A Distributed Software Architecture for Semiconductor Process Design

by

Matthew D. Verminski

B.S. in Computer Engineering, Tufts University (1996)

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

February 1998

© 1998 Massachusetts Institute of Technology
All rights reserved.

Signature of Author: ______________________________
Department of Electrical Engineering and Computer Science
January 15, 1998

Certified by: ______________________________
Michael B. McIlrath
Research Scientist, Electrical Engineering and Computer Science
Thesis Supervisor

Certified by: ______________________________
Donald E. Troxel
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: ______________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MAR 27 1998

A Distributed Software Architecture for Semiconductor Process Design

by

Matthew D. Verminski

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

Computer representations for semiconductor fabrication processes are currently not well defined for distributed computing. This work focuses on defining and implementing an open, distributed software architecture for designing semiconductor fabrication processes. The implementation of the architecture provides a common interface to semiconductor process repositories, a life cycle service to manage the distributed objects, various means for organizing distributed repositories, and clear component definitions to enable a trader to find and manage available services. The base interfaces and services are essential in the development of distributed and shared applications for semiconductor process design.

Thesis Supervisor:	Michael B. McIlrath
Title:	Research Scientist, Electrical Engineering and Computer Science
Thesis Supervisor:	Donald E. Troxel
Title:	Professor of Electrical Engineering and Computer Science

Acknowledgments

There are many people who have provided me with feedback, help, and support during the past year and a half of research. I cannot possibly thank them all, but I will try.

I would like to thank Professor Donald Troxel for providing me with the research assistantship that has funded my education at MIT. He has provided valuable feedback and insight as an advisor. I would also like to thank Michael McIlrath for providing a research topic and guiding me to its completion. Our weekly meetings kept the project on schedule and were the source of many critical ideas. William Moyne has shown the patient and courage necessary to program the first application with the software architecture. I would like to thank him for his help with the semiconductor process browser and the numerous poster sessions. Myron "Fletch" Freeman was instrumental in helping me recover from a few system crashes along the way.

I would like to thank my officemates, Yonald Chery, Manuel Perez, Brian Lee, Erik Pedersen, Jared Cottrell, Thomas Lohman, and Francis Doughty, for providing a friendly and innovative environment in which to work.

Many friends allowed me to balance my research and class work with outside activities. Brian Herrick, Justin Steele, Aaron Bradshaw, Christopher Centurelli, and Thomas Lampron ensured that I had breaths of real air and social interaction. Coach Don Megerle continues to be an unending source of advice and friendship. I have been incredibly fortunate to have so much emotional support from my mom, my dad, and my brother, Brian.

My work has been made possible by the support from the Defense Advanced Research Projects Agency (DARPA) contract #DABT 63-95-C-0088.

Table of Contents

1 INTRODUCTION 8

1.1 RELATED WORK 8
1.1.1 FABLE 9
1.1.2 Process Design Aid (PDA) 10
1.1.3 Berkeley Process Flow Language (BPFL) 11
1.1.4 Process Flow Representation (PFR) 11
1.1.5 Semiconductor Process Representation (SPR) Information Model 12
1.2 DISTRIBUTED SYSTEMS 13
1.2.1 Common Object Request Broker Architecture (CORBA) 13
1.2.2 SEMATECH CIM Framework 15
1.3 THESIS STATEMENT 17
1.4 ORGANIZATION OF THESIS 17

2 SYSTEM ARCHITECTURE 18

2.1 IDL FOR SEMICONDUCTOR PROCESS REPRESENTATION (SPR) 19
2.2 LIFE CYCLE SERVICE 20
2.3 REPOSITORY ORGANIZATION 21
2.4 TRADER SERVICE 22
2.5 APPLICATIONS 23

3 SEMICONDUCTOR PROCESS REPRESENTATION 25

3.1 OVERVIEW 25
3.2 DESCRIPTION 25
3.3 INTERFACES 26
3.3.1 Base Data Structures 27
3.3.2 Base Objects 28
3.3.3 Processes & Views 29
3.3.4 Process View 30
3.3.5 Effects 30
3.3.6 Equipment View 34
3.3.7 Environment View 34
3.4 SERVER IMPLEMENTATION 35
3.5 PERSISTENT STORAGE 37
3.6 EXAMPLE CLIENT INTERACTIONS 37

4 LIFE CYCLE SERVICE 39

4.1 OVERVIEW 39
4.2 DESCRIPTION 39
4.2.1 Factories 39

4.2.2 *Locating Factories* ... 40
4.3 INTERFACES ... 40
4.3.1 *Base Structures* ... 40
4.3.2 *Generic Factory* ... 41
4.3.3 *Factory Finder* ... 42
4.3.4 *Life Cycle Object* ... 42
4.4 IMPLEMENTATIONS ... 43
4.4.1 *Generic Factory* ... 43
4.4.2 *Factory Finder* ... 45
4.4.3 *Life Cycle Object* ... 45
4.5 EXAMPLES ... 45

5 ORGANIZING AND SEARCHING THE REPOSITORY ... 48

5.1 OVERVIEW ... 48
5.2 LIBRARY SERVICE ... 48
5.2.1 *Description* ... 48
5.2.2 *Interfaces* ... 49
5.2.3 *Implementation* ... 50
5.2.4 *Example* ... 50
5.3 QUERY EXTENSIONS ... 51
5.3.1 *Description* ... 51
5.3.2 *Interface* ... 52
5.3.3 *Implementation* ... 52
5.3.4 *Example* ... 53

6 LOCATING SERVICES ... 54

6.1 OVERVIEW ... 54
6.2 DESCRIPTION ... 54
6.3 SERVICE TYPES ... 56
6.3.1 *SPR Repository* ... 56
6.3.2 *SPR Library* ... 56
6.3.3 *Factory Finder* ... 57
6.4 EXAMPLES ... 57

7 SEMICONDUCTOR PROCESS BROWSER ... 61

7.1 OVERVIEW ... 61
7.2 FINDING SERVICES ... 61
7.3 LIBRARY SERVICE ... 62
7.4 SPR SERVICE ... 63
7.5 UTILIZING THE LIFE CYCLE SERVICE ... 64

8 CONCLUSION ... 65

REFERENCES ... 67

APPENDIX A - SPR IDL ... 69

APPENDIX B - SPR ODL ... 76

APPENDIX C - LIFE CYCLE IDL ... 82

APPENDIX D - CATALOG IDL ... 85

APPENDIX E - TRADER SERVICE DEFINITIONS ... 87

List of Figures

Figure 1: Semiconductor process design cycle 9
Figure 2: FABLE process layers 10
Figure 3: PFR layer abstractions 11
Figure 4: CAFE architecture 12
Figure 5: Object Management Architecture 15
Figure 6: CIM framework 16
Figure 7: Distributed software architecture for semiconductor process design 18
Figure 8: SPR server development 20
Figure 9: SPR views 20
Figure 10: Life cycle service interacting with SPR service 21
Figure 11: Library service indexing repository 22
Figure 12: A trader interacting with services 22
Figure 13: Selecting a simulator using the trader service. 23
Figure 14: Selecting a remote fabrication service using the trader service. 23
Figure 15: Process step divided into multiple views. 26
Figure 16: Deposit effect 32
Figure 17: Conformal deposit effect 32
Figure 18: Planar deposit effect 32
Figure 19: CORBA skeleton and stub interaction 36
Figure 20: SPR Java implementation hierarchy 36
Figure 21: Persistent SPR classes 37
Figure 22: Client and factory interaction 40
Figure 23: Factory finder interface used in conjunction with other location services. 40
Figure 24: Client creating a process object 44
Figure 25: Library service 49
Figure 26: Adding service types 58
Figure 27: Query with constraints 60
Figure 28: Query with preference 60
Figure 29: Browser interacting with trader service 62
Figure 30: Browser interacting with library service 63
Figure 31: Browser interacting with SPR service 64

List of Tables

Table 1: Base SPR data structures 28
Table 2: Foundation SPR interface objects 29
Table 3: Generic view and process interfaces 30
Table 4: Process view interface 30
Table 5: Effects view interface 31
Table 6: Simple effect location data structure 31
Table 7: Effect specific parameter interfaces 31
Table 8: Effects interfaces 33
Table 9: Equipment view interface 34
Table 10: Machine and equipment interfaces 34
Table 11: Environment view interface 35
Table 12: Basic environment interfaces 35
Table 13: Simple client navigation of a semiconductor process 38
Table 14: Basic life cycle data structures 41
Table 15: Generic factory interface 42
Table 16: Factory finder interface 42
Table 17: Life cycle object interface 43
Table 18: Life cycle client example 46
Table 19: Life cycle client with helper class 47
Table 20: Output of client interaction 47
Table 21: Catalog data structure 49
Table 22: Library interface 50
Table 23: Library example 51
Table 24: Output of library example 51
Table 25: Query data structures 52
Table 26: Query capable factory interface 52
Table 27: Query extensions example 53
Table 28: Example service type 55
Table 29: Repository service definition 56
Table 30: Library service definition 57
Table 31: Factory finder service definition 57
Table 32: Service offer for SPR Library 58

Chapter 1

Introduction

Building integrated circuits requires semiconductor fabrication processes. Current software systems for designing these fabrication processes are not built for distributed computing. This research implements and evaluates a distributed, platform independent software architecture for semiconductor process design. The architecture provides well-defined application programming interfaces for process repositories and surrounding services. To automate common network programming tasks, the interfaces use the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA).

This chapter begins by providing an introduction to previous work done in the area of semiconductor process representation. Next, distributed systems are discussed. A thesis statement follows this. The chapter concludes by presenting the organization of the thesis.

1.1 Related Work

A semiconductor process is affected and influenced by numerous steps during its life cycle from origination to fabrication and maintenance. New processes generally emanate from evolutionary changes to existing processes or the creation of new technologies. An example of an evolutionary change occurs when improved equipment is used. The introduction of a new technology, e.g. Silicon Germanium, might require creating an entirely new fabrication process.

When a new process is designed, the description can begin as a desired performance parameter specification. A process is then generated and simulated to satisfy the given constraint, adding specific information such as machine settings, environments, and effects to describe the process. An integrated circuit is then fabricated to collect data on actual performance characteristics of the process. The results are cycled back through each stage until the performance characteristics are finalized (see Figure 1).

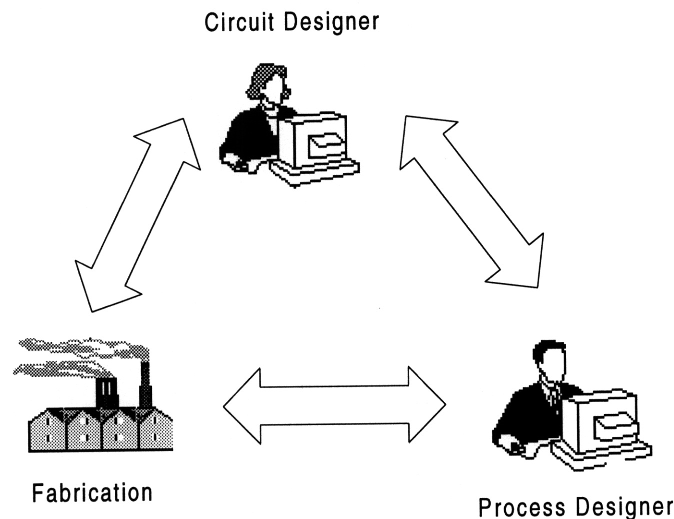

Figure 1: Semiconductor process design cycle

The design cycle for a semiconductor process demonstrates the need for an information model to communicate process information. The structure must encompass data that is viewed differently depending upon its position in the cycle. A variety of Technology Computer Aided Design (TCAD) tools and Computer-Aided Manufacturing (CIM) systems have been researched and designed to address this issue. The following sections outline several of these systems.

1.1.1 FABLE

Stanford University's FABLE was the first attempt to develop a structured representation for semiconductor processes [1]. The project generated a procedural language for representing processes. The language was highly structured with multiple abstraction levels for modeling semiconductor fabrication processes.

FABLE programs represent process steps with hierarchical layers. The five layers are *process*, *effect*, *treatment*, *settings*, and *physical* (see Figure 2). The *process* layer,

the highest level of abstraction, provides a general description of a process step (e.g. etch-oxide). The *effect* layer details any operation on the wafer (e.g. what and where is the oxide etched). The *treatment* layer corresponds to the specific operations needed to obtain the desired wafer state (e.g. how the oxide is etched). The *settings* layer describes the specific equipment needed for the step and its corresponding settings. Finally, the *physical* layer details the processing of the wafer.

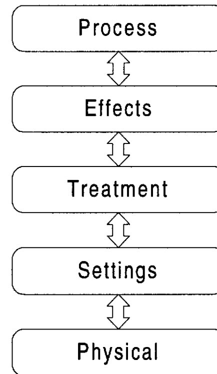

Figure 2: FABLE process layers

The FABLE project was not successful due to several limitations. The detailed structure proved very restrictive and not useable in a real manufacturing environment [2]. The language was not extensible and could not keep pace with rapidly evolving process technology. Lastly, FABLE was not able to interface to applications other than manufacturing.

1.1.2 Process Design Aid (PDA)

Building on the FABLE research, Stanford generated a second representation for semiconductor processes as part of the Process Design Aid (PDA) project [3]. An object-oriented process representation was coupled with PDA. It divided processes into a hierarchical structure where each process step was a class. The process step was constructed from the base class or inherited from it with additional attributes. The PDA process representation was flexible and extensible. This proved to be advantageous in that a user was free to add any attributes desired, and detrimental in that it was possible for some process descriptions to not be interpreted correctly by applications. Unlike FABLE, PDA interfaced well with simulators but lacked robust manufacturing support.

1.1.3 Berkeley Process Flow Language (BPFL)

The Berkeley Process Flow Language (BPFL) [2] was developed during the same time as PDA. BPFL was a programming-language-based approach for representing semiconductor processes. It represented semiconductor process information with a multiple *view* structure. For example, specific instructions for performing a process were in a *fabrication* view while a simulation description was in the *simulation* view. Other views included *equipment* and *material.* A wafer state model with standard control and processing instructions were also defined.

BPFL proved to be effective in a manufacturing environment due to its exception handling and systematic abstraction for process steps. Since BPFL did not provide extensibility, it became difficult to maintain the process representation.

1.1.4 Process Flow Representation (PFR)

As part of the Computer Aided Fabrication Environment (CAFE) at MIT, the Process Flow Representation (PFR) was developed [4]. PFR provides a representation for semiconductor processes. It follows FABLE by abstracting representation layers for processes. PFR uses a three level abstraction model. The *change-wafer-state* layer describes changes the semiconductor wafer undergoes during an operation. The *treatment* layer describes the physical environment surrounding the wafer during the changes. The *settings* layer details parameters for the equipment used to achieve the *treatment* (see Figure 3).

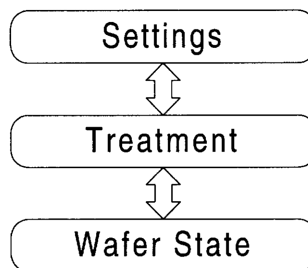

Figure 3: PFR layer abstractions

In addition, PFR supports unification of process data [4]. This allows for the association of multiple descriptions of processes. For example, a process simulator such as SUPREM-III [5] uses a specific format for input files. PFR unifies its internal view of the process with external views such as simulator input files.

PFR coupled with the overall CAFE architecture [6], provides a system for communicating process information between engineers, designers, and manufacturers of semiconductors (see Figure 4). However, the repository is limited to a specific hardware and software platform. This makes it difficult to share information with other repositories.

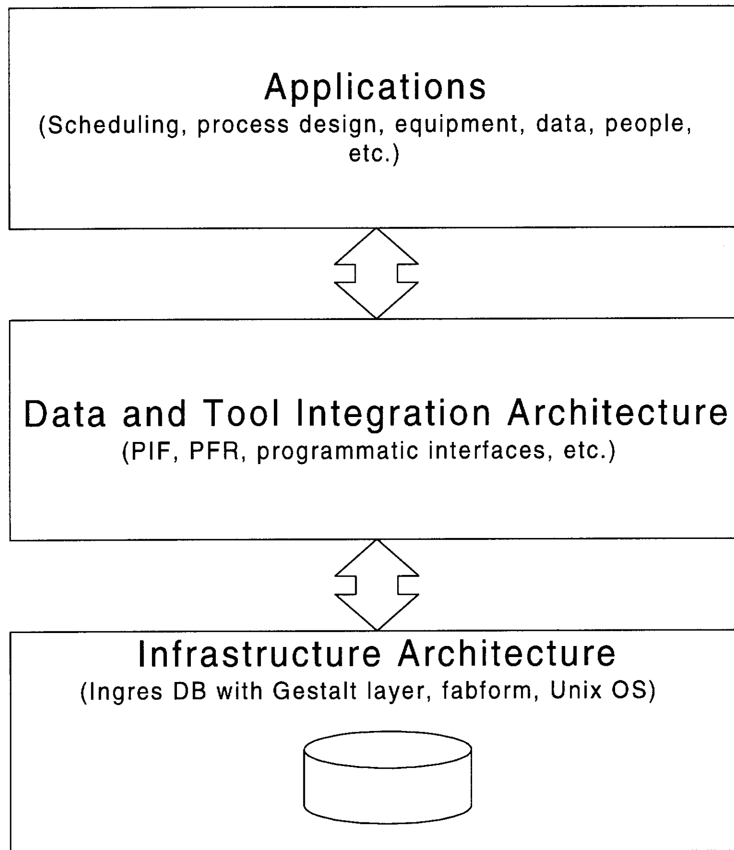

Figure 4: CAFE architecture

1.1.5 Semiconductor Process Representation (SPR) Information Model

There are other proprietary, commercial implementations in addition to the models detailed in the previous sections. Semiconductor representations are unique to specific laboratories and companies making it difficult to transfer process information between facilities. The varying models for semiconductor processing are not well suited for communicating information between repositories. Researchers at MIT and elsewhere in the TCAD/CIM community have addressed the need for a common representation of semiconductor processes by developing the Semiconductor Process Representation Information Model (SPR) [7].

SPR organizes process information into different views. The four main views are *effects*, *environment*, *equipment*, and *sub-processes*. The *equipment* view contains information about the machines for a process step. During the fabrication, the *equipment* produces a particular *environment* around a wafer, which in turn causes the *effects* on a wafer [7]. *Sub-processes* provide structure for dividing process information amongst multiple smaller steps.

A common interface to access semiconductor process repositories is essential for process designers to share and exchange information between repositories. The SPR information model incorporates the strengths of proven systems, in addition to addressing the weakness of previous models by providing extensible components for structured growth and expansion. The SPR information model can be used for exchanging information about processes within a distributed software architecture for semiconductor process design.

1.2 Distributed Systems

Software for designing semiconductor processes (e.g., process simulation, editing, and synthesis) is currently programmed for specific software and hardware platforms. Platform interoperability is essential for a distributed software architecture to incorporate legacy systems. The system should be able to integrate existing applications and reuse available services.

A distributed, platform independent software architecture provides numerous benefits for semiconductor processing research. Process designers can choose between any simulator, editor, or synthesis tool without being restricted by a particular operating system and hardware platform. Semiconductor fabrication benefits from the distributed software architecture. Due to expenses associated with fabricating semiconductors, lack of state-of-the-art equipment often limits research facilities. A distributed software architecture facilitates the sharing of remote resources and equipment.

1.2.1 Common Object Request Broker Architecture (CORBA)

The Object Management Architecture (OMA) is a consortium specification created by the Object Management Group (OMG). Including over 700 company members, the OMG is an organization whose purpose is to create vendor independent specifications for

object-oriented distributed computing [8]. The OMA's core is the Common Object Request Broker Architecture (CORBA). It provides interoperability among a variety of hardware and software platforms. CORBA defines an "Interface Definition Language (IDL) and Application Programming Interfaces (API) that enable client/server object interaction within implementations of Object Request Brokers (ORB)" [8]. In essence, CORBA is a technology that readily allows the development of distributed systems.

In addition to CORBA, the OMA consists of interfaces and functional specifications for common services and facilities (see Figure 5). These are defined as CORBA services and CORBA facilities. CORBA services focus on common object services for managing a distributed system [9]. Included are specifications for naming, event handling, persistent objects, life cycles, trader, querying, etc. CORBA facilities are of two types – domain (vertical) and common (horizontal) [10]. Domain facilities concentrate on specific vertical domains such as manufacturing, healthcare, telecommunications, and finance. Common facilities provide application functionality such as user interfaces, help facilities, and document management. The benefits of software reuse and interoperability is realized by using agreed upon standard interfaces for services. Applications utilizing the common interfaces are allowed to choose services between providers implementing the functionality.

To automate common networking programming tasks, programming interfaces for the distributed software architecture for semiconductor process design utilize CORBA. Furthermore, the architecture employs reusable CORBA services when appropriate.

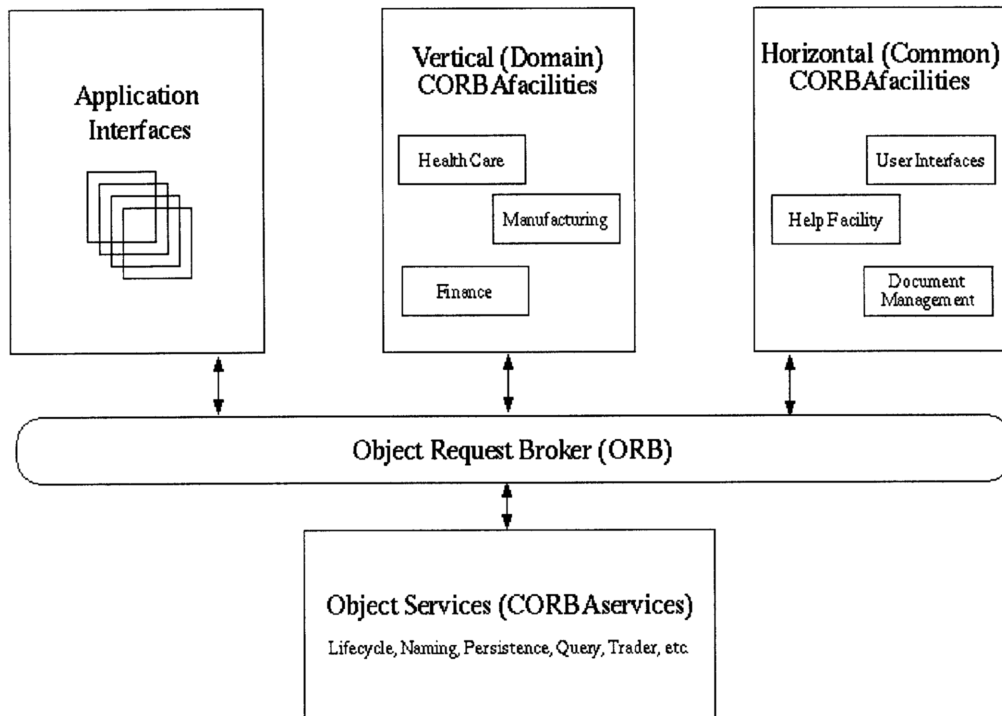

Figure 5: Object Management Architecture

1.2.2 SEMATECH CIM Framework

The OMG is focused on defining specifications for the common object services and facilities that are applicable to generic distributed systems. Industry consortia, such as SEMATECH, are left to generate interfaces for domain-specific facilities. SEMATECH, a consortium of U.S. semiconductor manufactures, works with government and academia to sponsor and conduct research in the area of semiconductor manufacturing. A CORBA based CIM framework is currently being developed by SEMATECH for semiconductor manufacturing. The CIM Framework is a vertical CORBA facility within the OMA (see Figure 5).

The objective of the SEMATECH CIM Framework is to provide a software infrastructure for integrating applications and sharing information for the manufacturing of semiconductors. The framework's intent is to establish an open industry standard for creating semiconductor CIM systems [11].

The CIM Framework defines the following component modules for managing a semiconductor manufacturing system [12]:

- Factory management
- Document management
- Labor management
- Machine management
- Material management
- Process definition management
- Process specification management
- Schedule management

The modules provide a basis for CIM applications such as process control, work-in-process tracking, machine control, etc (see Figure 6).

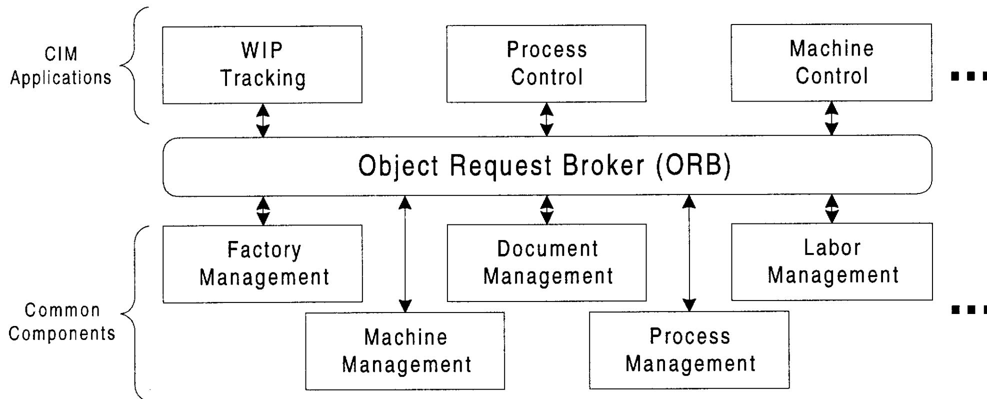

Figure 6: CIM framework

The focus of the CIM framework is to provide the appropriate manufacturing abstractions and services for the semiconductor industry. The scope of the framework encompasses all aspects of semiconductor manufacturing from billing and shipping to machine management. Within the framework, there is a need to communicate information about semiconductor processes. The current framework [12] has allocated components for process definition and specification management. Final component definitions have not yet been specified by SEMATECH. One intent of the research presented in this paper is to develop a process design representation that can be used in conjunction with the SEMATECH CIM framework.

1.3 Thesis Statement

Current computer representations for semiconductor processes are not well defined for distributed computing. Hardware and software dependencies, varied protocols for inter-communicating, and undefined interfaces for managing distributed data are several weaknesses of existing representations. The research presented in this thesis focuses on defining and implementing a distributed software architecture for designing semiconductor processes. The architecture addresses the need for a common representation for processes and implements surrounding services for managing the system based upon industry standards. The implementation of the services is done in Java [13] to provide platform independent components that can be used as the foundation for further distributed system development.

1.4 Organization of Thesis

This chapter has served to provide a brief background on current systems used for semiconductor process design. In addition, an overview of distributed systems was discussed. Finally, the problems and challenges of defining a distributed framework for semiconductor process design were presented.

In the following chapter, an overview of the entire system is presented. Then, Chapters 3 through 6 discuss the details of specific modules by exploring their interfaces and implementations. Applications that utilize the architecture are detailed in Chapter 7. Finally, Chapter 8 summarizes the thesis and details areas for further research.

Chapter 2

System Architecture

The distributed software architecture for semiconductor processing provides reusable, inter-operable services (see Figure 7). The architecture adheres to the philosophy of CORBA by using standard OMG interfaces for common object services such as *life cycle* and *trader* [14]. The combination of core services provides an architecture for distributed semiconductor process design.

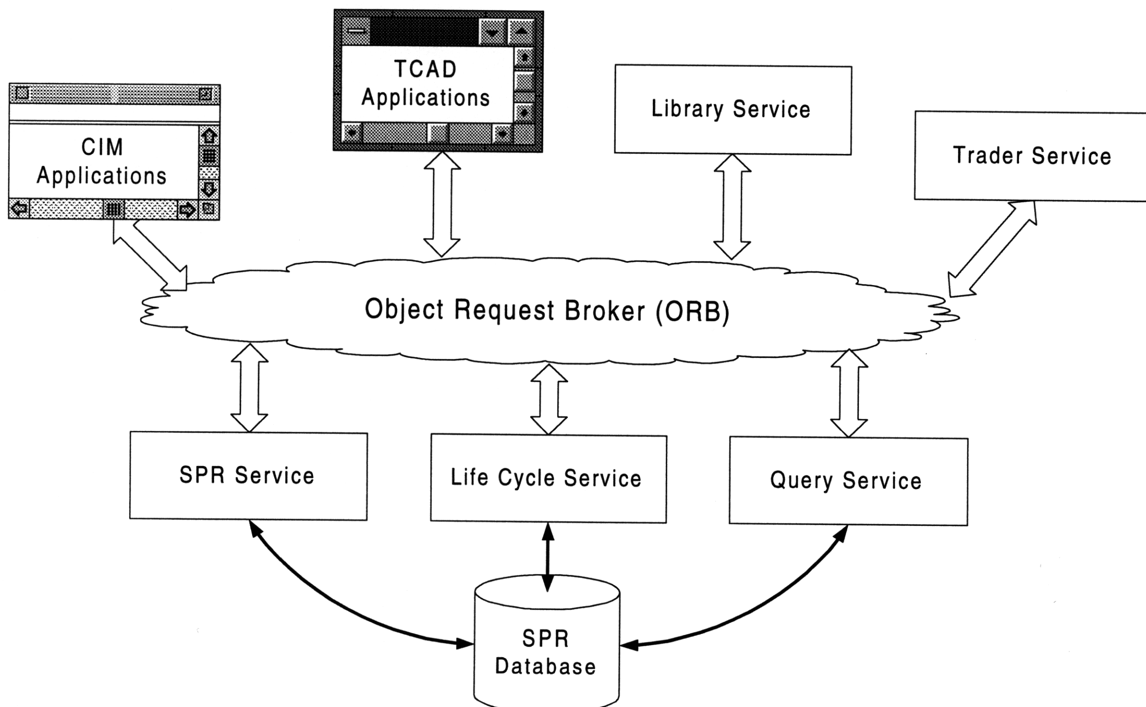

Figure 7: Distributed software architecture for semiconductor process design

The SPR service is the basis for accessing process information in the distributed software architecture. It utilizes a generic information model for describing semiconductor processes. Joining SPR are three supporting services for organizing and managing the distributed process objects. The *life cycle* service provides interfaces for creating, copying, moving, and removing the distributed objects. Extensions to the *life cycle* service enable querying process databases. A *library* service provides a common interface for organizing collections of processes. Lastly, a *trader* is used to find and manage available services.

In parallel with the implementation of the distributed software architecture, a semiconductor process browser was developed. The process browser application demonstrates aspects of the architecture implementation. It allows a user to view processes in distributed repositories. The process browser interacts with the four main services. It uses the *trader* service to connect to a *library* service. The *life cycle* service can be integrated for managing the creation and movement of the distributed objects. Once the browser has access to repositories, it uses the SPR interface to access information about processes.

2.1 IDL for Semiconductor Process Representation (SPR)

The SPR information model provides a standard medium for communicating information about fabrication processes [7]. It combines the needs of process designers (TCAD) and manufacturers (CIM). These characteristics make it suitable for defining the programming interfaces for a distributed software architecture for semiconductor process design. The SPR programming interfaces are defined in CORBA IDL and are used to access existing as well as new process repositories. The use of a standard interface allows process designers and manufacturers to integrate services and applications via a standard communications mechanism.

The implementation of the SPR programming interface consisted of three phases (see Figure 8). The first defined the IDL based upon the SPR information model document [7]. The second phase implemented the server in Java for developing and refining the interface definitions. After establishing the interfaces, the third phase created a persistent storage backend.

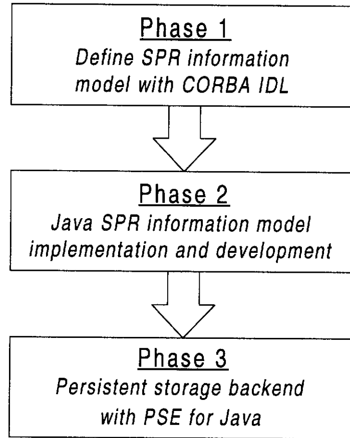

Figure 8: SPR server development

The server implementations closely reflect the SPR information model. The model provides an organized representation for process information. SPR organizes process information into four main views – *effects*, *environment*, *equipment*, and *process* (see Figure 9). The *process* view is a division of a process into sub-processes. The other views (*effects*, *environment* and *equipment*) describe what occurs during the *process*. The *equipment* for a process creates an *environment* that causes the *effects* [7].

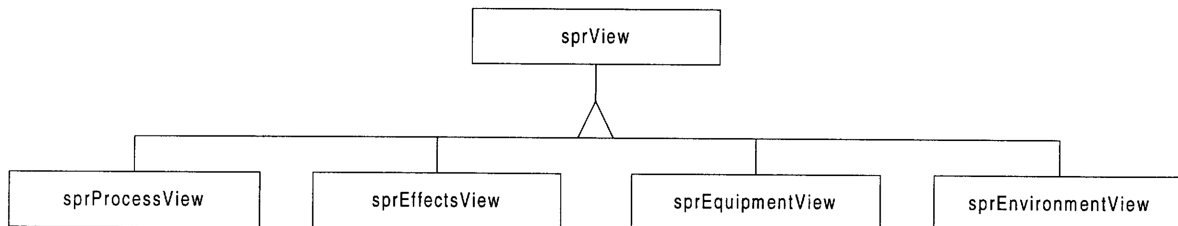

Figure 9: SPR views

2.2 Life Cycle Service

The *life cycle* service provides functionality for creating, deleting, copying, and moving distributed objects. Distributed systems need conventions for clients to manage and perform such operations on remote objects. The implementation of a *life cycle* service coupled with the SPR service is the core of the SPR repository.

The OMG defined *life cycle* service consists of three interfaces – a life cycle object, a factory finder, and a generic factory [9]. The base interface is the life cycle object. Object interfaces must inherit from the life cycle object to implement life cycle functionality. The life cycle object defines operations for *copying*, *moving*, and *removing* an object. Each subclass of the life cycle object implements these methods. The factory

finder interface provides a *find factories* operation used for obtaining factory references. A single reference to a factory finder interface enables a client to access any factory within its scope. The factory finder is a directory of factories that exist within its scope. The generic factory interface is the base for creating objects. Its *create object* method uses criteria to produce instances of objects.

There are various issues that are addressed in the *life cycle* service implementation. For copying and moving, the definition of shallow versus deep is clearly specified (e.g., copying a reference to an object versus copying an object's data). The removal of an object involves complex issues such as distributed garbage collection and dangling references. The *life cycle* service was developed in tandem with the SPR service (see Figure 10). Furthermore, the Java implementations of these services are coupled with a persistent storage backend.

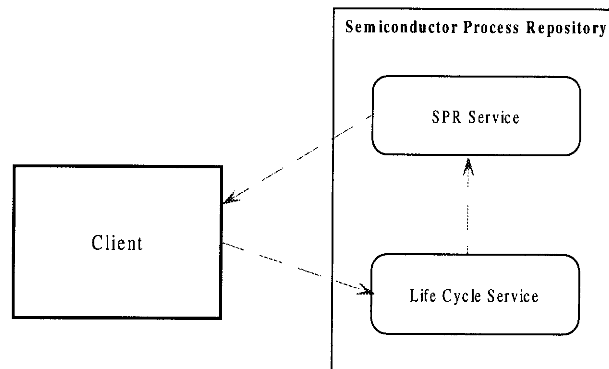

Figure 10: Life cycle service interacting with SPR service

2.3 Repository Organization

The organization of repositories was defined in a service separate from SPR to allow multiple abstractions and implementations. A *library* service organizes processes into catalogs. The processes of a catalog can be stored in repositories distributed across multiple systems. The *library* service organizes a distributed process repository as a single entity (see Figure 11).

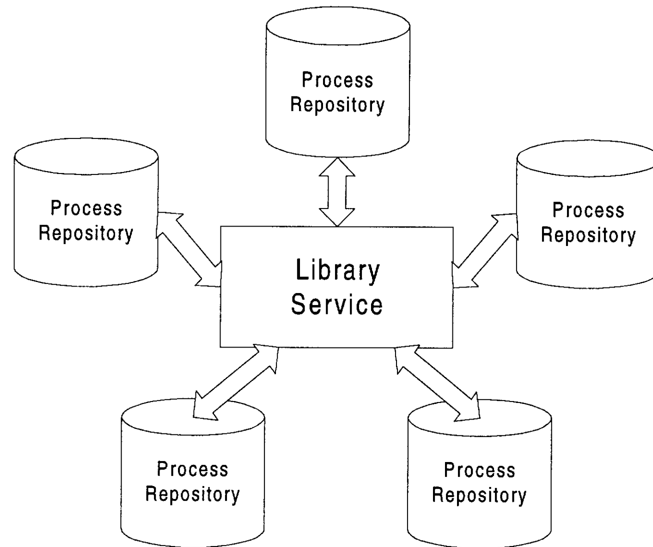

Figure 11: Library service indexing repository

For additional repository organization, *query* extensions were added to the *life cycle* service. This allows querying to utilize the underlying repository databases. The *query* extensions provide a generic interface masking the underlying database engine.

2.4 Trader Service

A *trader* facilitates the exporting and importing of services. A service advertises its functionality to a *trader* (exporting). The trader then takes requests from clients and matches the desired functionality with an advertised service (importing). Figure 12 shows the interaction of an advertised service (exporter), a client (importer), and a *trader*.

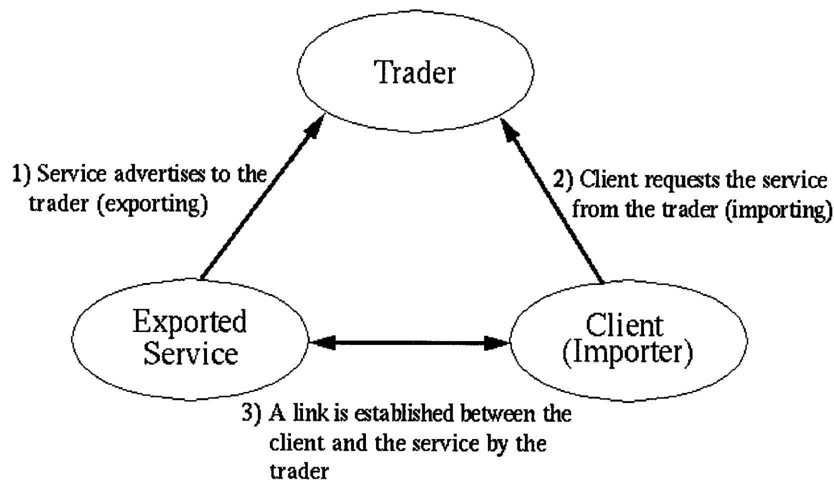

Figure 12: A trader interacting with services

The distributed software architecture incorporates a generic *trader* that client applications use to obtain services. For example, when a client needs a specific simulator, it passes a request to the *trader*. The *trader* then returns references to any qualified simulators. The program or user can select a simulator from these references (see Figure 13). Suppose that the client requests a remote fabrication service that uses a BICMOS process. The *trader* returns references to all the remote fabrication services that "advertise" a BICMOS process (see Figure 13). Finding a specific simulator or remote fabrication service are two examples of using the trader.

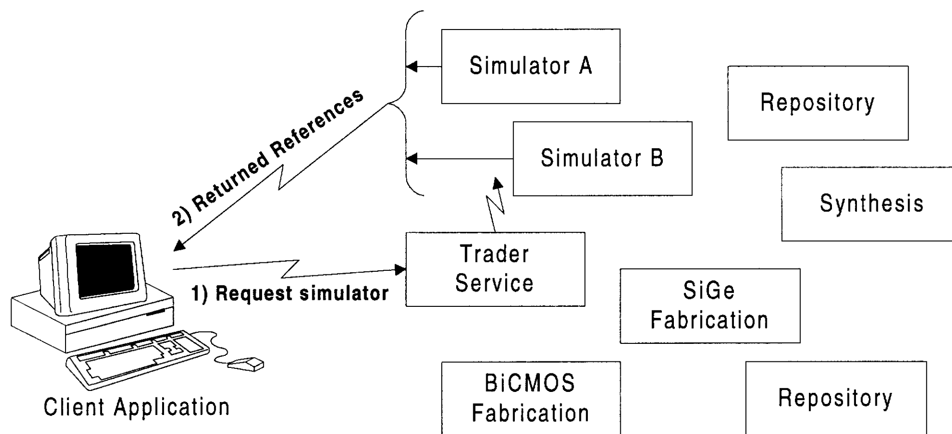

Figure 13: Selecting a simulator using the trader service.

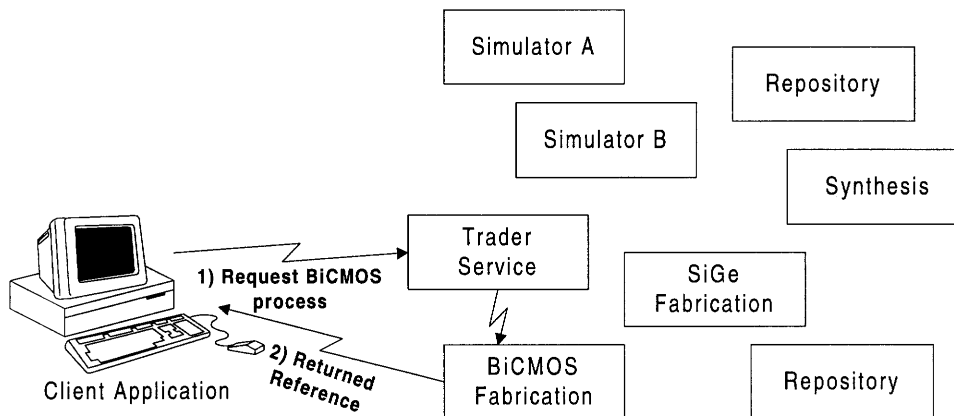

Figure 14: Selecting a remote fabrication service using the trader service.

2.5 Applications

In order to test and demonstrate the functionality of the distributed software architecture, a semiconductor process browser was developed. The process browser is a comparable concept to a web browser in that it accesses data from a heterogeneous collections of

services and machines across a network. The browser uses the distributed software architecture for semiconductor process design as the protocol for interacting with services. The browser allows a user to view processes. It can also be enabled to support other services such as simulators and fabrication tools as they become available through the *trader* service.

The browser demonstrates a client application interacting with the SPR, *life cycle*, *library*, and *trader* services. The *trader* is the main interface from the browser to the distributed services. The browser finds the *library* and other services via the *trader*.

Chapter 3

Semiconductor Process Representation

3.1 Overview

To share and distribute information efficiently, a standard data representation is necessary. The semiconductor process representation (SPR) is intended to provide this standard for communicating information about fabrication processes. Thus, SPR was chosen as the basis for the process information programmatic interfaces within the distributed software architecture. The interfaces are defined in CORBA IDL and are therefore language independent; repositories and clients can be implemented in multiple languages across platforms. The standard interfaces allow process designers and manufacturers to integrate services and applications seamlessly across global networks.

3.2 Description

The SPR model organizes information into multiple views. The model has four base views – *effects*, *environment*, *equipment*, and *process* (see Figure 15). In addition, developers are allowed to use generic or custom views for data unification and extensibility. Views encapsulate specific domain and discipline information about a process. The view abstraction is a mechanism for capturing the diverse information to support both TCAD and CIM applications.

Generally, a process is broken down into smaller stages. The *process* view provides this abstraction; it divides processes into sub-processes. The other base views depict details of the process. The *equipment* view contains timing and setting information for

machines. The *effects* view describes what changes are performed on a wafer. The thermal and chemical environment of the wafer during processing is detailed in the *environment* view. Possible custom views include legacy process representations and simulator specific instructions. The interfaces and components associated with these views are discussed in the following section.

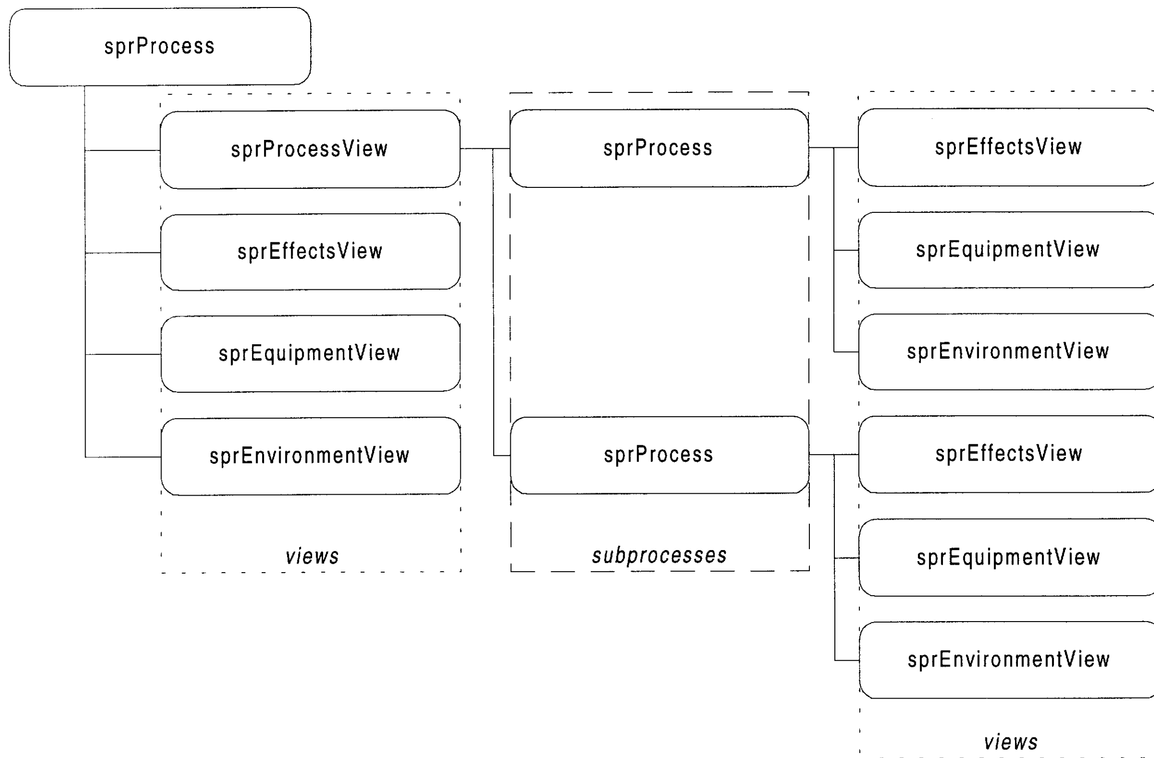

Figure 15: Process step divided into multiple views.

3.3 Interfaces

The SPR information model is extensible, laying the foundation for describing any semiconductor process. For practical application, SPR cannot possibly account for every process characteristic today and in the future. Thus, the model is extensible to account for growth and change. Extensibility is provided by a foundation of reusable interfaces for managing generic properties. These interfaces are used to construct view specific interfaces to access and organize semiconductor processes.

This section initially details the basic data structures. Then, the generic, reusable interfaces are described. Finally, the base SPR interfaces are illustrated.

3.3.1 Base Data Structures

There are two fundamental approaches for specifying data structures for a distributed environment. Data structures can be defined with the interfaces or strings may be used to pass data types. For the first case, all the information about the structure is tightly coupled with the interfaces. This guides implementations to adhere to the defined structures. In the latter case, strings must be parsed and unparsed according to an agreed upon specification in order for multiple implementations to operate equivalently. For the SPR implementation, the base structures are coupled with the interfaces to provide explicit definitions.

In addition to standard data types inherent with CORBA IDL, structures are defined for time duration, statistical floating point values, generic values, and units (see Table 1). A structure defines time duration as two integer values for hours and minutes and a floating-point value for seconds. Using a structure to represent time duration is not as extensible as an interface, but is beneficial in that the amount of information transmitted over the network is less. Additionally, the structure provides a restriction on a time duration's units to ensure that they are used equivalently by all applications. A correlating specification is that any time duration with minutes greater than sixty defines an unspecified value. Interfaces that use a time duration can provide specific methods for converting units of time to the duration structure. The statistical floating-point structure associates a standard deviation with a value. This is important for specifying process parameters that are digested experimental data or design specifications with tolerances. For specifying generic values, the **sprValue** union is defined to be either a primitive data type, a statistical floating-point value, or an **sprParameter** (defined on page 28). The **sprValue** type could have been implemented as a CORBA **any** type. **Any** enables the passing of all IDL defined interfaces and basic types. The **sprValue** was used instead of **any** because it provides additional information to implementations. Lastly, an enumeration is defined for possible units. This currently contains only a few basic units, but can be further expanded when the system has progressed beyond the current prototype.

``
struct sprDuration {
  long hours;
  long minutes;
  float seconds;
};
``

```
struct sprDuration {
  long hours;
  long minutes;
  float seconds;
};
```

```
struct sprStatFloat {
  float mean;
  float sdev;
};
```

```
enum sprValueType { BOOLEAN, SHORT, LONG, FLOAT, DOUBLE, STRING,
              PARAMETER, STATFLOAT };
union sprValue switch(sprValueType) {
   case BOOLEAN:    boolean      Boolean;
   case SHORT:      short        Short;
   case LONG:       long         Long;
   case FLOAT:      float        Float;
   case DOUBLE:     double       Double;
   case STRING:     string       String;
   case PARAMETER:  sprParameter Parameter;
   case STATFLOAT:  sprStatFloat StatFloat;
};
```

```
enum sprUnits { NONE, m, mm, nam, pm, sec, ns, ps, cSt, A};
```

Table 1: Base SPR data structures.

3.3.2 Base Objects

The foundation of the SPR information model defines a set of basic objects which are then used to provide extensibility for objects within the framework. Properties are used as an extensibility mechanism. A property is a name, value pair. The **sprProperty** interface is defined with interfaces for setting and retrieving a name and a value (see Table 2). Properties are useful for modeling characteristics that are not otherwise accounted for specifically by an object. Documentation and instructions are possible properties.

Properties alone do not provide extensibility. The **sprExtensibleObject** furnishes it by providing interfaces for managing a sequence of properties (see Table 2). This enables an object based on the **sprExtensibleObject** interface to contain a set of properties. The property sequence is used to model additional attributes.

Naming an object is a common convention. A name is useful for distinguishing between specific instantiations of objects. The **sprNamedObject** is an abstract interface used for attaching a name reference (see Table 2). This interface combines with **sprExtensibleObject** to form the **sprExtensibleNamedObject** interface.

The last two base interfaces define generic parameters for detailing primitive process information – **sprParameter** and **sprTimedParameter** (see Table 2). A parameter is a value associated with a unit of measure. A timed parameter is a start value and an end value with corresponding units. Using these basic parameters, constant and time-varying state information is described.

```
interface sprProperty : LifeCycle::LifeCycleObject {
  void   setName(in string inPropertyName);
  string getName();
  void   setValue(in sprValue inValue);
  sprValue  getValue();
};
```

```
interface sprExtensibleObject : LifeCycle::LifeCycleObject {
  void           setProperties(in sprPropertySeq inProperties);
  void           addProperty(in sprProperty inProperty);
  sprPropertySeq allProperties();
};
```

```
interface sprNamedObject : LifeCycle::LifeCycleObject {
  void   setName(in string inObjectName);
  string getName();
};
```

```
interface sprExtensibleNamedObject:sprNamedObject,sprExtensibleObject{};
```

```
interface sprParameter : LifeCycle::LifeCycleObject {
  void      setValue(in sprValue inValue)
    raises(sprInvalidValue);
  sprValue  getValue();
  void      setUnits(in sprUnits inUnits)
    raises(sprInvalidUnits);
  sprUnits  getUnits();
};
```

```
interface sprTimedParameter : sprParameter {
  void     setEndValue(in sprValue inEndValue);
  sprValue getEndValue();
};
```

Table 2: Foundation SPR interface objects

3.3.3 Processes & Views

The **sprProcess** interface is designed for encapsulating information about a semiconductor process (see Table 3). It is a named container derived from the **sprExtensibleNamedObject** interface. In addition to its inherited name and properties, methods are defined for managing the multiple views of a process object. Views are used to associate multiple descriptions with a process design.

The generic view interface, **sprView**, contains methods for defining and retrieving the view type (see Table 3). The four base views – *process*, *effects*, *environment*, and *equipment* inherit from this base interface. Additional views can be defined by extending this interface. Custom views enable the encapsulation of application specific data. An example is defining a custom simulation view. Defining additional views should be done with caution. The intent of the model is to specify common process information within the base views. A simulation view should only contain the instructions or transformations necessary for the simulator to use the base SPR views. It is not efficient to have multiple views containing the same process information.

```
interface sprView : sprExtensibleNamedObject {
  void   setViewType(in string inViewType);
  string getViewType();
};
```

```
interface sprProcess : sprExtensibleNamedObject {
  void       setViews(in sprViewSeq inViews)
    raises(sprNotUniqueViewType);
  void       addView(in sprView inView)
    raises(sprNotUniqueViewType);
  sprViewSeq allViews();
};
```

Table 3: Generic view and process interfaces

3.3.4 Process View

The **sprProcessView** provides structure for subdividing processes (see Table 4). It creates a structure to manage process information enabling the reuse of process objects. For instance, a sub-process for depositing a material only needs to be defined once. Then, multiple process views may reference this process.

```
interface sprProcessView : sprView {
  void          setSubProcesses(in sprProcessSeq inSubProcesses);
  void          appendSubProcess(in sprProcess inSubProcess);
  sprProcessSeq allSubProcesses();
};
```

Table 4: Process view interface

3.3.5 Effects

Effects represent changes that occur to a wafer. A sequence of effects within the **sprEffectsView** interface details the cumulative change in wafer state during a process

step (see Table 5). Applicable information within the *effects* view is often used by wafer state simulators.

```
interface sprEffectsView : sprView {
  void         setEffects(in sprEffectSeq inEffects);
  void         appendEffect(in sprEffect inEffect);
  sprEffectSeq allEffects();
};
```

Table 5: Effects view interface

A data representation for location is necessary for specifying an *effect*. The location defines where on a wafer the effect occurs. For the prototype implementation of SPR, a structure was defined for a simple effect location (see Table 6). The simple effect location contains a string defining a region of material coupled with a boolean specifying whether the region is exposed to the effect. More complex effect locations can be appended as necessary.

```
struct sprSimpleEffectLocation {
  string  MaterialRegion;
  boolean IsExposed;
};
```

```
enum sprEffectLocationType {SIMPLE};
union sprEffectLocation switch(sprEffectLocationType) {
   case SIMPLE: sprSimpleEffectLocation Simple;
};
```

Table 6: Simple effect location data structure

For *effects*, distance and material are defined as specific parameters (see Table 7). They are used to filter and narrow the broad scope of parameters. A distance parameter's value type is limited to being only a double or a string indicating "all" by the implementation. **sprMaterial** is implemented to only accept values of type string.

```
interface sprDistance : sprParameter { };
```

```
interface sprMaterial : sprParameter { };
```

Table 7: Effect specific parameter interfaces

Effects take a wafer state and transform it to new wafer state. Specific *effects* interfaces are defined beyond the base sprEffect in this prototype system (see Table 8). The base *effect* interface contains a location. The *change material effect* specifies a change from an old to a new material. The *deposit effect* describes the coating of a

vertical layer of material (see Figure 16). The *conformal* and *planar deposit effects* are subtypes of the *deposit effect.* The *conformal deposit effect* deposits a material that assumes the shape of the base layer (see Figure 17). The *planar deposit effect* deposits a level layer of material (see Figure 18).

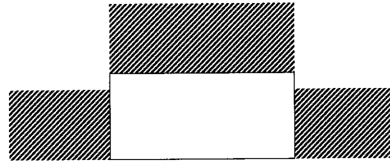

Figure 16: Deposit effect

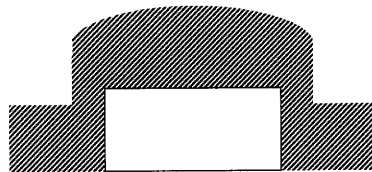

Figure 17: Conformal deposit effect

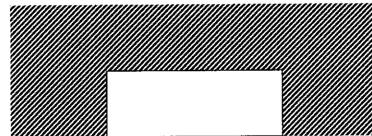

Figure 18: Planar deposit effect

The vertical *etch effect* has an interface for a thickness and a material. The thickness is a number or 'all' indicating that all the material should be etched. For a *strip etch effect*, the implementation ensures that the thickness is "all." Thus, the entire material layer is removed. For an *isotropic etch effect*, an equal thickness of material is removed across the entire wafer. The base interface for the *growth effect* contains the type of the material grown, the resulting thickness, and the depth consumed from the underlying layer.

```
interface sprEffect : sprExtensibleObject {
  void setEffectLocation(in sprEffectLocation inEffectLocation);
  sprEffectLocation getEffectLocation();
};
```

```
interface sprChangeMaterialEffect : sprEffect {
  void        setOldMaterial(in sprMaterial inOldMaterial);
  sprMaterial getOldMaterial();

  void        setNewMaterial(in sprMaterial inNewMaterial);
  sprMaterial getNewMaterial();
};
```

```
interface sprDepositEffect : sprEffect {
  void        setMaterial(in sprMaterial inMaterial);
  sprMaterial getMaterial();
  void        setThickness(in sprDistance inThickness);
  sprDistance getThickness();
};
```

```
interface sprConformalDepositEffect : sprDepositEffect { };
```

```
interface sprPlanarDepositEffect : sprDepositEffect { };
```

```
interface sprEtchEffect : sprEffect {
  void        setMaterial(in sprMaterial inMaterial);
  sprMaterial getMaterial();
  void        setThickness(in sprDistance inThickness);
  sprDistance getThickness();
};
```

```
interface sprStripEtchEffect : sprEtchEffect { };
```

```
interface sprIsotrophicEtchEffect : sprEtchEffect { };
```

```
interface sprGrowthEffect : sprEffect {
  void        setMaterial(in sprMaterial inMaterial);
  sprMaterial getMaterial();
  void        setThickness(in sprDistance inThickness);
  sprDistance getThickness();
  void        setDepth(in sprDistance inDepth);
  sprDistance getDepth();
};
```

Table 8: Effects interfaces

More effects such as *add field* and *diffusion* can be added to the model. The effects were expanded in detail to support initial applications and testing of the system with process design tools. The method used to extend the SPR IDL with additional effects can also be used to append the *environment* and *equipment* views.

3.3.6 Equipment View

The *equipment* view defines base interfaces for describing processing equipment. Additional parameters and equipment interfaces or properties can be used to provided more details about the machines and equipment settings. The *equipment view* interface contains two methods for describing the equipment setup and operation (see Table 9).

```
interface sprEquipmentView : sprView {
  void                   setMachines(in sprMachineSeq inMachines);
  void                   appendMachine(in sprMachine inMachine);
  sprMachineSeq          allMachines();
  void                   setSettings(in sprEquipmentStateSeq inSettings);
  void                   appendSetting(in sprEquipmentState inSetting);
  sprEquipmentStateSeq allStates();
};
```

Table 9: Equipment view interface

Equipment specific interfaces include *equipment state* and *machine* (see Table 10). The *equipment state* is a set of properties associated with a duration. The machine is a *named* entity with a set of properties. In a thorough SPR implementation, these two interfaces are the base for a collection of interfaces for fabrication equipment.

```
interface sprEquipmentState : sprExtensibleObject {
  void        setTimeDuration(in sprDuration inTimeDuration);
  sprDuration getTimeDuration();
};
```

```
interface sprMachine : sprExtensibleObject {
    void   setMachineName(in string inName);
    string getMachineName();
  };
```

Table 10: Machine and equipment interfaces

3.3.7 Environment View

The base interfaces are defined for the *environment view*. The physical state surrounding the wafer is described using a set of *environments* (see Table 11). Physical-based process simulators customarily utilize the *environment view*. The view may be created by collecting fabrication statistics. For example, data such as electron concentration and energy spectra collected from a plasma etcher may be used to create an *environment view*.

```
interface sprEnvironmentView : sprView {
  void              setEnvironment(in sprEnvironmentSeq inEnvironment);
  void              appendEnvironment(in sprEnvironment inEnvironment);
  sprEnvironmentSeq allEnvironment();
};
```

Table 11: Environment view interface

The *environment parameter* interface is a specific instance of a timed parameter. It is used to encapsulate models reflecting the state surrounding the wafer. Sets of *environment parameters* form the *general environment* interface (see Table 12).

```
interface sprEnvironmentParameter : sprTimedParameter { };
```

```
interface sprEnvironment : sprExtensibleObject{
  void        setTimeDuration(in sprDuration inTimeDuration);
  sprDuration getTimeDuration();
};
```

```
interface sprGeneralEnvironment : sprEnvironment {
  void setEnvironmentParameters
       (in sprEnvironmentParameterSeq inParameters);
  void appendEnvironmentParameter
       (in sprEnvironmentParameter inParameter);
  sprEnvironmentParameterSeq allEnvironmentParameters();
};
```

Table 12: Basic environment interfaces

3.4 Server Implementation

CORBA IDL translates into corresponding skeletons and stubs for an assortment of programming languages. Skeletons are code generated for the server-side implementation of objects. The stubs are code generated to enable client access to the objects. Client and server interaction such as managing network connections and transferring data are performed by skeletons and the stubs (see Figure 19). The SPR server implementation of skeletons was done in Java [13] to provide a platform independent prototype.

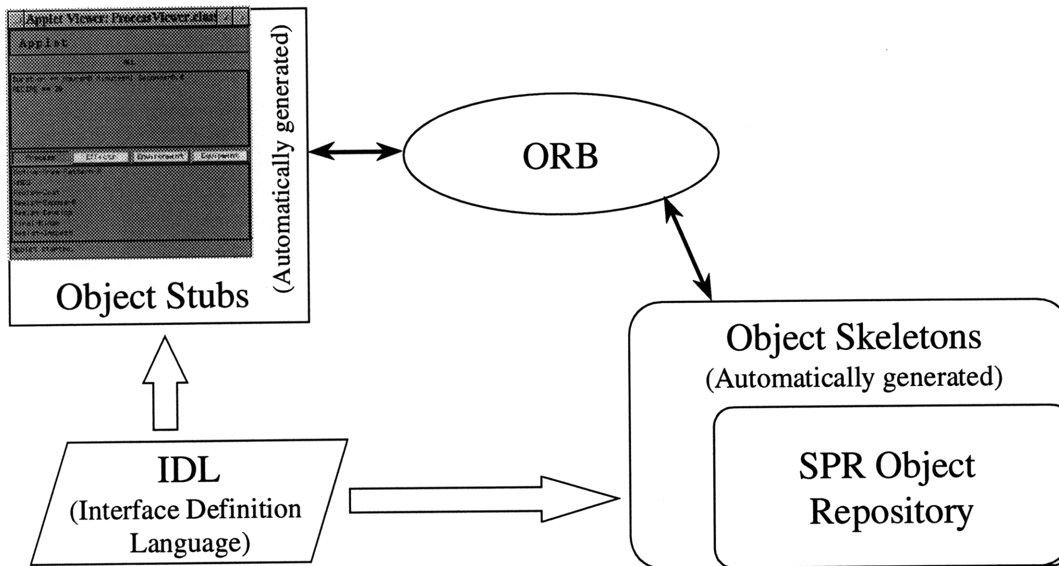

Figure 19: CORBA skeleton and stub interaction

There are two techniques for associating an implementation object with a skeleton – inheritance and delegation [15]. With inheritance, the implementation object extends the skeleton directly. Delegation associates an implementation object with the skeleton. Java does not support multiple inheritance. Ergo, the inheritance method of an implementation object must inherit directly from the skeleton. Consequently, the delegation method was chosen for implementing the SPR interfaces due to the extensive use of inheritance among the objects. The class hierarchy of the Java implementation closely follows the inheritance structure of the interfaces (see Figure 20).

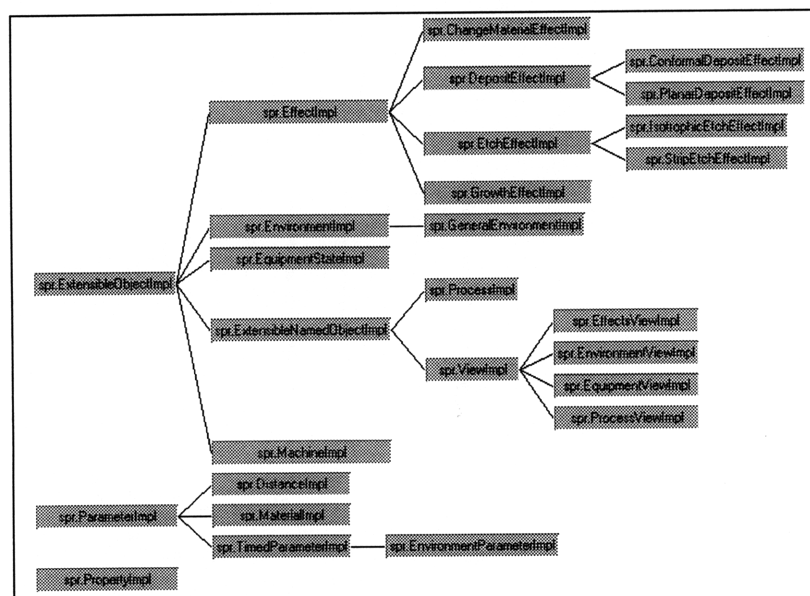

Figure 20: SPR Java implementation hierarchy

3.5 Persistent Storage

The initial SPR implementations were used for testing and developing the interface definitions. When the base SPR interfaces became finalized, a persistent implementation was created. PSE for Java [16] was employed to create an object-oriented database for the core set of SPR objects with attributes. As shown in Figure 21, the core classes are a subset of the interface hierarchy. An object model was defined (see Appendix D) and implemented in Java for storing process information persistently to disk. Within the defined database classes, methods convert distributed object references to strings for persistent storage.

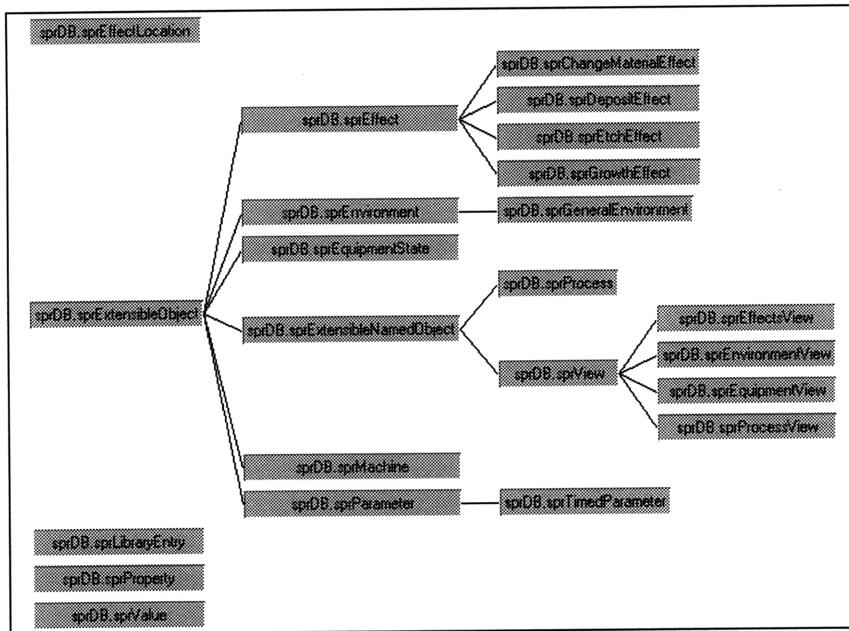

Figure 21: Persistent SPR classes

The development of the SPR and *life cycle* services were closely coupled. Chapter 4 details more specifically how the persistent classes interact with the CORBA implementations.

3.6 Example Client Interactions

The client example presented in Table 13 obtains a process reference, prints its name and the view types it contains. All the object method invocations appear as normal Java calls. The distributed mechanisms operating within this program are hidden to the programmer in the automatically generated stubs. Obtaining an initial reference to a process object has multiple approaches and is discussed with latter services such as the *library* and *trader* (see Chapter 6).

1	`sprProcess etchOxide = << obtain initial reference >>;`
2	`System.out.println(etchOxide.getName() + " process has views of type:");`
3	`sprView[] views = etchOxide.allViews();`
4	`for(int i = 0; i < views.length; i++) {`
5	`   System.out.println("   - " + views[i].getViewType());`
6	`}`

Table 13: Simple client navigation of a semiconductor process

Chapter 4

Life Cycle Service

4.1 Overview

The OMG *life cycle* service defines interfaces for creating, deleting, copying, and moving distributed objects [14]. In a distributed environment, operations provided by the *life cycle* service allow clients to manage objects in remote locations.

The *life cycle* service addresses object creation and location. Object creation is a mechanism for creating objects at a remote location. An interface is defined for producing remote objects. Object location entails how and where clients indicate remote object destinations while preserving the CORBA principal of location transparency. An interface is defined for locating remote objects.

4.2 Description

4.2.1 Factories

A factory is an object that creates and initializes new instances of objects (see Figure 22). With access to a factory, a client can create multiple objects instances. The factory determines the location of the new object and allocates the necessary resources for its implementation. Resources typically include memory or persistent storage mechanisms. A factory may also interact with a *trader* or *naming* service to register object instances.

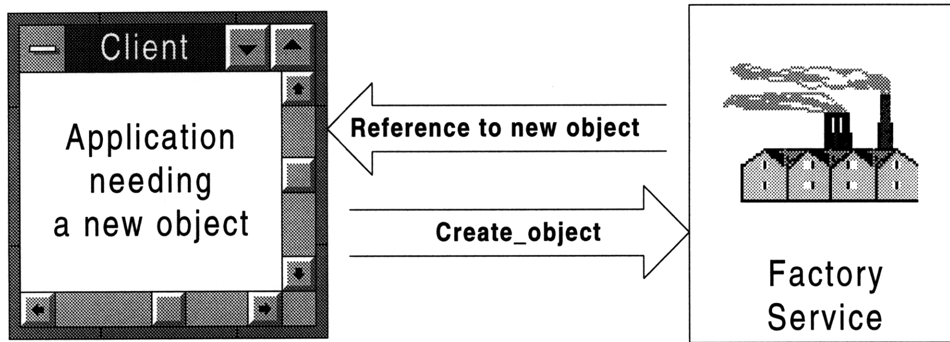

Figure 22: Client and factory interaction

4.2.2 Locating Factories

The notion of location is a major concern for any distributed system. Services such as *naming* and *trader* are two services that address this issue. Location is necessary to create, move, and copy objects. When an object is moved, a mechanism for specifying "where" the object is moving is needed. For the OMG Life Cycle service, the *factory finder* interface is used to determine location. The factory finder may be implemented as a standalone object managing available factories; or, it may be coupled with the organizational power of a *naming* or *trader* service (see Figure 23).

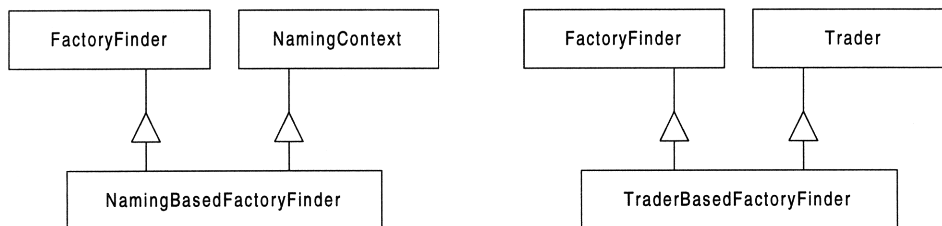

Figure 23: Factory finder interface used in conjunction with other location services.

4.3 Interfaces

The Life Cycle service consists of three interfaces – *generic factory*, *life cycle object*, and *factory finder*. These interfaces and some base data structures are discussed in the following subsections.

4.3.1 Base Structures

Three data structures are used with the life cycle service (see Table 14). A **Key** structure is defined as a name and a hostname. Keys are used to name and locate factories. A *factory entry* structure associates a key with a generic factory. Lastly, a *name value pair* structure is defined as a name with an associated value. The CORBA type **any** is used to

pass any primitive data type or defined interface. Sequences of the **Name Value Pair** structure are used to pass **Criteria** to *factories* and *factory finders*.

```
struct Key {
  string name;
  string hostname;
};
```

```
struct factory_entry {
  Key key;
  GenericFactory factory_ref;
};
```

```
typedef struct NVP {
  string name;
  any value;
} NameValuePair;
typedef sequence <NameValuePair> Criteria;
```

Table 14: Basic life cycle data structures

4.3.2 Generic Factory

The job of a generic factory is to match creation criteria specified by clients of the **GenericFactory** interface (see Table 15) with specific factory implementations [14]. The OMG defines *create_object* and *supports* methods for the **GenericFactory** interface. The methods *list_objects*, *remove_object*, and *shutdown* are custom extensions.

Create_object is a generic creation method that passes requests to specific factory implementations. It takes a key to identify the factory implementation and criteria to define how the object is initialized. The *supports* operation returns true if the generic factory can create an object given a factory key. The *list_objects* operation takes a key to identify a specific factory and returns a sequence of all the objects created by that specific factory. The *remove_object* method takes a factory key and object reference and removes the specified object from the factory's list of created objects. It returns true if the removal is successful. This operation is intended for use with persistent storage mechanisms. The *shutdown* method is for remotely shutting down a generic factory service.

```
interface GenericFactory {
  boolean supports(in Key k);
  Object create_object(in Key k, in Criteria the_criteria)
    raises (NoFactory, InvalidCriteria, CannotMeetCriteria);
  ObjectSeq list_objects(in Key k)
    raises (NoFactory);
  boolean remove_object(in Key factory_key, in Object obj_ref)
    raises (NoFactory);
  void shutdown();
};
```

Table 15: Generic factory interface

4.3.3 Factory Finder

Clients use a *factory finder* interface to specify the destination of a move or a copy operation. The **FactoryFinder** interface (see Table 16) has one OMG defined method – *find_factories*. The methods *add_factory* and *shutdownAll* are custom extensions.

The *find_factories* operation takes a key to identify the target generic factory. A sequence of factories with matching keys is returned by the operation. The *add_factory* method passes a key and a generic factory reference to be added to the factory finder's scope. The operation is used by generic factory implementations to notify the factory finder of its location. The *shutdownAll* method shuts down the factory finder service and all the factory objects within its scope.

```
interface FactoryFinder {
  Factories find_factories(in Key factory_key)
    raises(NoFactory);
  void add_factory(in Key factory_key, in GenericFactory factory_ref)
    raises (DuplicateKey);
  void shutdownAll();
};
```

Table 16: Factory finder interface

4.3.4 Life Cycle Object

Objects participate in the life cycle service by extending the **LifeCycleObject** interface. The OMG defines the *copy*, *move*, and *remove* operations for the **LifeCycleObject** interface (see Table 17). The methods *getMarker*, *setFactoryName*, and *getFactoryName* are custom extensions.

The *copy* operation makes a copy of the object. The factory finder and criteria determine the new object's location. A reference to the new object is returned by the

method. The *move* operation moves an object to the location specified by the factory finder and criteria. It returns its new reference. The *remove* method instructs the object to destroy itself. The object reference is no longer valid after the *remove* operation successfully completes. The *get_marker* operation returns a string containing the object's interface name. Factories notify an object that the factory created it by using the *setFactoryName* method. The *getFactoryName* operation is used to determine which factory created the object instance.

```
interface LifeCycleObject {
  LifeCycleObject copy(in FactoryFinder there, in Criteria the_criteria)
    raises(NoFactory, NotCopyable, InvalidCriteria, CannotMeetCriteria);
  LifeCycleObject move(in FactoryFinder there, in Criteria the_criteria)
    raises(NoFactory, NotMovable, InvalidCriteria, CannotMeetCriteria);
  void remove()
    raises(NotRemovable);
  string getMarker();
  void   setFactoryName(in string name);
  string getFactoryName();
};
```

Table 17: Life cycle object interface

4.4 Implementations

The components of the *life cycle* service were implemented in Java. The SPR implementations participate by implementing a defined functionality. In the case of the *life cycle* service, this is the *life cycle object* interface. The *generic factory* also participates by passing the task of object creation to object-specific factories. The *factory finder* is the only generic aspect of a *life cycle* service implementation.

4.4.1 Generic Factory

The *generic factory* object implements operations for *support*, *create_object*, *list_objects*, *remove_object*, and *shutdown*. The *generic factory* does not perform the actual creation. Specific implementation factories for individual objects that the generic factory supports perform the creation. For the distributed software architecture for semiconductor process design, the *generic factory* supports creating all the objects defined by the SPR service (see Chapter 3).

The specific factories for each SPR object extend from an abstract factory class that implements base methods such as listing the objects created and removing objects from this list. The abstract factory persistently stores the list of object references it creates.

This correlates to a second list of database objects. These two lists are stored to disk. They provide persistent storage for the repository objects. The specific factories implement create the object instance and link it to its respective database object.

The *generic factory* creates instances of all the specific factories that it supports. These factories and their respective keys are stored in a list that is searched when a client invokes the *support* method. The operation returns true if the *generic factory* finds the factory key in the list.

The *create_object* is passed a key identifying a specific factory for creating the object. The specific factory then creates the object based on the given criteria. The only criterion supported in the current implementation is to name the object. The object name is encoded within the interoperable object reference (IOR). If an object name criterion is not given, a unique random name is assigned to the object. Other criteria can be defined to initialize the object with values or to represent other constraints on the creation process. Figure 24 shows a client invoking the *generic factory* to create a process object. The *generic factory* passes the call to the process factory and returns a reference to the client.

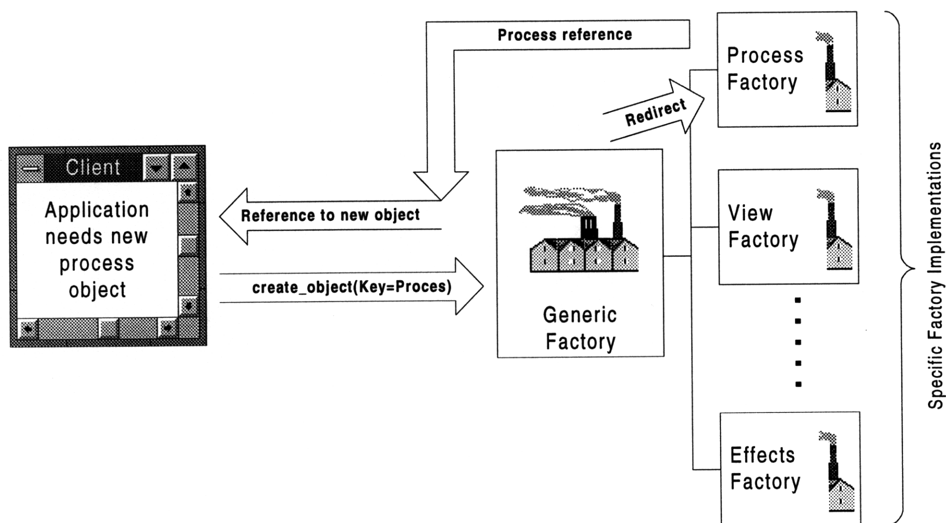

Figure 24: Client creating a process object

The *list_objects* operation returns a list of objects from the factory specified by the given key. The *remove_object* method deletes the persistent storage of an object given the factory key and the object reference. The *shutdown* operation commits all current data operations, closes the database, and turns off the factory service.

4.4.2 Factory Finder

The *factory finder* implementation keeps a list of all factories within its scope. The scope is defined as any factory that notifies the *factory finder* via the *add_factory* method. The *find_factories* operation cycles through the list of known factories and returns any that matches the given key. If a blank key is given (both name and host are empty strings), all the factories within the finder's scope are returned. The *shutdownAll* method cycles through all the factories within the finder's scope, calls their *shutdown* operation and then exits. Alternate implementations could utilize a *trader* or *naming* service to find factories.

4.4.3 Life Cycle Object

The *life cycle object* is the participatory aspect of the *life cycle* service. All SPR objects in the architecture inherit and implement the *life cycle object* interface. This enables the copying, moving, and removing of SPR objects.

The *life cycle object* implementation takes the approach of deep copying all objects except the *process view*. The *process view* is the only recursive object in that it references other *process* objects. Thus, only the references of the process view are copied. For removal, a process view deletes itself and its properties but not the processes that it references. The *remove* operation deletes an object via removing itself from memory and its persistent image. The *move* operation combines the *copy* and the *remove* methods. The two criteria implemented for the *copy* and *move* operations are for naming the new object and specifying a generic factory key within the *factory finder* scope.

A more expansive *life cycle object* implementation would add criteria for specifying different implementations and policies for copy, moving, and removing objects.

4.5 Examples

Two examples of a client interacting with the life cycle service are shown. The first set of code (see Table 18) binds to a factory finder, creates a process object using criteria, and then removes the object. The first line performs an initialization of the orb object. The need for obtaining an initial object reference is a bootstrapping problem for client applications. In this example, a universal resource locator (URL) is used to read a file containing a stringified IOR for a factory finder service (lines 2-9). The reference is then converted to a factory finder reference (lines 10-11). Keys are created to find the generic

factory (“MTL”) and specify the specific factory (“ProcessFactory”) to use (lines 12-13). Note that when the host field of the key structure is a blank string, the field is ignored. Criteria to specify an object name is then instantiated (lines 14-16). The factory finder binds to the “MTL” generic factory (line 17) and the factory creates a new process object (line 18-19). With a reference to the object, the name is set (line 20) and then the object’s remove operation is called (line 21).

```
            // initialize ORB
1           ORB orb = ORB.init(new String[0], new java.util.Properties());

            // bind to Factory finder via IOR
2           URL iorURL = new URL("http://gun.mit.edu/SPR/factoryfinder.ior");
3           URLConnection connection = iorURL.openConnection();
4           InputStream input = connection.getInputStream();
5           InputStreamReader reader = new InputStreamReader(input);
6           BufferedReader in = new BufferedReader(reader);
7           String buffer = in.readLine();
8           reader.close();
9           input.close();
10          Object obj = orb.string_to_object(buffer);
11          FactoryFinder ff = FactoryFinderHelper.narrow(obj);


            // create keys for the factories
12          Key specificKey = new Key("ProcessFactory", "");
13          Key genericKey = new Key("MTL", "");

            // create name criteria
14          NVP[] criteria = new NVP[1];
15          criteria[0] =new NVP("NewObjectName", orb.create_any ());
16          criteria[0].value.insert_string("Etch Oxide Object");

            // find factories
17          GenericFactory[] factories = ff.find_factories(genericKey);

            // create object
18          obj = factories[0].create_object(specificKey, criteria);
19          sprProcess etch = sprProcessHelper.narrow(obj);

            // set process name
20          etch.setName("Etch Oxide");

            // remove process object
21          etch.remove();
```

Table 18: Life cycle client example

The second example uses a helper client class for interacting with the *life cycle* service. The helper class (**spr.Base.Client**) was defined to reduce the amount of code necessary to interact with the *life cycle* service. As shown in the previous example, the

simple task of creating a process object takes multiple lines of code. The helper class contains operations to automate connecting to factory finders or factories and creating SPR defined objects. Table 19 shows the creation of a process object along with some properties and three views. The first line initializes the helper class with "MTL" as the default factory key. A new process is created and named "Etch-Oxide" (lines 2-3). Properties are created for the "version" and "date" and added to the process object (lines 4-9). *Process*, *equipment*, and *environment* views are created and added to the "Etch-Oxide" object (lines 10-17). Finally, the example from the previous chapter (lines 18-22) is used to output the view types of the process object (see Table 20).

```
1       spr.Base.Client helper = new spr.Base.Client(new Key("MTL",""));

2       sprProcess etchOxide = helper.newProcess();
3       etchOxide.setName("Etch Oxide");

4       sprProperty[] properties = new sprProperty[2];
5       sprProperty version = helper.createProperty("Version", 7);
6       properties[0] = version;
7       sprProperty date = helper.createProperty("Date", "05/26/97");
8       properties[1] = date;
9       etchOxide.setProperties(properties);

10      sprProcessView processView = helper.newProcessView();
11      sprEquipmentView equipmentView = helper.newEquipmentView();
12      sprEffectsView effectsView = helper.newEffectsView();
13      sprView[] mainViews = new sprView[3];
14      mainViews[0] = processView;
15      mainViews[1] = equipmentView;
16      mainViews[2] = effectsView;
17      etchOxide.setViews(mainViews);

18      System.out.println(etchOxide.getName() + " process has views of type:");
19      sprView[] views = etchOxide.allViews();
20      for(int i=0; i < views.length; i++) {
21              System.out.println("    - " + views[i].getViewType());
22      }
```

Table 19: Life cycle client with helper class

```
Etch Oxide process has views of type:
    - sprProcess
    - sprEquipment
    - sprEffects
```

Table 20: Output of client interaction

Chapter 5

Organizing and Searching the Repository

5.1 Overview

The SPR and *life cycle* services provide the core functionality for a distributed semiconductor process repository. A representation for communicating and managing semiconductor process information across a global computer network is the foundation of the repository. These services do not define interfaces for organizing and searching the process information. The organization was kept independent of the repository to allow multiple interfaces and implementations. A *library* service was defined to organize remote processes into catalogs and *query* extensions were added to the *life cycle* service for searching repositories.

5.2 Library Service

5.2.1 Description

The organization of repositories was defined separate from the SPR service to allow multiple abstractions and implementations. A *library* service organizes processes into catalogs (see Figure 25). The processes within a catalog can be stored in repositories distributed across multiple systems. Thus, a collection of processes from distributed repositories appear as a single entity. The interfaces of the *library* service provide operations for managing catalogs.

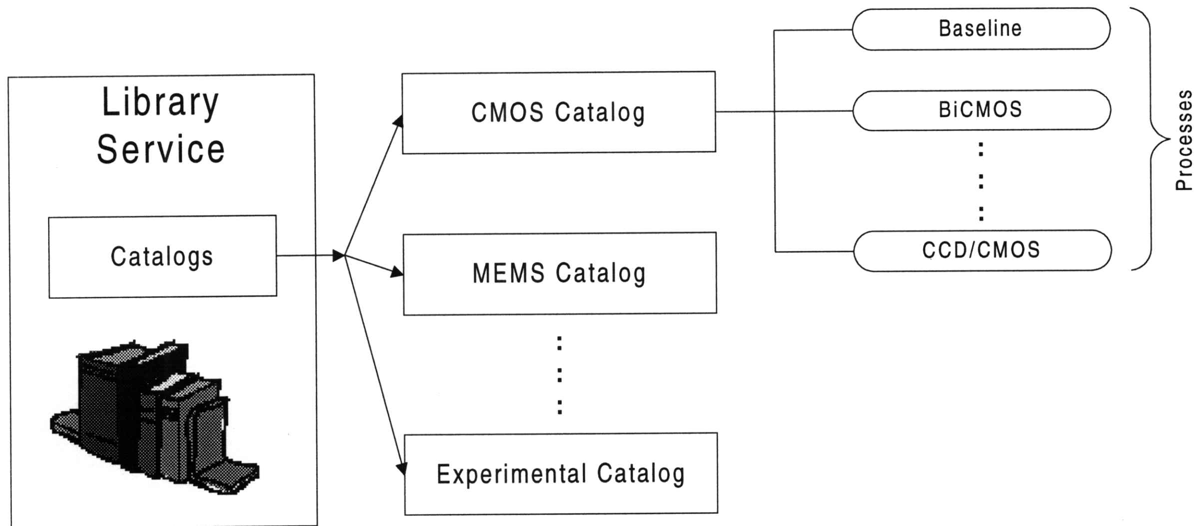

Figure 25: Library service

5.2.2 *Interfaces*

The **Catalog** is the only data structure defined for the **Library** interface (see Table 21). The structure consists of a name and a sequence of processes. It is used to group process references with an associated catalog name. The library is constructed as a collection of process catalogs.

```
struct Catalog {
    string name;
    sprCORBA::sprProcessSeq processes;
};
```

Table 21: Catalog data structure

The SPR library stores collections of semiconductor processes in catalogs. An example is a library for the Microsystems Technology Lab (MTL) at MIT. Some of the catalogs included would be for CMOS and MEMS processes. The **Library** interface (see Table 22) defines the methods *setName*, *getName*, *addCatalog*, *removeCatalog*, *allCatalogs*, *setProcesses*, *getProcesses*, *addProcess*, *removeProcess*, and *shutdown*.

The first set of operations manage the library. The *setName* and *getName* operations are for accessing and changing the library name. The *addCatalog* method appends the inputted catalog to the library's current collection. It raises a *DuplicateCatalogException* when a catalog with the same name already exists within the library. The *removeCatalog* operation deletes a catalog given a name. It raises a *CatalogDoesNotExist* exception if

there is no catalog with the given name. The *allCatalog* operation returns a sequence of strings containing the names of all the catalogs.

The second set of methods manage the processes within a catalog. The *setProcesses* operation is passed a catalog name and a sequence of processes to fill the catalog. The *getProcesses* method takes a string with a catalog name and returns the catalog's process sequence. The *addProcess* and *removeProcess* append and delete individual processes from a given catalog. All the catalog management operations raise a *CatalogDoesNotExist* exception if there is no catalog corresponding to the specified name.

```
interface Library {
        void setName(in string name);
        string getName();

        void addCatalog(in Catalog inCatalog)raises(DuplicateCatalogName);
        void removeCatalog(in string name) raises(CatalogDoesNotExist);
        StringSeq allCatalogs();

        void setProcesses(in string inCatalogName,
                          in sprCORBA::sprProcessSeq inProcesses)
          raises(CatalogDoesNotExist);
        sprCORBA::sprProcessSeq getProcesses(in string inCatalogName)
          raises(CatalogDoesNotExist);
        void addProcess(in string inCatalogName,
                        in sprCORBA::sprProcess inProcess)
          raises(CatalogDoesNotExist);
        void removeProcess(in string inCatalogName,
                           in string inProcessName)
          raises(CatalogDoesNotExist);

        void shutdown();
    };
```

Table 22: Library interface

5.2.3 *Implementation*

The Library skeleton is implemented in Java using the delegation method. The service implementation is a straightforward mapping to the IDL-defined interfaces. Following the SPR implementation, PSE for Java [14] is used to store the library name and catalogs of process references persistently. All process object references are converted to and from strings when writing and reading from disk.

5.2.4 *Example*

Table 23 shows an example client interaction with the *library* service. The example starts with the standard initialization of the orb and client helper objects (lines 1-2). It

then binds to a library (line 3). A new catalog is created and filled (lines 4-8). The catalog is added to the library (line 9). Lastly, a set of loops iterates through the library's collection of catalogs and its respective processes (lines 10-17). The output generated from this example is shown in Table 24.

```
1           ORB orb = ORB.init();
2           spr.Base.Client helper = new spr.Base.Client();

            // Bind to Library
3           Library libRef = helper.bindLibrary();

            // Create CMOS Catalog
4           sprProcess[] processes = new sprProcess[3];
5           processes[0] = baseline;
6           processes[1] = bicmos;
7           processes[2] = ccdcmos;
8           Catalog CMOS = new Catalog("CMOS", processes);

            // Add catalog to the library
9           libRef.addCatalog(CMOS);

10          System.out.println(libRef.getName() +
                                 "'s catalogs and processes:");
11          String[] cats = libRef.allCatalogs();
12          sprProcess[] sProcesses;
13          for(int i=0; i < cats.length; i++) {
14            System.out.println(" -> " + cats[i]);
15            sProcesses = libRef.getProcesses(cats[i]);
16            for(int j=0; j < sProcesses.length; j++) {
17              System.out.println("   o " + sProcesses[j].getName());
              }
            }
```

Table 23: Library example

```
MTL's catalogs and processes:
  -> CMOS
     o Baseline
     o BiCMOS
     o CCD/CMOS
```

Table 24: Output of library example

5.3 Query Extensions

5.3.1 Description

For additional repository access, *query* extensions were defined and added to the *life cycle* service. The additional method allows a query to utilize the underlying repository database. The intention of the query operation is to lay the foundation for a full *query*

service implementation. The query operation allows a *query* service to access the speed and efficiency of the repository's underlying database.

5.3.2 Interface

A **constraint** structure was defined to pass queries to factories. The structure contains a name for identifying the type of query, a value, and a predicate. The constraint structure defines a generic constraint mechanism for queries. Another possible implementation is to pass strings in a query language. A specification for the query language would be associated with the defined interfaces.

```
enum Predicate { EQ, NEQ, GTE, LTE, GT, LT };
typedef struct constr {
  string name;
  any value;
  Predicate predicate;
} Constraint;
typedef sequence <Constraint> ConstraintSeq;
```

Table 25: Query data structures

The **QuerableGenericFactory** interface extends the life cycle's **GenericFactory** interface to include a *query_objects* operation. The operation takes a specific factory key and a sequences of constraints as arguments. It evaluates the constraints for the factory specified and returns a sequence of objects that satisfy the constraints.

```
interface QueryableGenericFactory :: GenericFactory {
  ObjectSeq query_objects(in Key k, in ConstraintSeq constraints)
    raises (NoFactory, InvalidConstraint);
};
```

Table 26: Query capable factory interface

5.3.3 Implementation

Following from the *life cycle* service, the *query* extensions were implemented in Java closely interacting with the underlying object-oriented database. The prototype implementation of the *query* extensions only supports searching process objects with constraints for *identification* and *name*. Other objects and criteria can be defined and added to the current the implementation.

The implementation object for the **QueryableGenericFactory** interface passes a query to the specific factory identified by a key. The specific factory directly searches its collection of objects created. The method allows a collection of remote object references

to be efficiently searched, avoiding the overhead associated with iterating through a sequence of remote object references.

5.3.4 Example

An example using the query extensions binds to three factories, queries each factory for a process with a name equal to "etch-oxide," and then prints the results of each query (see Table 27). The first two lines of the code initialize the orb and client helper objects. Keys are then created for binding to three generic factories (lines 3-5). References to the factories are located via the factory finder (lines 6-11). A key to the specific factory to search is instantiated (line 12). Constraints are defined for searching for a process with a name equal to "etch-oxide" (lines 13-19). Finally, the query is performed at each factory (lines 20-22) and the results are output (lines 23-25).

```
1         ORB orb = ORB.init();
2         spr.Base.Client helper = new spr.Base.Client();

          // Create Specific Factory Keys
3         Key mtlKey = new edu.mit.mtl.LifeCycle.Key("MTL","");
4         Key sprKey = new edu.mit.mtl.LifeCycle.Key("SPR Research","");
5         Key experKey = new edu.mit.mtl.LifeCycle.Key("MTL Experimental","");

          // Bind to factories
6         GenericFactory mtlFactory
7             = ((helper.bindFactoryFinder()).find_factories(mtlKey))[0];
8         GenericFactory sprFactory
9             = ((helper.bindFactoryFinder()).find_factories(sprKey))[0];
10        GenericFactory experFactory
11            = ((helper.bindFactoryFinder()).find_factories(experKey))[0];

12        Key processFact = new Key("ProcessFactory", "");

13        constr[] constraints = new constr[1];
14        Any search = (orb.create_any());
15        search.insert_string("etch-oxide");
16        constraints[0] = new edu.mit.mtl.LifeCycle.constr(
17                            "Name",
18                            search,
19                            Predicate.EQ );

20        Object[] obj1 = mtlFactory.query_objects(processFact, constraints);
21        Object[] obj2 = sprFactory.query_objects(processFact, constraints);
22        Object[] obj3 = experFactory.query_objects(processFact, constraints);

23        System.out.println("MTL : " + obj1.length);
24        System.out.println("SPR : " + obj2.length);
25        System.out.println("Exper : " + obj3.length);
```

Table 27: Query extensions example

Chapter 6

Locating Services

6.1 Overview

The OMG defines a *trader* service to facilitate importing and exporting services [14]. It advertises and matches service capabilities to the appropriate client applications. The distributed software architecture for semiconductor process design utilizes a generic *trader* service, JTrader [17], allowing client applications to obtain services such as the *library*, *repository*, and *factory finder*.

6.2 Description

Traders are repositories of object references coupled with an interface type and a descriptive set of properties [15]. A service exports a *service offer* by advertising its functionality to the *trader*. The *service offer* contains a reference to the object coupled with specific property values (exporting). Client applications import services from a *trader* by requesting a desired functionality based on the interface type and properties of a *service offer* (importing).

A *service type* is a template used to ensure that *service offers* are grouped together with similar functional descriptions. It consists of an interface type and a defined set of properties (see Table 28). A *service type* allows clients to efficiently search and match offers within a trader. The *service type* encourages offered services and client applications to use common properties for classifying services.

```
service exampleServiceType {
   interface IDLinterface;
   mandatory property string name;
   property sequence<string> moreProperties;
};
```

Table 28: Example service type

A program exports a *service offer* to a trader. Clients use stubs generated from IDL to access services. The same client stubs can be used to access any service that implements the server-side skeletons. Thus, a client application can choose any advertised service that supports the defined IDL. Applications make a choice using constraints and preferences based upon the service properties advertised. Dynamically assigning services at runtime allows service implementations to be upgraded and enhanced without making changes to client applications.

The OMG defines trader interfaces for searching and managing a database of *service offers*. In addition, a trader can be linked to other traders. It utilizes links to pass queries to a larger pool of services. A group of linked traders is referred to as a *federated trader*. A trader may vary in the implementation level of OMG defined interfaces. Policies allow traders to define which interfaces are supported. A trader that supports all the OMG defined interfaces except linking is a standalone trader. JTrader [17] is a standalone trader implementation.

A scenario where a trader service is useful for semiconductor process design is locating software tools such as simulators. Currently, a process designer invokes a simulator by knowing the program location. If a new simulator becomes available, email, news, or other communication media notify users. Users then become responsible for "reprogramming" themselves to use the new simulator. A more efficient method for upgrading a simulator is to have users supply client applications (e.g. a process editor) with simulator requirements that automate the service selection using a trader. When new simulators become available that match the designers predefined requirements, the user can use the new simulator without knowledge of specific details such as location. Similar scenarios can be described for the interactions between the software architecture's foundation components such as the *library*, *generic factory* (repository), and *factory finder*.

6.3 Service Types

As part of the base software architecture, service types are defined for the *library*, *generic factory*, and *factory finder*. The service types are used to advertise the three fundamental interfaces used by clients for accessing, managing, and organizing SPR objects. When simulators and fabrication facilities are added to the architecture, appropriate service types should be defined.

6.3.1 SPR Repository

The foundation of a repository is the *life cycle* service's *generic factory* object. It contains interfaces for creating and accessing SPR object instances. A service type is defined for a SPR repository to advertise services that support the *generic factory* interface (see Table 29). The service type contains basic properties for defining a name, host, and version. Boolean variables are used to classify whether a factory is capable of storing objects persistently and if the factory supports query extensions. Last, a sequence of strings is used to classify the types of process objects that a repository supports. The classification is used for keyword indexing of repositories.

```
service sprRepository {
   interface GenericFactory;
   mandatory property string name;
   property string host;
   property string version;
   property boolean persistent;
   property boolean queryable;
   property sequence<string> processTypes;
};
```

Table 29: Repository service definition

6.3.2 SPR Library

The *SPR library* object is used to organize processes across multiple repositories. A SPR library service type is defined to advertise services that support the *library* interface (see Table 30). Similar to the repository service type definition, it contains basic properties for a name, host, and version. A boolean variable is used to classify whether the library persistently stores information to disk. A sequence of strings is used to classify the types of repositories that the library contains.

```
service sprLibrary {
   interface Library;
   mandatory property string name;
   property string host;
   property string version;
   property boolean persistent;
   property sequence<string> processTypes;
};
```

Table 30: Library service definition

6.3.3 Factory Finder

The *life cycle* service's *factory finder* is used to link multiple factories within a given scope. A factory finder service type is defined to advertise instances of *factory finders* (see Table 31). Again, the service contains basic properties to define a name, host, and version. Boolean variables are used to classify whether the factory finder supports OMG defined interfaces and if there are interface extensions. A sequence of strings details any customizations or variations from the standard OMG interfaces.

```
service FactoryFinder {
   interface FactoryFinder;
   mandatory property string name;
   property string host;
   property string version;
   property boolean OMGcapable;
   property boolean Extensions;
   property sequence<string> supportedIDL;
};
```

Table 31: Factory finder service definition

6.4 Examples

Adding service type definitions is the first step required to use the *trader* service. The trader implementation, JTrader [17], utilizes a graphical interface for managing the service type definitions. The service types for *SPR repository*, *SPR library*, and *factory finder* were defined using the service type manager (see Figure 26).

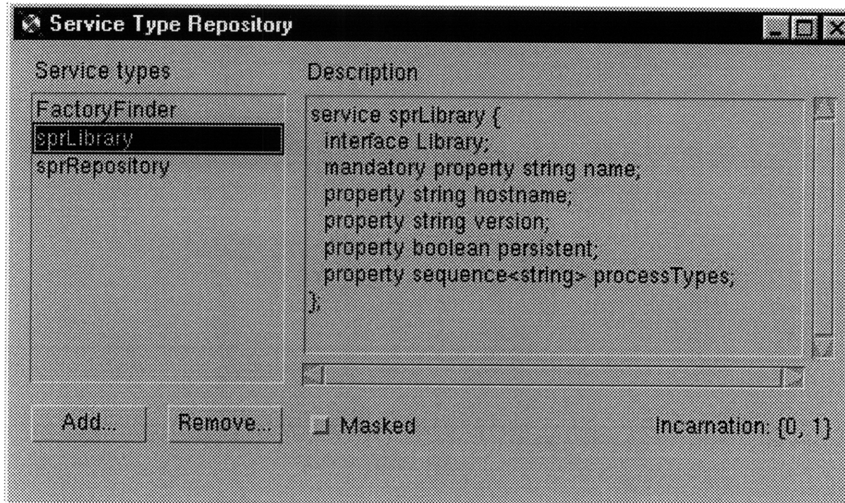

Figure 26: Adding service types

Once the service types have been defined in the *trader*, specific service implementations can be registered. Table 32 contains the Java code used to register a SPR library service offer. The first two lines use the helper function to bind to the *trader* service. Next, a reference to the registration module is retrieved (lines 3-4). Properties for a name and hostname are created (lines 5-16). Finally, the library implementation is registered using the *trader* service (lines 17-18).

```
        // Bind to Trader and retrieve interface to service repository
1       Lookup lookup = helper.bindTrader();
2       org.omg.CORBA.Object obj = lookup.type_repos();

3       Register reg = null;
4       reg = lookup.register_if();

5       int num = 0;
6       Property[] props = new Property[2];
7       props[num] = new Property();
8       props[num].name = "name";
9       props[num].value = orb.create_any();
10      props[num].value.insert_string("MTL");
11      num++;
12      props[num] = new Property();
13      props[num].name = "hostname";
14      props[num].value = orb.create_any();
15      props[num].value.insert_string("gunpowder.mit.edu");
16      num++;

        // bind to Library and offer the service
17      Library lib = helper.bindLibrary("MTL");
18      String id = reg.export(lib, "sprLibrary", props);
```

Table 32: Service offer for SPR Library

After the service offers are registered with the trader, queries can be performed on available services. Constraints and preferences are used for querying service properties. Constraints restrict the outcome of the search using boolean expressions of service properties. A constraint is required with *true* being the simplest. A preference is used to order the sequence services that match the given constraints. Preferences consist of a modifier and a boolean expression. Modifiers include minimum, maximum, random, with, and first. Preferences are an optional input. The default ordering is to return a sequence of service references in the order that the trader stores the matching offers.

The JTrader [17] implementation includes a simple graphical interface for demonstrating queries of the trader service. The application is used to perform a constraint and preference based query on available SPR libraries. Three available libraries are registered with the *trader*. Their names are "MTL," "SPR Research," and "ICL." The first search constrains the name to be equal to "SPR Research" without any preferences (see Figure 27). As expected, a reference to library named "SPR Research" is returned. The second query uses *true* as the constraint and a preference that the name is equal to "MTL" (see Figure 28). The resulting search returned all three libraries with "MTL" first in the sequence. The use of this simple query application demonstrates how any client application can use a *trader* service to locate services transparent to location.

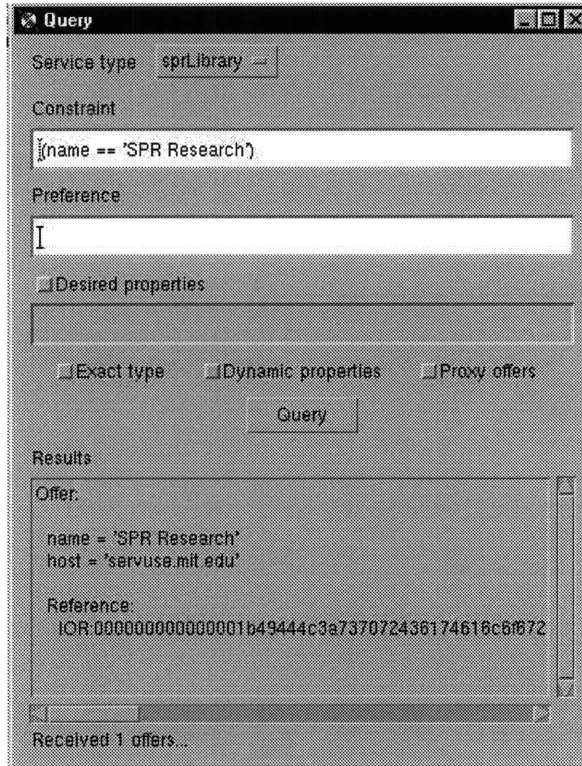

Figure 27: Query with constraints

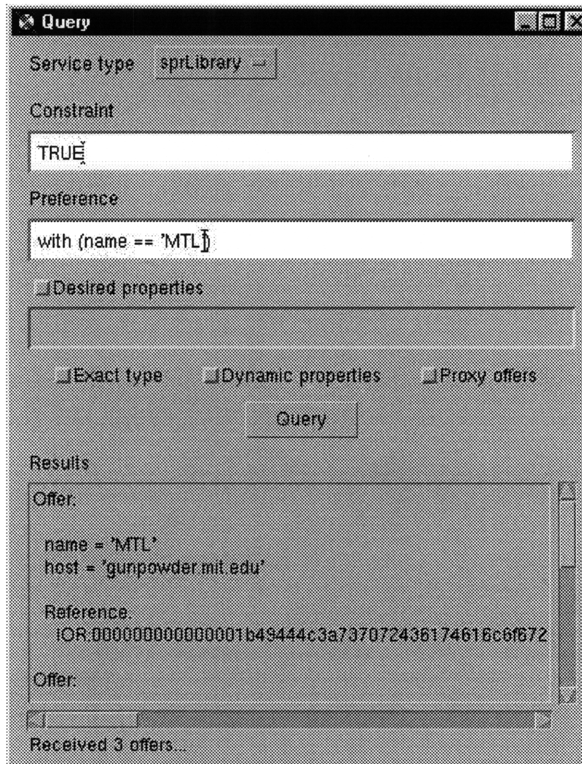

Figure 28: Query with preference

Chapter 7

Semiconductor Process Browser

In chapters 3, 4, 5, and 6, the base services of a distributed software architecture for semiconductor process design were detailed. This chapter introduces a semiconductor process browser that utilizes the architecture. Emphasis is placed on areas where the browser interacts with the *trader*, *library*, SPR, and *life cycle* services.

7.1 Overview

A semiconductor process browser was developed to test and demonstrate the functionality of the distributed software architecture [18]. The browser can access process information from a heterogeneous collection of repositories across global networks. The browser uses the components of the distributed software architecture for semiconductor process design to interact with repositories and surrounding services. The browser is a Java applet [13] that views semiconductor process information. It can be extended to support editing capabilities and other services such as simulators and fabrication tools as they become available through the *trader* service.

7.2 Finding services

The semiconductor process browser uses a *trader* service to retrieve initial object references. The *trader* service publishes a stringified IOR at a known URL location. The applet reads the IOR, converts it to an object reference, and binds to the trader. Once that applet is connected to the trader service, it can obtain references to services such as libraries.

Upon startup, the process browser queries the *trader* service to find all available SPR libraries. Users can connect to any of the libraries returned by the trader (see Figure 29). Information from the trader such as name, hostname, version, persistent, and process types can be displayed to users.

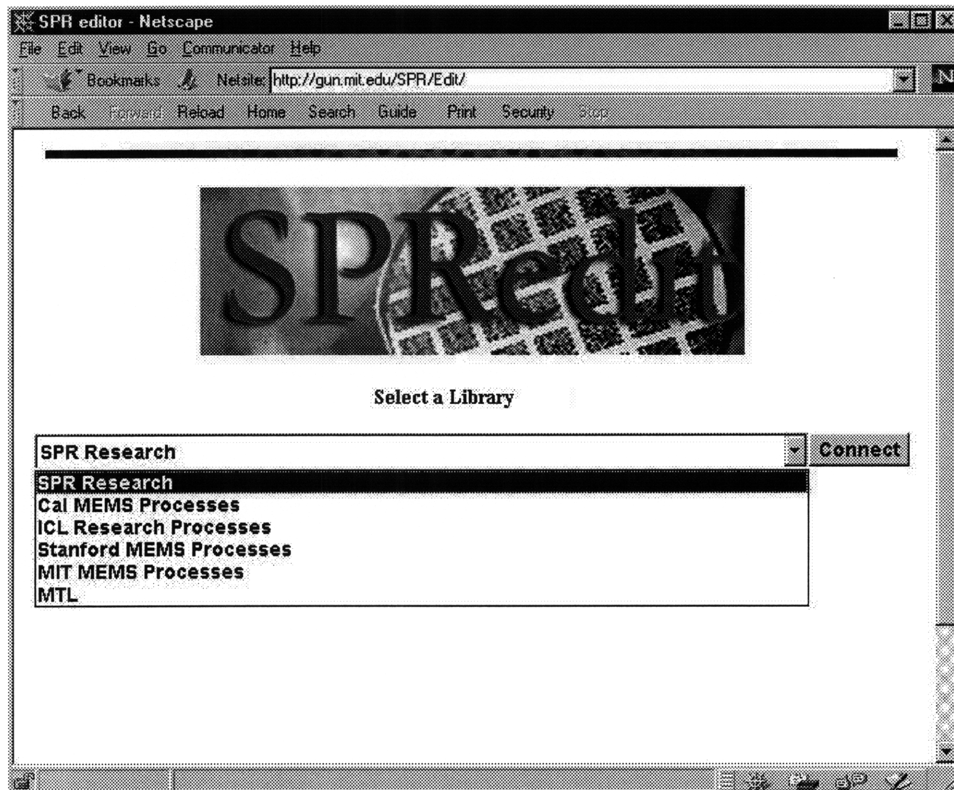

Figure 29: Browser interacting with trader service

7.3 Library service

Once connected to a library service, the browser retrieves and displays a list of catalogs (see Figure 30). Users select specific process catalogs to view. The applet uses the *allCatalogs* interface to retrieve the list of catalogs from the library server. The selection of a catalog brings up a new window with a list of processes. Upon selecting a catalog, the browser uses the library's *getProcesses* interface to retrieve a sequence of SPR process references.

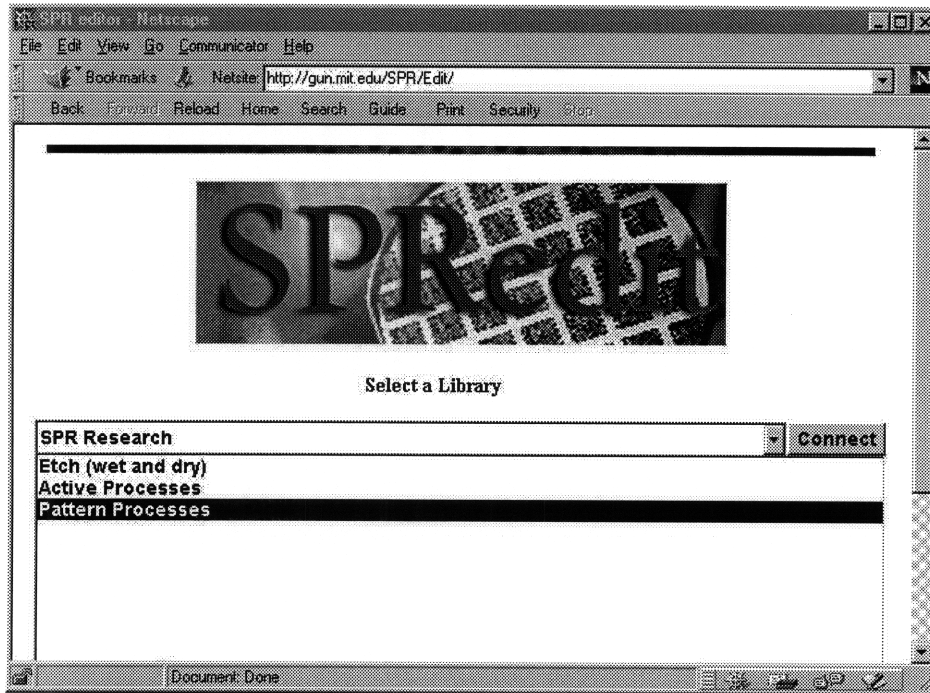

Figure 30: Browser interacting with library service

7.4 SPR service

With a sequence of processes, the SPR interfaces are used for accessing process characteristics and attributes (see Figure 31). The uppermost panel of the window displays the current location with the catalog and process. The example shown is in the sub-step oxide-boe-etch of the process backside-oxide-wet-etch within the catalog. The next panel contains object properties. The properties are retrieved via the *getProperties* interface, which all SPR objects support. In the example shown, properties such as duration, sink, and tank are displayed for the equipment view of the process step. Users can switch views of the process via the middle menu panel. References to views of a process object are gathered using the *allViews* interface. Lastly, view-specific attributes, such as machines for the equipment view, are displayed in the bottom portion of the window. The information is retrieved using view-dependent interfaces such as *getMachines* from the equipment view.

Figure 31: Browser interacting with SPR service

7.5 Utilizing the Life Cycle service

The current implementation of the SPR process browser does not allow editing of process information. Thus, there is no interaction with the life cycle service. Possible future research includes integrating editing capabilities to transform the browser into an editor. The life cycle service would be used for creating and managing remote SPR objects.

An editor would use the trader service to retrieve a reference to a repository for creating SPR objects. This entails binding to either a generic factory or a factory finder. For creating objects in only one remote location, a generic factory would be used. However, if a user needs to create objects at multiple remote locations, a factory finder would be more appropriate. Other life cycle operations such as copying, moving, and deleting objects are supported by each instance of SPR objects.

Chapter 8

Conclusion

Current computer systems for semiconductor process design are not well designed for distributed computing. Hardware and software dependencies, communication protocols, and lack of interfaces for managing distributed data are deficiencies of existing systems. Research presented in this thesis defines and implements a distributed software architecture for designing semiconductor processes. The architecture addresses the need for a common representation for semiconductor processes and implements surrounding services for managing and organizing repositories.

The semiconductor process representation (SPR) is used for communicating information about fabrication processes within the software architecture. The programmatic interfaces for processes are based on the SPR standard. The interfaces were defined such that repositories and clients can be implemented in multiple languages across platforms. The SPR interfaces allow process designers and manufacturers to communicate process information between services and applications across global networks.

An implementation of the OMG life cycle service has interfaces for creating, deleting, copying, and moving distributed objects [14]. In the distributed software architecture for semiconductor process design, the operations provided by the life cycle service are used by clients to manage SPR objects in remote locations.

Together, the SPR and life cycle services provide the core functionality for a distributed semiconductor process repository. This core does not include interfaces for

organizing and searching process information. Organizational and search interfaces were developed independent of the repository to allow multiple implementations. A library service was defined to organize remote processes into catalogs and query extensions were added to the life cycle service for searching individual repositories.

The architecture utilizes a generic trader service to facilitate matching client applications with the appropriate services. The distributed software architecture uses the trader with defined service types to enable client applications access to services such as the library, repository, and factory finder.

The creation of the semiconductor process browser demonstrates the functionality of the distributed software architecture. It can access process repositories across global networks. The browser uses the components of the software architecture to interact with distributed SPR repositories.

The main goal of this research was to define and implement platform independent interfaces enabling the development of distributed computing applications for semiconductor fabrication process design. A foundation of services have been defined and implemented to be reusable, extensible, portable, and easy to adopt. The generated modules can be used in the process management component of the SEMATECH CIM framework. The interaction of various client applications with the system will lead to further development and expansion of the distributed software architecture.

References

[1] H. L. Ossher and B. K. Reid, "Fable: A programming language solution to IC process automation problems," *Proceedings of SIGPLAN '83 Symposium of Programming Language Issues in Software Systems*, SIGPLAN Notices, pp. 137-148, ACM, June 1983.

[2] C. J. Hegarty, "Process-Flow Specifications and Dynamic Run Modifications for Semiconductor Manufacturing," PhD Thesis, *Electronic Research Lab. Memo 91.40*, U.C. Berkeley, April 1991.

[3] J. S. Wenstrand, "An Object-Oriented Model for Specification, Simulation, and Design of Semiconductor Fabrication Processes," *Integrated Circuits Laboratory Technical Report No. ICL91-003*, Stanford University, March 1991.

[4] D. S. Boning, *Semiconductor Process Design: Representations, Tools, and Methodologies*, PhD Thesis, Electrical Engineering, Massachusetts Institute of Technology, 1991.

[5] C. P. Ho, J. D. Plummer, S. E. Hansen, and R. W. Dutton, "VLSI process modeling – SUMPREM-III," *IEEE Trans. Electron Devices*, vol. ED-30, no. 11, pp. 1438-1452, November 1993.

[6] M. B. McIlrath, D. E. Troxel, M. L. Heytons, P. Penfield, Jr., D. S. Boning, and R. Jayavant, "CAFE - The MIT Computer-Aided Fabrication Environment," *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, Vol. 15, No. 2, p. 353, May 1992.

[7] D. Boning and M. McIlrath, "Semiconductor Process Representation Information Model Overview," Draft - V0.3.18 - March 15, 1994.

[8] Object Management Group, "OMG Background Information," <http://www.omg.org/omg00/backgrnd.htm>, April 1997.

[9] J. Siegel, *CORBA: Fundamentals & Programming*, John Wiley & Sons, 1996.

[10] T. J. Mowbray and R. Zahavi, *The Essential CORBA: Systems Integration Using Distributed Objects*, John Wiley & Sons, 1995.

[11] SEMATECH Incorporated, *CIM Framework Architecture Guide 1.0*, SEMATECH Technology Transfer 97103379A-ENG, 1997.

[12] SEMATECH Incorporated, *Computer Integrated Manufacturing (CIM) Framework Specification – Version 1.5*, SEMATECH Technology Transfer #93061697I-ENG, 1997.

[13] J. Gosling and H. McGilton, "The Java Language Environment: A White Paper" <http://java.sun.com:80/docs/white_papers.html>, May 1996.

[14] Object Management Group, *CORBAservices: Common Object Request Services Specification*, 1996.

[15] A. Vogel, *Java Programming with CORBA*, John Wiley & Sons, 1997.

[16] Object Design, “PSE for Java API Users Guide,” <http://www.odi.com/content/products/pse/doc_120/doc/apiug/index.htm>, December 1997.

[17] M. Spruiell, “JTrader 0.2,” <http://www.intellisoft.com/~mark>, December 1997.

[18] W. Moyne, private communication, August 1997.

Appendix A

SPR IDL

This appendix contains the base Semiconductor Process Representation (SPR) IDL.

```
/***************************************************************************
  IDL Declarations for Semiconductor Process Representation (SPR)

  Author:  Matt Verminski (mvermins@mit.edu)
  Date:    25-March-1997
  Revised: 31-October-1997

***************************************************************************/

#include "LifeCycle.idl"

module sprCORBA {
/***************************************************************************
  BASE INFORMATION MODEL - Guideline for how process information is organized.
***************************************************************************/

  /* INTERFACE DECLARATIONS */
  interface sprDistance;
  interface sprEffect;
  interface sprEffectsView;
  interface sprEnvironment;
  interface sprEnvironmentView;
  interface sprEquipmentView;
  interface sprEquipmentState;
  interface sprExtensibleObject;
  interface sprExtensibleNamedObject;
  interface sprGeneralEnvironment;
  interface sprGeneralEnvironmentParameter;
  interface sprMachine;
  interface sprMaterial;
  interface sprNamedObject;
  interface sprParameter;
  interface sprProcess;
  interface sprProcessView;
  interface sprProperty;
  interface sprTimedParameter;
  interface sprView;

  interface sprDepositEffect;
  interface sprChangeMaterialEffect;
  interface sprConformalDepositEffect;
  interface sprPlanarDepositEffect;
```

```
interface sprEtchEffect;
interface sprStripEtchEffect;
interface sprIsotrohicEtchEffect;
interface sprGrowthEffect;

/* Type definitions for specific parameter types */


/* TYPE DEFINITIONS and BASE DATA STRUCTURES */

// Sequences used to represent sets of objects
typedef sequence<sprEffect> sprEffectSeq;
typedef sequence<sprEquipmentState> sprEquipmentStateSeq;
typedef sequence<sprEnvironment> sprEnvironmentSeq;
typedef sequence<sprTimedParameter> sprEnvironmentParameterSeq;
typedef sequence<sprMachine> sprMachineSeq;
typedef sequence<sprProcess> sprProcessSeq;
typedef sequence<sprProperty> sprPropertySeq;
typedef sequence<sprView> sprViewSeq;

// Define duration as a struct of hours, minutes, seconds
// If hours and minutes are 99, then the duration is undefined.
struct sprDuration {
  long hours;
  long minutes;
  float seconds;
};

// sprStatFloat is a parameter that contains a mean value and its
// respective standard deviation
struct sprStatFloat {
  float mean;
  float sdev;
};

// Value is either a primitive data type or and sprParameter
enum sprValueType { BOOLEAN, SHORT, LONG, FLOAT, DOUBLE, STRING,
              PARAMETER, STATFLOAT };
union sprValue switch(sprValueType) {
   case BOOLEAN:    boolean      Boolean;
   case SHORT:      short        Short;
   case LONG:       long         Long;
   case FLOAT:      float        Float;
   case DOUBLE:     double       Double;
   case STRING:     string       String;
   case PARAMETER:  sprParameter Parameter;
   case STATFLOAT:  sprStatFloat StatFloat;
};

// Structure definitions for various Effect Locations
struct sprSimpleEffectLocation {
  string  MaterialRegion;
  boolean IsExposed;
};

// EffectLocation describes where on (or in) the wafer the effect occurs
// by specifying the material region to which the effect applies, and
// whether or not the region is exposed.
enum sprEffectLocationType {SIMPLE};
union sprEffectLocation switch(sprEffectLocationType) {
   case SIMPLE: sprSimpleEffectLocation Simple;
};

// Enumeration of possiblue units (used by Parameters)
enum sprUnits { NONE, m, mm, nam, pm, sec, ns, ps, cSt, A};
// fill in more later

// EXCEPTIONS
exception sprNotUniqueViewType { string sprViewType; } ;
exception sprInvalidValue { string reason; } ;
exception sprInvalidUnits { string reason; } ;
```

```
// EXCEPTION for invalid parameters
exception sprInvalidMaterial            { string reason; } ;
exception sprInvalidTimedParameter      { string reason; } ;
exception sprInvalidEnvironmentParameter { string reason; } ;
exception sprInvalidDistance            { string reason; } ;

/* INTERFACE DECLARATIONS */

// spr::Property
// -------------
// Name, value pair.  Useful for modeling "top-level parameters" and
// other possiblities such as documentation.  Effective mechanism for
// extending the information model.
interface sprProperty : LifeCycle::LifeCycleObject {
  void   setName(in string inPropertyName);
  string getName();

  void   setValue(in sprValue inValue);
  sprValue  getValue();
};

// spr::ExtensibleObject
// ---------------------
// ExtensibleObject is an abstract interface which contains a set
// (sequence) of properties.  Used to model additional attributes of
// objects.
interface sprExtensibleObject : LifeCycle::LifeCycleObject {
  void           setProperties(in sprPropertySeq inProperties);
  void           addProperty(in sprProperty inProperty);
  sprPropertySeq allProperties();
};

// spr::NamedObject
// ----------------
// NamedObject is an abstract interface to attach a name reference
// to objects.
interface sprNamedObject : LifeCycle::LifeCycleObject {
  void   setName(in string inObjectName);
  string getName();
};

// spr::ExtensibleNamedObject
// --------------------------
// ExtensibleNamedObject is an abstract superclass of named and
// extensible objects.
interface sprExtensibleNamedObject : sprNamedObject, sprExtensibleObject {
};


// spr::View
// ---------
// Contains specific information about a particular view of the process.
// There are four basic views that are derived from spr::View -
// spr::ProcessView, spr::EffectsView, spr::EnvironmentView, and
// spr::EquipmentView
interface sprView : sprExtensibleNamedObject {
  void   setViewType(in string inViewType);
  string getViewType();
};

// spr::Effect
// -----------
// Description of the change to the state of a wafer.  Take wafer states
// and transform them to new wafer states.
interface sprEffect : sprExtensibleObject {
  void              setEffectLocation(in sprEffectLocation inEffectLocation);
  sprEffectLocation getEffectLocation();
};

// spr::EquipmentState
```

```
// -------------------
// Describe important equipment settings and a time duration that is
// associated with the settings.
interface sprEquipmentState : sprExtensibleObject {
  void        setTimeDuration(in sprDuration inTimeDuration);
  sprDuration getTimeDuration();
};

// spr::Machine
// ------------
// Contains a name for a particular piece of machinery.
interface sprMachine : sprExtensibleObject {
  void   setMachineName(in string inName);
  string getMachineName();
};

// spr::Environment
// ----------------
// Description of the (perhaps time-varying) physical variables that
// determine the reaction of the wafer.
interface sprEnvironment : sprExtensibleObject{
  void        setTimeDuration(in sprDuration inTimeDuration);
  sprDuration getTimeDuration();
};

// spr::Parameter
// --------------
// Primitive items of process information.  They can describe constant
// or time-varying state information.
interface sprParameter : LifeCycle::LifeCycleObject {
  void      setValue(in sprValue inValue)
    raises(sprInvalidValue);
  sprValue  getValue();

  void      setUnits(in sprUnits inUnits)
    raises(sprInvalidUnits);
  sprUnits  getUnits();
};

// spr::Process
// ------------
// Named container holding information about a semiconductor process.
// It contains any number of spr::Views and spr::Properties.
interface sprProcess : sprExtensibleNamedObject {
  // Exception signal raised when an attempt is made to make
  // a view that is not a unique view type
  void       setViews(in sprViewSeq inViews)
    raises(sprNotUniqueViewType);
  void       addView(in sprView inView)
    raises(sprNotUniqueViewType);

  sprViewSeq allViews();
};


//** Specific Views **//

// spr::EquipmentView
// ------------------
// Information which describes the equipment setup and operation.  This
// may be generic ot an equipment class or specific.
interface sprEquipmentView : sprView {
  void                  setMachines(in sprMachineSeq inMachines);
  void                  appendMachine(in sprMachine inMachine);
  sprMachineSeq         allMachines();

  void                  setSettings(in sprEquipmentStateSeq inSettings);
  void                  appendSetting(in sprEquipmentState inSetting);
  sprEquipmentStateSeq allStates();
};
```

```
// spr::EffectsView
// ----------------
// Represents a change or changes in the static wafer state.  The
// cumulative effects specified by an effects view represents the change-
// in-wafer-state from inter-process interval before the process with which
// the effects view is associated to the immediately following interprocess
// interval.  Typically used by wafer-state "simulators".
interface sprEffectsView : sprView {
  void         setEffects(in sprEffectSeq inEffects);
  void         appendEffect(in sprEffect inEffect);
  sprEffectSeq allEffects();
};

// spr::EnvironmentView
// --------------------
// The specific dynamic physical environment (state) experienced by the
// wafer.  This is the environment the wafer actually experiences and is
// the basis for physically based simulators.  Environments are descriptions
// of thermodynamically intensive state variables (possibly time variant).
// Coupled with the wafer state they are used to model the physical
// (including chemical) process the wafer undergoes during the process.
interface sprEnvironmentView : sprView {
  void              setEnvironment(in sprEnvironmentSeq inEnvironment);
  void              appendEnvironment(in sprEnvironment inEnvironment);
  sprEnvironmentSeq allEnvironment();
};

// spr:ProcessView
// ---------------
// The sequential breakdown of subprocesses that together make up the
// process.
interface sprProcessView : sprView {
  void          setSubProcesses(in sprProcessSeq inSubProcesses);
  void          appendSubProcess(in sprProcess inSubProcess);
  sprProcessSeq allSubProcesses();
};

//** Specific Parameters **//

// spr::TimedParameter
// -------------------
// Contains parameter that has a start value (inherited from Paramter)
// and an optional end value.
interface sprTimedParameter : sprParameter {
  void     setEndValue(in sprValue inEndValue);
  sprValue getEndValue();
};

// spr::EnvironmentParameter
// -------------------------
// Environment are a specific instance of a timed parameter.
interface sprEnvironmentParameter : sprTimedParameter {
};

// spr::Distance
// -------------
// Used for measuring distance.  The value type is to be implemented
// such that it is double or a string indicating ALL
interface sprDistance : sprParameter {
};

// spr::Material
// -------------
// Used for specifying a material type.  The value type is to be implemented
// such that it is always a string.
interface sprMaterial : sprParameter {
};


//** Specific Environments **//
```

```
  // spr::GeneralEnvironment
  // ------------------------
  // Container of general environment parameters that are common to the
  // process that the envrionment is associated with.
  interface sprGeneralEnvironment : sprEnvironment {
    void setEnvironmentParameters(in sprEnvironmentParameterSeq inParameters);
    void appendEnvironmentParameter(in sprEnvironmentParameter inParameter);
    sprEnvironmentParameterSeq allEnvironmentParameters();
  };


//************************** END OF BASE SPR MODEL **************************//

// EFFECTS

  // spr::ChangeMaterialEffect
  // -------------------------
  // Changes the material from oldmaterial to newmaterial with possible
  // attributes residing in the properties.
  interface sprChangeMaterialEffect : sprEffect {
    void        setOldMaterial(in sprMaterial inOldMaterial);
    sprMaterial getOldMaterial();

    void        setNewMaterial(in sprMaterial inNewMaterial);
    sprMaterial getNewMaterial();
  };

  // spr::DepositEffect
  // ------------------
  // Simple effect to describe the vertical modeling of a one dimensional
  // deposit of a specific material for a specific thickness.  Base model:
  //
  //                   =====
  //                  |     |
  //              =====     =====
  //
  interface sprDepositEffect : sprEffect {
    void        setMaterial(in sprMaterial inMaterial);
    sprMaterial getMaterial();

    void        setThickness(in sprDistance inThickness);
    sprDistance getThickness();
  };

  // spr::ConformalDepositEffect
  // ---------------------------
  // Subtype of sprDepositEffect that conforms to the layer the material
  // is deposited upon:
  //
  //                  //===\\
  //                 //     \\
  //             =====       =====
  //
  interface sprConformalDepositEffect : sprDepositEffect {
  };


  // spr::PlanarDepositEffect
  // ------------------------
  // Subtype of sprDepositEffect that deposits a planar layer of material
  // upon the previous layer:
  //
  //              ===============
  //              =====     =====
  //              =====     =====
  //
  interface sprPlanarDepositEffect : sprDepositEffect {
  };

  // spr::EtchEffect
  // ---------------
```

```
  // Simplest etch effect is a vertical etch.  The material to be etched
  // is specified along with the thickness of the material to etch.  This
  // may be a specific number or all to indicate that all the material
  // should be etched.
  interface sprEtchEffect : sprEffect {
    void        setMaterial(in sprMaterial inMaterial);
    sprMaterial getMaterial();

    void        setThickness(in sprDistance inThickness);
    sprDistance getThickness();
  };

  // spr::StripEtchEffect
  // --------------------
  // The thickness for the strip etch effect is implemented such that it is
  // always ALL.  The material layer is removed everywhere.
  interface sprStripEtchEffect : sprEtchEffect {
  };

  // spr::IsotropicEtchEffect
  // ------------------------
  // An equal thickness of material is removed in all directions.
  interface sprIsotrophicEtchEffect : sprEtchEffect {
  };

  // spr::GrowthEffect
  // -----------------
  // Growth is defined by three parameters - the material grown, the
  // thickness of the resulting layer, and the depth of the underlying
  // layer which is consumed.
  interface sprGrowthEffect : sprEffect {
    void        setMaterial(in sprMaterial inMaterial);
    sprMaterial getMaterial();

    void        setThickness(in sprDistance inThickness);
    sprDistance getThickness();

    void        setDepth(in sprDistance inDepth);
    sprDistance getDepth();
  };

  // More effects such as sprAddFieldEffect, sprDiffusionEffect,
  // sprChangeMaterialEffect, etc. can also be added to the model

// END EFFECTS

};// module spr
```

Appendix B

SPR ODL

```
/**************************************************************************

  ODL Declarations for Semiconductor Process Representation (SPR)
  --------------------------------------------------------------

  Author:  Matt Verminski (mvermins@mit.edu)
  Date:    17-June-1997
  Revised: 15-August-1997

**************************************************************************/

module sprDB {

  // Define duration as a struct of hours, minutes, seconds
  // If hours and minutes are 99, then the duration is undefined.
  struct sprDuration {
    long hours;
    long minutes;
    float seconds;
  };

  // sprStatFloat is a parameter that contains a mean value and its
  // respective standard deviation
  struct sprStatFloat {
    float mean;
    float sdev;
  };

  // Value is either a primitive data type or and sprParameter
  enum sprValueType { BOOLEAN, SHORT, LONG, FLOAT, DOUBLE, STRING,
                 PARAMETER, STATFLOAT };
  union sprValue switch(sprValueType) {
     case BOOLEAN:    boolean      Boolean;
     case SHORT:      short        Short;
     case LONG:       long         Long;
     case FLOAT:      float        Float;
     case DOUBLE:     double       Double;
     case STRING:     string       String;
     case PARAMETER:  sprParameter Parameter;
     case STATFLOAT:  sprStatFloat StatFloat;
  };

  // Structure definitions for various Effect Locations
```

```
struct sprSimpleEffectLocation {
  string  MaterialRegion;
  boolean IsExposed;
};

// EffectLocation describes where on (or in) the wafer the effect occurs
// by specifying the material region to which the effect applies, and
// whether or not the region is exposed.
enum sprEffectLocationType {SIMPLE};
union sprEffectLocation switch(sprEffectLocationType) {
   case SIMPLE: sprSimpleEffectLocation Simple;
};

// Enumeration of possiblue units (used by Parameters)
enum sprUnits { NONE, m, mm, nam, pm, sec, ns, ps, cSt, A};

/* INTERFACE DECLARATIONS */

// spr::Property
// -------------
// Name, value pair.  Useful for modeling "top-level parameters" and
// other possiblities such as documentation.  Effective mechanism for
// extending the information model.
interface sprProperty
  (key name) {
  attribute string name;
  attribute sprValue value;
};

// spr::ExtensibleObject
// ---------------------
// ExtensibleObject is an abstract interface which contains a set
// (sequence) of properties.  Used to model additional attributes of
// objects.
interface sprExtensibleObject {
  relationship Set<sprProperty> properties;
};

// spr::NamedObject
// ----------------
// NamedObject is an abstract interface to attach a name reference
// to objects.
interface sprNamedObject
  (key name) {
  attribute string name;
};

// spr::ExtensibleNamedObject
// --------------------------
// ExtensibleNamedObject is an abstract superclass of named and
// extensible objects.
interface sprExtensibleNamedObject : sprNamedObject, sprExtensibleObject {
};


// spr::View
// ---------
// Contains specific information about a particular view of the process.
// There are four basic views that are derived from spr::View -
// spr::ProcessView, spr::EffectsView, spr::EnvironmentView, and
// spr::EquipmentView
interface sprView : sprExtensibleNamedObject {
  attribute string viewType;
};

// spr::Effect
// -----------
// Description of the change to the state of a wafer.  Take wafer states
// and transform them to new wafer states.
interface sprEffect : sprExtensibleObject {
  attribute sprEffectLocation location;
```

```
};

// spr::EquipmentState
// -------------------
// Describe important equipment settings and a time duration that is
// associated with the settings.
interface sprEquipmentState : sprExtensibleObject {
  attribute sprDuration timeDuration;
};

// spr::Machine
// ------------
// Contains a name for a particular piece of machinery.
interface sprMachine : sprExtensibleObject {
  attribute string getMachineName;
};

// spr::Environment
// ----------------
// Description of the (perhaps time-varying) physical variables that
// determine the reaction of the wafer.
interface sprEnvironment : sprExtensibleObject{
  attribute sprDuration timeDuration;
};

// spr::Parameter
// --------------
// Primitive items of process information.  They can describe constant
// or time-varying state information.
interface sprParameter {
  attribute sprValue value;
  attribute sprUnits  units;
};

// spr::Process
// ------------
// Named container holding information about a semiconductor process.
// It contains any number of spr::Views and spr::Properties.
interface sprProcess : sprExtensibleNamedObject {
  relationship Set<sprView> views;
};


//** Specific Views **//

// spr::EquipmentView
// ------------------
// Information which describes the equipment setup and operation.  This
// may be generic ot an equipment class or specific.
interface sprEquipmentView : sprView {
  relationship Set<sprMachine> machines;
  relationship Set<sprEquipmentState> settings;
};

// spr::EffectsView
// ----------------
// Represents a change or changes in the static wafer state.  The
// cumulative effects specified by an effects view represents the change-
// in-wafer-state from inter-process interval before the process with which
// the effects view is associated to the immediately following interprocess
// interval.  Typically used by wafer-state "simulators".
interface sprEffectsView : sprView {
  relationship Set<sprEffect> effects;
};

// spr::EnvironmentView
// --------------------
// The specific dynamic physical environment (state) experienced by the
// wafer.  This is the environment the wafer actually experiences and is
// the basis for physically based simulators.  Environments are descriptions
// of thermodynamically intensive state variables (possibly time variant).
```

```
// Coupled with the wafer state they are used to model the physical
// (including chemical) process the wafer undergoes during the process.
interface sprEnvironmentView : sprView {
  relationship Set<sprEnvironment> environment;
};

// spr:ProcessView
// ---------------
// The sequential breakdown of subprocesses that together make up the
// process.
interface sprProcessView : sprView {
  relationship Set<sprProcessSeq> subProcesses;
};

//** Specific Parameters **//

// spr::TimedParameter
// -------------------
// Contains parameter that has a start value (inherited from Paramter)
// and an optional end value.
interface sprTimedParameter : sprParameter {
  attribute sprValue endValue;
};

// spr::EnvironmentParameter
// -------------------------
// Environment are a specific instance of a timed parameter.
interface sprEnvironmentParameter : sprTimedParameter {
};

// spr::Distance
// -------------
// Used for measuring distance.  The value type is to be implemented
// such that it is double or a string indicating ALL
interface sprDistance : sprParameter {
};

// spr::Material
// -------------
// Used for specifying a material type.  The value type is to be implemented
// such that it is always a string.
interface sprMaterial : sprParameter {
};


//** Specific Environments **//

// spr::GeneralEnvironment
// -----------------------
// Container of general environment parameters that are common to the
// process that the envrionment is associated with.
interface sprGeneralEnvironment : sprEnvironment {
  relationship Set<sprEnvironmentParameter> environmentParameters;
};



// EFFECTS

// spr::ChangeMaterialEffect
// -------------------------
// Changes the material from oldmaterial to newmaterial with possible
// attributes residing in the properties.
interface sprChangeMaterialEffect : sprEffect {
  attribute sprMaterial oldMaterial;
  attribute sprMaterial newMaterial;
};

// spr::DepositEffect
// ------------------
// Simple effect to describe the vertical modeling of a one dimensional
```

```
// deposit of a specific material for a specific thickness.  Base model:
//
//                   =====
//                  |     |
//              =====     =====
//
interface sprDepositEffect : sprEffect {
  attribute sprMaterial material;
  attribute sprDistance thickness;
};

// spr::ConformalDepositEffect
// --------------------------
// Subtype of sprDepositEffect that conforms to the layer the material
// is deposited upon:
//
//                  //===\\
//                 //     \\
//             =====       =====
//
interface sprConformalDepositEffect : sprDepositEffect {
};


// spr::PlanarDepositEffect
// ------------------------
// Subtype of sprDepositEffect that deposits a planar layer of material
// upon the previous layer:
//
//              ===============
//              =====     =====
//              =====     =====
//
interface sprPlanarDepositEffect : sprDepositEffect {
};

// spr::EtchEffect
// ---------------
// Simplest etch effect is a vertical etch.  The material to be etched
// is specified along with the thickness of the material to etch.  This
// may be a specific number or all to indicate that all the material
// should be etched.
interface sprEtchEffect : sprEffect {
  attribute sprMaterial Material;
  attribute sprDistance thickness;
};

// spr::StripEtchEffect
// --------------------
// The thickness for the strip etch effect is implemented such that it is
// always ALL.  The material layer is removed everywhere.
interface sprStripEtchEffect : sprEtchEffect {
};

// spr::IsotropicEtchEffect
// ------------------------
// An equal thickness of material is removed in all directions.
interface sprIsotrophicEtchEffect : sprEtchEffect {
};

// spr::GrowthEffect
// -----------------
// Growth is defined by three parameters - the material grown, the
// thickness of the resulting layer, and the depth of the underlying
// layer which is consumed.
interface sprGrowthEffect : sprEffect {
  attribute sprMaterial material;
  attribute sprDistance thickness;
  attribute sprDistance depth;
};
```

```
  // More effects such as sprAddFieldEffect, sprDiffusionEffect,
  // sprChangeMaterialEffect, etc. can also be added to the model

// END EFFECTS

};// module spr
```

Appendix C

Life Cycle IDL

This appendix contains the Life Cycle IDL. Modifications to the OMG standard are noted.

```
module LifeCycle{

  // Forward declarations
  interface FactoryFinder;
  interface LifeCycleObject;
  interface GenericFactory;

  // A simple structure containing two strings is used to represent the key
  // to objects.  One field contains the object name while the other contains
  // the object's hostname where it resides.
  struct Key {
    string name;
    string hostname;
  };

  typedef GenericFactory Factory;

  // A factory entry structured is useful to keep track of what factories
  // available for use by the factory finder implementation.
  struct factory_entry {
    Key key;
    GenericFactory factory_ref;
  };

  typedef sequence<GenericFactory> Factories;
  typedef sequence<Object> ObjectSeq;

  typedef struct NVP {
    // use basic string type until Istring has been implemented
    // Naming::Istring name;
    string name;
    any value;
  } NameValuePair;

  typedef sequence <NameValuePair> Criteria;

  // Exceptions
  exception NoFactory {
    Key search_key;
  };
```

```
exception DuplicateKey {  // addition for add_factory
  Key search_key;
};
exception NotCopyable { string reason; };
exception NotMovable { string reason; };
exception NotRemovable { string reason; };
exception InvalidCriteria{
  Criteria invalid_criteria;
};
exception CannotMeetCriteria {
  Criteria unmet_criteria;
};

// Clients pass factory finders to move and copy operations for
// scoping purposes.
interface FactoryFinder {
  // The find_factories operation is passed a key to identify the
  // desired factory.  If the factory exits, it is returned.
  Factories find_factories(in Key factory_key)
    raises(NoFactory);

  // The add_factory operation is passed a key to identify the factory
  // that should be added to the list and a reference to the factory.
  // Used by implemenations of factories to notify factory finders that
  // they exist.
  void add_factory(in Key factory_key, in GenericFactory factory_ref)
    raises (DuplicateKey);

  // The shutdown method is for turning a factory finder and its respective
  // factories off
  void shutdownAll();
};

// Defines copy, move, and remove operations.  Objects participate in the
// life cycle service by supporting this interface.
interface LifeCycleObject {
  // The copy operation makes a copy of the object.  The copy is located
  // in the scope of the factory finder passed as the parameter.  The
  // operation returns a reference to the new object.  The new object is
  // intialized from the existing object.
  LifeCycleObject copy(in FactoryFinder there, in Criteria the_criteria)
    raises(NoFactory, NotCopyable, InvalidCriteria, CannotMeetCriteria);

  // The move operation on the target moves the object to the scope of the
  // factory finder passed as the parameter.  It returns the new object
  // reference.
  LifeCycleObject move(in FactoryFinder there, in Criteria the_criteria)
    raises(NoFactory, NotMovable, InvalidCriteria, CannotMeetCriteria);

  // Remove instructs the object to cease to exist.  The object reference
  // for the target is no longer valid after remove successfully completes.
  void remove()
    raises(NotRemovable);

  // Implementation name for narrowing object references without calling
  // ORB specific methods
  string getMarker();

  // Methods for setting and returning the name of the factory that created
  // the object
  void   setFactoryName(in string name);
  string getFactoryName();
};

// The generic factory interface defines a generic creation operation that
// specific factory instances should implement.
interface GenericFactory {
  // The supports operation returns true if the generic factory can create
  // an object given a key.
  boolean supports(in Key k);
```

```
    // The create_object operation is passed a key that is used to identify
    // the object to be created.
    Object create_object(in Key k, in Criteria the_criteria)
      raises (NoFactory, InvalidCriteria, CannotMeetCriteria);

    // The list_objects operation is passed a key to identify a specific
    // factory and they returns a sequence of objects that have been created
    // by that factory.
    ObjectSeq list_objects(in Key k)
      raises (NoFactory);

    // The remove_object operation takes in a factory key and an object
    // reference to remove the object from the list of objects created by
    // the factory.  It returns true if the removal is successful.
    boolean remove_object(in Key factory_key, in Object obj_ref)
      raises (NoFactory);

    // The shutdown method is for turning a persistent factory off
    void shutdown();
  };


  // Query Extentions
  enum Predicate { EQ, NEQ, GTE, LTE, GE, LE };

  typedef struct constr {
    string name;
    any value;
    Predicate predicate;
  } Constraint;

  typedef sequence <Constraint> ConstraintSeq;
  exception InvalidConstraint { string reason; };

  // The generic factory interface defines a generic creation operation that
  // specific factory instances should implement.
  interface QueryableFactory : GenericFactory{
    // The query_objects operations takes in a factory key and a constraint
    // that is then evaluated and returns the objects that satisfy the
    // contraint.
    ObjectSeq query_objects(in Key k, in ConstraintSeq constraints)
      raises (NoFactory, InvalidConstraint);
  };

};
```

Appendix D

Catalog IDL

This appendix contains the Catalog IDL for the organization of SPR repositories via libraries.

```
#include "spr.idl"

module sprCatalog {

    // EXCEPTIONS
    exception DuplicateCatalogName { string reason; } ;
    exception CatalogDoesNotExist { string reason; } ;

    typedef sequence<string> StringSeq;

    struct Catalog {
        string name;
        sprCORBA::sprProcessSeq processes;
    };

    interface Library {
        // interfaces for naming the library
        void setName(in string name);
        string getName();

        // interfaces for creating, deleting, and retrieving Catalogs
        void addCatalog(in Catalog inCatalog)
          raises(DuplicateCatalogName);
        void removeCatalog(in string name)
          raises(CatalogDoesNotExist);
          StringSeq allCatalogs();

        // interfaces for setting and retrieving processes from a Catalog
        void setProcesses(in string inCatalogName,
                          in sprCORBA::sprProcessSeq inProcesses)
          raises(CatalogDoesNotExist);
        sprCORBA::sprProcessSeq getProcesses(in string inCatalogName)
          raises(CatalogDoesNotExist);

        // interfaces for adding and removing individual processes from
        // a Catalog
        void addProcess(in string inCatalogName,
                        in sprCORBA::sprProcess inProcess)
          raises(CatalogDoesNotExist);
```

```
        void removeProcess(in string inCatalogName,
                           in string inProcessName)
          raises(CatalogDoesNotExist);

        void shutdown();
    };
};
```

Appendix E

Trader Service Definitions

```
service sprRepository {
   interface GenericFactory;
   mandatory property string name;
   property string host;
   property string version;
   property boolean persistent;
   property boolean queryable;
   property sequence<string> processTypes;
};

service sprLibrary {
   interface Library;
   mandatory property string name;
   property string host;
   property string version;
   property boolean persistent;
   property sequence<string> processTypes;
};

service FactoryFinder {
   interface FactoryFinder;
   mandatory property string name;
   property string host;
   property string version;
   property sequence<string> supportedIDL;
   property boolean OMGcapable;
   property boolean Extensions;
};
```