operate-Machine/Next-Operation Cycle Resumed

Once the data has been incorporated back into CAFE, the normal operate-machine/next-operation cycle resumes on the opset SubProcess 3, which contain the remaining machine operations:

```
MACHINE: TCL coater          DATE: 08/20/95 19:24:19     USER: merklin
PROCESS: NIL                 WAFERS: Lot cookie1: 1 2
Start 08/20/95 19:36:20
Settings
         INGREDIENTS                   cookie dough
End   08/20/95 19:51:20
Comments:


MACHINE: TCL oven1           DATE: 08/20/95 19:51:38     USER: merklin
PROCESS: NIL                 WAFERS: Lot cookie1: 1 2
Start 08/20/95 19:51:40
Settings
         TEMPERATURE                   350F
End   08/20/95 20:51:38
Comments:


MACHINE: TCL c-inspector     DATE: 08/20/95 20:55:35     USER: merklin
PROCESS: NIL                 WAFERS: Lot cookie1: 1 2
Start 08/20/95 20:55:36
Readings
         CONSISTENCY                   vapory and volatile
         TASTE                         mmm...mmm...mmm...
End   08/20/95 21:01:27
Comments:
```

The cookies start in solid form, but once placed in the mouth they vaporize, letting the cookie vapor carry the delicious flavor of dark chocolate, golden honey, and cinnamon for an exquisitely nebulous experience.

6.3.3 Traveller Report

After all the tasks have been completed, the operate-machine/next-operation cycle ends, and we can enjoy our cookies. Because we only made two prototypes, we want to preserve what was done by printing out the process history. From the Fabrication Menu we can select **Generate-Traveller** to get such a print out. Below is a sample printout of a traveller report for the processing of the lot **cookie1**.

```
               Traveller Report for Lot: cookie1

                     Description:      Cookie-1 Recipe
                     Created on:       08/20/95
                     Owner:            merklin
                     Responsible User: merklin
                     Priority:         NORMAL
                     Status:           ACTIVE
                     Wafersets:        cookie1
                                       cookie1-1

**** 1. Opset Task: Cinnamon Implant   Opset: NOT AN OPSET ****
MACHINE:  TCL implanter        USER: merklin
WAFERS: Lot cookie1: 1 2
```

```
Start  08/20/95 14:36:47
Settings
           INGREDIENT                      cinnamon
End  08/20/95 14:47:00
Comments:

**** 2. Opset Task: Creme Filling Deposition   Opset: NOT AN OPSET ****
MACHINE:  TCL tubeA1           USER: merklin
WAFERS: Lot cookie1: 1 2
Start  08/20/95 14:49:55
Settings
           INGREDIENTS                     vanilla creme
End  08/20/95 14:59:55
Comments:
the operator scorched himself on the tube... we'll need to add some safety
precautions.

**** 3. Opset Task: Brown Sugar Sprinkling   Opset: NOT AN OPSET ****
MACHINE:  TCL sprinkler        USER: merklin
WAFERS: Lot cookie1: 1 2
Start  08/20/95 15:11:24
Settings
           INGREDIENTS                     fine brown sugar
End  08/20/95 15:21:04
Comments:

**** 4. Opset Task: Inspection   Opset: NOT AN OPSET ****
MACHINE:  TCL c-inspector      USER: merklin
WAFERS: Lot cookie1: 1 2
Start  08/20/95 15:29:30
Readings
           CREME_LAYER_THICKNESS           5.5 cm
End  08/20/95 15:31:18
Comments:

**** 5. Opset Task: Chocolate Chip Deposition   Opset: NOT AN OPSET ****
MACHINE:  garcon2:RCF depositor  USER: mrq
WAFERS: Lot cookie1: 1 2
Start  08/20/95 16:20:24
Settings
           INGREDIENTS                     chocolate chip
           TYPE                            chunky
End  08/20/95 16:48:28
Comments:
 (:encoding-started-on "Sun Aug 20 16:20:24 EST 1995") (:decoding-completed-on
"Sun Aug 20 16:33:46 EST 1995")

**** 6. Opset Task: Golden Honey Dip   Opset: NOT AN OPSET ****
MACHINE:  garcon2:RCF dipper   USER: mrq
WAFERS: Lot cookie1: 1 2
Start  08/20/95 16:51:26
Settings
           INGREDIENTS                     honey
End  08/20/95 17:01:27
Comments:

**** 7. Opset Task: Peanut Butter Spread   Opset: NOT AN OPSET ****
MACHINE:  garcon2:RCF spreader  USER: mrq
WAFERS: Lot cookie1: 1 2
```

```
Start  08/20/95 17:12:17
Settings
           INGREDIENTS                  peanut-butter
           GRADIENT                     parallel planar
End  08/20/95 17:25:17
Comments:

**** 8. Opset Task: Inspection   Opset: NOT AN OPSET ****
MACHINE:  garcon2:RCF c-inspector  USER: mrq
WAFERS: Lot cookie1: 1 2
Start  08/20/95 17:53:31
Readings
           TOTAL_THICKNESS              5.9 cm
End  08/20/95 17:55:35
Comments:
not as good as expected... we'll have to talk more about modifying the process
to enhance the thickness.

**** 9. Opset Task: Cookie Dough Preparation   Opset: NOT AN OPSET ****
MACHINE:  TCL coater           USER: merklin
WAFERS: Lot cookie1: 1 2
Start  08/20/95 19:36:20
Settings
           INGREDIENTS                  cookie dough
End  08/20/95 19:51:20
Comments:

**** 10. Opset Task: Cookie Bake   Opset: NOT AN OPSET ****
MACHINE:  TCL oven1            USER: merklin
WAFERS: Lot cookie1: 1 2
Start  08/20/95 19:51:40
Settings
           TEMPERATURE                  350F
End  08/20/95 19:51:38
Comments:

**** 11. Opset Task: Cookie Inspection   Opset: NOT AN OPSET ****
MACHINE:  TCL c-inspector      USER: merklin
WAFERS: Lot cookie1: 1 2
Start  08/20/95 20:55:36
Readings
           CONSISTENCY                  vapory and volatile
           TASTE                        mmm...mmm...mmm...
End  08/20/95 21:01:27
Comments:
The cookies start in solid form, but once placed in the mouth they vaporize,
letting the cookie vapor carry the delicious flavor of dark chocolate, golden
honey, and cinnamon.
```

Part 4

Summary

7 Conclusion

This document has described how the MIT CAFE system can be extended to provide users a remote fabrication process model. Part 1 described some background on the current CAFE architecture and the fabrication process paradigm, and outlined a useful set of requirements that extend the process paradigm for remote fabrication, including the need for interoperability between databases. Part 2 introduced the reader to the notion of objects and some of the specific objects in the CAFE database, and then proceeded to describe a possible framework for supporting interoperability, the Virtual Object Manager. The three modules of the VOM that implemented the mechanisms needed for interoperability were then discussed, using the CAFE database and system in particular as the platform to implement the necessary functions and routines. The Object Management Module was responsible for keeping track of information being shared and transferred to other databases. The Export/Import Tools were used to translate objects from database form into a more transportable form and vice versa. The Remote Message Passing Tools implemented the information transport mechanisms, with the particular example in section 4 using TCP/IP sockets. Part 3 then built upon these basic tools of the VOM and describe CAFE-specific application modules that were added to give remote fabrication functionality to the user. Section 6 provided a walk-through example of remote fabrication between two prototype databases using a sample process flow, displaying the various tools and software involved and how they worked together.

8 Future Work

Despite the fair amount of work done to extend CAFE to support remote fabrication (approximately 8600 lines of development source code), the author initially approached the project with a minimalistic view: figure out in the barest of bones how to implement a remote fabrication system for CAFE without overhauling the entire CAFE system. Whether or not this minimalistic goal was achieved, there is still much work to be done to the prototype described in this document. Issues relating to crash recovery, data security and integrity have not been touched upon, and are very important for interoperability to be reliable. The framework of the Virtual Object Manager needs considerably more research and study. Some important questions are: What should the encoded form of an object from a database be so that it can be understood by all types of databases? While the export/import tools in this document described one way to transfer object information, perhaps a study in exporting and importing objects using the CORBA (Common Object Request Broker Architecture) specifications may provide a better solution; What is the minimal set of information required when referring to IC fabrication and processing, and how should this information be presented in as neutral way as possible (i.e., not biased towards any one CIM system)? MIT has the PFR (Process Flow Representation), Berkeley has the BPFL (Berkeley Process Flow Language), and other CIM systems each have their own formats and requirements, and translating information from one format to another may not always be possible; What are the requirements for interoperability between radically heterogeneous systems, and how do we cope with information sharing and synchronization between such systems? This document focussed on homogeneous interoperability (CAFE to CAFE interoperation), and research in resource sharing and distributed programming in heterogeneous systems might prove helpful in arriving at a standard message passing format between such heterogeneous systems. These are questions that may be difficult to answer straightforwardly, and hopefully this document has helped the reader become more aware of the issues involved and provided a foundation to build upon future work.

Appendix A

Maxims discovered during the Remote Fabrication project

(in no particular order of discovery)

BOVE'S THEOREM

The remaining work to finish in order to reach your goal increases as the deadline approaches.

CARLSON'S CONSOLATION

Nothing is ever a complete failure; it can always serve as a bad example.

LAW OF CONTINUITY

Experiments should be reproducible. They should all fail in the same way.

CROPP'S LAW

The amount of work done varies inversely with the amount of time spent in the office.

DEMIAN'S OBSERVATION

There is always one item on the screen menu that is mislabelled and should read "ABANDON HOPE ALL YE WHO ENTER HERE".

GILB'S LAW OF UNRELIABILITY

1) At the source of every error which is blamed on the computer you will find at least two human errors, including the error of blaming it on the computer.

2) Any system which depends on human reliability is unreliable.

3) Undetectable errors are infinite in variety, in contrast to detectable errors, which by definition are limited.

4) Investment in reliability will increase until it exceeds the probable cost of errors, or until someone insists on getting some useful work done.

GUMPERSON'S LAW

The probability of a given event occurring is inversely proportional to its desirability.

HINDS' LAW OF COMPUTER PROGRAMMING

1) Any given program, when running, is obsolete.

2) If a program is useful, it will have to be changed.

3) If a program is useless, it will have to be documented.

4) Any given program will expand to fill all available memory.

5) The value of a program is proportional to the weight of its output.

6) Program complexity grows until it exceeds the capability of the programmer who must maintain it.

7) Make it possible for programmers to write programs in English, and you will find that programmers cannot write in English.

JENKINSON'S LAW

It won't work.

MURPHY'S LAWS

1) If anything can go wrong, it will (and at the worst possible moment).

2) Nothing is as easy as it looks.

3) Everything takes longer than you think it will.

MURPHY'S FOURTH LAW

If there is a possibility of several things going wrong, the one that will cause the most damage will be the one to go wrong.

NINETY-NINETY RULE OF PROJECT SCHEDULES

The first ninety percent of the task takes ninety percent of the time, and the last ten percent takes the other ninety percent.

O'TOOLE'S COMMENTARY ON MURPHY'S LAW

Murphy was an optimist.

PEER'S LAW

The solution to a problem changes the problem.

RHODE'S COROLLARY TO HOARE'S LAW

Inside every complex and unworkable program is a useful routine struggling to be free.

RULE OF ACCURACY

When working toward the solution of a problem it always helps you to know the answer.

THOREAU'S THEORIES OF ADAPTATION

1) After months of training and you finally understand all of a program's commands, a revised version of the program arrives with an all-new command structure.

2) After designing a useful routine that gets around a familiar "bug" in the system, the system is revised, the "bug" taken away, and you're left with a useless routine.

3) Efforts in improving a program's "user friendliness" invariably lead to work in improving user's "computer literacy".

4) That's not a "bug", that's a feature!

Bibliography

[1] McIlrath, Michael B., Troxel, Donald E., Heytens, Michael L., Penfield, Paul Jr., Boning, Duane S., and Jayavant, R. "CAFE - The MIT Computer-Aided Fabrication Environment," I.E.E.E Transactions on Components, Hybrids, and Manufacturing Technology, Vol. 15 No. 2, May 1992.

[2] Boning, Duane S. "Semiconductor Process Design: Representations, Tools, and Methodologies," Ph.D. Thesis, Massachusetts Institute of Technology, June 1991.

[3] Boning, Duane S., and McIlrath, Michael B. "Guide to the Process Flow Representation: Version 3.1," Internal CIDM Memo Series No. 93-17, September 23, 1993.

[4] Fischer, Gregory T., and Troxel, Donald E. "Writing PFRs for Use in Fabrication," Internal CIDM Memo Series No. 93-18, October 13, 1993.

[5] Troxel, Donald E. "Tasks and PFR based Fabrication," Internal CIDM Memo Series No. 94-1, January 10, 1994.

[6] Rowe, Lawrence A., Williams, Christopher, and Hegarty, Christopher. "The Design of the Berkeley Process-Flow Language," Computer Science Division - EECS, University of California, Berkeley, CA 94720, July 25, 1990.

[7] Carney, John C. "An Implementation of a Layered Message Passing System," Proposal for Master's Thesis Research, Internal CIDM Memo Series No. 94-15, October 13, 1994.

[8] Carney, John C. "Message Passing Tools for Software Integration," M.S. Thesis, Massachusetts Institute of Technology, June 1995.

[9] Steele, Guy L. Jr., "Common Lisp: The Language (Second Edition)," Digital Press, c1994.

[10] Keene, Sonya E. "Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS," Symbolics, Inc., Addison-Wesley Publishing Company, c1989.

[11] Schwartz, J., and Westfechtel, B. "Integrated Data Management in a Heterogeneous CIM Environment," Computers in Design, Manufacturing, and Production: 1993 I.E.E.E. CompEuro, I.E.E.E. Computer Society Press, Los Alamitos, CA, c1993, pp. 248–257.

RDA = Remote Database Access

[12] Messina, S. "RDA Integration in an Open System Architecture for CIM," Computers in Design, Manufacturing, and Production: 1993 I.E.E.E. CompEuro, I.E.E.E. Computer Society Press, Los Alamitos, CA, c1993, pp. 232–238.

[13] Scalas, Maria Rita, Cappelli, Alessandro, and De Castro Christina. "A Model for Schema Evolution in Temporal Relational Databases," Computers in Design, Manufacturing, and Production: 1993 I.E.E.E. CompEuro, I.E.E.E. Computer Society Press, Los Alamitos, CA, c1993, pp. 223–231.

ESPRIT = European Specific Research and Technological Development Programme in the field of Information Technology

[14] "CIMOSA: Open System Architecture for CIM," ESPRIT Consortium AMICE, New York, c1993.

[15] Scheer, August-Wilhelm. "CIM: Computer Integrated Manufacturing: Towards the Factory of the Future," New York, c1991.

[16] Foster, Alden T., Goodman, Richard V., and Roos, Daniel. "Remote Computer Processing for Computer-Aided Teaching Applications," Cambridge, M.I.T. Civil Engineering Systems Laboratory Research Report R69-05, 1969.

[17] Boubekri, Nourredine, Khalil, Tarek M., and Kabuka, Mansur. "Human-Machine Interface in Remote Monitoring and Control of Flexible Manufacturing Systems," Ergonomics of Hybrid Automated Systems I. Proceedings of the First International Conference on Ergonomics of Advanced Manufacturing and Hybrid Automated Systems, New York, NY, 1988, pp. 311–318.

[18] Ramanathan, Geetha, and Alagar, V.S. "Specification of Real-time Distributed Database Systems," Computer Systems and Software Engineering: 1992 I.E.E.E. CompEuro, I.E.E.E. Computer Society Press, Los Alamitos, CA, c1992, pp. 101–106.

[19] Kündig, Albert. "From Computer Communication to Computer Supported Co-operative Work," Advanced Computer Technology, Reliable Systems and Applications: 1991 I.E.E.E. CompEuro, I.E.E.E. Computer Society Press, Los Alamitos, CA, c1991, pp. 184–190.

[20] Albano, Antonio, and Ghelli, Giorgio. "Object-Oriented Database Programming Languages," Advanced Computer Technology, Reliable Systems and Applications: 1991 I.E.E.E. CompEuro, I.E.E.E. Computer Society Press, Los Alamitos, CA, c1991, pp. 726–734.

[21] Ceri, S., Tanca, L., and Zicari, R. "Supporting Interoperability between New Database Languages," Advanced Computer Technology, Reliable Systems and Applications: 1991 I.E.E.E. CompEuro, I.E.E.E. Computer Society Press, Los Alamitos, CA, c1991, pp. 273–281.

[22] Scholtz-Reiter, B. "CIM Interfaces: Concepts, standards and problems of interfaces in Computer-Integrated Manufacturing," Chapman & Hall, New York, c1992.

[23] Ranky, Paul G. "Computer Networks for World Class CIM Systems," Biddles Limited, The Book Manufacturers, Guildford, Surrey, UK, c1990.

[24] Perris, F. A. "An integrated approach to project and quality management in engineering software development," Proceedings of the Institution of Mechanical Engineers, Eurotech Direct '91: Computers in Engineering Industry, Mechanical Engineering Publications Limited, c1991, pp. 1–6.

[25] Kubota, Katsutoshi, Okuba, Tsuneo, and Tokeuchi, Hideaki. "An Integrated Database System for Effective Correlation Analysis to Improve LSI Manufacturing Yield," 1993 Proceedings, Fifteenth I.E.E.E./C.H.M.T. International Electronics Manufacturing Technology (I.E.M.T.) Symposium, October 1993, pp. 91–96.

[26] Guo, Ruey-Shan, Slama, Mike, Griffin, Ralph, and Holman, Ken. "A Work Cell Manufacturing System for VLSI Fabrication," 1993 Proceedings, Fifteenth I.E.E.E./C.H.M.T. International Electronics Manufacturing Technology (I.E.M.T.) Symposium, October 1993, pp. 200–205.

[27] Yoshizawa, Masahiro, and Sakurai, Tetsuma. "An LSI Delivery Management Method using Lot-Sampling Scheduling," 1993 Proceedings, Fifteenth I.E.E.E./C.H.M.T. International Electronics Manufacturing Technology (I.E.M.T.) Symposium, October 1993, pp. 195–199.

[28] Meieran, Eugene S. "Intelligent Manufacturing Systems," 1993 Proceedings, Fifteenth I.E.E.E./C.H.M.T. International Electronics Manufacturing Technology (I.E.M.T.) Symposium, October 1993, pp. 323–327.

[29] Baudoin, Claude R., and Kantor, Jeffrey P. "Software Engineering for Semiconductor Manufacturing Equipment Suppliers," 1993 Proceedings, Fifteenth I.E.E.E./C.H.M.T. International Electronics Manufacturing Technology (I.E.M.T.) Symposium, October 1993, pp. 337–352.

[30] Nutt, Gary J. "Open Systems," Prentice Hall Series in Innovative Technology, Prentice–Hall, Inc., Englewood Cliffs, New Jersey, c1992.