Analysis and Specification of Office Procedures

by

Jay S. Kunin

February 1982

This research was supported in part under
a contract with Exxon Enterprises, Inc.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

Abstract

Conventional approaches to "office automation" focus on the lowest common denominator of office work: typing, filing, filling in forms, *etc.* As a consequence, the process of office systems analysis lacks tools and techniques that address the office in terms of business functions rather than as manipulation of paper artifacts. The Office Specification Language (OSL) and its associated analysis methodology have been developed as a means of implementing a *functional* approach to office procedure analysis and description.

OSL is based on several premises derived from a study of office work and office systems analysis at a functional level:

- There exist high-level constructs common to a wide variety of disparate offices. A structured, formal language built upon such standardized abstractions can be useful in helping an analyst approach, understand, and describe the operations of many offices.
- Office procedures deal with (abstract) *objects*, not paper forms. Forms and other documents are not basic to office operations; they are mechanisms for organizing and transmitting information about some more fundamental object. Therefore office analysis and specification should focus not on forms, but rather on the underlying business requirements that must survive any change in system implementation.
- Office procedures are fundamentally simple; their apparent complexity is not inherent, but due to a myriad of special cases, historical accretions, and implementation details. Identification of a procedure's core requirements is the framework upon which analysis should be based. Such an understanding is a prerequisite to effective reorganization of and design of support systems for office functions.

OSL is postulated to be of utility for office analysis and systems design. Field tests of the language and methodology have shown that our basic approach is effective for analysis purposes, and have identified directions for further improvements.

Key Words: Office Automation, Office Analysis, Office Systems, Systems Analysis, Specification Languages, Integrated Office Systems

---

This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science in January 1982 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Acknowledgments

A grateful mind
By owing owes not, but still pays, at once
Indebted and discharg'd.

- John Milton

The standard set of acknowledgments begins with the thesis advisor. Mine shall be no different, though Mike Hammer, I believe, is. He has been a teacher, a colleague, a mentor, a patron, and a wonder to observe. He has taught me a great deal about the office automation business, among other things. Without his consistent encouragement I would not have begun, much less completed, a Ph.D. I hope he's satisfied.

The other members of my committee also contributed to my education in general and this thesis in particular. Irene Greif did an impressive job of learning both the concepts and the details of this work in a short time; her comments and assistance, particularly in teaching the field study courses, were especially helpful. Marvin Sirbu brought to our group a unique perspective, valuable in its vision and thoughtfulness. He has been very supportive of me and my aspirations; his confidence in my abilities occasionally exceeds my own, which has encouraged me to live up to his expectations.

Two of the giants upon whose shoulders I stand have also been good friends and role models of sorts. Mike Zisman told me years ago that an office specification language is a hard problem, and it has taken me too long to prove him correct. He has long been a source of encouragement, and I'm grateful for his advice and enthusiasm. Dennis McLeod provided the initial existence proof of the entire process, and generously contributed portions of his work that mine could proceed. His reputation remains, long after his departure for more appropriate surroundings, as spiritual guide and encouragement for future generations of graduate students.

Several of my colleagues have been of particular help in the development of OSL. Craig Zarmer came upon the scene initially as a guinea pig, and in proving his value and talent as a serious participant in research provided me with both intelligent commentary and a great many hours of support work. Juliet Sutherland and Sandor Schoichet have also been active participants in much of this work; each helped me wrestle with some of the underlying issues, and both have made valuable contributions of their own. I am also very grateful for the assistance of all those who

participated in the OSL field studies. Several of these were of particular help in providing thoughtful and detailed feedback: Vernon Skipper, Ed Landry, Jim Murphy, Lars Roesch, Jack Barlow, Elaine Beckwith, Bill Harris, and several others who must remain anonymous.

My present and former associates in the Office Automation Group (and the erstwhile Database Systems Group) at LCS, along with the rest of the folks on the second floor and assorted hangers-on, have helped create an environment that is always an adventure, and often a joy. Several of them have been particularly responsible for much of my sanity and insanity over the years. Stan Zdonik is especially guilty; may he follow me soon. Special technical thanks are also due to Dave Lebling (Lord of Rmode), Tim Anderson (Utility Guru), Larry Rosenstein (Scribe Seer), Jan Schoof, and James Hung, for their essential contributions.

My family and friends have provided the base upon which my efforts have been possible. My parents in particular have consistently made all the opportunities available, and I can never repay their contribution. Betsy Broucek and Phil Bernstein have been constant sources of support and encouragement through my graduate career. Finally, Gabrielle Silber's patience and caring have allowed me to complete this work with renewed enthusiasm and expectation.

Table of Contents

Chapter One

Introduction

1.1 Office Automation

The advent of "office automation" as a label for numerous efforts in product marketing, business analysis, and research has brought to light a number of significant barriers to the penetration of the office environment by computer technology. While "office automation" has no fixed meaning, it does represent the confluence of computer and communications technology, systems and data processing practice, organizational development, and business strategy. However, if, as we assert, the goal of office automation is to improve the realization of office functions, [16] the reality is that few effective models exist for most parts of the improvement process.

The labor-intensive nature of office work provides significant motivation for the application of computer technology to offices. However, there has not as yet been much progress in the development of office systems, beyond the design of rather rudimentary computer-based tools and the attempt to adapt conventional information systems to office needs. There appear to be two major problems preventing large-scale use of computer systems in improving office functioning. First, neither computer nor management scientists have yet provided a clear approach to the architecture of systems for offices, primarily because it is not sufficiently understood or agreed just what such systems should do. Second, even if there were such an understanding of the general requirements of an "office automation" system, it is still the case that individual offices differ significantly in their purposes, organization and operation. This diversity requires that even

systems designed specifically for office needs be tailored for individual applications; yet contemporary technology for the construction of complex custom software systems produces results that are expensive, error-prone, and difficult to change. The research in office automation represented by this thesis has as its goal an approach to solutions to these two problems: an analysis of the office domain, in order to provide a framework for understanding the requirements of office information systems; and, using that framework, the development of new tools for the analysis and description of such systems.

The successful movement of software technology into new areas depends critically upon the ability of systems analysts and designers to understand the needs and structure of the applications. As we see the movement of hardware from the data processing organization to the end user, in the guise of distributed data processing, word processing, professional support systems, or other vendor-defined terms, we witness a parallel growth in the desire and attempts of users to conform the capabilities of such systems to their particular needs. Some of these efforts have led to impressive successes (*e.g.*, [26, 33]), and there has arisen a new software subfield in the programming of word processing system macro facilities; yet in many cases we see the lessons of the past 25 years of DP experience being relearned by new users, in a painful and costly manner.

This experience with customization of general-purpose hardware (whether minicomputers, "intelligent" terminals, or word processors) points to perhaps the most effective means of utilizing such technology in the future. That is the development of office-specific systems that meet their users' particular needs and are adaptable to the inevitable changes in those needs. [18] Users' requirements, however, go beyond the functionality provided by isolated tools, and even beyond the need for an integrated, easy-to-use interface to those tools. In fact, what we see in the extensive development of small scale applications is a need for support of

office *procedures.* We believe that office systems, to enhance most effectively the productivity of the organization, should support rationalized office procedures, as well as low-level generic office tools.

However, there is a major barrier to the realization of this goal — we do not sufficiently understand office procedures in the abstract to provide general methodologies for their analysis, specification, and rationalization. Current practice is to develop office systems from the bottom up, starting with programming of repetitive word processing tasks and evolving to "list processing" or "records management," the use of a word processor to perform small DP applications. The paradigm of problem analysis followed by system design and then implementation is not often followed, or indeed perceived as useful.

The lack of systems designed to address office procedures is due, among other things, to an absence of good models on which to base the analysis, design, and construction of office systems. While we might look to the data processing environment for models and techniques, we find that such tools do not easily address the aspect in which office procedures differ most significantly from dp processes — their semi-structured nature [57]. That is, office procedures consist of a combination of both structured (algorithmic) and unstructured (judgmental) kinds of activities. Without appropriate tools for examining and describing semi-structured procedures, it is difficult to understand what goes on in an office, to communicate between analysts and office workers and between analysts and designers, or to provide a framework in which knowledge can be transferred from one project to the next.

The goal of this thesis is to develop a framework for understanding office procedures that can be used to analyze, explain, communicate, and optimize the operations of a variety of office situations. This understanding is incorporated in a

model of office procedures and a formal specification language based upon that model. Ancillary to the language, and based on the same model, is a methodology for office procedure analysis that incorporates the language constructs as a guide, and a formal specification as a goal.

In general terms, our work is meant to investigate issues that may lead to a "theory" of office work. Such a theory would provide a framework for office analysis, systems design and implementation, product development, education and future research in the field. We do not imply that all office work can or should be formally described; we will have more to say about this subject in later sections. Rather, we believe that there are some commonalities that underlie much of the domain, and that a conceptualization in terms of some fundamental structures will serve to enhance the capabilities of those who must deal with it. This thesis describes an investigation into the nature and utility of some of those structures.

1.2 Outline of the Thesis

The remainder of this document describes a project that addresses some of the gaps in our understanding of office work and the application of computer-based technology thereto. In Chapter 2 we examine contemporary practice in office systems analysis, relevant experience from related fields, and current office automation research. We describe our *functional* approach to office automation and hypothesize that an *office specification language,* designed to meet a number of requirements that we develop, can serve a major role in effecting that approach. We then provide an outline of our research project in the development of a particular office specification language, called "OSL," as a vehicle for testing that hypothesis.

In Chapter 3 we define the premises, following from our functional approach,

upon which the design of OSL is based. We provide a summary of the language, and then explain the key concepts of OSL and how its major features are derived from the overriding premises. Chapter 4 is an annotated example of an OSL office specification that should serve to illustrate the nature, if not all the details, of the language; the full language reference is in Appendix A. Chapter 5 describes a methodology for using OSL in office analysis. This technique embodies the OSL framework and approach, and is a guide to using OSL to develop an understanding of an office and produce an English and/or OSL description of it. Chapter 6 describes the research methodology that provided information by which OSL has been developed and tested. The data includes a series of field studies in which volunteers from the office automation staffs of several cooperating firms were taught both OSL and OAM, an office analysis methodology [48] that incorporates the techniques described in Chapter 5. We also discuss the results of the studies and their implications for guiding the development and use of OSL. Chapter 7 is a summary with suggestions for research directions in office procedure analysis and specification.

The full, formal definition of OSL is provided in the Appendices. Appendix A is the language reference manual; Appendix B is the formal grammar.

Chapter Two

Office Procedure Analysis

In this Chapter we examine the nature of contemporary practice in office analysis and identify a number of problems that confront the office procedure analyst. The most critical of these is the lack of an adequate model of office operations that can be used as a framework for analysis and description of office procedures. We suggest that such a framework can be defined and used as the basis of a specification language for office procedures.

2.1 What is Office Automation?

The term "office automation" is perhaps the most visible buzzphrase in the contemporary computer environment. It has been applied to such disparate products and ideas (ranging, for example, from word processors to PABXs to decision support systems to personal computers to distributed computing) that it has been rendered nearly meaningless. We can, however, categorize most commercial systems available under the rubric of office automation as components of an "electronic desk" model. That is, vendors of these products take a typical office environment as the *eidos* of what office systems should do, and thereby seek to provide electronic analogs to the tools used to perform common tasks in "unautomated" offices. In a manner similar to the development of the dictating machine as a replacement for the secretary taking shorthand, or the office copier replacing carbon paper and mimeograph, computer technology has produced electronic calculators to replace mechanical devices, word processing systems that replace paper drafts with electronic ones, and computer-based message/filing

systems that are beginning to take over some functions of the postal and telephone systems and file cabinet.

Most current research and development in OA tends to continue along the lines of this electronic desk model. More sophisticated hardware and better human-engineered text processing software allow for more efficient production of paper documents [23, 12]. The anticipated large-scale availability of computer networks, as well as particular corporate needs, has spurred development of message systems and associated storage and retrieval facilities [7, 37]. In the field of information management, much effort is spent toward providing interfaces that make possible sophisticated use of a database by computer-naive personnel [35]. Other work with major applications in offices includes the development of scheduling systems and similar "personal assistant"-type programs [39, 11, 19], decision support systems [25] and personal computer software.

While these items have a significant place in the spectrum of office automation work, it is necessary to understand that the electronic desk is only a part of the automated office. By providing the office worker with electronic tools, we attempt to make his actions more efficient. Yet if we are to bring significant changes to the office in the form of lower personnel costs and increased effectiveness of office processes, we should seek not only to mechanize office tools, but also to automate office functions [58]. That is, it is not really the actions of the office worker that we wish to address, but the larger issue of the functions that require the existence of the office. Thus, our conception of office automation in its broadest terms is the use of computer-based technology to increase the efficiency and effectiveness of office functions [18]. Note that this definition does not explicitly deal with office workers themselves; rather we are concerned with the more fundamental question of why they are in the office at all.

2.2 Understanding Office Work

The "office" (in "office automation") is often defined in practice by a group of people or by physical layout. Organization charts provide an indication of management responsibility, while floor plans give a different feel for organizational boundaries; both are often obsolete. Regardless of definition, the notion of "the office" as the target of office automation efforts is widespread and, we believe, incorrect. Our thesis is that there is no such thing as "the office" and that such a concept is counterproductive. Rather, there are many different kinds of offices, each with its own set of characteristics that should indicate the role, if any, that automated equipment can play therein. Therefore automation methodologies and tools that address the office as a fixed target are doomed to irrelevance and failure. The means by which offices can be differentiated and described is an important area of research that is, in part, addressed by this thesis.

When one watches people working in an office, it is often difficult or impossible to ascertain the structure, function, or purpose of their actions. One observer may look at a particular office and see typing, filing, telephoning, and conversation; a second would look at the same situation and describe the activities as communications, information management, and decision-making. A third observer might say that what is happening is the processing of admissions applications, evaluation of applicants, and the selection of next year's freshman class. While all these descriptions might be correct, they are clearly of differing utility to those who would "automate" that office. This multiple perspective points out a key issue in office systems analysis: the need to address the appropriate level(s) of abstraction. Indeed, one writer has suggested that the office itself is "a place for transacting abstractions." [42]

Historically, and still most commonly, the least common denominator of office

activities has served as the basis for the majority of analysis. Whether for time and motion studies aimed at improving efficiency [14, 2] or productivity analysis aimed at selling or justifying word processing equipment [38], office analysis in terms of time spent typing, filing, thinking, making telephone calls, *etc.* has formed the basis of office improvement methodologies. Examined at this level, office work is a tenuously connected series of locally oriented activities. The difference between the work of a file clerk and that of a department manager appears to be one of relative frequency of each activity, rather than inherent structure. The *purpose* of their work is not apparent in such a low-level analysis and thus is largely ignored [47].

Similarly, work in various aspects of computer and management research has as its goal the enhancement of the information-handling operations described above. In particular, the data processing and communications disciplines have recently offered simple though higher level taxonomies of office work. Communications, information management, and decision-making are elements of various "communications audit" or "information retrieval management" methodologies, such as [3]. While useful for some purposes, such techniques also assume that office work is the sum of those particular parts. Again the *goal* of the activities is not the point, merely their cataloging and enumeration.

As has been strongly suggested elsewhere [18], by focusing on these lower levels of abstraction, an analyst cannot provide for the efficient specification and implementation of office systems; he is led, rather, to mechanize the existing ways of doing things. In fact, we believe that office automation must take a more abstract, functional view of office work [17]; and further, that an adequate understanding of office work at the functional level does not currently exist. Indeed, it is the absence of a coherent model that provides an overall structure for office work that makes office systems analysis difficult and often ineffective.

How might we describe the actions of office workers in higher-level terms? One way would be to distinguish among various types of workers. As we have noted, the "office" in office automation is not a homogeneous idea — there are many kinds of offices. Similarly, there are many kinds of office work and office workers. It seems unlikely that a single technique or technology will be appropriate to all such workers or offices. Some efforts, particularly the low level activity-oriented approaches, concentrate on clerical and secretarial workers. Contemporary work in decision support systems, computer-aided design, and personal computer software addresses primarily the professional worker: the engineer, financial analyst, *etc.* Past efforts in management information systems addressed the middle- or upper-level managerial worker, even the chief executive officer. However, in spite of a number of serious efforts in these areas, for example [36, 25, 13], little in the way of practical tools for office analysis have emerged.

Whereas a classification of office activities by job title can be helpful in some areas, analysis by function may be more so. Rather than looking at clerical, secretarial, managerial, professional and executive workers, we may choose to examine administrative, procedural, or principal support functions, which may be carried out by any of the workers in an office. In fact, as we have argued elsewhere, [17] it is our thesis that *functional* analysis is the appropriate way to address office systems, and that analysis and specification of offices and office procedures should be directed along functional lines. [16]

Following this functional approach provides a framework for understanding, and presumably improving upon, the details office activities and how they operate to effect the office's functions. A crucial dimension in analyzing office activities is

task structure, as described by Gorry & Scott Morton in the context of management information systems [13]. A structured task is one that is amenable to algorithmic specification, as in most accounting activities for example. Unstructured tasks are those that are not amenable to such description, and that inherently require human intelligence in their execution; these are judgmental, decision-making tasks. Most office procedures consist of a combination of both kinds of activities, and may appropriately be called semi-structured processes. For example, the processing of an application for admission at the MIT Admissions Office consists of some structured activities, which ensure that the application is complete and valid, followed by an unstructured decision process in which the admissions director and his staff apply their experience, knowledge and judgment to the applications in order to select those to admit. This in turn is followed by the structured activities of sending letters, awaiting replies, distributing various copies of class lists, and maintaining certain records. (Note that "structured" does not mean "simple"; algorithms for implementing structured tasks may be large and complex.)

An office system should provide facilities for handling both structured and unstructured activities. Existing structure should be exploited by transferring control of activities to the system. By this we do not mean just the proper and timely invocation of structured activities; it is equally important that as much as possible of the bookkeeping of invocation, tracking and data handling be handled automatically by the system for all tasks, whether they can be expressed algorithmically or not. What activities then remain to be done by office workers are just those unstructured, judgmental tasks at which humans are good and computers are not. It is the system's *control* of the overall operation that is the essence of automation; the system may not always know *how* to do an activity, but if it understands *when* that activity must be done, by whom, and in what context, it can provide an integrated framework for the support of structured and unstructured

operations. It is by combining the capabilities of supporting both types of tasks in a single system environment that computer technology can best be used to achieve gains in office productivity. Consequently, any office analysis and description methodology should then be able to account for structured and unstructured tasks in the context of some higher-level structured model.

A perception of the abstract structure of the office domain, such as we are advocating, can certainly guide the builders of automated office systems. However, it is also our goal to provide, beyond this conceptual framework, specific tools and methodologies for the analysis of office systems and the implementation of appropriate computer-based information systems. The research described in this thesis is part of an effort to determine whether there is any underlying and fundamental structure to office work in general, and to office procedures in particular. By structure we mean a model of office operations that will allow for elucidation, description, communication, reorganization, and measurement of office work in a manner that is both usable and useful.

An initial part of this research has been an analysis of the types of activities that are important in offices. There are two major uses for the results of an analysis of this kind. First, the process of defining a set of constructs widely applicable in the office domain provides significant insight into the structure of that field. One immediate use of this information is the identification of the most appropriate targets for automation. In general, our work provides a framework in which to approach further research into office systems analysis. The relatively simple model of office work that we have developed can serve as a working hypothesis to be elaborated (or discarded) by experiment. The second important use for our results in this area is to form a foundation for the design of a formal higher-level language for the description of office processes and an analysis methodology that reflects that foundation. The design and field testing of our language constitute the major part of this research.

2.3 Automating Office Procedures

A key concept in our functional approach to office automation is that of an office *procedure*, a high-level construct that organizes and orders the individual activities of the office. We believe that a great many office functions are realized by semi-structured procedures. Such procedures intersperse structured tasks with judgmental ones, in the service of some particular function which is necessary to the organization. As will be discussed later, it is one of our premises that not only do these procedures exist in many settings, but that they are fundamentally simple in outline; their apparent complexity (or nonexistence) is due to the difficulty of recognizing the procedure being performed underneath the implementation details.

Many office procedures as they now exist are so overlain with obsolete and *ad hoc* activities that they are nearly indistinguishable from chaos. We believe that any effective office automation effort must first seek to rationalize the procedures in the office: to understand them in terms of the overall goals and functions of the office, and to adapt, reorganize, or eliminate them where appropriate. Our goal is to enable, through new tools and methodologies, the analysis and description of office procedures. This should greatly improve our ability to rationalize, and where feasible automate, office procedures.

As we noted earlier, and have argued extensively elsewhere [17], an automated office system is an integrated and interconnected collection of components under the supervision of an intelligent control program. This intelligence must be specific to the office in question: systems designers must recognize the fact that no two offices are exactly alike in their operations, procedures and interactions; and that generic "office automation systems" that require adherence to some "standard" modes of performing business functions will not provide for the most efficient realization of those functions. Thus, the "automation" of an office will require the

entire cycle of analysis, specification, design and implementation, in order to determine the requirements of the office and to embed this "knowledge" of what the office's procedures are into the system.

How do we provide such knowledge to a system? Even given a framework that will guide our analysis of the office, how do we make use of that general understanding in constructing office-specific systems? In practice, how do we best provide for the expression of the requirements of a particular office; in what form might a specification be presented so that it provides the requisite communications among analyst, office worker, and system implementor?

This is the fundamental problem with the functional approach to office automation: the issue of building office-specific information systems in a cost-effective fashion. Since each office is unique, installing off-the-shelf generic products into an office environment is no longer appropriate when attempting to realize functionally oriented office systems. Instead, a system development effort is required, in which the operations of the office in question are analyzed, its needs assessed, and a custom system designed and implemented for it. The last stage of this process will entail the construction of software that is specific to the particular office in question. This software will embody knowledge of the office's operation; it will automate selected clerical tasks, control the assorted devices employed in the system, serve as the intelligence that organizes and orders the steps of the office procedure as a whole, and provide tools that support office workers both in their specific functional activities and for their personal information managment needs. This software is clearly specific to the office in question. In other words, custom software must be produced for each office information system.

Unfortunately, the system development process outlined above requires highly trained personnel (systems analysts and designers) who must exercise ingenuity in

analyzing the operation of the office in question, defining its needs, and designing and then implementing a system. Moreover, experience has repeatedly shown that complex software systems produced by conventional means tend to be error-prone, costly to construct, and difficult to change. (See, for example, [53].) They have generally been restricted to large DP problems or special-purpose applications, where the risk and expense is justified by volume or lack of alternatives. If we are to be successful in building and installing custom office systems on a wide scale, we must seek new means to produce them.

Let us look more closely at the process of office system construction and the problems inherent in it. In the first stage, the current operations of the office are studied and their shortcomings identified, and the general capabilities of the automated system to be built are defined; in the next, specifications of this system are produced. These are the tasks of the office systems analyst, who then designs the structure of a system that will meet these specifications. A programmer finally reduces them to code. The major sources of difficulty in this process lie with the analyst's activities rather than the programmer's. The programmer need only seek to implement a system that meets the requirements given him; the analyst has the responsibility of constructing these specifications. His is a challenging and creative job; yet the analyst lacks any useful methodologies or tools to employ in analyzing an office or specifying a system for it. His task as a whole lacks structure; there are few guidelines or principles for him to employ.

One particular problem the analyst faces is that he has no effective notation or language in which to express himself. Many errors in software systems arise from the fact that the original specifications for the system are unclear, incorrect, or incomplete; this derives from the fact that they are poorly expressed in a language unsuitable for the purpose. Office analysts may use English or "structured English" to describe the current operation of an office as well as to specify the desired

functionality of an automated system that is to be built. Alternatively, they may utilize flow charts, screen definitions, or other DP-derived techniques. Each of these approaches suffers from its own set of problems.

The requirements of an office might be expressed in a detailed procedural specification of the activities of the office at the level of common contemporary general-purpose programming languages, but that approach has several shortcomings. First, such a detailed level of description would embody and perpetuate the existing task structure, rather than identify the function that is being realized; it would subordinate the end to the means. Second, such a description would be inflexible. Offices (like other complex systems) are dynamic; they are constantly evolving to match their changing environments and to meet new needs. Consequently, an office description must be able to evolve as well; one expressed at a fine level of procedural detail will find it difficult to accommodate change. Finally, constructing such a procedural description would be a major programming undertaking, and would suffer from the usual problems of unreliability and high cost. Furthermore, construction of each office system would start from the same point, with little advantage accruing from one system to the next.

Alternatively, the specification of office procedures might be expressed in English. Although rich and expressive, English (like all natural languages) is imprecise and ambiguous, and consequently not effective for the accurate specification of systems. Many errors in software systems arise from the fact that English specifications are unclear, incomplete, or incorrect. Natural language specifications do not bridge the gaps between experts in the office domain (workers, managers, office analysts) and those in the systems domain (vendors, system designers). The results are familiar: systems that are expensive to build, difficult to maintain, and impossible to adapt, and which do not solve the users' problems.

2.4 An Office Specification Language

We believe that many of the problems discussed above can be significantly mitigated by providing the analyst with a problem-oriented office specification language. This is a formal language for describing in high-level and machine-independent terms the operation of an office system (either manual or automated). It may be thought of as a notation in terms of which an office systems analyst can express himself, both for describing an existing office operation and for specifying the requirements of an office support system to a system designer. A specification language is a formal language, with well defined syntax and semantics; thus, any description expressed in it is unambiguous and open to a single interpretation. Furthermore, the primitives of a high-level language are based on the natural structures and vocabulary of office work so that the language user can express himself in terms natural to the application. Such a language can serve as an effective means for describing in a precise, natural, and understandable way the operation of an office's procedures.

There is a variety of potential uses for such a language. The principal one is as a communications mechanism between office systems analyst and the system designer. Because of its formality and precision, specifications expressed in the language can be clearly understood and interpreted by the system architect who uses them as a basis for his design effort. The use of this language would enable an office systems analyst to describe more precisely to a designer the system that is to be developed; this improved communication can have a major and positive impact on the systems thus produced, improving their quality and lowering their cost. The use of such a language facilitates the jobs of both the analyst and the system designer.

A second major use of the language is in the way it can impose a structure on the entire process of office analysis and system specification. By providing the

analyst with high-level primitives in terms of which he is to express a system, a specification language effectively gives him a basic set of concepts, in the form of templates, syntax rules, and semantic definitions, with which he is encouraged to analyze office operations. Thus, the analyst is presented not just with a set of disconnected language features but with an approach to their employment, a perspective on office operation that provides a conceptual framework in terms of which to analyze and describe office operations. The analyst will be able to readily express himself in terms familiar to him while suppressing irrelevant detail.

There are several other plausible uses of an office specification language, not directly related to the process of constructing automated office support systems. A formal specification language for office procedures could serve as an effective mechanism for expressing existing manual office operations. In current practice, English is the language employed in systems and procedures manuals; however, as is well known, these manuals are usually incomplete, difficult to read, and obsolete. Well-organized specifications in a high-level language can be used as a reference for office workers in many office environments; such a language, by enforcing modularity in its use, can reduce considerably the effort needed to keep specifications up to date. Related uses might include the training of new employees, and the recording of organizational history in a way that survives the coming and going of individual office personnel. The formal specifications of an office procedure could also be subjected to various analytic techniques in an effort to identify bottlenecks and problem area in its operation; this can highlight those areas of the procedure most in need of rationalization and redesign. Even outside the context of an automation effort, the use of a specification language could be an effective tool in understanding and reorganizing an office's operations.

Obviously, the mode in which an office specification language is used depends in part on the application for which it is being employed. However, in general,

specifications would be written by a trained office systems analyst who has been instructed in the use of the language. This person would not necessarily be a computer expert; he might be a manager, a staff professional, a secretarial or clerical worker, or a specialist dedicated to this task. The analyst must possess two important skills: an understanding of office work, and an ability to analyze and describe office operation in a systematic fashion.

As suggested above, we believe that this language could be used both for prescriptive and descriptive purposes; that is, to describe an existing office operation as well as a new and proposed one. In fact, such uses are often demanded in the context of an evolving office system. An analyst must first construct a description of the system as it is currently configured and use that as a basis for developing specifications of a new and improved system. It is rarely feasible to institute a revolutionary change in the process of an automation effort and to dramatically restructure an entire office operation; rather, the new office system must evolve from the old one. Consequently, at some suitable level of abstraction, the specifications of the new system should be virtually identical to those of the old one. It is only at the level of mechanism and implementation that the two become distinguishable. Thus, it is appropriate that a specification language be multi-tiered, with the topmost levels expressing the implementation-independent structure of the office and only the more detailed levels serving to identify the particular way in which the general structure is being instantiated.

The problems of office systems analysis, specification, and construction discussed above lead us to four major criteria for an office specification language. First, the language should be formal, *i.e.*, have well defined syntax and semantics. It should have a limited vocabulary of constructs that can be combined only in specific ways. The meaning of a function should be clearly expressed by the specification text; a legal specification would have a unique meaning. Thus, the problems of

ambiguity and imprecision encountered in using English descriptions are to a large degree eliminated. Also, the properties of formal languages provide opportunities for various kinds of completeness and consistency checks of specifications, further reducing errors.

Second, the language should be natural and problem-oriented. Both the overall organization of a specification, and the constructs of the language in which it is expressed, should reflect the semantics of offices and office work. That is, the language primitives should correspond to office activities and structures, so that the description of an office procedure will be expressed in terms meaningful to those familiar with the application domain. In this manner, the communications needs of all those involved in the design, implementation, and use of office systems can be met. By incorporating a standard and natural logical structure for office specifications, the language should support the process of performing the analysis and writing the description. The specification of a procedure should be expressed at a level of abstraction corresponding to the purpose of the function, rather than in terms of the low-level task structure used to implement it. The focus should be on what the process does, rather than on the details of how it does it. By embedding this functional orientation in the structure of the language itself, the benefits provided by the holistic approach to office automation discussed above can be realized more efficiently.

Third, the language should be modifiable. Both the well-known arguments about the effects of modularized, modifiable software on the costs of maintenance, and the dynamic nature of office procedures themselves oblige any office specification to be easily changeable. As offices evolve in their goals, resources, and constraints, so too must the procedures evolve to meet unanticipated situations. If the specifications of an office system cannot easily be modified to deal with new requirements, they will soon become obsolete and unused.

Finally, a critical requirement for an office specification language is a hierarchical strcture. We have noted the variety of uses to which such a language might be put, and it is likely that the same level of detail will not be appropriate for different uses. Yet if our approach is correct, the same overall structure, reflecting the inherent structure of office work, should serve as a unifying framework for all applications of the language. Thus, a hierarchical design, in which details can be added in an organized manner to whatever detail is required for the current use of a specification, will be necessary to the effective use of an office specification language.

We believe that a language that meets these criteria would not only provide a meaningful framework with which to approach office automation, but also serve as a major tool for the production of efficient, effective office information systems. The development of a set of design principles, the design of a particular language (which we call the Office Specification Language (OSL)) in conformance to those principles, and the field testing and evaluation of the language, constitute the major portion of this thesis.

We note that restricting an analyst's purview through the mechanism of a formal language and methodology is not without risk. There is an important tradeoff between generality and specificity in analysis tools and methodologies: the more a technique is designed to apply to a specific universe of discourse, the more helpful it is to a user in that field and the more it reduces the demands made upon him. Conversely, a tool that is usable in a variety of applications cannot be particularly efficient for any of them. OSL is therefore not expected to be appropriate for describing all conceivable office procedures. OSL embodies a particular perspective and approach to office work and its description that, we believe, matches a large number of office procedures, although certainly not all of them. A major goal of the field studies described in Chapter 6 was the more precise

determination of the universe of application for which our approach and our particular language is relevant.

We will outline our research effort in more detail shortly; first, however, we turn to an investigation of related work that has served as the basis for, or otherwise influenced, our efforts.

2.5 Related Work

Little work in the area of functional office analysis and specification has been reported. Office analysis techniques in general have two major sources: the process-based approaches that underlie some industrial engineering practice, and data processing systems analysis methods. Both disciplines provide useful insights and techniques, but neither addresses the major needs of office analysis.

There is an extensive literature on "administrative" systems analysis, generally based upon a variation of time and motion studies, *e.g.* [14, 2]. These efforts, however, have generally addressed the lowest level of office tasks: typing, phone-answering, *etc.* As we have strongly suggested, however, a focus on these levels of analysis cannot provide for the efficient specification and implementation of office systems; it leads, rather, to mechanizing the existing ways of doing things. Similarly, PERT/CPM techniques are useful in describing scheduling constraints among various activities, and can be adapted to office procedure descriptions, but must be significantly augmented to satisfy the goals of an overall structure and well defined syntax and semantics that we have identified.

The "Playscript" procedure [34] is an interesting effort in office description that bridges the administrative and computer viewpoints. This approach requires office procedures to be described, in English, in a highly structured format, including

restrictions on sentence style and length, page layout, and general procedure structure. Without naming them as such, Playscript encourages the use of such programming concepts as iteration, case statements, and subroutines. This approach provides a partially structured syntax but lacks the defined semantics necessary to assist the analyst in his tasks, as we have previously described.

There has also been some research on the informal procedures so necessary to the social operations of a workplace (see for example, [50, 54]). While this work points out some significant issues in organizing an office to implement the needed functions, it again does not recognize or address the nature of those functions. The framework that we have developed provides an initial organization to the office application area.

The DP/computer science approach to office systems research is characterized by a search for effective formal models for describing office work. The best example is Xerox' Information Control Net model [8]. While useful for describing the information needs and flow of various procedures, the model lacks any semantics that relate it to office work; it could equally well be used to describe an auto assembly line. This is not a criticism of the model, which itself is an effective means of specification; in fact, it could be used as an intermediate ("compiled") representation of the process-flow component of OSL. Its inadequacies as an office analysis and description tool stem from its low-level approach.

The Office Form Flow Model [30] and the Automated Workflow Control model [5] are similar in their major design premise: that offices should be analyzed in terms of forms or information flowing from one work station to another. Neither addresses the *content* or *purpose* of the information.

The literature on specification languages for office applications might provide

some guidance to previous research and development in this area, though we have found no evidence of such studies *per se*. We first note that our usage of the concept of a specification language differs from that typically employed in the computer science literature [31]. The common usage refers to a general-purpose facility for describing the behavior of individual modules of a large software system. By contrast, our perspective is domain-specific and implementation-independent.

Some earlier work has been done in areas related to our effort, generally in the context of specification languages for conventional data processing applications. For example, the Time Automated Grid [22] and Accurately Defined Systems [32] languages are designed for describing file processing applications; the latter is primarily a documentation tool. The Business Definition Language [15] takes a high-level, nonprocedural approach to the description of highly-structured business DP tasks. The Problem Specification Language [51] is a more general language for defining information system requirements. The System Specification Language, a part of the Protosystem automatic programming project, is designed to handle "a subset of the class of all batch oriented dps's [data processing systems]." [43] Although a number of general principles of system specification can be derived from these languages, their scope is inadequate for the flexible, interactive, and semi-structured nature of the office functions that we are addressing.

Recently, several attempts have been made to design languages specifically for the office domain. Barber and Hewitt [4] are using a form of the Actor formalism [20] to specify the activities of office workers and the communications among them, primarily in an effort to find means of symbolically proving, simulating and modifying procedures. IBM's System for Business Automation Programming Language [60] is based upon the Query-by-example relational database query language [59]. The user programs in a forms-oriented graphical environment; primitives in the language include database access, forms editing and control, and

similar functions. The Xerox PARC Officetalk system [41, 40] uses a similar forms-oriented interface and programming-by-example paradigm; although it attempted to allow the writing of procedural descriptions of forms processing with a small set of filing, communications, and forms editing primitives, the major thrust of the work was in the user interface, and the language aspects of the project seem to be moribund [10].

A major problem with all these languages is that they do not deal directly with office function. They are oriented toward the tasks of individual office workers; in an office implemented using one of these systems, a worker merely does electronically what he currently does on paper. Information flow is described in terms of forms passing among different persons (although in several languages the forms may contain some "intelligence" as to how they are to be handled). As noted above, a description at this level embodies the current task structure, rather than the overall function that those tasks are implementing. As a result, the utility of these languages for high-level specification is limited, and the cost effectiveness of replacing manual systems with electronic ones that require people to perform the same tasks can be difficult to demonstrate.

The Office Procedure Specification Language [57] does deal more directly with office functions. It primarily allows description of documents and communications patterns. Primitives are at the level of document definition and movement and procedure instantiation; more complex processing is expressed by programs written in a general-purpose programming language (APL). The structure and syntax of the language reflect the underlying general representation scheme for asynchronous, concurrent process, the Augmented Petri Net model. The major shortcoming of this approach, which it shares with all of the above office description languages, is that it lacks any constructs at a level higher than "send message" or "file document." Nor do any of these languages result in highly-structured or readable specifications.

The OFFIS "specification language" [6] is an attempt to address the issues of higher-level objects and relationships in the office. OFFIS attempts to use these items as the basis for describing the communications needs of the office. However, the oversimplification of office operations necessary to provide a "compilable" model frustrates the utility of this approach.

As an example of type of high-level constructs missing from previous work, consider the activity SELECT that we have defined in OSL. This is an abstraction that reflects the semantics of a common decision situation in which a specified number of objects must be chosen from a given set. We could specify a particular selection processes in terms of the primitives available in any of these languages, but since selection is an activity common to many offices, it seems reasonable to define a "selection" as a construct of the specification language. Further, in many cases we cannot describe algorithmically the manner in which a selection is to be made (*e.g.*, select a widget vendor from among the 30 in the catalog); nevertheless, by describing the activity as a selection with a certain set of parameters, we have provided, to an analyst, manager, or computer-based system, a great deal of information about how to support the data management, tracking, and communications needs of the person(s) charged with the decision. We have specified, at an appropriate level of abstraction, the function to be performed, and the context in which it is to be carried out. Should an algorithm for making a selection be developed at some later time, it can be implemented without changing the context of the overall specification. Thus, whether the specification is descriptive or prescriptive, the selection decision is a fundamental part of the procedure's goal; the details are relevant only for specific implementations.

2.6 Summary and Research Outline

As we have discussed, our approach to office automation is based upon the thesis that there is a high degree of commonality of structure among functions performed by a wide variety of offices. An analysis of these functions can lead to a better understanding of the problem domain in abstract terms, which understanding can then be exploited in the design of tools and methodologies to assist in the analysis, specification and automation of office procedures. The examination and elucidation of the structure of office functions is the primary goal of this research.

It is important to note that office automation, as we perceive it, is not really a new field in computer science. It is rather an application area that provides a fruitful setting for new developments in systems design, programming languages, and other areas of computer systems research. Contrasts are sometimes drawn between the fields of office automation (particularly in its word processing incarnation) and data processing, with the idea that they are entirely separate problems requiring different technologies for their solution. This philosophy holds that offices are such an unstructured environment that more and better mechanized electronic tools are all that computer science has to offer it. We believe that this attitude is incorrect, that there is a strong analogy between the two fields, and that, in fact, they are really much the same; OA is merely in its first stages of development. For example, the fundamental structure of accounting processes was understood well before the invention of the computer. Early in the history of the development of computer systems, it was recognized that this abstraction of accounting functions could be exploited in terms of algorithms that could execute the functions; the major result was the moving of much of the detail work of accounting (*e.g.*, payroll, accounts receivable), and control of that work, from people to machines. In other words, the understanding of the common, abstract structures of the accounting domain led to the cost-effective *automation* of many accounting functions. Thus was born data processing.

More recently, researchers in management science developed an understanding of the inventory control function, in terms of a simple, but algorithmic, model (the Economic Order Quantity formula). The incorporation of this knowledge of inventory control into computer programs has changed it from a skilled management function into a structured, and therefore automatable, one — one that is now considered another DP function. What we are engaged in is a similar investigation of the office domain, in order to bring an understanding of its structure to bear upon the development of tools for building office systems. We expect that as more of this structure is understood, and as the trend toward the distribution of DP functions to end users continues, the tools, techniques and technologies of OA and DP will look more and more alike. (Indeed, in the fifties and sixties, the term "office automation" was applied to just those functions that we now consider normal data processing, as the first wave of office clerical tasks was subsumed by computer applications programs [52, 21, 9, 45].) Consequently, we believe that the application and extension of several important fields of computer science research, particularly high-level language design and semantic data modeling, to the problems of supporting and enhancing the operations of the office, is a natural path in the continuing development of the uses of computer technology.

Our overall goals are to develop a more fundamental understanding of the office domain; to design, based on that understanding, tools and techniques for the analysis, specification and construction of office information systems; and, more generally, to provide a framework for further research in office automation, Contemporary techniques in the "office automation" field do not address the operations of the office at a sufficiently high level of abstraction to achieve significant gains in operational effectiveness. We believe that the appropriate level is that of functional office procedures, and that these procedures are currently insufficiently understood in the abstract to allow for the cost-effective construction

of systems to automate or support them. Finally, we have argued for the utility of a high-level language for the description, specification and documentation of these procedures, and for an associated methodology to support their analysis.

The research project reported in this thesis consists of several components. First, we have examined a number of existing office case studies and supplemented them by conducting a large set of studies of our own. These cases, plus a study of the related work described in the previous section, led to the formulation of our functional approach to office automation and to the concept of, and requirements for, an office specification language, as previously discussed. We have designed OSL to address these requirements. In so doing, we have developed a concomitant methodology for using the OSL premises and design principles to assist an analyst in understanding an office and constructing a specification. (The structure of OSL and its attendant methodology have also been the genesis of a more extensive research project in the development of a full-scale office analysis methodology, as described in [48] and discussed in Chapter 5.)

We have designed OSL as essentially a working hypothesis of its underlying premises. In order to examine the adequacies of OSL's concepts and of its particular features, specifically with respect to its learnability, usability, and range of applicability, we have engaged in a set of field studies with personnel from several cooperating firms. The results of those studies and the implications for future research in office automation are the major product of this effort.

Chapter Three

The Design of OSL

The overall goal of our OSL design effort has been to develop a language whose structure assists an office systems analyst in constructing specifications that are clear, unambiguous and natural descriptions of office procedures. These specifications should uncover and highlight the basic structure of the procedure rather than focus on the details involved in its implementation. As we have stated, our approach is a functional one. That is, we do not attempt to capture in a specification all of the mechanisms associated with an office procedure — any such "complete" specification would be overwhelming in its size and complexity. In addition, it is unlikely that an office systems analyst (in any finite amount of time) will be able to uncover all possible variations of the procedure. Most important, the implementation details of an office procedure continually evolve as office workers develop new techniques to solve old problems or face previously unencountered difficulties. Consequently, we have not sought to achieve any elusive "completeness." Instead, an OSL description focuses on the purpose of the procedure, rather than on its mechanics. This has been accomplished by including in the language primitives that express the goals of office activities in application terms. In order to achieve this goal, we have sacrificed completeness in another way as well. OSL is not appropriate for describing all conceivable office procedures; it embodies a particular perspective and approach to office work and its description which, we believe, matches a large number of office procedures, although certainly not all of them. In particular, as should become clear in the discussions to follow, OSL is designed to be of most utility in a more- rather than less-structured environment. This is a consequence both of the evolutionary nature of business

processing model development described in Chapter 2 and the common-sense observation that structured tools and techniques are most effective in such an environment. Thus, we expect that OSL and its associated analysis methodology will be most appropriate for those offices whose missions require extensive processing and organized recordkeeping, and involve decision-making at the *operational* level. The Admissions Office and OSP are examples of such situations; purchasing, claims processing, letter of credit, and other such regularly structured operations are also examples of the operational decision support systems that we are addressing. In contrast, we do not believe that offices with essentially strategic missions, such as those of CEO's, corporate staff, or research, contain a sufficient ratio of structured to unstructured operations to be amenable to our approach. The OSL premises and design choices, discussed in this Chapter, reflect this orientation toward operational offices.

In this Chapter we describe the nature of OSL and the motivations for its major concepts. We assume in these discussions a familiarity on the part of the reader with the basic structure and features of the language. To that end, we begin with a discussion of the premises upon which OSL is based, and follow that with a summary of the language. We do not attempt a narrative description of the whole of OSL; a complete language reference manual, including all definitions and syntax description, is provided in Appendix A. A formal grammar of the language is defined in Appendix B.

We illustrate a number of ideas in this Chapter with examples drawn from two MIT offices: the undergraduate admissions procedure of the Admissions Office [29] and the research program administration function of the Office of Sponsored Programs [44]. Annotated examples of OSL specifications from both offices are presented in the following Chapter.

3.1 Premises

OSL can be regarded as an initial experiment in developing a "theory" of office procedure analysis based upon our functional approach to office automation. Several premises thus underlie our design effort, and are critical in understanding OSL.

Our first premise is necessarily that there *is* structure in an office. We look at an office as a *system.* The components of this system include space, equipment, information, and (especially) people. Further, an office as an organizational unit has a particular *mission*, a set of goals defined in terms of the business of which it is a part. [18, 17] The activities of office workers are, or should be, designed to effect that mission. Indeed, the equipment, information, space, and other system components incorporated in an office should be there to help people carry out the office's mission. An office's mission is carried out via one or more *functions*, each of which is based on a simple model of resource management, and is implemented by means of a set of procedures.

Our second premise is that there is a high degree of commonality of structure and activity among procedures in different offices. In other words, we believe that there are fundamental semantic structures in the office application domain that recur in many different contexts. This commonality can be exploited by identifying the structures that are repeatedly used in natural descriptions of different office procedures and embedding them in a formal language. The user of the language will then find that it provides him with just those problem-oriented features that he wishes to use; he will not have to build up a description of an office procedure from lower-level and more general constructs. Consequently, the design of OSL is based on an extensive familiarity with the application environment, both from examination of related work (described in the previous Chapter) and of office case studies performed as part of this research (described in Chapter 6).

Our third premise is that office procedures are basically *simple.* An important conclusion of our analysis is that most office procedures are fundamentally simple processes that are often obscured by implementation details and disorganized exception handling. However, when it is eventually uncovered, the basic structure of the office procedure is often relatively easy to comprehend and describe, given the appropriate set of primitives. The "complexity" of office procedures is often an artifact; a goal of OSL and its related methodology is to manage and even avoid this complexity.

A consequence of the notion of office procedures as essentially simple is that we should be able to derive a reasonably simple model to describe their basic structure. OSL incorporates such a model as the basis of analysis and description, focusing on what we call the "main line" of the procedure. This approach provides a framework of the underlying fundamental structure of the office's activities. It makes it easy for a reader of a specification to get an overview of the procedure and for an analyst to concentrate first on the "big picture." It gives a simple, canonical structure on which to base the investigation and description of the special cases, variations and exceptions that make up the apparent observed complexity of the procedure. Finally, it helps distinguish between the current operation and the higher-level goals that we are attempting to uncover and that will presumably survive a change in implementation.

Our final premise is that paper documents are *not* the fundamental focus of office procedures. Rather, documents are artifacts of the current implementation of the procedures. A document is a device for collecting and transmitting information about a more abstract object that is the true focus of the procedure. For example, the MIT admissions office's admissions procedure deals with *applications.* An application is a complex object that has many attributes, some of which are represented by documents (*e.g.*, a high school transcript), others are themselves more abstract ideas (*e.g.*, a reviewer's evaluation) that are represented *on* paper.

The concept that paper is not basic to office procedures, that indeed an office procedure is something more abstract that can be implemented in a variety of different ways, is a primary distinguishing characteristic or our work. This abstract *object orientation* has several consequences. First, it forces clarification of business goals, by the process of deciding which objects are important. Taking documents for granted as the key to office procedures provides no incentive to think this problem through. Second, objects expose the difference between functions and procedures: functions *manage* objects while procedures *change the state of* objects until they are no longer of interest to the procedure. Third, object orientation enables a clear statement of the main line of the procedure as a sequence of ***desired*** object state changes. Finally, it provides a context for understanding the activities in a procedure.

3.2 Language Overview

The premises discussed above are reflected in OSL's structure, as well as its specific features. A holistic view of office specification is central to the structure of the language. OSL provides canonical high-level office-oriented constructs for the specification of both data and control structures. Such constructs provide a framework for the organization and presentation of a specification and also act as a guide to the analyst in structuring his task. This framework in turn provides for the readability and naturalness of expression necessary for using OSL in documentation and training. Finally, OSL provides a built-in structure for all specifications; the office description as a whole has a standard format, and each of its components can be decomposed in a uniform way.

The remainder of this Chapter elucidates the structure and features of OSL, how they are derived from the fundamental premises of the previous section, and

how they satisfy the requirements for an office procedure specification language described in the previous Chapter. We do not here attempt a narrative description of the full language. Annotated examples are provided in the following Chapter, the language reference manual is included as Appendix A, and the formal grammar is presented in Appendix B. We assume assume a basic familiarity with these appendices; the balance of this section provides a brief overview of the language.

An OSL *specification* is a description, in the OSL formal notation, of the structure and operations of some part of an organization. An office specification in OSL consists of two major components: a description of the application domain with which the office is concerned, and specifications of the procedures performed in the office. The former, called the *environment*, provides a (static) context for the description of the procedures. It effectively expresses a model of the world of the office; it describes the objects on which the office is focused, the organizational context of the office, the documents and forms that the office processes, and the information that it needs to utilize. It establishes the vocabulary for the description of the office's operations, identifying key components of the office's context and their relevant relationships. In the case of the OSP, this contextual information describes a world consisting of proposals, contracts, funding agencies, researchers, laboratory directors, and the like. The description is couched in terms of a variant of the SDM [35], a data modeling mechanism originally developed for describing the information content of databases. The key feature of the SDM is that it models an application rather than data; thus the specification includes a direct description of the office and its environment. This enables the specification to distinguish substance from artifact; a procedure can access information it requires by directly referring to the appropriate attribute of an entity, without caring whether that information is captured on a form, in a database, or elsewhere. This "schema" thus naturally expresses the static semantics of the office in terms with which a reader is likely to be familiar.

The description of the office environment is expressed in terms of entities and their attributes, inter-entity relationships, and entity collections. Associated with the description of an office entity is the definition of those documents related to it (for example, a proposal document is associated with each proposal entity), as well as constraints on the attributes of the entity (for example, that the principal investigator on a contract must be a faculty member). Specialized entity and relationship types (such as people, agreements, schedules, logs, supervision, and the like) are built into the language, since they recur frequently and they possess special semantics. Included in this environmental specification is a description of the offices in the organization and the lines of communication and authority that connect them. Such an organizational description is particularly valuable when a function is realized by means of related procedures executed in different offices. The local office context describes the people in the office, their roles, responsibilities and authority, as well as the files maintained in the office. The environmental description also identifies the primary objects of the office. These are the entities that are the major focus of the office activities and around which the descriptions of the procedures are organized. In the OSP, the primary objects are proposals and contracts; in the admissions procedure, applications.

The dynamics of the office are captured in the *operations* part of the specification. This part describes the procedures — the activities performed in the context, and with the vocabulary, provided by the environment part. Just as an appropriately-designed data model can serve as the basis for natural descriptions of the environment, so too a simple but powerful model of office activities can be applied to procedural specification. As mentioned, OSL incorporates a canonical set of structures for process description that are based on three concepts: an orientation around objects; hierarchically structured and modular descriptions; and an emphasis on the identification of exceptions and other special cases.

In many cases, a set of office procedures fits into the framework of a *function.* A function concerns the management of a *resource* (some class of entities) over time, and is also described using a simple canonical model. This functional overview may incorporate procedures from a number of different physical offices.

3.3 Procedure Specification

The key to the design and use of OSL is its model for and presentation of office procedure specifications; OSL describes office operations in terms of procedures, the organizing structure of office work. A procedure description in OSL indicates *what* is to done and *when* it is to be done, within a standard structure and in terms of a restricted set of activities and events. As we have noted, our assumptions are that most office procedures are fundamentally simple processes, obscured by implementation details, and that many office procedures have a similar underlying structure. The OSL procedure description is structured so as to highlight these basic operations and separate them from the special cases.

The premise, and subsequent observation, that there is some commonality of structure among procedures operating in divers offices leads to the use of a standard *template* for the organization and presentation of all OSL procedure specifications. This template, shown in Figure 3-1, applies to every OSL procedure and is meant to provide both a common presentation structure and a framework for the analyst in pursuing the information required for his description. Beyond the framework for analysis, such a canonical organization serves a research purpose: by encouraging a uniform method of discovery and presentation of a wide range of office procedures, we provide a means of gathering a collection of procedure case studies in a format suitable for further analysis. The availability of such a database is critical to the development of more comprehensive and higher-level models of office operations.

For example, it is necessary to provide a wide range of examples at the procedure level if we are to develop a taxonomy of procedure types; we shall return to this issue in the discussion in Chapter 6 of results obtained in field studies of OSL.

```
Procedure name

Focal object

Responsible

Main line

Timing constraints

Quantitative information

Variations

Exceptions

Details
```

Figure 3-1: Procedure Structure Template

Every procedure in OSL is concerned with a *focal object*, an entity that is the focus of office activities; the procedure as a whole is described in terms of the evolving history of its object and the goals of the various steps in processing it. An OSL procedure is not meant to specify an exact prescription that must be followed, but rather an idealized goal. That is, a procedure represents the history of processing of an object in the case where everything "works correctly." This history is called the main line of the procedure. In "executing" an office procedure, a worker's goal is in a significant sense to make the object end up in a final state *as if* it had followed the main line. (This conceptualization of office procedures, as a set of normative rules whose result is what office workers seek to achieve, has also been proposed independently by Suchman in a similar context. [49]) It is the handling of all the special cases and problems that cause the procedure to deviate from the ideal that is the essence of the semi-structured nature of office procedures. The OSL procedure specification mechanism provides a structure in which to represent various levels of deviations and details, all based upon the specified main line.

Each of the entries in the procedure template is derived from the need to express a particular kind of information about the procedure; each of our key premises is reflected in one or more sections of the procedure structure. The "focal object" section identifies the object around which the procedure is oriented. The main line of the procedure is described in its own section. The principles of hierarchical decomposition and separation of special cases from "normal" processing result in individual sections for describing processing variations, exception handling, and lower-level details of processing steps. Finally, sections titled "Procedure name," "Responsible," "Timing constraints," and "Quantitative information" reflect a need to provide information *about* the procedure as well as a specification of its processing requirements. Each of these sections is described in further detail in the following.

3.3.1 Focal Object

The focal object of the procedure, as we have noted, is some entity that may or may not have a tangible embodiment. One of our fundamental premises is that forms, or more generally, documents, are not in and of themselves the foci of office procedures and that therefore office procedure analysis should not focus on forms management or on counting, cataloging, and following paper around the office. OSL enforces this rule in two important ways by requiring that the focal object of a procedure not be a document. It may, and in general will, have one or more documents associated with it; but those documents *per se* are not the critical factors in analysis or specification. (As described later in this Chapter, the syntax of document definition in OSL enforces the notion of documents as artifacts by requiring the identification of the abstract object to which the document refers.) Examples of focal objects include the **application** object in the admissions office and the **contract** in the OSP. The former is a *record*, an abstraction that represents a

certain set of information about the applicant, provided by the applicant, his high school, his references, the College Board, the admissions interviewer, and the reviewers. The OSP contract is an *agreement*, an entity most often represented by the document that defines it. Each object, like all other entities referenced in the operational part of a specification, is described and characterized in the environment part of the specification.

3.3.2 Responsible

An important consequence of the semi-structured nature of office procedures (as opposed, for example, to rather structured accounting and other DP procedures) is the requirement of frequent human intervention for decisions. Our functional approach to office systems analysis emphasizes the examination of a procedure as fulfilling some business goal, and which therefore should be under the purview of some individual. While others may carry out the procedure, the "responsible" party is both in charge of the overall operation, and the reference point for handling any exceptions beyond the competence of the staff. The "responsible" entry in the procedure template is used to name the role that is responsible for the procedure's operation, and to which any otherwise-specified exceptions will be referred. (This entry is by definition an OSL *role* rather than a *person*; see the Section on the environment part of an OSL specification for a discussion of this distinction.) For example, the Director of Admissions is responsible for the admissions procedure, though he is not directly involved in most of the day-to-day operations and decisions involved in the admissions procedure; rather, he designs and oversees it.

3.3.3 The main line procedure model

The main line of the procedure, separate from any special cases and implementation details, is a key construct in our approach to office analysis and

specification. Therefore the facilities in OSL for expressing the requirements of the main line of office procedures are critical to its utility. The nature of these facilities is derived from the perspective of our (previously described) object orientation, whereby a procedure's processing specification is expressed as a series of desired changes in the *state* of its focal object from an initial to a final, desired state. We define the necessary processing to transform the object from one state to the next as a *step.*

The paradigm of office activities from which the OSL procedure model is derived is therefore as follows. An object is initially in a quiescent state. Some event then requires the office's personnel to transform the state of the object to some other, desired state. If this desired final state cannot be achieved through immediate processing, the object is transformed to an intermediate state until further processing can be accomplished. The procedure terminates upon the object's reaching a final state, at which point in time the response to the initial, invoking, event has been completed.

An OSL procedure thus expresses the progress of its focal object through a succession of states. States are stages in the execution of the procedure at which no further processing can be done until the occurrence of some *event*, an autonomous occurrence that is beyond the control of the office. That is, the object "waits" in a state until some specified event requires or allows some processing to be accomplished. Example states in the admissions office include **complete**, **reviewed**, **admitted**, and **accepted**. All procedures start in (the quiescent) state **null** and end in (the quiescent) state **done**. In order that an object reach a state (recall that the goal of a procedure is to move its object to the final **(done)** state), it must have suffered the processing specified in the steps preceding that state.

When the procedure is in a given state, it is waiting for the occurrence of one of

a set of expected events; when of these events occurs, a corresponding *step* is executed, at the conclusion of which the procedure is left in some other state. During the execution of a step, the object is considered to be between states.

A state machine-like formalism is appropriate for expressing these state/event/step relationships and determining the overall control structure of the procedure. The OSL model, following our office procedure paradigm, is as follows. The object is in some (initially **null**) state; some expected event occurs, causing an associated step to be executed; completion of the execution of the step causes the object to enter another state; if this new state is **done** the procedure is complete, otherwise the object "waits" in the new state for some designated event. For example, the following is an informal representation of part of the admissions procedure's main line, illustrating states, events, and consequent steps:

```
State:  complete
Event:  January
Step:  Review applications; end in state reviewed

State: reviewed
Event: February
Step:  Choose freshman class; end in state admitted

State:  admitted
Event:  Receive acceptance
Step:  Update files; end in state accepted
```

The model illustrated by this example provides a basic framework for procedure descriptions, and is fundamental to OSL's structure and analysis methodology. However, the descriptions are in English; they lack the precision that we require. We therefore need to address in more detail the specification of both events and the processing of objects represented by steps, in order to extend the model to include formal representations of both types of information.

3.3.4 Events

As we have seen, office procedures depend critically upon events, occurrences over which the office responsible for a procedure has no direct control. We can categorize relevant office events, and thereby identify the important aspects of an event model, by examining the *ways* in which they are uncontrollable. We find three major categories of such uncontrollability of events: those dependent upon real time; those dependent upon a specific change in the office's environment; and those dependent upon a specific action (or inaction) of some party external to the office. These categories give rise to the following types of events, each of which is defined as an OSL construct with syntax as in the examples.

3.3.4.1 Events based on real time

A basic *time* event is a specific date and/or time. It may be explicit, for example,

```
January 1
```

or relative to some other event:

```
Event 2 + 1 week
```

3.3.4.2 Events based on environment changes

An *environment* event is the raising of a particular condition in the office's environment due to the change in status of some entity. Such a change may be specific to one attribute or merely reflect any change in state. For example,

```
When APPLICATION is updated
```

Some environment-related events are concerned with the state of processing of a procedure, rather than the state of some entity. An *activity* event reflects such changes caused by the initiation or completion of a specified activity in some step of the procedure. For example,

```
Complete 2.3
```

In some cases the environment state that triggers a procedure or a step may not be definable *a priori*; that is, the determination of the appropriate conditions is left to the judgment of some decision-maker. A *trigger* event is defined in OSL as an explicit command of a designated person or role. It indicates that the next processing step is to occur only when determined by the designate, independent of any other occurrence. For example,

```
By Director
```

3.3.4.3 Events based on external parties

Often processing of an object must be suspended pending the arrival of information from an outside party. Because the relevant events are in fact the *arrival* at the office of such information, these occurrences may be characterized as *communication* events. A communication event may be further categorized as representing one of three types of conditions:

- It may be the receipt of a specified communications entity (often a document). The specification may be absolute (in terms of the document identification or other parameters), or relative (related to some local information, *e.g.*, matching a local entity in two or more attributes). For example,

```
Receive Preliminary
```

- It may be the receipt of a response to a specified message. For example,

```
Receive response to SSGR
```

- It may be the non-receipt, after a specified time interval, of a specified communication or response. For example,

```
Not receive response to SSGR after 2 months
```

3.3.5 Step Specifications

Steps represent the actual processing required to move the object to its next state. The nature of the specification of this processing is another important aspect of the design and utility of OSL. The design of the OSL step specification semantics and syntax is an attempt to answer two fundamental questions about the nature of office systems analysis and specification: What are the basic operations of office work, and how can these operations be usefully described?

Our answers to both questions are derived from our functional approach to office automation and our object-oriented procedure model. From this perspective, we find that there are three categories of actions that must be expressed:

1. routine manipulations of objects. These are primarily actions concerned with the existence and properties of objects, and include the following operations:

    - creation of a new object

    - removal of an object from the environment

    - changing of some characteristic of an object

    - transmitting an object to another site

    - placing a record of an object in a long-term archive

    - adding an object to a set

    - removing an object from a set

2. actions that require some decision. These, primarily judgment-based, actions can be further subdivided into

    a. basic decisions, concerned with whether a single object satisfies certain conditions, including:

- verifying the correctness of an object

- approving the creation/issuance of an object (by a person with authority to do so)

- evaluating an object and recording a judgment

- negotiating the characteristics of an object among several parties

b. aggregate decisions, those made simultaneously about several objects of the same type, due to the interdependence of the decisions about the individual objects: selecting a subset of a set of objects

allocating objects among several consumers

partitioning a set of objects into several groups]

3. actions that concern the control of procedures and the handling of exception conditions:

- initiating a procedure

- terminating a procedure

- calling a (subroutine) procedure

- restarting an action that caused an exception

- notifying a party responsible for handling an exception

It is our thesis that these actions form a complete set of operations for manipulating office objects. Each of these operations therefore serves as the basis of one or more OSL *activities*. Activities are the fundamental units of office work and are the basic active constructs of OSL; a step consists of a structured set of activities that specify the required processing. The OSL activity set is designed to be precise

and limited. It is based upon the types of actions described above and the class/entity model of objects used in the environment description (see the section below and Appendix A); it thus includes the following types of activities:

Basic	concerned with the existence and state of entities and classes of entities; several activities (SET, CALCULATE, REVISE) are defined to deal with different reasons for and semantics of changing entity states
Decision	concerned with decisions about entity instances; the effect of such an activity is to set the value of a specially designated attribute of the entity to which it is applied
Aggregate	concerned with decisions about the manipulation of subsets of entity instances
Control	concerned with control of the procedure

The OSL activity set is meant to provide a complete vocabulary for describing office processing. Each activity has a well-defined meaning (see Appendix A for definitions) and no other terminology is used in the main line to describe the processing operations of a step. Note that while an activity is a semantically-meaningful process; it may or may not be a structured one. For example, the activity GROUP indicates a subsetting decision. There are cases in which an algorithm is applicable (*e.g.*, group applicants according to age) while others may be inherently a judgmental process (*e.g.*, group applicants into accepted, rejected, waitlisted). We characterize both operations as grouping, and leave the details of particular implementations to lower levels of description (see the section on Details).

Activities define particular types of manipulations of entities or sets of entity instances. The full syntax and semantics of activity specification follow from the requirement that the language have a natural, and therefore easy to use, construction. Therefore OSL activity specification is modeled on a sentence

structure, in which the activity itself is the "verb." The sentence is completed by specifying the "subject," the actor responsible for performing the activity; and the "object," the entity (or entities) upon which the action is performed.

The subject is, by default, the role designated as responsible for the procedure, but any party defined in the environment may be specified as the subject of a particular activity. The object is, by default, the focal object (or the set of objects represented by the class designated as the focal object) of the procedure. However, since any entity may be the object of a particular activity, several "modifiers" are necessary to permit the identification of a particular instance or set of instances of the object class. OSL therefore includes syntax for specifying "any," "all," "first," and "last" instance of a given class, as well as facilities for defining an instance in terms of attributes matching those of the focal object.

As we have noted, the main line of the procedure represents the "normal" course of events, the (possibly rare) case in which everything works correctly. Activities then correspond to *milestones* in performing a step; specified activities must have been performed before the step is considered complete. Within the step, the various activities are organized with a control structure that allows both serial and parallel execution, as well as branching. The ordering of activities is again meant to indicate *suggestions* or *goals*; the specified processing path is the normal one, not the required one. The branching constructs also allow for alternative processing depending upon the result of a decision activity.

3.3.6 Timing Constraints

A *timing constraint* is a temporal relationship between two events in a procedure. For example, it may be necessary that some activity be completed by the end of a quarter, or that an event occur within one week of another event. A timing

constraint therefore expresses an important piece of information about the operation of the procedure, and a violation of such a constraint is an important form of exception (see below).

OSL provides both a separate section in the procedure structure and a syntax for expressing timing constraints. Using events defined in the main line, a timing constraint may be absolute or relative; that is, it may state that an event must occur before, at, or after a specific date/time or one defined relative to another event.

3.3.7 Quantitative Information

Procedure statistics provide a set of figures for various timings and counts in the current implementation of the procedure. While the need for and the specifics of the numbers required depends heavily upon the use of the specification, the OSL standard structure provides a section for such data as is commonly gathered in existing case studies. This information includes: average total time an object spends in the procedure, the number of objects active at any time, the frequencies of various exceptions, and the probabilities at various branch points.

3.3.8 Beyond the Main Line

A major aspect of our approach is the focus on the main line of the procedure as the basis for analysis and description. That is, we believe that an understanding and elucidation of the main line is a prerequisite to effective office procedure analysis. The OSL structure and analysis methodology have been designed to support that view. However, once having developed a model for analyzing, and a syntax for describing, the main line, we are then faced with the question of how to deal with the myriad "non-main-line" details that represent the complexity of real office procedures. In order to do this, we need to look again at just what the main

line model is meant to represent, and how it (deliberately) fails to capture the totality of the procedure.

We first recall that the main line is defined as the history of the focal object in the "normal" case, and in which no problems arise. Therefore, we need to provide not only for "unusual" object histories but also for problem situations; the former are called *variations* in OSL, the latter *exceptions.* Further, we have stated that the purpose of an OSL-based office analysis and description effort is *not* a detailed account of the current implementation of the office, nor a complete specification suitable for defining to a programmer a system design. Nevertheless, for either of these purposes, or for others for which an OSL specification may be used, there will be a need for some additional level of relevant explication of the procedure. OSL provides a section in the procedure template for this next lower level in the specification hierarchy (after functions, procedures, steps, and activities).

The main line describes the normal history of processing for focal objects of the class defined in the procedure specification. In reality, however, not all instances of the focal object's class are identical, and often different instances cannot be treated exactly alike. There are several reasons for which processing of individual instances of the focal object class may differ; we have identified three key types of processing variability:

1. decision-based

2. event-based

3. object-based

The first represents some decision about the object made in the course of the (main line) procedure. For example, in the admissions office procedure some applications (focal objects) are accepted and some are rejected; clearly the actions

required after that decision vary with the decision. Such processing requirements, which are dependent upon the result of a decision made in the main line, are properly alternative histories of the focal object in the main line. These decision-based alternatives are provided for in the OSL formalism by branching constructs in the main line state diagram.

The second kind of process variability is based upon the occurence of events, which are by definition outside the office's control. In cases where multiple events may lead to transitions from a given state, consequently triggering different steps, the variability is inherent in the procedure specification, and is therefore represented in the main line using the normal state/event/step mechanism.

3.3.8.1 Variations

The third reason for differences in histories of object instances is the existence of some inherent characteristic of the instances that require alternative processing. OSL *variations* are such *anticipated* deviations from the main line state/event/step sequence that are based upon an *a priori* attribute of the focal object. For example, the Admissions Office handles both foreign applications and those from students requesting "early action" somewhat differently from each other and from the normal, main line, processing. The former are routed through the Foreign Students' Office before being reviewed and are grouped into admitted/rejected subsets a week after the main line; the latter are reviewed and grouped early. In both cases, the required processing for each application is evident upon its receipt; the main line for that particular instance is different from the norm, but equally well defined.

In keeping with OSL's goals of modularity, control of complexity, and focus on the main line, variations are described as modifications to the main line, and are placed in a separate section of the specification. The specification of a variation uses

the same syntax as the main line, and in addition includes the attribute values that define when the variation is to be used. The variation itself is defined as a set of states, events, and steps that are to be added to the main line, and/or events and steps that are to replace existing ones. The semantics of a variation specification indicate that the result of applying these amendments is a new state diagram describing the main line processing requirements for the specified subset of focal objects.

3.3.8.2 Exceptions

Branches in the main line and variations are types of normally anticipated procedure requirements. An *Exception* is an abnormal condition that prevents some part of the procedure from being accomplished. A key aspect of OSL is its approach to exceptions, which has three major facets: the separation of exception specifications from the main line; the exception specification schema that associates exceptions with specific activities and events as well as with particular aspects of the procedure as a whole; and the predefined set of exception conditions.

As we have noted, special cases and exceptions are often the source of the perceived complexity of office procedures; by organizing the description of exceptions, we provide a means of making the overall specification more organized and readable. OSL takes a hierarchical approach to the specification of exceptions. Each level of procedure and activity description has an associated list of exceptions. These are classified in terms of the nature of the event that gives rise to them; instances include violation of timing constraints, waiting for an event that doesn't happen, invalid data values, unavailable personnel, and activity-specific errors (*e.g.*, an inadequate set of resources from which to make a stipulated selection). OSL provides a vocabulary of standardized exceptions both for analytic and descriptive purposes, and, as discussed earlier, also provides several activities (*e.g.*, NOTIFY,

RETRIGGER, REPEAT) for use in describing responses to exceptions. The descriptions of the exceptional situations and the responses to them are separate from the main line to maintain modularity and manage the complexity of the specification.

There are three types of predefined exceptions in OSL: timing constraint violations (discussed earlier), activity-specific, and general. Each OSL activity has associated with it a set of exceptions, each of which identifies a potential problem specific to it. For example, the SELECT activity, which indicates that a subset is to be created from a set of entities, has associated with it an exception, called "insufficient," which occurs when not enough entities are available to form the needed subset. Thus exceptions that might reasonably be expected to occur occasionally can be anticipated by the analyst and, if special processing is necessary, described in an appropriate place in the OSL specification.

General exceptions apply to the procedure as a whole, rather than particular activities or events. These exceptions include "missing personnel," "lost document," "backout," and "cancellation." Again, these are anticipatible occurrences that would require specific attention by those responsible for the procedure.

All predefined exceptions have associated with them a default response, "notify the responsible person and wait." In an OSL specification, it is necessary to describe exception handling only when a non-default action is necessary. In such cases, the syntax and vocabulary for describing the response is identical to that used for describing the main line procedure.

Of course, the nature of exceptions renders it impossible to anticipate all possible problems with any procedure. Therefore, the Exception section of the OSL procedure template is used for the specification of *ad hoc* exceptions, those that are

identified by office workers or anticipated by an analysis. Again, the standard vocabulary and state/event/step paradigm are used to defined the exception and the required response to it, if other than DEFAULT.

3.3.8.3 Details

As we have discussed, an OSL specification *per se* is not meant to be a complete description for all purposes. Rather, it is meant to describe *what* is to be done, in terms of the "lowest level" activity constructs; the control structure itself represents goals rather than firm requirements.

Not everything about a procedure can (or should) be expressed in the OSL formalism. In most cases, however, additional information relevant to the intended use of the specification must be provided to the user. Rather than allow arbitrary annotation of a specification, the design of OSL allows such annotation consistent with its hierarchical approach to procedure description. The Detail section of the procedure template is used for more explicit comments about the activities, and each comment is keyed to a specific activity. Examples of information that might be provided concerning activities include algorithms for implementing them, policies used in carrying them out, names of people with particularly useful talents. That is, Details include anything interesting the analyst has to say about how an activity is performed in a specific implementation.

3.4 Functions

OSL is primarily a language for describing office procedures. However, we might also ask how procedures, which deal with the transformation of entities from one state to another, fit into a higher-level context. Our functional approach to office automation, as described in Chapter 2, is the overall framework in which we

wish to approach procedure analysis and specification. Therefore, the concept of an office function is realized in our formalism as the top-level dynamic structure in OSL.

Recall that when a procedure is invoked, its focal object is in a "quiescent" state. By gathering all the procedures that deal with a particular class of entities as their focal objects into one overall construct, we can define a context in which, in general, the objects exist in a quiescent state until acted upon by one of the procedures. So long as a procedure merely transforms the state of the object, the object returns to a quiescent state upon termination of the procedure. Only when a procedure calls for the permanent disposal of its object is an object removed from the system. A set of procedures concerned with the management of a particular class of entities over time is thus defined as an OSL *function.* The purpose of such a function is the management of all instances of that class and information about them. The entity instances are called *resources* in this context. The actions involved in this management task include monitoring anticipated events and responding to unanticipated ones. The major function of the OSP, for example, is the management of sponsored research; the resources in this case are sponsored research programs.

Following this definition, OSL incorporates a simple model for functions that divides the life cycle of a resource into three phases. First is an *initiation* phase, in which the resource is created or is initially brought under the purview of the function. The major, or *management,* phase of the function involves controlling the resource, principally by reacting to external events and satisfying certain predefined requirements. Such requirements include information about the status of the resource that may need to be provided in the form of regularly scheduled reports; regular reports required or expected from others; and other non-regular but expected events. Finally, the *terminating* phase disposes of an instance of the

resource that is of no further interest. In general, a function deals with multiple individual instances, each of which is in some state of a procedure, or in a "quiescent" state in which nothing need occur.

All activity involved in a function is described by procedures; the function provides the framework that indicates when the procedures are invoked. Typically, several procedures are associated with a single function; at the same time, a procedure may be associated with several functions. In fact, a single office may be responsible for an entire function, for several functions, or for parts of one or more functions. For example, the undergraduate admissions procedure represents merely the initiating procedure of a function that can be described as "manage students"; the admissions office executes this procedure, but the Registrar's Office is responsible for most of the function, terminating when a student graduates or otherwise leaves the university.

This *functional* approach to office systems analysis provides a unifying framework for, and a mechanism for transcending the sometimes artificial barriers between, the related activities of multiple offices. By addressing office functions rather than offices *per se*, the analyst is led to a more complete picture of the overall goals of various procedures and the reasons behind their current implementation. This approach is useful not only in forming an understanding of an organization's operations, but also in addressing the rationalization of procedures and the redesign of work in order to achieve more effectively the organization's mission [17].

OSL uses a template form for organizing a function specification. Each of the entries in the template is used to define, with a standard syntax (see Appendix A), a particular information component of the OSL function model:

FUNCTION <name>

Resource: name of class (defined in environment description)

Responsible: name of role

Initialization: the event that initiates action for an object, and the name of the procedure that is thereby invoked

Required reports received:
a list of entries, each of which indicates how often the report is expected, the name of the procedure invoked when received, and the name of the procedure invoked when not received on time

Required reports generated:
a list of entries, each of which includes the report name (as defined in the environment section) and the required interval, as well as the name of the procedure invoked to generate the report

Other events: list of anticipated-event/invoked-procedure entries, each indicating an event requiring some processing relevant to the resource, but for which no particular scheduling is known

Termination: the event causing termination of the resource, and the name of the procedure thereby invoked

Quantitative information:
a section used to describe relevant quantitative information: numbers of resource instances, frequency of events, time extent of procedures, etc.

3.5 The Office Environment Model

As we have noted, the operational part of an OSL specification describes the dynamics of the office's activities. As we have seen, the procedure and function specifications make extensive reference to entities and classes in the environment. We therefore need to be concerned with the development of effective means of

describing those items in the environment part of the specification. An OSL environment is a description of the static structure of the office. It describes the things the office deals with, both within and outside of its control. It is the environment description that provides the vocabulary for the objects of the operational part of the specification. In essence, the environment is a "model" of the office, or the relevant entities therein and the relationships among them.

Techniques for application modeling of information environments have been the subject of research and development for at least two decades in the fields of database management (data models) and artificial intelligence (semantic representation). We therefore turn to these areas for the development of the OSL environment model; we are looking for a modeling mechanism that satisfies the following requirements, derived in Chapter 2:

- It should be application-oriented; the model should be based upon office constructs rather than computer or other formal language concepts.

- It should be well defined. The syntax and semantics should be clear and unambiguous, and a given model should have a unique interpretation.

- It should be sufficiently rich to express effectively a wide variety of office environments, but sufficiently limited that its structure and constructs provide some inherent guidance to the analyst in its use.

We have drawn the ideas and structure of the OSL environment modeling facility from the SDM, a high-level semantic data modeling technique developed for database systems [35]. It was designed as a general application modeling mechanism that answers most of the needs that we have defined. We have therefore adopted its important features and made several changes to adapt it to the office applications with which we are concerned. (Specifically, we have modified the SDM by adding the notion of built-in entity types and associated classes; we have also eliminated the

concept of events as separate semantic objects and have streamlined the model in other minor ways. Descriptions and syntax rules of the model are found in full in the Appendices.) The remainder of this Chapter discusses the derivations of the major features of the OSL environment model (henceforth "OEM").

3.5.1 Entities

Inasmuch as our approach is heavily object-oriented, the description of objects is a key aspect of the modeling facilities. To avoid confusion with the "focal objects" of procedures, and following SDM terminology, objects are called *entities* in this context. An OSL environment is a collection of entities, the physical or intangible things ("nouns") in the office's world. An entity is anything, whether concrete (*e.g.*, an employee, a document, a widget, an order of Peking ravioli) or abstract (*e.g.*, a program, an account, a job) that is used, manipulated, referred to, or otherwise relevant to, the area, its people, or its operations. An entity is some *real* thing; it need not correspond directly to a document or an entry in a file.

3.5.2 Classes

Entities are organized into collections called *classes*. A class is a named, homogeneous collection of entities of a single *entity type*, (*e.g.*, a class of documents, a class of grade reports, a class of high schools, a class of applications, a class of applications from foreign students). The entities that make up a class are its *members*. Any particular entity is an *instance* of its type.

The OEM formalism is designed both with two related goals in mind. As we have discussed, the OSL structure and vocabulary constitutes a basic, generic model of office environments and operations. It is this model that provides the framework for analysis and specification that is one of the major purposes of this work. Within

these guidelines, the model must also provide facilities for the analyst to construct a model of his specific office from the "raw materials" of the language. These two requirements result in two kinds of classes: those that are built in to OSL, and those that are defined by the user of OSL to describe his particular environment.

A *built-in class* is an implicit class defined by, and consisting of all entities of, a built-in entity type. As the name implies, each of these classes is part of the definition of OSL, and they are not declared as part of any specification. A built-in class name may be used, however, in any place in an environment specification in which a class name is called for; in particular, they serve as the parent classes for the definition of many derived classes. (For example, the built-in entity type EMPLOYEE defines the built-in class EMPLOYEES.)

The classes built in to OSL encompass a range of very generic office entities: employees, organizational units, documents, roles, *etc.* They were chosen so as to include a broad base of commonplace items that provide a starting point for analysis, consistent with OSL's overall approach, without being overly restrictive. The analyst uses the built-in classes as the base for constructing classes specific to his application. Such a *derived class* is defined in terms of some other class(es) in the environment. Thus a member of a derived class is also a member of one or more other classes, including one built-in class; this built-in class defines the type of the member. For example, the class ADMISSIONS-STAFF describes a set of entities that is a restriction (see below) of the class MIT-EMPLOYEES. Each member of ADMISSIONS-STAFF is a an entity of type EMPLOYEE, and a member of class MIT-EMPLOYEES. The class MIT-EMPLOYEES is also a derived class, defined in terms of the built-in class INTERNAL-EMPLOYEES.

3.5.3 Attributes

A key aspect of the SDM formulation, which we have adapted for use in the OEM, is that every entity has a set of *attributes* that describe its characteristics and relate it to other entities in the environment. An attribute is some characteristic of an entity; for each entity, there is some specific *value* for each of its attributes. The value of an attribute is either some entity in the environment, or some set of such entities. (For example, consider a particular member of class MIT-EMPLOYEES; each attribute will have a particular value, such as Name = "Paul Gray" (a member of the class NAMES), Rank = "President" (a member of the class RANKS), Office = "3-208" (a member of the class MIT-OFFICES).

Classes may also have attributes, describing characteristics of the class as a whole, rather than those of individual members. For example, the number of members currently in a class is an attribute of the class itself, not of any of its members.

The possible values of an attribute may be described simply by specifying the class from which its value is to be drawn; this is a *primitive attribute.* Alternatively, an attribute's set of possible values may depend directly, and by a specified rule, upon its relationship to something else in the environment; such a rule defines a *derived attribute.*

A built-in entity type includes in its definition a set of attributes (including, of course, their value classes) that provide a means for characterizing particular entities of that type. As described above, the type defines a built-in class whose members consist of all entities of that type. For example, the built-in entity type EMPLOYEE defines the built-in class EMPLOYEES. Each EMPLOYEE entity includes such attributes as "Name," which is an entity of type NAME describing the name of the employee; and "Supervisor," an entity of type EMPLOYEE that describes the

employee's supervisor. Each entity of a given type has at least those attributes; when used in defining a built-in class, any additional attributes specific to, and characteristic of, the members of the built-in class are added.

One of OSL's primary premises — that documents are artifacts and not the fundamental objects of office work, and that therefore they should not be the focus of office analysis and description — is enforced through the mechanism of built-in entity attributes. For example, DOCUMENTS is a built-in class, one of whose (required) attributes is "Refers"; the analyst, in defining a document, must identify the abstract object about which it is carrying information. He is thus forced to confront the document's nature as artifact and to delve into the office's operations in with a more abstract, functional, view.

In the definition of a class, each attribute of the members is described in terms of its *value class*, that is, the class from which its values can be drawn. (For example, the attribute "Supervisor" of the class MIT-EMPLOYEES has the value class MIT-EMPLOYEES; thus the value of the "Supervisor" attribute of an MIT-EMPLOYEE must be a member of the class MIT-EMPLOYEES.)

Again, the attributes of the built-in classes provide a base set of structures from which the analyst builds his application model. The mechanisms for expressing the characteristics of a particular office environment include the definition of new attributes to further characterize derived classes, and the definition of relationships among classes explicit in the attribute's value class.

3.5.4 Entity instances

It is sometimes the case that we wish to define as part of the environment not a class of entities but an *instance* of a given entity type. While most entity instances

are created dynamically (as part of a procedure, described in the operational part of a specification), in some cases an instance is in fact a part of the static environment. A common example is the definition of the office of interest, which is generally a member of a class of offices or organizational units belonging to that organization.

OSL provides several methods of defining instances. The most important is that of *hierarchies.* There exist in organizations several formal hierarchies, as well as numerous informal ones; the two major formal hierarchies are built in to OSL. The *organizational hierarchy* indicates how the reporting relationships of the various organizational units are structured. The hierarchy is described as a table of instances of internal organizational units, giving values for the attributes NAME, PARENT and SUPERVISOR of each. In this manner, a linear representation of an organizational chart can be built. The *personnel hierarchy* indicates the supervisory relationships among people in the organization. This hierarchy is described as a table of instances of "roles" (a built-in entity type describing a set of job responsibilities; see Appendix A), giving values for the attributes NAME, OFFICE, REPORTS-TO and CURRENT-HOLDER of each. Such a hierarchy provides information that might be useful for tracing responsibilities or locating substitutes for absent workers. Note that these two built-in hierarchies are related to each other through the ORGANIZATIONAL-UNIT attribute of the personnel hierarchy and the SUPERVISOR attribute of the structural hierarchy.

3.5.5 Using the modeling facilities

The mechanisms of OSL environment specification provide for the definition of relationships among entities. Built-in classes indicate how entities may be members of several related classes. Attributes directly relate entities to each other; since the value of an attribute is an entity (or class of entities), an explicit relationship is indicated. In particular, the several mechanisms available for

expressing the derivation of (derived) attribute values provide a rich set of models for inter-entity relationships.

The OSL environment modeling facility allows for the description of a wide variety of office situations. It provides the mechanism for specifying sufficient detail about the application to build a usable context in which the operational specifications can be interpreted. The detailed rules for defining classes and their relationships are found in the reference manual, Appendix A. Chapter 5 includes some guidelines for use of the facilities in constructing environment descriptions.

3.6 Environment Structure

For purposes of clarity and reference, there is a standard structure to an OSL environment specification, encompassing an "identifications" part and a "definitions" part. The identifications part is a summary of the environment; an entry in this section (except for special cases such as in the organizational context and the second and third subsections of the internal context) is simply the name of the entity class and the built-in class of which it is a restriction, *e.g.*,

```
MIT-EMPLOYEE is INTERNAL-EMPLOYEE
```

The identifications section is divided into three subparts, each of which has a different role in the environment definition. The first, called the "organizational context," identifies the relevant aspects of the organization of which the office is a part. The definition of the organization itself (an instance of type "ORGANIZATION") and the organization and personnel hierarchies (or equivalent organization charts) are included here. This section, if complete, would be identical for each office in the organization; thus it might be shared with other office specifications and ultimately located elsewhere rather than replicated for each office. Thus this information is placed in its own section to enhance sharing and severability.

The other two sections of the identification part serve to distinguish between that part of the world over which the office has some control (the "internal context") and that with which it must interact but over which it has no control (the "external context"). This distinction is critical in the definition of steps and events (*cf.*) in the procedure specifications. Thus the external context section identifies the entities external to the organization that are relevant to the office being described. The internal context identifies all entities of consequence within the office. The internal context is further divided into subsections dealing with people and roles; documents and other communications; and names. The entries in each section or subsection are listed alphabetically.

The definitions part of the environment provides the full OSL definition of the attributes of each class identified in the Identification section. Definitions of all classes, regardless the section with which they are identified, are listed alphabetically.

Chapter Four

An OSL Procedure Specification

In this Chapter we present an annotated example of a description of an office procedure using OSL. The example chosen is the undergraduate admissions procedure of the MIT Admissions Office, an English writeup of which is found in Appendix C. We also provide an example of an OSL function specification, using the sponsored research function from MIT's Office of Sponsored Programs [44]. Our purpose is not to detail every feature of OSL (which can be found in the OSL reference manual in Appendix A) or to provide a line-by-line narrative of the example, but rather to illustrate the nature and use of the language by examining an OSL representation of a real office procedure.

An OSL specification includes operations and environment parts. In general, to understand a specification one reads the operations part, referring to the environment for descriptions of entities as needed. The environment thus serves as a "dictionary," defining the nouns used in the operational description in terms of the OSL built-in entities and application building facilities (which are assumed to be understood by the reader). The design of the environment description, as discussed in Chapter 3, facilitates this usage by placing all definitions in a separate part of the environment, listed alphabetically. We will not, therefore, point out each class definition in the environment as it is encountered in the operational part of the specification.

[A note on syntax: Following the convention defined in the language reference manual (Appendix A), class names in the examples are in all upper case, attribute names are in lower case with initial capitals.]

4.1 An Example Procedure

We begin the examination of the admissions procedure in Figure 4-1 by noting that its name is "Admit-Freshman-Class" and its focal object is an "APPLICATION." Examining the entry for APPLICATION in the environment, we find that it is a class of type RECORD (that is, an abstraction representing an organized set of information about some object, in this case an APPLICANT) with a long list of attributes. Some of these attributes are documents that form part of the visible "application," while others are information added by various admissions personnel in their evaluation, review, and decision process.

The main line of the procedure (Figures 4-1 and 4-2) consists of states **null**, **waiting**, **complete**, **reviewed**, **admitted**, **accepted**, **coming**, and **done**. The procedure is invoked (Step 1 entered from state **null**) when Event 1 occurs: a Preliminary application card is received. ("Preliminary" is the name of an attribute (of the focal object APPLICATION, by default) whose value is defined as a DOCUMENT of the class PRELIMINARY-APPLICATION-CARD (defined in Figure 4-8).) Step 1 specifies that initial processing includes verification of the preliminary application, creation of an APPLICATION object, selection of an interviewer, and transmission of several documents. Note that whereas the interview report form is sent explicitly to the interviewer ("Interviewer" is an attribute of APPLICATION that has as its value a member of the class EDUCATIONAL-COUNCIL, a person with a name and address), the Final application forms are sent, by default, to the Address of the APPLICATION focal object, which in this case is the address of the APPLICANT to which the APPLICATION refers. The control structure in step 1 indicates that there is no serial requirement concerning the two activities numbered "1.4a" and "1.4b." The step terminates, as do all steps, when there is nothing more that the office can do but await some external event, in this case the receipt from the applicant of all the final application forms. This event initiates Step 2, in which little

MIT Admissions Office

Operations

```
Procedure: Admit-Freshman-Class
Object: APPLICATION
Responsible: Director
Main-line:

    State null
    Event 1:  Receive Preliminary
        1.1 Verify Preliminary
        1.2 Create APPLICATION
        1.3 Select Interviewer from INTERVIEWERS
        1.4a Send Final-application-forms
        1.4b Send Interview-report to Interviewer
        End in waiting

    State waiting
    Event 2: Receive Final-application-forms
        2.1 Verify Forms.Final-Application.Check
        2.2 Send Check to MIT-OFFICES(Name="Cashier")
        End in complete

    State complete
    Event 3:  January 20, Year-applying-for
        3.1a Send S-S-G-R,L-list
        where CB-scores = "unknown" add
            3.1b Send CB-letter
        3.2a AA Select Faculty-review.Reviewer from FACULTY
        3.2b AA Select Staff-review1.Reviewer from ADMISSIONS-STAFF
        3.3a Faculty-review.Reviewer create Faculty-review using Application
        3.3b Staff-review1.Reviewer create Staff-review1 using Application
        where abs(Faculty-review.Rating - Staff-review1.Rating) > 1 add
            3.4 AA Select Staff-review2.Reviewer from ADMISSIONS-STAFF
            3.5 Staff-review2.Reviewer create Staff-review2 using Application
            end in reviewed
```

Figure 4-1: Admissions Office Main Line

activity is required; this step terminates in state **complete**, awaiting the January "round-up" actions.

Step 3 is initiated by the time event "January 20, Year-applying-for"; the last part of the date specifies that the value is to be found for any APPLICATION by looking at the value of its "Year-applying-for" attribute. This step primarily involves the selection of one reviewer from each of two classes (FACULTY and

ADMISSIONS-STAFF) and their creation of REVIEW entities related to the APPLICATION. If the values of the "Rating" attributes of these two REVIEWS differs by more than one, the procedure (based upon office policy) requires a third review. The object is then in the **reviewed** state, awaiting a decision.

The admissions decision process, beginning in February, is specified in Step 4 (Figure 4-2). Activity 4.2 represents the decision as a GROUP activity whereby all members of the APPLICATION class are placed into one of three subclasses. Each of these classes is defined as a subset of APPLICATION, and each subset definition refines the value classes of several attributes of APPLICATION. For example, the class ADMITTED (Figure 4-7) refines the APPLICATION attribute "Letter," which is a member of class LETTER, to "ACCEPTANCE-LETTER," which is a (member of a) restriction of class LETTER. Similarly, the APPLICATION attribute "Decision" is defined as a member of the (NAME) class DECISION; the definition of the ADMITTED class refines the value of the "Decision" attribute to the specific value "admitted."

The GROUP decision creates three possible processing paths, all of which happen to diverge at the termination of Step 4. Those applications that have been rejected end this step in state **done**; no further processing is required. No further processing is possible on waitlisted applications until April 15 (Event 8); they end Step 4 back in state **reviewed**. No further processing is possible on admitted applications until the applicant returns an acknowledgment; until then, the application is in state **admitted.**

The remainder of the main line is straightforward, specifying the limited processing involved with applicants acceptance or rejection of the admissions offer and subsequent archiving of the records after the new freshman class matriculates in September.

```
    State reviewed
    Event 4:  February 20, Year-applying-for
        4.1 Calculate Scholastic-Index
        4.2 Group APPLICATIONS into {ADMITTED, REJECTED, WAIT-LISTED}
        4.3 Create Letter
        4.4 Send Letter
        where Decision = "admit" end in admitted
        where Decision = "reject" end in done
        where Decision = "waitlist" end in reviewed

    State admitted
    Event 5:  Receive Reply
        5.1 Create ADMITTED.Acknowledgment
        5.2 Send ADMITTED.Acknowledgment
        5.3 where Acceptance = "refuse" add
            5.4 Send E3 to Financial-Aid
            end in done
        5.5 Send AAC to Financial-Aid
        end in accepted

    State accepted
    Event 6: July 15, Year-applying-for
        6.1 Send ADMITTED to MIT-OFFICES(Name="Freshman-Advisory")
        end in coming

    State coming
    Event 7:  September 30, Year-applying-for
        7.1 Send ADMITTED.E3 to MIT-OFFICES(Name="Freshman-Advisory")
        7.2 Archive each APPLICATION
        end in done

    State reviewed
    Event 8:  April 15, Year-applying-for
        8.1 Group WAIT-LISTED into {ACCEPTED, REJECTED}
        8.2 Create Letter
        8.3 Send Letter
        where Decision = "reject" end in done
        where Decision = "admit" end in admitted

Timing constraints:
    1. Event 2 < Event 4

Quantitative Information:
    Procedure statistics:
        Objects: 4500
        Variations:
            1. 7%
            2. 15%
        Branching:
             Step 4 → admitted: 44%
             Step 5 → accepted: 50%
```

Figure 4-2: Main Line, cont.

The only timing constraint specified represents the fact that all final application

materials must be in (Event 2) before the time for the admissions decision (Event 4). Violation of this constraint constitutes an exception, whose handling (if not the default action) is specified in the Exceptions section later in the specification template. The quantitative information presented in this includes several basic statistics: the total (average) number of objects in the procedure at any time; the percentage of objects to which each variation applies; and the average probability of important branch points in the main line.

The admissions procedure has several exceptions and two variations defined (Figure 4-3). The one timing constraint defined earlier gives rise to an exception when violated; the only action required in this case is the sending of a letter to the applicant, indicating that he is late in getting in some materials. Two application-specific exceptions are specified. (All other application-specific exceptions are therefore handled via the default mechanism.) The "unable to verify" exception to Activity 1.1 (a VERIFY) is handled by sending a letter and terminating the procedure for that application; receipt of a new preliminary application with all the needed information will initiate the procedure *de novo*. The second exception specified involves the lack of a check for the application fee with the application, which gives rise to another "unable to verify"; this results in a letter to the applicant but otherwise does not affect processing.

One general procedure exception is specified, in particular a cancellation where Event 4 (the date for decision) has occurred. (Again, cancellation before that time, since it is not specified, results in the default action for cancellation, namely termination of the procedure for that object.) The processing specified is simply the sending of a letter and subsequent termination.

Two variations are specified in the admissions example: "early decision" and "foreign." The former involves those applicants who ask on their applications for

```
Exceptions:
    Timing constraint:
        1. Send LATE-LETTER
    Activity-specific:
        1.1 Unable to verify:
            1.1 Send Preliminary,Problem-letter to Student
            1.2 Terminate
        2.1 Unable to verify:
            1.1 Send Check-letter
    General:
        Cancellation:
            where > Event 4
                1.1 Send CANCEL-ACKNOWLEDGMENT
                1.2 Terminate

Variations:
    1. where Early-decision = "T":
        add:
            State reviewed
            Event 9:   November 30, Year-applying-for - 1
                9.1 Calculate Scholastic-Index
                9.2 Group APPLICATIONS into ADMITTED, DISCOURAGE
                9.3 Create Letter
                9.4 Send Letter
                where Decision = "admit" end in admitted
                end in reviewed
        replace:
            Event 3:   November 20, Year-applying-for - 1

    2. where Foreign = "T":
        delete:
            Event 1
            Step 1

        add:
            State null
            Event 10: Receive Preliminary
                10.1 Send Preliminary to MIT-OFFICES(Name="FSO")
                end in wait-FSO-ok

            State wait-FSO-ok
            Event 11: Receive Preliminary.FSO-reply
                11.1 Verify Preliminary
                11.2 Create APPLICATION
                11.3 Select Interviewer from INTERVIEWERS
                11.4a Send Final-application-forms
                11.4b Send Interview-report to Interviewer
                End in waiting

        replace:
            Event 3. January 31, Year-applying-for
```

Figure 4-3: Admissions Procedure, cont.

this treatment; the latter is required for those applications received from outside

North America. Since both of these conditions are *a priori* attributes of the focal objects, they are treated as variations.

The early decision variation adds one event/step unit and replaces one event. Event 3 is the date upon which the application review process is begun (Step 3); in this variation that event specification is changed from January to November. The addition of Event 9 and its associated Step provides the mechanism for specifying the early decision process, which begins at the end of November (*vs.* February for the normal decision process). Note that the event is a transition from state **reviewed**, so that the variation provides two possible exits from that state: the normal, main line one and the new one. The decision process in Step 9 leads to either state **admitted** or **reviewed** in both cases the application rejoins the main line for further processing.

The foreign variation involves the deletion of the initial event/step unit and its replacement (by addition) by two units: the first in which the preliminary application (in this case a FOREIGN-PRELIM, a restriction of PRELIMINARY-APPLICATION-CARD) is sent to the Foreign Student Office for approval, and the second triggered by receipt of that approval and rejoining the main line. The only other change required by this variation is in the date of the review process, which is started later than would be the case for non-foreign, main line applications.

Environment

Organizational Context

```
Define Instance of ORGANIZATION
    Address =  "77 Massachusetts Avenue
          Cambridge, MA 02139"
    CEO = "President"
    Name = "MIT"
```

Organization Hierarchy

Name	Parent	Supervisor
MIT-Corporation	None	Chairman
Office-President	MIT-Corporation	President
Office-Chancellor	Office-President	Chancellor
Office-VP-Admin	Office-President	VP-Admin
Office-DSA	Office-Chancellor	Dean-Student-Affairs
Financial-Aid	Office-VP-Admin	Director-Finaid
Career-Planning	Office-VP-Admin	Director-CPP
Registrar	Office-Chancellor	Registrar
Freshman-Advisory	Office-DSA	Chairman-FAC
Foreign-Student	Office-DSA	Director-FSA
Cashier	Office-VP-Admin	Comptroller
Admissions	Office-VP-Admin	Director

Personnel Hierarchy

Name	Organizational-Unit	Reports-to	Current-holder
President	Office-President	Chairman	P. Gray
Chancellor	Office-Chancellor	President	W. Rosenblith
VP-Admin	Office-VP-Admin	President	J. Wynne
Director	Office-Admissions	VP-Admin	P. Richardson

```
MIT-EMPLOYEE is INTERNAL-EMPLOYEE

MIT-JOB is ROLE

MIT-OFFICE is INTERNAL-ORGANIZATIONAL-UNIT
```

External Context

```
APPLICANT is PERSON

EDUCATIONAL-COUNCIL is PERSON

REGISTRANT is APPLICANT

SCHOOL is ORGANIZATION
```

Figure 4-4: Admissions Office Environment

Internal Context

AA = ADMINISTRATIVE-ASSISTANT

ADMINISTRATIVE-ASSISTANT is ADMISSIONS-STAFF

ADMISSIONS-STAFF is MIT-EMPLOYEE

ADMITTED is APPLICATION

APPLICATION is RECORD

DISCOURAGE is APPLICATION

EVALUATOR is MIT-JOB

REJECTED is APPLICATION

REVIEWER is MIT-ROLE

WAIT-LISTED is APPLICATION

Documents & Communications

AAC

ACCEPTANCE-LETTER

ACKNOWLEDGMENT

CB-LETTER

CB-SCORE

CHECK

CH-LETTER

E3

FINAL-APPLICATION-FORM

FOREIGN-PRELIM is PRELIMINARY-APPLICATION-CARD

INTERVIEW-REPORT

LATE-LETTER

LETTER

LETTER-REPLY

Figure 4-5: Internal Context

PRELIMINARY-APPLICATION-CARD

PROBLEM-LETTER

RECOMMENDATION-FORM

REVIEW

SCHOOL-REPORT

SEVENTH-SEMESTER-GRADE-REPORT

S-S-G-R = SEVENTH-SEMESTER-GRADE-REPORT

Names

ACCEPTANCE

DECISION

GRADE

RATING

Figure 4-6: Internal Context, cont.

Definitions

```
AAC is DOCUMENT
    Refers:  APPLICATION

ACCEPTANCE is NAME
    {accept, refuse}

ACCEPTANCE-LETTER is LETTER
    where Result = "admit"
    Reply: LETTER-REPLY

ACKNOWLEDGMENT is DOCUMENT
    Refers: APPLICATION
    Text: TEXT (common)
    To: Refers.Student

ADMINISTRATIVE-ASSISTANT is ADMISSIONS-STAFF
    where Title = "AA"

ADMISSIONS-STAFF is INTERNAL-EMPLOYEE
    Name: NAME
    Job: ROLE (multiple)

ADMITTED is subset of APPLICATION
    Acceptance: Letter.Reply.Answer
    Acknowledgment: ACKNOWLEDGMENT
    Decision: "admit"
    Letter: ACCEPTANCE-LETTER

APPLICANT is PERSON
    Address: ADDRESS
    Application: APPLICATION
    Birthdate: DATE
    High-school: SCHOOL
    Id#: SOC-SEC-NO (unique)
    Name: TEXT
```

Figure 4-7: Definitions

```
APPLICATION is RECORD
    Address: Refers.Address
    CB-letter: CB-LETTER (optional)
    Check: CHECK
    Check-letter: CH-LETTER (optional)
    Chem/physics: BOOLEAN
    Constituents:
        Answer: Letter.Reply.Decision
        Boards: CB-SCORE (multiple)
        Forms:
            Evaluation-forms: RECOMMENDATIONS (multiple)
            Final-application: FINAL-APPLICATION-FORM
            Secondary-school-report: SCHOOL-REPORT
        Interview-report: INTERVIEW-REPORT
        Preliminary: PRELIMINARY-APPLICATION-CARD
        S-S-G-R: SECONDARY-SCHOOL-GRADE-REPORT
        Transcript: TRANSCRIPT
    Decision: DECISION
    Early-decision: BOOLEAN
    Faculty-review: REVIEW
    Foreign: BOOLEAN
    Interviewer: Interview-report.Interviewer
    Letter: LETTER
    L-list: LAUNDRY-LIST
    Minority: BOOLEAN
    MITID: Student.Id#
    Name: Refers.Name
    Refers: APPLICANT
    Scholastic-Index:
        S-I1: NUMBER
        S-I2: NUMBER
    School: Student.High-school
    Staff-review1: REVIEW
    Staff-review2: REVIEW (optional)
    Student: Refers
    Year-applying-for: YEAR

CANCEL-ACKNOWLEDGMENT is DOCUMENT
    Refers: APPLICATION
    Text: TEXT
    To: Refers.Student

CB-LETTER is DOCUMENT
    Refers: APPLICATION
    Text: TEXT
    To: Refers.Student
```

Figure 4-8: Definitions, cont.

```
CB-SCORE is DOCUMENT
    From: College-Board
    Id#: TEXT (unique)
    Refers: APPLICATION
    Score: (multiple)
        Score: NUMBER
        Test: TEXT

CHECK is DOCUMENT
    Refers: APPLICATION

CH-LETTER is DOCUMENT
    Refers: APPLICATION
    Text: TEXT (common)
    To: Refers.Student

DECISION is NAME
    {admit, reject, waitlist}

DISCOURAGE is subset of APPLICATION
    Decision: "Discourage"

EDUCATIONAL-COUNCIL of PERSON
    Name: TEXT
    Address: ADDRESS

E3 is DOCUMENT
    Refers: APPLICATION
    Same: Address,Faculty-review,Interviewer.Name,Name,School,
       Staff-review1,Staff-review2

FINAL-APPLICATION-FORM is DOCUMENT
    From: Student
    Payment: CHECK
    Refers: APPLICATION

FOREIGN-PRELIM is PRELIMINARY-APPLICATION-CARD
        where Foreign = "T"
    FSO-Reply: FSO-REPLY

FSO-REPLY is COMMUNICATION
    OK: BOOLEAN
    Refers: FOREIGN-PRELIM

INTERVIEW-REPORT is DOCUMENT
    Evaluation: EVALUATION
    Interviewer: EDUCATIONAL-COUNCIL
    Rating: EVAL-SCORE
    Refers: APPLICATION
```

Figure 4-9: Definitions, cont.

```
LATE-LETTER is DOCUMENT
    Refers: APPLICATION
    Text: TEXT
    To: Refers.Student

LAUNDRY-LIST is DOCUMENT
    Refers: APPLICATION
    Text: TEXT
    To: Refers.Student

LETTER is DOCUMENT
    Body: TEXT
    Provisional: not(Refers.Chem/physics)
    Refers: APPLICATION
    Result: Refers.Decision
    To: Refers.Student,Refers.School,Refers.Interviewer

LETTER-REPLY of DOCUMENT
    Answer: REPLY
    From: Refers.Student
    Refers: ACCEPTANCE-LETTER

MIT-EMPLOYEE is INTERNAL-EMPLOYEE
    Id#: TEXT (unique)
    Name: TEXT
    Office: MIT-OFFICE
    Role: ROLE
    Supervisor: MIT-EMPLOYEE
    Title: TEXT

MIT-OFFICE is INTERNAL-ORGANIZATIONAL-UNIT
    Name: TEXT (unique)
    Parent: MIT-OFFICE
    Supervisor: MIT-EMPLOYEE

PRELIMINARY-APPLICATION-CARD is DOCUMENT
    From: Student
    Refers: APPLICATION
    Same: Address,Age,Foreign,High-school,Name,Year-applying-for

PROBLEM-LETTER is DOCUMENT
    Refers: PRELIMINARY-APPLICATION-CARD
    Text: TEXT
    To: Refers.Student

RATING is NAME
    NUMBER where > 4 and < 11
```

Figure 4-10: Definitions, cont.

```
RECOMMENDATION-FORM is DOCUMENT
    Evaluating: Refers
    Evaluator: EVALUATOR
    From: Evaluator
    Recommendation: TEXT
    Refers: APPLICATION

REGISTRANT is ADMITTED
        where Acceptance = "accept"

REJECTED is subset of APPLICATION
    Decision: "reject"

REPLY is NAME
    {accept, refuse, defer}

REVIEW is COMMUNICATION
    Comments: TEXT
    Rating: NUMBER
    Refers: APPLICATION
    Reviewer: REVIEWER

REVIEWER
    merge members of FACULTY, ADMISSIONS-STAFF

SCHOOL-REPORT is DOCUMENT
    From: Student.High-school
    Eval-score: NUMBER
    Evaluation: EVALUATION
    Refers:  APPLICATION
    Student: Refers

SCHOOL is ORGANIZATION
    Address: ADDRESS
    Applicants: Invert APPLICANTS on High-school
    Guidance-counselor: EMPLOYEE
    Name: NAME

SEVENTH-SEMESTER-GRADE-REPORT is DOCUMENT
    Copy: Name
    Refers: APPLICATION
    Report: TEXT
    To: Refers.School

WAIT-LISTED
    Subset of APPLICATIONS
    Decision: "waitlist" (common)
```

Figure 4-11: Definitions, cont.

4.2 An Example Function

Figure 4-12 is a simple example of the kind of structuring mechanism provided by the function template. It represents the sponsored research administration function of the MIT Office of Sponsored Programs. The function's resource is the class of sponsored research programs, and a particular assistant director is responsible for each program. The resource is initialized by receipt of a PROPOSAL object, which invokes the procedure named "Proposal-negotiation." (The procedures, as well as the environment, are not shown here.) Once a program is initiated (the initiating procedure is terminated), there is one report that is required to be received during the management phase of the function: a monthly financial report (from the accounting office). The procedure "Financial-reporting" is invoked upon receipt of the report and the procedure "No-financial-report" is invoked if the report is not received on schedule.

Three "other events" are defined for this function, each of which (by definition) is a normal, expected event but without any regular scheduling. Each event (receipt of purchase or travel requests for expending the funds available in the account, and receipt of a notice changing some attribute of the program) invokes the named procedure. Finally, termination is invoked when the AUDIT entity (a final report from the accounting office) is received, and the named procedure is invoked. The OSP function description also includes quantitative information concerning the numbers of programs and people involved, as well as average frequencies of the "other events" for the class of resource being managed.

Office of Sponsored Programs

```
Function Sponsored-Research-Administration

    Resource: SPONSORED-RESEARCH-PROGRAMS

    Responsible: AD-Resp

    Initialization:  Receive PROPOSAL: Proposal-negotiation

    Required reports received:
        1. Monthly: FINANCIAL-REPORT: Financial-reporting; No-financial-report

    Required reports generated:
        none

    Other events:
        1. Receive PURCHASE-REQUISITION: Process-PR
        2. Receive TRAVEL-REQUEST: Process-TR
        3. Receive CHANGE-REQUEST: Change

    Termination:  Receive AUDIT: OSP-Terminating

    Quantitative Information:
        Number of resources: 800
        Number responsible: 11
        Number of personnel: 20
        Frequencies:
            1. 200/mo
            2. 30/mo
            3. 10/mo
```

Figure 4-12: OSP Function Specification

Chapter Five

Office Procedure Analysis Using OSL

We hypothesized and have found that one of the primary uses for OSL is as a tool for assisting in the process of office procedure analysis. Perhaps the most important criteria for any analysis methodology are that it be both learnable and usable. The purpose of developing such techniques is to short-circuit to some extent the process by which an analyst develops his intuition and skills. An analysis methodology, by providing a number of guiding principles and rules of thumb based upon them, incorporates much of the knowledge about the office environment that would otherwise need to be acquired through extensive experience in office systems analysis. These guidelines can also streamline the analysis process by focusing the analyst's efforts in a manner that has proven to be most effective; it thereby reduces his confusion about the most efficient way to proceed. Further, adherence to a common approach provides longer-term advantages to an analyst as well as his organization. Repeated use of a methodology allows it to be tuned to the specific needs and idiosyncrasies of an organization. It also can provide for consistency among analysis staff members, reducing training and allowing for more effective integration of systems designed for different parts of the organization.

We have elucidated in previous Chapters the framework and premises that underlie OSL. This Chapter describes an analysis methodology that we have developed, based upon those premises, concepts, and structures, to provide guidance for the analyst in developing his understanding of an office and constructing an OSL description of it.

5.1 Context

As discussed previously, OSL is built upon a particular model of offices and their operations. While it cannot be expected to fit all offices well (or even any offices exactly), that model serves as an overall framework for thinking about and describing the office. The major concepts of this model — functions managing resources, procedures processing objects, classes as collections of entities, object orientation rather than artifact counting, abstractions of implementations — provide the starting point for approaching the office and beginning to write the specification. Indeed, our approach forms a major underpinning of, and therefore shares many principles with, OAM, an Office Analysis Methodology developed in parallel with this work [48]. When constructed in conjunction with OAM, an OSL description of an office can help the analyst understand the structure of the office he is studying and assist him in organizing both his interviews and his writeup. (OAM incorporates guidelines for addressing the overall organizational analysis. As well as the technical issues of office procedure analysis with which we are concerned here, OAM pays special attention to a number of related areas of concern, including project planning, behavioral and organizational issues, interview techniques, and documentation standards. The OAM writeup format, the outline of which is shown in Figure 5-1, illustrates its common structure with OSL and its reliance on most of the OSL design premises. In practice (as described in Chapter 6), users have found it relatively easy to derive an English (OAM) writeup from their OSL descriptions of the office.)

5.2 OSL Skeletons

The use of OSL as a tool in the analysis process is based upon the concept of an OSL *skeleton.* A skeleton is a description of an office using the OSL *structure* but a

I. Introduction

    A. Mission

    B. Organization

    C. Overview of functions, resources, procedures, and objects

II. Procedure descriptions

    A. For each procedure:

        1. Environment

        2. Inputs and outputs

        3. Core procedure steps

        4. Major alternate procedure paths

        5. Databases

        6. Local exception handling

        7. Quantification

III. General exception handling

IV. Collected database descriptions and document samples

Figure 5-1: OAM Writeup Outline (from [48
)]

combination of abbreviated OSL and concise English for describing the *content* of the specification. The key notion here is to provide only as much information as is necessary at any stage of analysis, but in such a format that the need for and organization of the next level of data can be incorporated in an obvious and compatible manner. With this increasing complexity of information content comes the need for increasing formality. Therefore an initial skeleton would include describe events with phrases and steps with sentences; these descriptions would be expanded at later stages in the analysis process into the formal syntax of OSL event and activity specification.

A skeleton therefore forms an important bridge toward the ultimate OSL

description. By requiring the analyst to describe the broad outlines of the procedure in the OSL structure as soon as he begins organizing his interview notes, the skeleton serves as both an enforcing mechanism for the OSL approach and a tool for eliciting further detail for expansion into the final formal description.

Figures 5-3 and 5-2 present an example of an OSL skeleton used in deriving the Admissions Office example in Chapter 4. In the environment skeleton (Figure 5-3), only the important classes are named, each with an identification of its entity type. A few attributes are included, but only the class APPLICATION, which is the focal object of the procedure, is characterized by several attributes. In general, the environment part of an OSL skeleton (as with that of a full specification) includes only enough information to explain the operational part. The latter, in this example (Figure 5-2), consists of only the procedure name, the focal object, the responsible role, and an outline of the main line. Events are described as phrases, either already in OSL syntax (if convenient for the analyst) or readily translatable to it. Each step is identified as a short sentence that describes the general nature of the required processing.

5.3 Building the Specification

The starting point of the office specification is the set of functions that the office is (completely or partially) responsible for implementing, and the resources that those functions manage. The analyst's initial interview with the office manager provides sufficient information to start the construction of the organizational context section of the environment part of the initial skeleton, and to sketch out the major entities and classes of the internal context. It will sometimes be the case that there already exists some organizational description that can be used in part or in whole; such a description would come from a previous OSL description of one or

MIT Admissions Office

```
Procedure Process-applications

Object: APPLICATION

Responsible: Director

Main-line:

    State null
    Event 1: receive preliminary application
        1. Send out application material
            end in waiting

    State waiting
    Event 2:  Receive final application
        2. Maintain records about application
        end in complete

    State waiting
    Event 3:  January review
        3. Get the applications reviewed
        end in reviewed

    State reviewed
    Event 4:  February roundup
        4. Decide whether to admit applicant and let him know
        end in admitted

    State admitted
    Event 5:  Receive acceptance of admission offer
        5. Process an addition to freshman class list
        end in coming

    State coming
    Event 6:  September registration
        6. Clean out the files
        end in done
```

Figure 5-2: Admissions Office Operations Skeleton

more parts of the organization. Otherwise, the analyst should sketch out a sufficient subset of the organization to ensure a useful context for the office description. The organization and personnel hierarchies can be placed in the skeleton, starting with the office being described and its director and proceeding upward. Any other offices in the organization that are known to be referenced can be added, as well as their branches on the hierarchy.

```
APPLICANT is PERSON
    High-school: SCHOOL

APPLICATION is RECORD
    Refers: Student: APPLICANT
    Year-applying-for: YEAR
    Constituents:
        Final-application: FINAL-APPLICATION-FORM
        Secondary-school-report: SCHOOL-REPORT
        Evaluation-forms: RECOMMENDATIONS (multiple)
        Transcript: TRANSCRIPT
        Interview-report: INTERVIEW-REPORT
        Boards: CB-SCORE (multiple)
    Preliminary: PRELIMINARY-APPLICATION-CARD
    Reviews:  REVIEW

REVIEWER
    merge members of FACULTY, ADMISSIONS-STAFF

SCHOOL is ORGANIZATION
    Applicants: APPLICANT

ADMISSIONS-STAFF of INTERNAL-EMPLOYEE

PRELIMINARY-APPLICATION-CARD of DOCUMENT

FINAL-APPLICATION-FORM is DOCUMENT

CB-SCORE is DOCUMENT

INTERVIEW-REPORT of DOCUMENT

SCHOOL-REPORT is DOCUMENT

LETTER is DOCUMENT

CHECK is DOCUMENT
```

Figure 5-3: Admissions Office Environment Skeleton

In understanding the mission of the office, the analyst also attempts to identify the major functions and resources, as well as the important procedures of each function. For the operational part of the specification, the result of the initial interview should be a reasonably complete template for each function, with the procedures and events identified and described in skeleton form. Each set of resources is represented as a class in the environment, and each function constitutes a major portion of the operational specification. The function template serves as a

guide, indicating what kinds of procedures might be used to carry out the management of the resource. The initiating procedure, for example, is identified as the one that causes the resource to be created, or is the response to the resource first being "noticed" by the office; if the latter, it is probably the case that some other office or an outside party actually creates it. The terminating procedure is the one followed when the resource is no longer of interest (destroyed, sent elsewhere permanently, *etc.*) As the names of the procedures are placed in the skeleton, their focal objects are identified and noted in the environment; the objects, if they are not the same as the resource of the overriding function, are related to that resource in some way, and this relationship is also defined in the skeleton.

During the first round of staff interviews, the analyst identifies all the entities of significance to the office, their important attributes, and the ways in which they are related to each other (as expressed in class and attribute derivations). It will usually be up to the analyst to bring his perspective and expertise to bear upon the situation in order to define the more abstract entities; most interviewees will tend to describe their work in terms of the forms and other documents with which they work, rather than the objects to which those documents refer. The feature of OSL that requires that all documents be defined as referring to some other, more abstract, object forces the analyst to confront this issue early in his study. As we have explained in earlier chapters, an OSL specification deals primarily with abstractions rather than implementations. For example, most forms are artifacts; their real purpose is to communicate information about some other entity. Thus, the way to think about documents is to ask what the information contained in it is about; the answer to the question should be some abstract entity that is the object of some procedure or the resource managed by a function.

Temporary files are almost always artifacts also; usually they are physical implementations of procedure states. Recall that a state represents the situation in

which processing of the object cannot continue until some event beyond the office's control occurs. A temporary file is often used to store documents while waiting for the event. Such information is best represented as procedure states. Permanent files should remain explicitly described as entities of type FILE or ARCHIVE, as appropriate.

The goal of the staff interviews is to develop an understanding of the main line of each procedure, as well as the important variations. Constructing the skeleton descriptions of these pieces serves as means of organizing the analyst's impressions and allowing him to develop a coherent picture of the office and its operations. It also serves as a starting point for resolving inconsistencies, filling in incomplete information, and investigating exceptions.

A rough guideline for defining the main line of a procedure is to determine the history of the most common case in which everything goes right. If a somewhat different path is taken by a well-defined subset of objects, then that path is either a variation on the main line, or a separate procedure entirely; the distinction is based upon the amount of overlap, and is a matter of judgment. The events that separate steps in the procedure are, as described in Chapter 3, ones over which the office has no control. The focal object "rests" in some state until the event triggers further processing. Thus, waiting for a communication from outside, or waiting for a particular day, are examples of situations that would be reflected as procedure states.

It is important that the analyst not become overwhelmed with details and lose sight of the essential structures. By allowing the development of a skeleton to guide his information gathering, the analyst is encouraged to write down initially only what is necessary to describe the important ideas; the details are added later in the predefined structure. In general, development of the environment and operations

parts proceed in parallel, with the need to refer to entities (and their relationships) in the operational specification driving the construction of the appropriate classes and attributes in the environment specification. Once the main-line and variation steps are defined, the procedure description is completed in as much detail as is desired. Steps are broken down into activities, and the partial ordering of those activities defined. The list of exceptions for each activity should be examined to determine whether a non-default handler needs to be described for any, and any other exceptions that are identified should be described. As activities are described, any references to entities and attributes are again checked against the environment, and any needed additions are made. When the operational part is complete, the environment should contain descriptions of everything that is referenced. The environment is then be checked for consistency, and any needed name classes defined.

When describing the environment, the analyst should rely on the built-in entity types. By defining base classes with a suitable set of attributes, it should be possible to model most of the entities of interest in the office environment. It may, however, be necessary to invent other types (*i.e.*, use the entity type ENTITY); such inventions, though, should only be used as a last resort.

Entity definitions should be nonoverlapping; that is, only one class specification should define an entity. It can often be difficult, particularly in the case of an abstract entity for which no obvious concrete analog exists, to decide what is part of the entity and what is a related, but independent, entity. Consider, for instance, the entity GRANT/CONTRACT (of type AGREEMENT) in the OSP. This is an abstract entity (which has associated with it several documents, including one usually called the "grant" or "contract"). It is neither a proposal (which leads to it) nor a sponsored research program (which it defines), though it is obviously related to both. GRANT/CONTRACTS is therefore specified as a separate base

class, and is related to both proposals and research programs through several attributes of each.

By the time the second round of interviews is finished, it a virtually complete OSL description of the office is developed. This is then examined for any further missing information, which is obtained in further staff interviews or in the final interview with the office manager.

Chapter Six

Development Methodology and Field Studies

The development of OSL has been an iterative process, involving studies of office procedures and refinement of the language concepts and features. In this Chapter we describe the design methodology and the development of the key concepts underlying OSL. We also describe several field studies conducted by industry personnel using OSL, discuss their results and evaluate OSL in light of their experiences.

6.1 Case Studies

The initial data used to develop the OSL structure and vocabulary was obtained from a number of external sources, primarily those noted in the "Related Work" section of Chapter 2. In particular, a number of researchers trying to understand office applications made available case studies particularly those from Xerox PARC [41] and Wharton [57]. However, rather than rely upon published reports, we derived the bulk of the data used in developing the first version of OSL from an ongoing series of formal office case studies, which are an important component of this and related research. We initially conducted a series of about fifteen studies at MIT, ranging from purchasing, admissions, travel, student accounts, and payroll to academic department headquarters, a graduate administrative office, and a co-operative program administrative office [29]. Later studies, performed with the benefit of early versions of OSL and its underlying approach, included offices concerned with sponsored research administration [44], a physical plant dispatching operation [56], and the MIT Industrial Liaison Office [55]. In addition, our case

study series has included examinations of the administrative operation of a commercial television station [24] and the regional sales office of a manufacturing firm [46]. Less formal studies include the international division of a major Boston bank [26] and a large British insurance broker [27]. Finally, we have had access to about 30 case studies provided by a half-dozen cooperating firms [47].

Although our data describes a number of different office situations, we have concentrated on the semi-structured kinds of procedures that provide the basis for the OSL constructs. It is clear that functions accounting for the highest volume of work in typical office situations would benefit most from the development of automation tools. (Note that those business functions that occur in the greatest volume (accounting, inventory, *etc.*) were the first to be analyzed and automated.) As we noted in Chapter 3, it is the operational and, to a lesser extent, the managerial control functions (as categorized by Anthony [1]) rather than the strategic planning functions, which are of the greatest immediate interest.

Analysis of the case study data resulted in the development of the outlines of the procedure and function models underlying OSL. The basic premises of our approach were also determined by these studies: that most office procedures are fundamentally simple processes, obscured by implementation detail and disorganized exception-handling; that office specifications should be couched in terms of natural primitives of office work, and should suppress irrelevant detail; and that specifications should be written in a well-defined language, in which a given description has a single, unambiguous meaning.

The key to the OSL development process has been the identification of common and recurring structures in the case studies, and the embodiment of those structures in various features of the language. Clearly, such analytic data is most conducive to the development of lower-level models; the number of (lower-level)

activities that can be identified in any set of examples is much greater than the number of (higher-level) functions. This inherent problem — abstracting effective models from limited examples — is reflected in OSL's detailed activity taxonomy, its less complex procedure model, and its very simple function model.

Concomitant with the development of the language has been the development of the methodology, described in Chapter 5, for analyzing offices and constructing specifications in OSL. As noted, this has been an iterative approach; the analysis of case studies, parallel evolution of language and analysis methodology, and application to new case studies forms the design loop.

6.2 Field Studies

OSL is a tool that incorporates our particular perspective on office automation. In order to evaluate its utility, its ease of learning, and the validity of our overall approach, we conducted a set of field studies with a number of cooperating firms. The participating organizations evinced a willingness to participate in a research project, with the risks that such an effort entails. The individuals involved agreed to try our approach and techniques, both to experiment with new tools that may assist them in doing their jobs better, and to explore the benefits that may accrue from presenting their specific problems to us as case studies upon which further research will be based. They also agreed to provide us with evaluations of how effective our materials are and how they might be changed. In these studies we presented two 2½-day training seminars covering an earlier version of OSL as well as OAM (the office analysis methodology, described in Chapter 5, whose development was an outgrowth of the OSL research effort) to personnel from the participating companies. They then returned to their organizations to apply our techniques to operational offices there, and to provide us with feedback on their experiences. The

current version of OSL, as presented in Appendix A, resulted directly from the feedback received during the courses, as well as analysis of the field test results. Suggestions for additional research directions that were elicited from these experiences are discussed in Chapter 7.

Approximately 40 people representing seven firms attended courses designed to present both an appreciation of our approach to office automation, and OSL (and OAM) as means of implementing that approach. The firms include three insurance companies (called below "A," "B," and "C"), a research laboratory ("D"), a chemical company ("E"), a consumer products company ("F"), and an industrial products manufacturer ("G"). The attending personnel had a wide variety of training and backgrounds, including data processing systems analysis, industrial engineering, line management, secretarial, and shop foreman experience.

Seven OSL studies were conducted and submitted to us for evaluation; the sites of these studies include:

- a marketing support office in company "E" (7 procedures)
- a manufacturing production control office in company "E" (2 procedures)
- an engineering support office in company "E" (1 procedure)
- a large industrial engineering organization in company "F" (~20 procedures)
- a graduate school admissions office (performed independently by a participant from company "D") (3 procedures)
- a volunteer organization's membership management function (also performed by a company "D" employee) (3 procedures)
- a documentation library in company "A" (5 procedures)

In addition, groups from companies "A," "D," "E," and "F" sent written evaluations of their experience using OSL. Personnel from companies "B" and "C," who were unable to conduct studies in-house at the time the courses were concluded, also provided valuable verbal reports of their impressions of OSL and its potential utility to them. Company "G" decided after attending the training course not to proceed with the field trials. Company "B" is currently conducting a study using OSL.

6.3 Results

Overall, the courses and studies have been quite fruitful in helping us understand and enhance our material. While some inadequacies (not unexpectedly) have appeared, we have been gratified by the response of those undertaking studies. Most participants found our approach sensible and easy to understand. OAM was received well, particularly by those with little or no experience in office or systems analysis. OSL is more difficult to teach and learn; as with any formal language, use is the best teacher. The reaction of those who have used it, however, has been positive, particularly in the way that it structures the analysis process and organizes the documentation at various stages of a project. In fact, several users have used several levels of OSL skeletons as replacements for, rather than in addition to, the draft English writeups recommended by OAM.

The major value of OSL to its users has been in its approach to office analysis, which also underlies OAM. The control of complexity provided by initial concentration on the main line, and subsequent relation of variations and exceptions to it, has been a useful revelation for many users. Similarly, the concept that documents are artifacts, and not the key to understanding offices, has been very well received. Other positive aspects of OSL include its functional approach and its

uniform notation. Following are several positive comments from users about OSL and its approach:

> Using OSL is definitely a forcing mechanism keeping descriptions implementation-free. We found, as you have suggested, that basic procedures at OSL level/in OSL terms are straight forward — it's the uneven, historically developed implementation that makes them appear complex. ["A"]
>
> Although the edges are still rough, doing the English description after the OSL was fairly straight forward. ["A"]
>
> The process of producing the OSL description results in a depth of understanding of office procedures that goes beyond the final product. ["D"]
>
> It is much easier to locate information in an OSL description than in an English one. ["D"]
>
> OAM/OSL is a very thorough methodology, which if faithfully followed should yield an abundant and detailed understanding of "what's supposed to happen." ["F"]
>
> Application of the analysis methodology is quite useful in building credibility for the analyst. ["F"]
>
> Resulting documents from OAM/OSL (particularly OAM) are useful as training material. ["F"]

Company "E," which made extensive use of the techniques, provided the following list of advantages to the OSL approach:

- emphasis on "mainline" functions
- function input/output analysis
- interviewing of study office population
- "micro-analysis" level [referring to activities and steps]
- use of a uniform language [by a corporate office automation group]
- addresses interface of functions and [organizational] groups
- transferable skill [from trained analysts to novices]
- good for structured office environments

- concept of "artifacts" [documents as not being the focus of office procedure analysis]

A number of specific shortcomings of OSL and its application methodology were also pointed out by the field test community. The same firm that provided the list of advantages above also identified a number of disadvantages to the approach:

- time consuming (cost/benefit)
- high level of detail
- unique language (must be learned)
- poor in the unstructured office environment
- does not address people related issues
- "macro-analysis" level

The following comments also illustrate these and similar problems:

> The process is too tedious to use broadly. . . . While we recognize that analyst efficiencies would improve in subsequent studies, we still do not think that we can get the study period down to an acceptable duration. ["F"]
>
> The study results are quickly outdated. ["F"]
>
> Resulting documents are too voluminous. ["F"]
>
> OSL good for documenting process requirements, but very general. ["F"]
>
> We shared OSL descriptions with interviewees because they expressed an interest, but this confused them. ["A"]
>
> We have some concerns about OAM/OSL limitations because of the functional, vertical-slice approach as opposed to a process-oriented, through/across functional units approach. ["A"]
>
> The description represents the framework within which decisions are made — the decision processes themselves are only crudely represented. ["D"]
>
> Specific problems focused on the use of the language which was quite detailed and cumbersome. ["E"]

An analysis of the approximately 35 procedures submitted by the participants produced some interesting data about the use of specific OSL features. First, most procedures included at least two steps; few degenerate procedures were used. A total of 121 steps/events was defined. Of these events, 74% were specified as communications events, and all of these were simple "receive" specifications; none was a non-receipt event or a receipt of a response to a specific request. "Time" events constituted 17% of the event specifications, primarily periodic (monthly, annually, *etc.*) and a very few relative to some other event. Nine percent of the event specifications were "trigger" events, that is, based upon explicit command of someone in the office.

A total of about 450 activity specifications was used. After adjusting for misuse of activities due to misunderstanding by users, the distribution shown in Figure 6-1 was obtained.

25%	CREATE
25	SEND
10	EVALUATE
8	NEGOTIATE
6	VERIFY
4	REVISE
3	SELECT
3	ARCHIVE
3	PERFORM
3	APPROVE
2	SET
2	INITIATE
2	DELETE
2	LOG
1	CALCULATE
<1	REPEAT, GROUP, FILE, RETURN
0	ADD, REMOVE, ALLOCATE, NOTIFY, RETRIGGER, TERMINATE

Figure 6-1: Distribution of activities used in field studies

Figure 6-2 restates the activity distribution in the four categories of office operations defined in Chapter 3.

Branching constructions in the main line were never used; in a few cases

64%	Basic
27	Decision
4	Aggregate
5	Control

Figure 6-2: Distribution of activities by category

alternatives showed up as variations. In general, exceptions were specified where and as appropriate, while variations were only occasionally noted and not specified according to OSL syntax. Details were specified only rarely.

One study, the engineering support office, was written only in skeleton format, without using formal OSL syntax.

The use of built-in entity types in the environment descriptions was fairly straightforward. Most entities were documents and communications, and often "referred" to other documents and communications. Heavy use was also made of persons, roles, organizational units, and employees. More abstract types also appeared, particularly agreements and records. There was little attempt to define new types; virtually all entities were defined in terms of the existing, very general, built-in entities.

6.4 Evaluation

Ideally, we would like to be able to evaluate separately at least the following aspects of the field studies:

- the overall effectiveness of our approach to office analysis and specification
- the applications for which OSL is most and least appropriate
- the utility of the OSL analysis process

- the utility of the OSL specification product

- the specific features (especially the activity "verbs" and built-in entity "nouns") of the language

- the learnability of the language

- the effectiveness of the teaching in the field study courses

- the effects of user background on the utility of the approach and tools

In conducting field tests, rather than controlled experiments, however, it is generally difficult or impossible to determine the cause of any particular effect. Nevertheless, we have been able to draw a number of significant conclusions from a combination of debriefings, evaluation by the user community, observation, and examination of the results of users' efforts.

As we stated in Chapter 2, we expect that OSL should have a variety of uses. The major ones, of course, are as a tool for an analyst in developing his understanding and presentation of the office he is studying, and as a means of communicating requirements to a system architect. The field tests have shown quite clearly that the former goal has been addressed well by the OSL approach. Virtually all participants noted explicitly that exposure to the functional approach to office automation was to them the primary value of the exercise. The premises derived from that philosophy, upon which OSL is based — isolation of the main line, object orientation, structured analysis and presentation — have generally taken hold in the participating firms. For example:

> We will likely use a composite of methods previously used within our Company with learnings from [OAM/OSL] plowed in. . . . I feel that we have not rejected application of OAM/OSL but have merely picked off the nuggets from the process which are truly applicable to our situation. ["F"]
>
> The concept of "mainline" functional analysis vs. traditional job analysis as the

> important dimension to productivity improvement studies was found to be extremely useful and has been incorporated into the Office Applications Development Group (OAD) study approach. Not dealing directly with "artifacts" of the study office, but focusing on the intent of the documents for identifying alternate ways to enhance information and/or communication transfer has also been internalized. The concept of a uniform language to define office activities has become part of analysis techniques and jargon used by [OAD] to standardize communication between analysts and client groups. ["E"]

We therefore believe that our approach is fundamentally sound and that further development of the ideas derived from our first principles should be fruitful. The major problem with the use of OSL as an analysis tool is its extensive detail; this has proven to be a mixed blessing. While detail is helpful in enforcing rigor upon the process, in the absence of a clear need for a system design as a follow-on to the analysis the value of such detail is limited. Several directions for further research suggest themselves in this context; they and others are discussed in Chapter 7.

The second principal use hypothesized for OSL remains untested. We have as yet had no experience with system design based upon an OSL specification; none of the participating companies has gone beyond the analysis stage. The synthetic use of OSL in system design is at heart even more of a creative process than the analytic use tested in the field studies. It is therefore necessary to examine a significant number of examples of that synthesis process in order to elicit basic principles of design upon which a methodology can be based.

We do have one relevant design case, however; one of our internal case studies [55] done with an early version of OAM was used as the basis of a request for quotations and subsequent vendor selection and system design. While that office is currently the subject of a post-implementation study, several conclusions are clear. First, the writeup, based upon the OSL approach, including function and procedure

models, was considered by both users and vendors as an excellent description of the office and its information system support context. Second, the system finally developed was based heavily around a database system, whose data definition schema had to be developed with a knowledge of the office's functions and procedures. Had an OSL description been available, it would have served as an effective base for design of the database (from the OSL environment) and its transactions (from the OSL operations specification).

While we continue to believe that an OSL-type specification can be of significant utility as a design tool, full validation of that belief will await further studies. In particular, we feel that OSL as currently defined does not yet allow for the expression of sufficient detail to lead directly to a design effort. However, the *structure* of an OSL specification does appear to be an effective way of approaching the problem, and therefore an extended OSL, encompassing lower levels of detail within the same hierarchical framework, is a plausible approach to a system design tool. This issue is also discussed in more detail in Chapter 7.

Other uses originally posited for OSL included aid in training new employees and service as a procedure reference manual. These are examples of the generic use of OSL as a communications device, through which understanding of office procedures can be transmitted from those who write descriptions to those who read them. Results on this score have been limited and mixed. One analyst, quoted above, found that sharing OSL with office personnel caused confusion; another found OSL very helpful in explaining his understanding of the office to several of his interviewees. In general, the detailed, formal nature of OSL precludes its ready understanding by laymen. This conclusion argues for a simpler language, though such a result would undoubtedly involve serious cost to OSL's utility for its major purposes. As an intermediate step, the concept of the OSL skeleton, as described in Chapter 5, was developed from the results of the initial field study. This mechanism

retains the structure of the OSL approach without requiring the formal syntax. While it was designed as a tool for the analyst to use in building his final OSL description, the skeleton was found to be very useful in communicating with interviewees and in substituting, in a number of cases, for the English writeup that is the product of the OAM process.

The complexity of the language was in fact a recurring theme in the evaluations. While some of these criticisms are valid, we believe that they overstate the case. OSL is indeed a formal notation, requiring the precision of usage and understanding that such a language entails. Though many had some analytical training, particularly in industrial engineering and DP systems, most of the users in the field studies had had little or no experience in formal language usage. Thus some of their troubles stem from being forced to work in an unusually demanding context after only a three-day training period. As with any complex skill, use of a formal language is the best teacher. For example, the following comment was received from a user in the process of the case study:

> Just a comment on writing OSL... I'm having much less trouble than I expected. In fact, I'm even enjoying it a little. It's a lot less painful than the English language description! ["F"]

Internal users have found that their second study using OSL is significantly easier than the first, indicating that learning to use the language well is both possible and helpful. The only field test user to perform two studies remarked that the familiarity with the language gained in his first study made a significant difference in the time spent and the difficulty encountered in his second. Several of the written reports also indicated the participants' belief that the learning curve had a significant effect.

We might attempt to distinguish problems with usability from those of

learnability; the two concepts are often (and perhaps must be) antithetical. Part of the users' problems can be ascribed to less than ideal teaching materials. The development of the course materials and presentations was another iterative process. While we believe that the most recent version of the course is reasonably effective, we are not yet satisfied that it is the best approach to the subject. More and better written materials provided before the course, effective problem assignments, and a more interactive style would undoubtedly serve the users better.

All the complaints about complexity should not be taken at face value, however. We feel that some of the users' difficulty was due to OSL's in fact achieving one of its design goals: its use forced analysts to be clear and explicit in representing their understanding of the office in OSL terms, which required them to deal with both an approach and a level of precision that was unusual for them. All users admitted the validity and utility of the approach, if not the precision.

We now turn to the question of OSL's specifics. The OSL specifications returned to us for evaluation indicated that many of the features were not used, particularly branching structures and non-main-line specifications (variations, exceptions, details). Most of the built-in entity types were used and attributes more or less correctly defined, but features allowing descriptions of complex relationships in the environment were not in evidence. In general, the environment modeling examples were typical of unsophisticated, new users. The major failing in the environment descriptions was the excessive concentration on documents and other artifacts; the inability to identify more abstract objects is partly a problem in users' perceptions and partly due to an insufficient quantity of cases to allow us to identify a larger number of generic office objects and embed them in the language.

The activity set has undergone several changes as a result of the field tests. As noted in Figure 6-1, by far the most common activities used were SEND and

CREATE; these were almost always used with reference to some document. It is clear that the language does not go far enough in enforcing our object orientation — it is too easy for a user to deal primarily with documents and other artifacts. Some document-based activities in earlier versions were removed from the language to help prevent this "easy way out," but more work needs to be done to mold the language to the underlying principles. Further, the large usage of two activities suggests that further investigation into the nature of those actions would provide useful insight into their intentions and allow several more precise activities to be defined. Similarly, the overwhelming use of communications events as step triggers is certainly a reflection of office realities, but also suggests that further differentiation by means of intended use or source of the information would lead to a more useful language.

OSL, as expected, has been found to be of most value in more process-oriented offices. Since OSL's primary focus has been on office procedures, those offices are most appropriate for the detailed OSL descriptions. For example:

> Also the level of detail precluded the ability to effectively use OSL for the analysis of an office with many unique functions and procedures; e.g., a staff/decision office environment with poorly defined procedures and many exceptions and variations. A more structured office with limited functions and well defined procedure is best suited for the OSL analysis approach. ["E"]

In less structured offices, the skeleton description form has been found to be quite useful, while a full OSL specification is inappropriate.

A number of features of the current version of OSL and its use, as presented in this thesis, were derived directly from results of the field studies. The most important of these is the notion of the skeleton itself, and its use as an ongoing tool in the analysis process and the development of the final OSL specification. Other important effects on OSL as a result of the studies included: the elimination of

some of the environment modeling mechanisms that were unused due to their complexity; changes in the activity portfolio; variations in the syntax of the procedure description; and reorganization of the presentation format of the environment description to make it more useful as a reference. In general, the language has been somewhat streamlined to make it more useful for analysis, but it retains the hierarchical organization so as to provide *structure* for more detailed descriptions necessary for system requirements specifications. Finally, the field studies brought to light a number of important areas for further research, which are discussed in Chapter 7.

Chapter Seven

Summary and Directions for Further Research

OSL has been designed to embody our approach to office automation in the form of an analysis and description tool for the office analyst. It provides both a framework for the analyst's approach to understanding an office's operations and a communications mechanism for expressing that understanding. Our work has focused on the problems of increasing our knowledge of office operations and embedding that knowledge in OSL; our goal is to make that knowledge more accessible to analysts who have not had the training or experience to develop an intuition for their task.

7.1 Summary

In this document, we have described our approach to office systems analysis, detailed the design of OSL, presented an outline of a methodology for analysis and specification of office procedures, and reported on field studies of OSL and its use. At base, we have two fundamental theses. The first is that our *functional* approach to office automation, which focuses upon the *business* goals of an office rather than its information artifacts and current implementations, is an effective means of understanding and supporting the "automation" needs of an organization. The second is that, within this context, an office specification language can be a useful tool for office analysis, for the design of office support systems, and for communications among various people interested in the functioning of an office. A corollary of this thesis is that such a language can be designed and used effectively. Our efforts have been directed toward the development of an office specification

language, along with an associated methodology for its use in office analysis, that embodies our approach.

Our study of the office domain and the nature of office systems analysis has elicited a number of principles upon which the design of OSL is based. The framework provided by these premises, through the OSL structure, vocabulary, and usage guidelines, is unique in its approach. We have found both in our own development work and in field tests of OSL that tools based upon this approach can be and are of significant utility to office analysts in a variety of business situations. The key distinguishing aspects of our formulation include:

- the functional approach to office analysis, which views procedures as implementing pieces of functions that may span several parts of an organization in furtherance of a business-oriented mission, rather than as inherently necessary or fundamental operations;

- the use of a formal specification language not only as a descriptive tool but as an analytic one, providing continuing guidance to the analyst throughout the process of analysis and description of an office;

- the development at several levels of detail of models of office operations and the office environment;

- the notion that office procedures are fundamentally simple, but are obscured from easy analysis and understanding by special cases, historical accretions, and exception handling;

- the object-oriented approach to procedure analysis, looking at a procedure as dealing fundamentally with changing the state of an (abstract) focal object;

- the concept of an office procedure as the idealized history of an object, as a set of processing goals rather than strict requirements;

- the belief that forms are not critical to understanding office operations, that they are merely collections of information about other, more important and fundamental objects, that in fact they are artifacts that should be dealt with as implementation details;

Several major design premises, developed from these principles, have been implemented directly as specific language features:

1. There exists structure in offices, and there are structures common to disparate offices that can be used to develop models that are helpful in the analysis and specification of office operations. This idea has been implemented in OSL through the use of templates for structuring functions and procedures, and by the development of models of functions, procedures, and environments, and taxonomies of activities, events, and exceptions, that are embedded in the OSL syntax and semantics.

2. Office procedures are basically simple; apparent complexity can be explained as special cases historically grafted onto the underlying procedure. OSL's procedure model implements this notion: the "main line" identifies the core of the processing requirements and serves as the point of reference for describing the special cases. Those cases themselves are further structured into variations, exceptions, and timing constraints, each of which are described in separate sections of the procedure template and keyed to specific parts of the main line specification.

3. Office procedures are not algorithms, nor are they strict rules that must be followed. An OSL procedure represents the normal history of processing some entity; equivalently, the goal of those executing an office procedure is to make the result appear *as if* the nominal procedure had been followed. The implementation of this idea is embodied in OSL's notion of a focal object for each procedure, and the procedure specification as the idealized history of processing the object.

4. Documents are not the appropriate focus of office analysis; they are merely artifacts of the current implementation. OSL and its attendant methodology enforce this principle by requiring that the analyst identify and specify the entity to which each document defined in the environment refers.

7.2 Evaluation of OSL

Field tests of OSL and its methodology have basically confirmed the utility and value of our overall approach, while identifying several problems in its current implementation. We postulated three kinds of uses for OSL: analysis, system design, and communications among people (analysts, workers, managers, designers, *etc.*) concerned with the office's operations. In general, all of our premises have been of demonstrated utility in office analysis, and most in communication; for the latter, more development needs to be done on the specific OSL structures. Data on design utility is extremely limited, though the overall structures and the treatment of documents as artifacts appear to be helpful.

The major question about the use of OSL in office analysis appears to be that of too much detail. Many users found that the level of analysis and specification detail required for an OSL study was excessive compared to the value of the results. We believe that some of these criticisms stem from inadequacies in the field tests, specifically a lack of explicit goals of the participants beyond self-education and assisting in a research project. Yet there are some possible solutions to whatever real problems lie beneath the question. The OSL skeleton seemed to satisfy a number of needs, and further development of that notion would be fruitful. OSL's activity set needs to be refined, based upon the results of the studies and further research efforts. The environment modeling facilities could be further streamlined to eliminate those mechanisms that are confusing and/or unused. Finally, a higher-level taxonomy of procedure types, functions, and offices could be identified, replacing some of the modeling effort currently needed with less-costly classification.

In contrast to its use for analysis, OSL appears to be inadequately detailed for system design. Since we have little hard data on design use it is difficult to

determine specific design needs. Our experience does indicate, however, that the major problem stems from an inability to capture sufficient detail in the environment and the activity specifications; a system design effort requires, among other items, that a database be defined and specific information support requirements for decision activities be described. Several approaches to a more usefully detailed OSL are possible. The first is to use a full database data definition language modeling facility for the environment, whether a semantic data model or a more familiar one, in order to encourage gathering sufficient data to design the database. However, this approach raises the dangerous possibility that the model will be driven by artifacts rather than fundamental abstractions, and should probably only be used by experienced analysts; the value of any semantic data model, and OSL's in particular, is its elimination of implementation dependencies from the modeling mechanism.

Design could also be assisted by extending OSL's semantics and formal syntax to a level of detail below that of activities. This would require further understanding of the information requirements of each of the OSL activities, once an optimal set has been determined. We have provided an initial hypothesis as to the requirements for and nature of this activity set. Further work would be useful both in refining the membership of that set and in exploring activity semantics to the point at which defining system support requirements for each is possible.

Another way in which the design detail problem could be approached is by embedding some of the existing models into a system. By assuming that all systems designed with this methodology will share certain underlying similarities, such as the OSL procedure model, part of the analysis and design problem can be turned into a customization problem. Such constraints upon the system's functionality could significantly reduce the level of detail needed in the analysis and specification phase, at the obvious cost of being totally inapplicable for some types of offices.

The final use posited for OSL, which encompasses what we have generally called communications, appears to suffer from excessive overhead. By this we mean that the mechanism associated with an OSL description of an office is rather forbidding to the casual reader of specifications. The skeleton idea is an appropriate means of addressing this problem. However, we believe that there is a limit to the gain achievable from such an approach. We feel that the problem of overhead in an office specification is to a large extent inherent: offices are complex places, and there is much that must be described to present an adequate understanding of its operations. The value of a formal language is in fact due to its detailed structure and vocabulary, which once learned allow a reader to shortcut the need to explain each office from basic principles.

7.3 Research Directions

The problems with OSL and the possible solutions discussed above point out several directions for further research in this area. We have, of course, only begun the process of understanding how offices work and developing a theory of office work that can guide the development of office systems. In fact, the issue of transition from understanding of an office's current operations and desired functioning, to the design of a system to support that functioning, is clearly the next step in any research effort.

In the previous section we identified several areas that are ripe for extending the work begun in this thesis. We have several specific suggestions for plausible follow-ons. The first is the development of more case studies, which are needed in order to provide a larger database for analysis. It is necessary to have a reasonable number of examples of any construct before general models can be abstracted. While a single office study may provide dozens or hundreds of examples of activity-

level constructs, it will only elicit one or a few function examples. Since the current function model is quite simple, it is not sufficiently helpful in the analysis process. In fact, taxonomies at all levels, particularly abstractions such as functions, procedures, activities, and entities, would be of extreme value in analysis and specification.

Second, the gap between analysis and design needs to be bridged. It is very difficult, in a research setting, to experiment with realistic systems designed for specific office situations. A long-term cooperative program is needed to establish a database of analysis/design efforts in real settings. From such a database, attempts can be made to develop models and methodologies that feed back critical system design parameters to the analysis and specification techniques. This research should involve two types of studies. The first, an empirical approach, would follow systems from the analysis effort (using OSL or a derivative) through system design and implementation, to post-implementation studies; the goal would be to examine the ways in which the specification influenced (or didn't influence) the design, and how the ultimate use of the system reflects both the design and the specification. The second type of study is more analytical, and would involve examination of existing successful systems that provide effective functional support. Working backward to see how they were in fact built, would permit abstracting constructs, within the context of the (possibly modified) OSL structure, that would lead to the definition of activities, entities, events, *etc.* that are more effective in guiding an analysis toward a real system design.

Finally, the interaction between structured and unstructured parts of office procedures needs to be explored. Even at the OSL activity level, many tasks are judgmental in nature and system support for them can and must be in the form of (operations-level) decision support systems. We need to develop an understanding of how information and support needs and capabilities differ based on the type of

decision to be made (*i.e.*, due to the particular activity process required), and to design analysis tools that provide guidance for the gathering of the appropriate information.

In all cases, the need for field studies is crucial. In developing tools for office analysts and designers there appears to be no substitute for the feedback gathered from actual use by actual analysts on actual projects. The latter is especially important, as an effort made simply for educational or cooperative purposes fails to meet the real needs of any of the parties.

7.4 On Research in Office Automation

A common thread runs through our comments in the previous sections: an approach to office automation from the design, or "back" end, rather than from the "front," or analysis, end. While we would argue that either approach is inadequate in itself, our experience with OSL has led us to believe that the next steps in office automation research should begin at the design end. The paradigm of office analysis and specification as leading to a system design, whether completely manual or including computer-based system support, is no doubt an effective one, but it requires that some further thought be given to the nature of such systems. In particular, if we are going to address the system support of semi-structured office procedures, as we discussed in Chapter 2, we need to pay more attention to the premises upon which those support systems are to be designed. By understanding the design process, we can work backwards to determine analysis needs, for example, the level of detail needed to specify adequately support for various activities.

We feel, based upon extensive experience in designing and observing office

systems, that the database is the key facility in any automated office support system. Therefore the OSL model of procedures as key and the environment description being a reference may be skewed. An alternative design (and therefore analysis) approach that is based upon the same basic premises would have an OSL environment specification drive the development of the database schema, expressing the relationships, operations, and information used in the office in an object-oriented manner. The procedures and functions would then provide the information for developing transactions definitions.

An alternative approach is to rely less on complete customization via specification and design, and, as mentioned in the previous section, embed a certain amount of basic functionality in a system which is then customized by the analyst. The usual tradeoffs apply here: the more generic the system functionality, the more work for the analyst and designer, but the more requirements it can ultimately satisfy. This approach would argue for a more experimental, rather than analytic, approach to basic office automation research. (Though it must be pointed out that true experimental research in office automation is not merely the usual constructing of a system and seeing what happens; as in any field, hypothesis formation and testing is a prerequisite to the utility of observations.)

Much of our discussion revolves implicitly or explicitly around the question of the appropriate level of abstraction for office analysis and specification tools. Our fundamental approach is based upon office functions, which are the purpose of the office's existence. Yet contemporary system design still deals primarily with low-level, high-volume, computer-oriented applications such as word processing, electronic mail, and database management. Few users have been successful in achieving an effective integration of these generic tools to support the needs of their specific office functions. The best support for such operations is still based upon some form of custom software development; witness the lengths to which vendors of

word processors will go to be able to claim any kind of "programmability" for their systems. The crucial questions are then: what is the appropriate paradigm for designing effective functional office support systems, and what kinds of analysis and design tools are most appropriate in that context? The OSL model provides multiple levels of abstraction, each of which could be expanded, given further investigation, into a usable approach to customization:

- At its lowest level, we have begun to identify a number of distinct types of decision activities; this categorization could serve as the basis for more detailed "operational decision support" system analysis and design.

- At the procedure level, the OSL model allows for expression of the key requirements for and external effects upon the processing of abstract objects. Expanding upon this approach might lead to the implementation of the kind of general system substrate mentioned earlier; the goal would then be to find the appropriate abstractions needed to describe ways of defining implementations of specific procedures.

- The top level of the OSL model is the function. A more extensive study of the structure of office functions may lead to the definition of a set of "generic applications"; such a taxonomy would permit easier initial categorization of office functions and allow for specialized analysis and design techniques keyed to the unique requirements of each application.

Each of these levels of abstraction leads points to a distinct form of analysis and design tools. We believe that each is valuable in itself, and that all of them, suitably integrated, are necessary to provide for the effective development of office support systems.

As a final point, we would observe that the separation of the analysis function from system design is an attractive idea, but one in which we no longer have much confidence. It is just not clear, in an effort whose overall goal is the development of

a system, that an analysis done by someone without the background to appreciate the issues of system design will be of much use, at least with today's tools. Whether or not we can develop better analysis tools, based upon the design of effective office support systems, will determine whether "the office of the future" is to be anything more than a more expensive version of the office of today.

Appendix A

OSL Reference Manual

Note: This manual makes use of examples drawn from the OSL descriptions of the MIT Admissions Office undergraduate admissions procedure and the MIT Office of Sponsored Research sponsored research management function [28] (Chapter 4).

A.1 Definitions

A.1.1 Specifications

An OSL *specification* is a description, in the OSL formal notation, of some part of an organization. Organizations are normally divided physically into *offices*, and the unit of study in many analysis projects is the office. OSL takes the view that the fundamental unit of analysis is the *function* (which will be defined shortly); an office may implement part or all of one or more functions. Thus, an OSL specification may be a description of an office, consisting of various parts of functions, or of a function, encompassing one or more offices. In this document, we use the term "area" to refer to the the parts of the organization that are of interest in any particular specification.

A specification describes both the (static) context and the (dynamic) operations of the area, and the structure of a specification reflects this dichotomy. This structure has two pieces: the *environment* or static part, and the *operational* or dynamic part.

A.1.2 Environment

The environment part of an OSL specification describes all the "things" that are in or of concern to the area; these are termed *entities*. Most entities are organized into collections called *classes*; entities have *attributes* that describe their characteristics and relate them to other entities in the environment. In essence, the environment is a "model" of the area, of the relevant entities and the relationships among them. The environment part of the specification defines the vocabulary used in the operational part of the specification: anything named in the operations specification is either built into the language or defined in the environment part using the OSL environment modeling facilities.

A.1.2.1 Entities

An entity is anything, whether concrete (*e.g.*, an employee, a document, a widget) or abstract (*e.g.*, a program, an account, a job) that is used, manipulated, referred to, or otherwise relevant to, the area, its people, or its operations. Any particular entity is an *instance* of its *type*; for example, the manual that you are reading is a specific instance of all the entities of type MANUAL (and also of type DOCUMENT).

A.1.2.2 Classes

A class is a named, homogeneous collection of entities of a single type, (*e.g.*, a class of manuals, a class of purchase requisition forms, a class of corporations, a class of corporate employees, a class of corporate officers who hold more than 1000 shares of stock). The entities that make up a class are its *members*. In the following, we shall use "a FOO" to mean "a member of the class FOO."

There are two kinds of classes: those that are built-in to OSL, and those that are defined by the user of OSL to describe particular aspects of an environment.

A *built-in class* is an implicit class defined by, and consisting of all entities of, a built-in entity type. As the name implies, each of these classes is part of the definition of OSL, and they are not declared as part of any specification. A built-in class name may be used, however, in any place in an environment specification in which a class name is called for; in particular, they serve as the *parent classes* for the definition of many *derived classes*. (For example, the built-in entity type EMPLOYEE defines the built-in class EMPLOYEES.)

A *derived class* is defined in terms of some other (parent) class(es) in the environment. Thus a member of a derived class is also a member of one or more other classes, including one built-in class; this built-in class defines the type of the member. For example, the class MIT-EMPLOYEES is a derived class, defined in terms of the built-in class INTERNAL-EMPLOYEES (which is therefore the parent class of MIT-EMPLOYEES). Thus each member of MIT-EMPLOYEES is an entity of type INTERNAL-EMPLOYEE, and is also a member of the (built-in) class INTERNAL-EMPLOYEES. The class ADMISSIONS-STAFF in turn describes a set of entities that is a restriction (*q.v.*) of the class MIT-EMPLOYEES. Each member of ADMISSIONS-STAFF is therefore an entity of type INTERNAL-EMPLOYEE, and also a member of the classes INTERNAL-EMPLOYEES and MIT-EMPLOYEES.

A.1.2.3 Attributes

An attribute is some characteristic of an entity; for each entity, there is some specific *value* for each of its attributes, and these values serve to define the individual entity and to distinguish it from other entities in its class. The value of an attribute is either some entity in the environment, or some set of such entities. For example, consider a particular member of class MIT-EMPLOYEES; each attribute will have a particular value, such as Name = "Paul Gray" (a member of the class

TEXT), Rank = "President" (a member of the class RANKS), Office = "3-208" (a member of the class MIT-OFFICES).

Classes may also have attributes, describing characteristics of the class as a whole, rather than those of individual members. For example, the number of members currently in a class (*e.g.*, the number of MIT-EMPLOYEES) is an attribute of the class itself, not of any of its members.

The possible values of an attribute may be described simply by specifying the class from which its value is to be drawn; this is a *primitive attribute.* Alternatively, an attribute's set of possible values may depend directly, by a specified rule, upon its relationship to something else in the environment; such a rule defines a *derived attribute.*

A.1.2.4 Value Classes

In the definition of a class, each attribute of the members is described in terms of its *value class*, that is, the class from which its values can be drawn. For example, the (primitive) attribute "Telephone-number" of class MIT-EMPLOYEES is specified as having as its value class MIT-PHONES, indicating that for a particular MIT-EMPLOYEE the value of his telephone number may be any member of the class MIT-PHONES.

A.1.2.5 Attribute Semantics and Built-in Entity Types

A built-in entity type includes in its definition a set of attributes (including, of course, their value classes) that provide a means for characterizing particular entities of that type. As described above, the type defines a built-in class whose members consist of all entities of that type. For example, the built-in entity type EMPLOYEE

defines the built-in class EMPLOYEES. Each EMPLOYEE entity includes such attributes as "Name," which is an entity of type NAME describing the name of the employee; and "Supervisor," an entity of type EMPLOYEE that describes the employee's supervisor. Each entity of a given type has at least those attributes; when used in defining a built-in class, any additional attributes specific to, and characteristic of, the members of the built-in class are added.

A.1.2.6 Modeling of Relationships

The mechanisms of OSL environment specification provide for the definition of relationships among entities. Built-in classes indicate how entities may be members of several related classes. Attributes directly relate entities to each other; since the value of an attribute is an entity (or class of entities), an explicit relationship is indicated. In particular, the several mechanisms available for expressing the derivation of (derived) attribute values provides a rich set of models for inter-entity relationships.

A.1.3 Office Operation Specifications

As we have noted, the operational part of an OSL specification describes the functions of the office in terms of actions that are performed upon entities in the environment. In this section, we define the important concepts needed to read and write OSL operation specifications.

A.1.3.1 Functions

A function is the set of all actions and information concerned with the management and maintenance of some class of entities. These entities are called the function's *resources.* A function and its resources are defined not specifically as

information processing or document handling, but in terms of some business goal of the organization. Typical resources might include people, accounts, time slots, *etc.*; in the OSP, the resources are "sponsored research programs."

A function is always "operating," since the *need* to manage resources exists independent of the existence of any particular object. This management is effected through a set of *procedures*; each procedure is invoked upon the occurrence of a particular *event* of importance to the resource. When no procedure is being executed, an extant resource is said to be in its *quiescent state*. When any procedure is invoked due to an event concerning a resource, the resource is said to be in an *active state*; it returns to the quiescent state upon termination of the procedure (and any other procedures that may have been invoked by the original procedure).

A.1.3.2 Procedures

A procedure is a structure that specifies how some entity (or set of entities) is to be processed from an initial state to a final state. A procedure's basic components are a set of *activities* and *events*. The entity processed by a procedure is called the procedure's *object*; the object is related in some way to the resource being managed by the function(s) of which the procedure is a part. (For example, in the OSP, there is a procedure that processes the object "Proposal," a member of the value class (PROPOSALS) of the "Proposal" attribute of the SPONSORED-RESEARCH-PROGRAM resource.) In contrast to a function, a procedure has a specific invocation and a definite termination.

An OSL procedure is not meant to specify an exact prescription that must be followed, but rather an idealized goal. The procedure represents the history of processing of an object in the case where everything "works correctly." This history is called the *Main line* of the procedure. In "executing" an office procedure, a

worker's goal is in a significant sense to make the object end up in a final state *as if* it had followed the main line. It is the handling of all the special cases and problems that cause the procedure to deviate from the ideal that is the essence of the semi-structured nature of office procedures. The OSL procedure specification mechanism provides a structure in which to represent various levels of deviations and details, all based upon the specified main line.

A.1.3.3 Activities

Activities are the lowest-level operations defined in OSL. The set of activities defined by OSL represents all the possible actions that can be taken with regard to entities or classes of entities. An activity defines a particular type of manipulation of one or more entities; it may be thought of as a "verb" of the language, by which the "subject" (the person responsible for the procedure, or someone designated by him) operates on the object. An activity describes a semantically-meaningful process; it may or may not be a structured one. For example, the activity GROUP indicates a subsetting decision. There are cases of grouping in which an algorithm is applicable (*e.g.*, group applicants according to age) while others may be inherently a judgmental process (*e.g.*, group applicants into accepted, rejected, waitlisted). At the activity level, we characterize both operations as GROUP operations, and leave the details of particular implementations to lower or later levels of description.

A.1.4 Syntax

Unless otherwise specified, the following conventions hold for all descriptions in this manual:

- All names must begin with a letter and consist of letters, numbers, and hyphens.

- Class names are written in all capitals; example: `MIT-EMPLOYEES`

- Attribute names are written in small letters with initial capitals; example: `Supervisor`

- Language literals are written in lowercase: `receive`

- Metasyntactic words that represent a set of possible values are in italics; example: *attributes*

- Angle brackets surrounding an item indicate that zero, one, or many of the items may be used, separated by commas or carriage returns; example: <*attributes*> means that any number of attributes may be specified.

- Curly brackets surround optional items (separated by semicolons). They may be specified or not, as required; example: {*option*}

- Square brackets surround a set of items of which exactly one must be chosen; example: `[option1, option2, option3]`

- Descriptive instructions that are not actually a part of a syntax definition are in a special bold italic typeface; example: ***explanation***

- Specific values of text classes are enclosed in quotation marks; example: CEO = "Paul Gray"

A.2 Operational Specifications

Operational specifications in OSL include several levels of abstraction. At the highest level a *function* represents the management of a set of entities, the *resources* of the function, over time. (The OSP function "Sponsored Research Administration" represents the management of sponsored research programs.) The format of an OSL function description provides an overview and summary of the procedures and events relevant to the management of its resources. Each major

event invokes a procedure, which describes the needed processing. A procedure is constructed of *steps*, *states*, and *events*. A step consists of a group of *activities* that should be completed before further processing is done. Activities can be further explicated in a "Details" section, which is the lowest level of operational specification defined in OSL.

In an operational specification, references to classes and entities in the environment use the same forms as those in the environment specification itself. (See "References" in the Environment section for details.) In addition to those rules, there is a "local context" established for each function and procedure; this context is determined by the specified class of resources being managed, or the specified focal object being processed, respectively. Within this context attribute names can be used unambiguously to refer to attributes of the resource or focal object.

A.2.1 Functions

A function is the set of operations necessary to manage a set of resources. The operations are specified in terms of procedures, each of which performs the processing necessary at some point in the "life" of the resource. That life begins with *initialization*, which may take one of two forms: either the resource is "created" by a procedure that is invoked upon the occurrence of some event, or the resource exists external to the purview of the function and through some event is first brought under its control. (In the OSP function "Sponsored-Research-Administration," the resource, a sponsored research program, is created at the end of an initiating procedure that is invoked by the receipt of a proposal.) In either case, an initiating event invokes the initiating procedure, at the end of which the resource is in the *quiescent state* the first time. The resource's life ends with *termination*; a terminating event invokes the terminating procedure, which performs

the processing necessary when a resource is destroyed or is otherwise of no further interest.

The form of an OSL function description is a template that provides "slots" for the listing of major events in the life of the resource: those that cause the resource to be created and terminated, and various events that remove it from its quiescent state (*i.e.*, that require some processing to be performed). Along with each event is specified the name of the procedure that describes the required processing.

The following is the function template; each entry is explained in detail below:

```
FUNCTION name
        Resource:
        {Structure:}
        Responsible:
        Initialization:
        {Structure-initialization:}
        Required reports received:
        Required reports generated:
        Other events:
        Termination:
        {Structure-Termination:}
        Quantitative information:
```

A.2.1.1 Resource

The resource managed by the function is some class of entities. Note that while an entity may be a resource of several functions, in practice a function and its associated procedures represent all operations of importance to a single entity class. The resource is specified simply as the name of some class in the environment:

```
Resource: Class name
```

A.2.1.2 Structure

A structure is a framework for organizing resources in some manner. (Several kinds of structures are built-in entity types, such as SCHEDULES (structures of APPOINTMENTS) or PROJECTS (structures of TASKS).) Some functions are concerned both with individual resource instances and a particular structure of those resources. In this case, the structure is defined as the name of a class or an entity in the environment:

`Structure:` *class name*

A.2.1.3 Responsible

"Responsible" is the name of some role defined in the environment. This is the person responsible for the supervision of the function, and, unless otherwise specified, the one to whom all questions are directed and all exceptions are reported. It may be described as a specific instance of some role or as a class of type ROLE.

`Responsible:` [*role*; *instance* ***of some role***]

A.2.1.4 Initialization

The *initiating procedure* for a resource is the one that causes it to be "created" or otherwise brought for the first time into the purview of the function. This procedure results in the resource being in the quiescent state for the first time. The initialization specification includes a description of the event that invokes the initiating procedure, and the name of that procedure.

`Initialization:` *event*: *procedure name*

A.2.1.5 Structure initialization

Where the function includes a structure, the event and procedure that create the structure are also specified:

`Structure-initialization:` *event*: *procedure name*

A.2.1.6 Required reports received

In this section are listed all regularly-scheduled inputs relevant to the resource. These generally take the form of reports, but may in fact be any kind of communication expected on a regular basis. The specification includes the report period; the name of the document or message expected; and the names of two procedures, the first to handle processing when the report is received, and the second (optional) procedure that is followed if the report is not received on time. If the "late report" procedure is omitted, then the default action is to report the problem to the responsible party.

`Required-reports-received:`

`[<`*period*: *communication name*: *procedure name*: *procedure name*`>; None]`

A.2.1.7 Required reports generated

The function may require that periodic reports concerning the resources managed be produced. In this section the period, name of report, and procedure to produce the report, are named for each such required output.

`Required-reports-generated:`
`[<`*period*: *communication name*: *procedure name*`>; None]`

A.2.1.8 Other events

In this section are listed all events that are anticipated but not on any regular schedule. The specification includes the event and the name of the procedure invoked to perform the required processing and return the resource to its quiescent state.

```
Other-events: enumeration of
                    <event: procedure name>
        or
Other-events: None
```

A.2.1.9 Termination

Upon the occurrence of some event, a resource is no longer of interest within the context of a function. At that point, the terminating procedure is invoked to handle any required processing.

```
Termination: event: procedure name
```

A.2.1.10 Structure termination

A structure, if it exists, may have a different terminating condition from any of its constituents. (Consider, for example, the processing required at the end of an appointment, as contrasted with the processing required at the end of a day's schedule of appointments.) The structure termination is specified in the same manner as the resource termination.

```
{Structure-termination: event: procedure name}
```

A.2.1.11 Quantitative information

Quantitative information at the functional level includes: typical number of resources managed; number of personnel responsible for the function (in cases

where volume or other considerations require various people to fill the "responsible" slot at different times or with different resource instances); number of personnel needed to implement the function; frequency of initiation and termination of resources; and expected frequencies of events listed in the "Other Events" subsection.

```
Number of resources: number
Number responsible: number
Number of personnel: number
Initiation: frequency
Termination: frequency
{Other Events:
                <event number. frequency>}
```

A.2.2 Procedures

A procedure is a formal expression of the processing required as the result of some event pertinent to a resource. The purpose of a procedure is to move some entity (its *object*) from an initial to a final state. A procedure normally exists as part of one or more functions; its object is either a resource that a function is managing, some constituent of a resource, or an entity otherwise related to a resource.

A procedure specification consists of several building blocks: identification of the object and the role responsible for overseeing operation of the procedure; a "main line" process describing the normal actions and their ordering; quantitative information about the operation of the procedure; and subordinate processes describing expected variations from the main line, handling of exception conditions, and some details about individual activities.

Two basic principles of office procedures are reflected in OSL procedures. First, they are relatively loosely structured; the ordering information, particularly of the activities within a step, but also of the steps themselves, should indicate the processing steps in the "ideal" case, when nothing goes wrong. A procedure can be

thought of as representing the history of processing of the object in that ideal case. A procedure is thus an indication more of goals than of strict requirements. It is the job of the responsible party to assure that the goals are met; whether the actions in any particular case are, or can be, taken in the specified order is not particularly pertinent. The second principle follows from the first: the main line description, stripped of special cases and exceptions, is basically simple. A typical such process may have only a few steps and states. The complexity of real procedures is expressed in OSL in the structure of variations, details and exceptions.

The "meaning" of a procedure specification is derived from the following model of office procedures, which in turn is derived from the preceding two principles: The procedure provides instructions about which activities should be performed in order to move the object to a final state. Most objects would follow the main line. There are variations to the main line processing that take effect only for objects with particular attribute values. There are exceptional conditions that may occur, making it impossible to follow the normal processing requirements. Upon the occurrence of such an exception, if there are processing instructions associated with that exception, then they are followed until normal processing can be resumed. If no processing is specified for the exception, then the *default action* is to notify the responsible agent of the problem; after he has dealt with the problem, normal processing is continued.

The following is the structure of a procedure specification. Each entry is described in detail below.

```
Procedure name
Object:
Responsible:
Main line:
Timing Constraints:
Quantitative Information:
Variations:
Exceptions:
Detail:
```

A.2.2.1 Focal Object

The focal object (or just "object") of a procedure is some entity in the environment. It is either the same as the resource managed by a function of which the procedure is a part, some constituent of that resource, or an entity somehow related to it.

`Object:` *class name*

A.2.2.2 Responsible

This entry of the procedure template specifies the name of a role that is responsible for the operation of the procedure, and to which all otherwise-unspecified exception conditions be reported. This may be the same role as that responsible for a function of which the procedure is a part, but alternatively may be any role defined in the environment.

`Responsible:` *class name* `[of some role or party]`

A.2.2.3 Main line

The *main line* describes the normal course of processing, or the history of the object in the ideal case. It consists of a set of *events*, *states* and actions that make up a description of the way in which the object is acted upon by the office in order to achieve some goal. The actions are represented by *steps*, which consist of individual

activities that apply to the object. Ordering ordering information is provided by a formalism (related both to state machines and Petri nets) in which *state transitions* are related to specific events that give rise to associated states. A state represents the situation in which further processing cannot be done by the office until the occurrence of an event beyond its control; the "wait" may be due to an outside agent (*e.g.*, proceed upon the receipt of some document), or to the date or time (*e.g.*, proceed on Thursday).

A procedure specification is a description of a *goal* and a set of *requirements* for achieving that goal. It is not a description of any particular *implementation* of those requirements. It is to be interpreted as a set of instructions indicating what should be done and, ideally, in what order; it does not specify exactly how those instructions should be carried out. The goal of a procedure is essentially to move the object to a *final state*, after which no further actions are required.

The main line description has three components: a set of *events*, a set of *steps*, and a set of *states*. These are specified in a set of state/event/step triplets, indicating the *transitions* from one state to another, the event that causes the transition, and the processing required for the transition:

```
state state name
        Event.event number  event specification
        Step.step number
                <activity specification>
```

Each event and step have a unique number within a procedure. Event specifications are discussed in section A.2.3; step specifications are discussed in section A.2.2.3.1. There may be any number of transitions out of a state, each defined by a different event and invoking a different step. A step may terminate in one of several states, depending upon the processing defined within the step; the step specification includes the state in which each branch (if more than one) of the

step terminates. All procedures start in the step named **null** that is exited upon the invoking event. All procedures terminate in the step named **done.**

A.2.2.3.1 Steps

A step is a partially-ordered set of activity specifications. (Activity specifications are described in section A.2.4.) Each step in a process has a unique step number. Within a step, each activity has a prefix number that defines its ordering relative to the other activities within the step. This ordering again represents the sequence in which activities are carried in the "ideal" or normal case. "Partial ordering" means that some activities may be carried out in any order relative to each other, while some *must* follow others. Those activities that may be carried out at the same time or in any relative order have the same prefix number with a unique small letter appended to each. The following illustrates the structure of a step:

```
2.1. Activity1 specification
2.2. Activity2 specification
2.3a. Activity3 specification
2.3b. Activity4 specification
2.3c. Activity5 specification
2.4a. Activity6 specification
2.4b. Activity7 specification
2.5. Activity8 specification
```

In this step (#2), ACTIVITY2 follows ACTIVITY1. ACTIVITY3, ACTIVITY4, and ACTIVITY5 all follow ACTIVITY2, but since they have the same prefix number (2.3) they can occur in any order. (The small letters serve only to identify each activity uniquely.) The activities with prefix number 2.4 (ACTIVITY6 and ACTIVITY7) must follow *all* the activities with prefix number 2.3, but again, there is no ordering specified among them. ACTIVITY8 (prefix 2.5) follows *both* ACTIVITY6 and ACTIVITY7.

A.2.2.3.2 Branching Within Steps

The step syntax also provides for the specification of alternate paths within the step. Such alternates occur when the processing requirements depend upon the value of some attribute that is either not part of the object or is not known *a priori.* (See section A.2.2.6 for a discussion of such *a priori* object variations.) The form for such specifications is:

```
where attribute expression add
        <activity specification>
        {end in state name}
or
where attribute expression end in state name
```

where *attribute expression* has the form

```
attribute = value
        or
attribute expression or attribute expression
        or
(attribute expression) and (attribute expression)
        or
arithmetic function expression
```

and each activity specified has a unique prefix number within the process.

The meaning of the first form of this specification is as follows. For any object, if the attribute expression is true when that object is being processed, then the step for that object includes all the activities specified (and indented) after the *add.* If there is no *end* the step continues with the activity immediately after indentation. If there is an *end* statement, the step does not include any activities after the end, and the named state is reached when all the added activities have been performed. If the attribute expression is false, then the step does not include the added activities.

The second form of alternate path syntax simply specifies that if the attribute expression is true then the step is terminated and the named state reached.

As an example of this syntax, consider the following fragment of an undergraduate admissions procedure:

```
. . .
5.2 Send ADMITTED.Acknowledgment
5.3 where Acceptance = "refuse" add
        5.4 Send E3 to Financial-Aid
        end in done
5.5 Send AAC to Financial-Aid
end in accepted
```

For refused admissions, the step includes activities 5.2, 5.3, and 5.4 and terminates after activity 5.4 in state **done**. For accepted (not refused) admissions, the step includes activities 5.2, 5.3, and 5.4 and terminates in state **accepted.**

A.2.2.4 Timing constraints

A timing constraint is an expression that defines a temporal relationship between two events. Its purpose is to state that some event must occur before, at, or after either an absolute date/time or one defined relative to another event. For example, it might be desired to specify the fact that one event is to occur within six months of another event. Violation of a timing constraint is an *exception condition* (*q.v.*).

The *Timing constraints* entry in a procedure template is used to define any constraints upon the events specified in the main line entry. Each timing constraint in a procedure has a unique number. The form for describing timing constraints is:

constraint# . *event#* *relation* *event expression*

where *relation* is:

[<; >; =]

event expression is

event# *arith* *time*

arith is

[+; -]

and *time* is

[*a specific date/time*; *an interval*]

The following is an example of a timing constraint section:

```
1. Event 2  <  Event 3 + 6 months
2. Event 4  =  October 31
```

In this example, timing constraint 1 specifies that Event 2 is supposed to occur within (less than) three months after Event 3 occurs. If this does not happen, then timing constraint 1 is violated, raising an exception condition. Similarly, Event 4 is supposed to occur on October 31; should it occur earlier, later, or not at all, the constraint is violated.

A.2.2.5 Quantitative information

This section provides a set of figures for various timings and counts in the current implementation of the procedure. While OSL provides suggestions for what kind information might be expressed in this subsection, any quantitative information that is of use in describing the procedure can be described. The numbers listed include the nominal total elapsed time for one complete execution of the procedure; the number of responsible people and the number of people working on the procedure, as in the functional specification; the number of objects in some stage of the procedure at any time; frequency of exceptions; and probabilities of transitions emanating from branch points (either multiple events leading from states, or steps terminating in multiple states).

```
Total time: time
Number responsible: number
Number of personnel: number
Objects: number
Exceptions:
        <exception#: frequency>
Variations:
        enumeration of
        <variation#: probability>
Branching:
        <step# → state name: probability>
        <state name (event#): probability>
```

A.2.2.6 Variations

As noted, the main line expresses the goal structure for processing normal objects. In some cases, there are differences among individual objects that require somewhat different processing; these different processing requirements are called *variations.* Variations apply only to objects that are known to be variants *a priori*; that is, a variation is characterized by some attribute of the object whose value is known at the time the procedure is initiated. A variation then specifies the processing required for that object. For example, in a university admissions office procedure, there might be variations in the normal procedure for those applications (objects) that are from minority, female, or foreign students, and for those applications that request an early decision. (In contrast, event-based and decision-based variability, which stem respectively from multiple possible events invoking different steps in transition from the same state, and from different processing paths determined by a decision made as part of the procedure, are handled in OSL as part of the main line specification.)

The "Variation" entry of the procedure template identifies the object attributes that determine when the variation is to be used, as well as the alternate processing requirements. Since the differences from the main line are generally minor, the variation is expressed as one or more processes that are superimposed upon the main line. Each variation is given a unique number, and both the characteristic attribute value for the variation, and steps, states, and events necessary to handle it are specified. The format is:

variation number.**where** *attribute expression*:
{delete:
{<**Event** *event number*>}
{<**Step** *step number*>} }
{add:
{<*event/step specification*>} }
{replace:
{<*event specification*>}
{<*step specification*>} }

The meaning of the variation specification is that for each object whose attributes satisfy the attribute expression, the steps and events in the "delete" specification are removed from the main line; the states, events and steps in the "add" specification, if any, are added to the main line; and the events and steps in the "replace" specification replace identically-numbered events and steps in the main line. The result of these operations is then treated as if it were the originally specified main line for each affected object.

When there are several variations in one procedure that are not mutually exclusive (that is, that are not defined by different values of the same attribute), they must not replace any of the same steps or events. For example, suppose that there is one variation in the admissions procedure for applications from foreigners, which replaces steps 2 and 4 of the main line. Then a variation for applications from American citizens who are children of alumni could also replace steps 2 and 4. However, consider another variation for applications from females, which replaces steps 3 and 4. Clearly, for female foreigners, the specification of step 4 would be ambiguous; it is therefore not allowed. The appropriate structure for such a situation is to define each variation specifically enough that the ambiguity does not occur. Thus, in our example we would define the "female" variation just for step 3, the "foreign" variation just for step 2, and a "foreign female" variation for step 4. Since the attribute values that define the variations are known at the beginning of the procedure, it is possible to determine then which variations apply to each object.

A.2.2.7 Exceptions

This entry of the procedure template is used to specify processes to be used to handle specific exception conditions. As noted previously, the default action for all exceptions is to suspend the procedure, notify the responsible agent of the exception, and resume processing when that agent has dealt with the problem.

There are two types of exceptions that appear in a procedure specification: *predefined exceptions* and *ad hoc exceptions*. Predefined exceptions are those that are built into OSL; these are defined below. Ad hoc exceptions are those that are not built into OSL, but are identified by the analyst in writing an OSL specification.

A.2.2.7.1 Predefined exceptions

There are three kinds of predefined exceptions in OSL:

Activity-specific exceptions

Each individual activity (*q.v.*) defined in OSL has associated with it a set of named exceptions, each of which identifies a potential problem specific to the activity.

Timing constraint violations

Each timing constraint (*q.v.*) defines an exception that occurs if it is violated.

General exceptions

There are several exceptions defined that cover possible problems with a procedure in general. These include:

- *Missing personnel*: Someone responsible for making a decision is not available.

- *Lost document*: A document containing information that is unavailable elsewhere in the environment has been lost.

- *Backout*: A decision that was made at some step is reversed.

- *Cancellation*: The entire procedure is terminated abnormally.

A.2.2.7.2 *Ad hoc* exceptions

In designing a language such as OSL, it is impossible to anticipate all possible exception conditions. Therefore, the procedure template allows for the specification of *ad hoc* exceptions, those that are identified by office workers or anticipated by an analyst. Each of these exceptions is identified by a unique name, and some procedure for handling it is specified. The general procedure description for exception handling uses the same state/step/event model as the main-line; alternatively, it may be the special procedure DEFAULT, indicating that the default exception action, as described above, is used.

A.2.2.7.3 Exception specifications

The form for the Exception-handling entry is:

```
Timing constraint:
        <timing constraint#: procedure>
Activity:
        <activity#:
                <exception name: procedure>>
General:
        Missing personnel:
                procedure
        Lost documents:
                <document name: procedure>
        Cancellation:
                procedure
        Backout:
                procedure
        Ad hoc:
                <exception name: procedure>
```

where *exception name* is either a built-in activity exception name or a unique name for an *ad hoc* exception.

A.2.2.8 Details

The detail entry provides a place for the expression of more detail about individual activities than is necessary or desirable in the main line procedure. Such detail may involve partial algorithms for making a choice, names of external

procedures (such as statistical calculations) that can be used to perform an activity, or simply another layer of structure expressed in the OSL model or another language.

The format of the detail entry keys each detail to a specific activity. Its purpose is to provide a structure for whatever information about activity implementation is desirable for a given specification. For example:

```
2.3  Select first three if time is critical.
```

A.2.3 Event Specification

An event specification describes the condition upon which the event occurs. Event specifications are used in function specifications to indicate when procedures are invoked, and in process specifications to indicate when steps are invoked. There are several types of events in OSL, each of which has its own form for specification.

Events may be compounded by combining individual event specifications. A *compound event* is defined with the following syntax:

event specification
or
compound event or *event specification*
or
(*compound event*) and (*compound event*)

where an *event specification* is one of the forms (trigger event, time event, environment event, communications event, or activity event) defined in the following subsections.

A.2.3.1 Trigger Event

A trigger event occurs upon the explicit command of an authorized person, and is used in a function specification to indicate that a procedure invocation is dependent upon that person. The form is

by *role*

A.2.3.2 Time Event

A time event occurs upon a specified date or date/time. This time may be specified either in absolute terms (*e.g.*, September 1) or relative to another event. The forms are

on [*date*; *time* after/before *event#*]

A.2.3.3 Environment Event

An environment event occurs when a specified condition in the environment obtains. This condition may be based upon the value of an attribute, or upon some action that changes an entity. The forms are

when *Attribute* = *Value*
or
when *Entity* is [updated; created; deleted]

The first form states that the event occurs when the attribute first attains the given value. The second form states that the event occurs when the specified action is taken.

A.2.3.4 Communication Event

A communication event occurs when a specific communication entity (*q.v.*) is received, or when a communication entity is not received after a specified time. We define a *receipt event* as one of the following forms:

```
receive communication
        or
receive communication with Attribute = value
        or
receive communication matching entity on attribute(s)
        or
receive reply to communication
```

where *communication* is a specified entity of type COMMUNICATION or one of its derivatives (DOCUMENT, MEMO, *etc.*). Then a communication event is one of the following forms:

```
[receipt event; no receipt event after time]
```

where *time* may be either an absolute date/time or a time relative to another event, expressed as an *event expression* (as defined in the *timing constraint* subsection).

A.2.3.5 Activity Event

An activity event occurs upon the initiation or completion of some activity. Activities are identified by their complete prefix number.

```
[complete; start] Activity#
```

A.2.4 Activities

Activities are the fundamental operational constructs of OSL. Although some activities have a more specialized syntax, the general form for an activity specification is:

{subject} activity name {arguments} {modifier} {predicate} {source}

The *subject* names the person or role that is to perform the activity; if omitted, the default is the responsible role specified for the procedure.

Some activities have optional *arguments* that serve to define more precisely the actions that they represent.

The *predicate* is the thing upon which the activity acts. It may be any entity or class; if omitted, the default is the entity specified as the object of the procedure.

Modifiers specify more precisely the predicate to be acted upon, and take one of the following forms:

```
matching on Attribute name(s)
first
last
any
each
```

Each of these modifiers serves to identify a particular member or members of the predicate class: "first" and "last" define particular members of classes that have some ordering defined; "any" specifies an arbitrary single member of the class; and "each" specifies that all members of the class are to be acted upon in the same manner. The "matching" modifier identifies those predicate class members that have (the specified) attribute names and values matching those of the object.

The *source* provides general information about where information is obtained to perform the activity. It may take one of the following forms:

```
using entity
consulting [role: party]
```

Using indicates that the named entity provides useful information for performing the activity. *Consulting* is used to name someone who is used as an information resource for the implementation of the activity.

Each activity has a more specific syntax, and these are described in the subsections below, along with its meaning and the possible exceptions to it. For each activity, the following are given in the definition: the meaning of the activity, the syntax, and the set of activity-specific exceptions and their meanings. Curly brackets ({ }) are used in the syntax descriptions to enclose optional parts of the

specification. Square brackets ([ ]) enclose a set of options from which one is to be chosen.

Activities are divided into four categories, each of which involves a particular type of manipulation of entities or processes:

- **Basic activities**, which are concerned with the existence of classes and entities, and their attribute values;
- **Decision activities**, which are concerned with various decisions that pertain to entity instances;
- **Aggregate activities**, which are concerned with decisions that pertain to groups of entity instances;
- **Control activities**, which are concerned primarily with the control structure of a procedure, and are most commonly used in exception-handling processes.

A.2.4.1 Basic Activities

A.2.4.1.1 Create

Create a new instance of a given class. This activity has the effect of adding a new entity (such as a document, a program, an employee), with all appropriate attribute values, to the environment. The Detail subsection may be used to specify how the values for the attributes are to be set.

`Create` *class name*

Exceptions:

repeated unique attribute value
: an attribute of the created entity that has the "unique" characteristic was set to a value already used by another entity in the class.

bad attribute value

an attribute of the created entity was set to a value that was not in the value class specified in the class definition.

can't create

the instance cannot be created for any reason (*e.g.*, a document instance is to be created but there are no forms available).

A.2.4.1.2 Delete

Remove an entity permanently from the environment. Examples might be the destruction of a document, the deletion of a completed project, the termination of an employee. Any attempt to reference the entity after it is deleted is an error.

`Delete` *instance*

Exceptions:

doesn't exist

the named instance does not exist in the environment

A.2.4.1.3 Set

Change a (primitive) attribute value. This activity is used whenever a primitive entity is changed, reflecting a change in the environment. (Note that values of derived attributes change without explicit action.) Attributes that do not have the "mandatory" characteristic may have the value "unknown"; SET is used when the value becomes known.

`Set` *Attribute* `= value`

Exceptions:

bad value

the value to which the attribute was set is not a member of the value class specified in the class specification.

A.2.4.1.4 Calculate

Perform a specified mathematical operation on the attributes of an entity or class. The particular expression may be specified; alternatively, the name of a (non-OSL) procedure to perform the operation may be specified.

```
Calculate attribute = formula
        or
Calculate attribute = routine name(routine argument(s))
```

Exceptions:

error Some error occurred in carrying out the calculation

A.2.4.1.5 Revise

Review an existing entity and change the value of a TEXT attribute. This activity is used in situations where some significant unstructured (text) portion of an entity must be changed, usually as part of a rewriting action.

```
Revise attribute
```

Exceptions:

none

A.2.4.1.6 Archive

Place some information about an entity in an archive file. The attributes of the archive file, which is defined in the environment, specify what information is to be saved. If no archive is specified, then the archive specified in the "Archive" attribute of the entity is used.

```
Archive {entity} {in archive}
```

Exceptions:

nonexistent entity
The specified entity does not exist.

nonexistent archive

The specified archive does not exist.

A.2.4.1.7 Send

Transmit an entity to another location; the entity is generally of type COMMUNICATION or a derivative. (It is meaningless to Send an abstraction, or to transmit an organization). A destination is any entity of type PARTY or ROLE (or derivatives) defined in the environment; if not given explicitly in the activity specification, the destination is the value of the "To" attribute of the entity. A list of destinations may be specified, in which case the activity is equivalent to a number of SEND activities, each with a single destination, in any order.

Since communications entities always refer to some more fundamental entity (see the discussion in the Environment section), the notion of a "copy" has a slightly different meaning here than in usual practice. When an approved entity (*q.v.*) is to be transmitted, then the approval characteristic is also transmitted; this is normally implemented by sending a physical, signed "original." The SEND activity allows the specification of "copy" transmittal. Such a copy is identical to an original except that the value of the approved attribute(s) is "copy"; the meaning is that this is a copy of which the original is authorized. A copy of an unauthorized entity would be identical to the "original"; the value of the approved attribute is either "unapproved" or "unknown."

`Send {(copy)} {`*entity*`} {to` *destination(s)*`}`

Exceptions:

nonexistent entity

The specified entity does not exist.

wrong destination

The specified destination does not exist, or is otherwise incorrectly specified.

communications failure

The transmittal cannot be accomplished for any other reason.

A.2.4.1.8 Add

Include an entity in a specified subset. A subset (*q.v.*) is a subclass whose members are only those added to the subset by an ADD activity. The class named in the activity specification must be defined in the environment as a subset.

`Add` *entity* `to` *class*

Exceptions:

incorrect operation

The specified entity is not a member of the parent class of the specified subclass.

subclass overflow

Adding the entity to the subclass violates a constraint on the size of the subclass.

A.2.4.1.9 Remove

Remove an entity from a specified subset.

`Remove` *entity* `from` *class*

Exceptions:

wrong entity

The specified entity is not a member of the specified subclass.

subclass underflow

Removing the entity from the subclass violates a constraint on the size of the subclass

A.2.4.2 Decision Activities

A.2.4.2.1 Approve

Approval is the authorization or sanction of an entity by a person legally entitled to do so; it is the abstraction of which a signature is the most common implementation. The APPROVE activity has the effect of changing the value of the specified attribute from whatever it was ("unknown" "unapproved" "approved" "conditionally approved") to "approved." The attribute must be of type APPROVAL (*q.v.*). If no attribute is specified, then the entity must have only one attribute of type APPROVAL.

Approve *attribute* {*entity*} {**by** *role*}

Exceptions:

not approved
: The specified person cannot or will not approve the entity. This is considered an exception because the appearance of an APPROVE activity in a process specification means that in normal operation the approval will occur.

unavailable
: The required person is unavailable.

nonexistent entity
: The specified entity does not exist.

nonexistent attribute
: The specified attribute does not exist.

A.2.4.2.2 Verify

Confirm the correctness of information. Normally used with communications entities to check that field values are consistent and do not violate any constraints, this activity may also be used to represent verification that all components of an

abstract entity are present and satisfy any constraints, or that the value of some attribute is correct.

`Verify` {*attribute(s)*} {*entity*}

Exceptions:

not verified
: There is something wrong with the entity or attribute being verified.

nonexistent entity
: The specified entity does not exist.

nonexistent attribute
: The specified attribute does not exist.

A.2.4.2.3 Evaluate

Examine an entity (or some physical representation or aspect of it) and record findings by setting a specified text attribute. The attribute must be of type EVALUATION. A role indicating who is to do the evaluation may be specified.

`Evaluate` {*attribute(s)*)} {*entity*} {`by` *role*}

Exceptions:

nonexistent entity
: The specified entity does not exist.

nonexistent attribute
: The specified attribute does not exist.

unavailable\The person who is to do the evaluation is not available.

unable to evaluate
: The person who is to do the evaluation cannot do so for any reason (other than absence).

A.2.4.2.4 Negotiate

Come to an agreement (with one or more other parties) about some aspect of an entity; the agreement is reflected in the value(s) of the relevant attribute(s). This activity reflects the common situation in which a decision must be reached, but a single person does not have the required information and/or authority to make the decision himself. The others involved in the negotiation may be either of type PARTY or ROLE.

`Negotiate` *attribute* `with [`*role(s), party(s)*`]`

Exceptions:

nonexistent attribute
: The specified attribute does not exist.

nonexistent party
: One of the roles or parties specified does not exist.

unavailable
: One of the roles or parties specified is unavailable.

unable to agree
: The negotiation has failed to produce a decision.

A.2.4.3 Aggregate activities

A.2.4.3.1 Select

Create or add to a subset by picking one or more entities from a specified class; SELECT indicates that more entities are available than are needed. The first class name in the activity specification identifies some class in the environment that is defined to be a subset. The class from which the subset is selected must be either the parent of the subset class (in which case the parent name can be omitted) or

another subset of the same parent. If *n* is omitted, it is assumed to be one; it specifies the number of entities to be selected.

```
Select {n} class {from class}
```

Exceptions:

nonexistent class
: One of the specified classes does not exist.

insufficient
: There are not *n* entities in the "from" class.

unable
: The selection cannot be made for some reason (other than insufficient entities)

A.2.4.3.2 Allocate

Distribute entities from a given class among several subsets; ALLOCATE indicates that fewer entities are available than are needed. The operation is to take some number (*n*) of members of the "from" class, and add each of them to one of the "to" subsets. If *n* is omitted, then all the members are allocated. The "to" subsets must be defined in the environment to be subsets of the "from" class.

```
Allocate {n} from class to classes
```

Exceptions:

nonexistent
: One of the specified classes doesn't exist.

insufficient
: There are not *n* members of the "from" class to be allocated.

A.2.4.3.3 Group

Partition the members of a class into subsets.

`Group` *class* `into` *subsets*

Exceptions:

none

A.2.4.4 Control activities

A.2.4.4.1 Notify

Inform someone of an exception condition. (Note that if the person to be notified is the person responsible for the procedure, then the activity specification is equivalent to the default exception-handling process.)

`Notify` {*role*}

Exceptions:

nonexistent
: The specified person does not exist.

unavailable
: The specified person is unavailable.

A.2.4.4.2 Retrigger

Reset a timing constraint. This causes the timing constraint to be set for the specified time in the future.

`Retrigger` *constraint number* + *time*

Exceptions:

none

A.2.4.4.3 Initiate

Cause a procedure to be invoked.

`Initiate` *procedure name*

Exceptions:

nonexistent
: The procedure specified is nonexistent.

A.2.4.4.4 Terminate

With a procedure name as object, cause all processing associated with the named procedure to stop, regardless of its current state. (This will cause an exception in the halted procedure.) Without an object, terminate the current step.

`Terminate` {*procedure name*}

Exceptions:

nonexistent
: The procedure specified is nonexistent.

nonactive
: The procedure specified is not active.

A.2.4.4.5 Perform

Initiate another OSL procedure; this differs from the INITIATE activity in that a PERFORM activity is not finished until the named procedure reaches its final state, whereas an INITIATE activity is finished as soon as the named procedure is started.

`Perform` *procedure name*

Exceptions:

unable The named procedure cannot be completed, for any reason.

A.2.4.4.6 Return

This activity is used to specify that an entity is to be sent back to the party, role, or organizational unit from which it came. RETURN is used primarily in processes that handle exceptions to the VERIFY activity when an *approval* attribute is being verified. In such a case, the defaults for the RETURN activity are that the unapproved entity is to be returned to whoever did not approve it.

```
Return {entity} {to entity}
```

Exceptions:

none

A.2.4.4.7 Repeat

REPEAT is used in an activity-specific exception handler to indicate that the activity is to be repeated *n* times before any further exception condition is raised. If *n* is omitted, it has the value 1.

```
Repeat {n}
```

Exceptions:

none

A.3 Environment Specifications

This section describes the components of an OSL environment specification, including definitions of the built-in entity types.

A.3.1 Overall Structure of an Environment Specification

The environment specification is divided into two major parts: "Identifications" and "Definitions." All class definitions in an environment are organized in alphabetical order, in the Definitions part, for easy reference. Preceding these definitions, the Identifications part is a summary of all the class names and their types. This summary is organized in a particular order, with the following outline:

Organizational context

Classes describing the relevant aspects of the organization of which the office is a part. Includes:

- The instance definition for the organization itself.
- The organization hierarchy and personnel hierarchy instance tables.
- Class type descriptions of other relevant aspects of the organization.

External context

Class type descriptions of items external to the organization that are of interest to the office being described.

Internal context

Classes describing relevant aspects of the office itself. This includes special subsections providing separate identification of the class names for the following kinds of classes:

- Documents, communications and their derivatives
- Names

A class type description is one of the following forms:

class-name is [*class-type*; *class derivation*]
or
class-name = *class-name*

where *class-type* is the name of a built-in or derived entity type, and the "=" represents an alias definition. Each of these items is described in the following sections.

A.3.2 Class Definitions

An OSL environment is not just a collection of entities. Rather, the relevant entities in the office's "world" are organized into classes, each representing a set of entities of the same type. There are two kinds of classes in an OSL description: built-in classes and derived classes (defined relative to some other class(es) in the environment). For example, the built-in entity type EMPLOYEE implicitly defines the built-in class EMPLOYEES; we may define a (derived) class of MIT-EMPLOYEES of type INTERNAL-EMPLOYEES; subsequently, we may define the class OSP-EMPLOYEES as a particular subclass of MIT employees (*i.e.*, by the value of some attribute).

There is also a kind of class whose members are of the special built-in type *name*. Names serve to represent other entities, and take the form of numbers or strings of characters. There are several built-in name classes in OSL; name classes may also be defined as required to specify the environment.

Each class has a *class name*, which must be unique within the environment. A class may also have an *alias*, a second unique name by which the class is known; the alias is usually shorter than the class name, and is used only for convenience.

A.3.2.1 Built-in Classes

Each built-in entity type defines a built-in class consisting of all entities of that type. Since these classes are defined as part of OSL, they do not appear as part of any environment definitions. They may be used, however, as the value class of any attribute. See the subsection A.3.5 for definitions of these items.

A.3.2.2 Derived Classes

A derived class is one that is defined in terms of other classes. Each member of a derived class is also a member of one or more of the classes from which the derived class is defined. A class in terms of which a derived class is defined is called the derived class's *parent class*. A derived class may have one or more parent classes, each of which may be a built-in or a derived class.

There are several kinds of possible class relationships; each kind is reflected in one of the possible means of defining a derived class. *Restriction* and *subset* provide for derivation of a class in terms of a single parent; *merge-members*, *common-members*, and *missing-members* provide for derivation in terms of multiple parent classes.

A.3.2.2.1 Restriction

A restriction defines a class as consisting of all members of the parent class that have a particular attribute value or set of values. There are two forms for defining restriction classes. One is used only when the parent class is a built-in class:

class name { = *alias* } is *parent class*
<*attribute specifications*>

Some of the attribute specifications are "type attribute restrictions"; these are specifications of type attributes (defined in subsection A.3.3.3.1) in the form

attribute name: *value class*

where the *value class* is either the class defined as the value class of the attribute in the built-in type definition (subsection A.3.5), or a defined subclass (restriction or subset) of that value class.

The second form for defining a restriction class can be used with a built-in or derived parent class:

class name { = *alias* } is *parent class*
where <*attribute restriction*>
<*additional attributes*>

An attribute restriction has one or more of the following forms:

predicate
(*restriction*)
not *restriction*
restriction or *restriction*
restriction and *restriction*

where a *predicate* has one of the following forms:

chain comparator chain
chain comparator constant
is a value of *attribute name* of *class name*

A *chain* is a form of reference (described in Section A.3.3.2.2) and a *comparator* is one of:

=
~=
>
<
>=
<=
is in
is not in
contains
does not contain

and a *constant* is any number or a string of characters surrounded by double quotes (" ").

The attributes of the derived class and its members include all those of the

parent class and its members. These *inherited attributes (q.v.)* need not be included in the definition of the derived class (unless they are part of an attribute restriction specification); only attributes that apply only to the derived class and its members need be specified.

The derived class provides a means of defining interesting subclasses of a given class, and allowing additional attributes to be assigned to the entities in that restriction. Note that entities that are members of the restriction class are simultaneously members of the parent class, and that membership of any member of the parent class in a restriction class can always be determined by referring to the value(s) of its restricted attribute(s).

An example of a restriction class:

```
ACCEPTANCE-LETTER is LETTER
        where Result = "admit"
        Reply: LETTER-REPLY
```

This example defines the class of research coordinators as consisting of all the jobs (roles) in the OSP whose name is "R-C"; the parent class OSP-JOBS is itself a restriction of the class MIT-JOBS. Each research coordinator job (member of the class RESEARCH-COORDINATORS) is characterized by the set of attributes that defines all OSP jobs (which in turn includes the set of attributes that defines all MIT jobs), as well as by an additional attribute indicating which kinds of programs each research coordinator is responsible for.

A.3.2.2.2 Merge-members

The merge-members derivation defines a class as containing all the members of each parent class. (This is equivalent to a set-theoretic union operation). The format for such a derivation is:

class name is
merge-members in <*parent class name*>
<*additional attributes*>

An example of a merge-members derivation is:

```
REVIEWER
        merge members of FACULTY, ADMISSIONS-STAFF
```

This example defines researchers as being any faculty or senior staff.

A.3.2.2.3 Common-members

The common-members derivation defines a class as containing only those entities that are members of all of the parent classes. (This is equivalent to a set-theoretic intersection operation). The format for such a derivation is:

class name is
common-members of <*parent class name*>
<*additional attributes*>

An example of a common-member derivation might be:

```
ADMISSIONS-FACULTY
        common-members of ADMISSIONS-STAFF, FACULTY
```

This example defines the class whose members both hold faculty positions and work in the admissions office.

A.3.2.2.4 Missing-members

The missing-members derivation defines a class as containing all those entities that are members of one parent class but not of another. (This is equivalent to a set-theoretic difference operation.) The format for such a derivation is:

class name is
missing-members of *parent class name 1* not in *parent class name 2*
<*additional attributes*>

An example of a missing-members derivation might be:

```
ADMISSIONS-NON-FACULTY
        missing-members of ADMISSIONS-STAFF not in ADMISSIONS-FACULTY
```

defining the class of all employees of the admissions office who are not faculty.

A.3.2.2.5 Subset

A subset class derivation is similar to a restriction class derivation in that it defines the derived class in terms of a single parent. A subset simply defines a subclass whose membership is determined specifically by an activity; this is in contrast to a restriction derivation, in which the membership of the derived class is defined *a priori* in the class specification. The subsection on Activities describes those activities that add members to and remove them from subsets. The format for a subset class definition is:

class name
        subset of *parent class name*
        <*additional attributes*>

An example of a subset would be a class WAITLISTED-APPLICANTS, defined as a subset of class APPLICANTS, and representing those applicants who were explicitly placed in the class WAITLISTED-APPLICANTS. In this case, no attribute of a member of the class APPLICANT would indicate *a priori* whether that applicant is or is not also a member of WAITLISTED-APPLICANTS, although a derived membership attribute may be defined for that purpose (see Section A.3.3.4.2.)

A.3.2.2.6 Redundancy in Class Derivations

There are two related issues that are of significance in using the various derivation facilities. First, the merge-members, common-members, and missing-members derivations make sense only when all the parents have reasonably similar member types. It would probably make no sense, for example, to merge a class whose members are of type EMPLOYEES with one whose members are of type

AGREEMENT; therefore, OSL requires that the parent classes of multiple-parent derived classes be subclasses of the same built-in type.

Second, because many classes may be derived from a single parent, it is often the case that there is a choice in how to derive a particular class. (In the missing-members example above, the same class of admissions non-faculty could have been derived by a restriction of ADMISSIONS-STAFF, by a common-members derivation of ADMISSIONS-STAFF and MIT-NON-FACULTY, or several other derivations. All these derivations would result in the same derived class.) This redundancy is provided in OSL to allow the analyst to choose the derivations that most closely match the office he is modeling. All equivalent derivations are "correct"; the best is the one that most reflects the environment.

A.3.2.3 Name Classes

Names are special entities that serve as representations, rather than as descriptions of actual entities; name entities have no attributes other than a value. They are thus the "lowest level" of the OSL environment specification mechanism. All name classes are derived from the built-in type TEXT, which includes anything that can be written with the standard character set (including the digits 0-9).

There are two basic means of name class definition. The first is *enumeration*, in which all possible values of the name are specified:

class name **is NAME**
{*value*, *value*, *value*, . . .}

The second is by derivation from any other name class(es), using the same derivation mechanisms described in Section A.3.2.2. A name class may also be defined by a combination of these methods.

A.3.3 Attributes

The attributes of an entity (or class) describe the properties of the entity (class), and may serve to indicate relationships to other entities (classes) in the environment. All attributes are defined using the same syntactic mechanisms. The syntax of attribute definition is:

Attribute name: *value determinant* {(*characteristics*)}

The *value determinant* describes either the class from which the value may come, for a *primitive attribute* (*q.v.*), or the way in which the value is derived, for a *derived attribute* (*q.v.*). Characteristics apply only to primitive attributes, and designate certain special properties of the attribute's possible values (see Section A.3.3.4.1).

A.3.3.1 Attribute Names

An attribute name must be unique within its class. By convention, all attribute names are written in small letters with initial capitals

A.3.3.1.1 Hierarchical names

Attribute names may be *hierarchical*, that is, an attribute name may refer to several component attributes. For example,

```
Employee-id:
        Last-name: TEXT (mandatory)
        First-name: TEXT (mandatory)
        Middle-initial: TEXT
        ID#: INTEGER (unique)
```

Reference may be made either to the entire hierarchy ("Employee-id" in this case) or to any of its components. A component reference is formed by concatenating the names of each level of the hierarchy, separated by periods, *e.g.*, "Employee.Middle-initial".

Hierarchical attributes need not have names for all the components, if the hierarchy is always referred to as a unit. An example of such an attribute is:

```
Fiscal-approval:
        APPROVAL by AD-Resp
        APPROVAL by RC-Resp
```

A.3.3.1.2 Alternate names

An attribute may have more than one name, any of which serves to identify the attribute. Alternate names are indicated by the following form:

Attribute name: *Attribute name*: *value determinant*

This capability is most useful when it is desired to attach an alternate identifier to a type attribute. For example, the class PURCHASES in Appendix I is of built-in type TRANSACTION (*q.v.*), which has a type attribute "Party1"; the following attribute definition provides for a more convenient name for the attribute, which can be referred to as either "Party1" or "Purchaser":

```
Purchaser: Party1: SPONSORED-RESEARCH-PROGRAMS
```

A.3.3.2 References

References to classes, entities, and attributes must be made for various purposes in an environment specification. The following describes the formats used for referencing these items.

A.3.3.2.1 References to Classes

Since classes are uniquely named within an environment, the class name serves as a sufficient reference to the class.

A.3.3.2.2 References to Attributes; Chains

Within a class, attribute names are unique; therefore, when referring to an attribute within its class, as, for example, in defining the derivation of another attribute (*q.v.*), the attribute name is sufficient.

When referring to an attribute of an arbitrary class, the name of the attribute is concatenated with the name of the attribute. (For example, since the form "MIT-EMPLOYEES" refers to the set of members of the class named MIT-EMPLOYEES, the form "MIT-EMPLOYEES.Supervisor" refers to the set of "Supervisor" attributes of the members of class MIT-EMPLOYEES.)

These concatenated formats are examples of a general reference format called a *chain.* Since the value of an attribute is an entity of some class, it may have attributes of its own. It is possible to reference an attribute of an entity that is the value of an attribute of an entity, *ad infinitum.* Thus, the set of supervisors of employees would be referred to as "MIT-EMPLOYEES.Supervisor"; the set of supervisors' names would be "MIT-EMPLOYEES.Supervisor.Name"; the set of supervisors' supervisors' names would be "MIT-EMPLOYEES.Supervisor.Supervisor.Name"; for a given member of MIT-EMPLOYEES, his supervisor's supervisor's name would be "Supervisor.Supervisor.Name" and so on.

A chain is defined to be either the name of a class, or the name of an attribute, or the name of a class followed by any number of attributes, or the names of any number of attributes, concatenated together in any length, so long as the chain follows some path defined in the environment.

A.3.3.2.3 References to Entities

In order to refer to a particular member of a class (rather than the class itself, or an arbitrary entity), some means must be available to identify the particular member of interest. Such identification is made possible by the use of "unique attributes" (*q.v.*). By specifying the value of a unique attribute, a unique member of a class is designated:

class-name(*unique-attribute-name*=*value*)

For example:

```
MIT-EMPLOYEES(Id="012-34-5678")
```

Note that such a reference may be used in a chain:

```
MIT-EMPLOYEES(Id="012-34-5678").Supervisor.Supervisor.Name
```

would refer to one specific value, rather than the set of names described above.

A.3.3.3 Kinds of Attributes

There are three kinds of attributes, distinguished by their applicability: member attributes, common attributes, and class attributes.

A.3.3.3.1 Member attributes

Member attributes are those that describe an aspect of individual members of a class. (For example, the name of an employee). This is the default attribute kind, and an attribute that is not otherwise specified will be a member attribute.

There is a special kind of member attribute called a *type attribute.* Type attributes are those attributes that are associated with the definition of the built-in type (as shown in subsection A.3.5).

A.3.3.3.2 Common Attributes

Common attributes describe some aspect of individual members of a class that has the same value for each member of the class. (For example, the value of the attribute "Organization" for all members of the class MIT-EMPLOYEES is "MIT".)

A common attribute is specified by placing the word "common" in the characteristics subsection of the attribute specification.

A.3.3.3.3 Class Attributes

Class attributes describe properties of the class itself, not its individual members. (For example, the class ADMISSIONS-STAFF might have a class attribute "Number" to indicate the number of members of the class.)

A class attribute is specified by placing the word "class" in the characteristics subsection of the attribute specification.

A.3.3.4 Attribute Values

Each attribute has a value. This value may be one or more entities, or an entire class. The value determinant part of the attribute specification indicates the possible values of the attribute, and may include the means by which that value is derived. If just the value class of the attribute is specified, the attribute is a primitive attribute; a derived attribute specification includes a derivation expression describing how the attribute value is found.

A.3.3.4.1 Primitive attributes

Primitive attributes are those whose values may be any member of the value class (or any members, in the case of a multiple-valued attribute). Except in the case

of mandatory attributes (see below), the value of a primitive attribute may also be the special value "unknown." Primitive attributes are specified in the following form:

Attribute name: *value class* (*characteristics*)

The value class may be the name of a built-in class (*i.e.*, the class of all members of some built-in entity type), or of any class defined in the environment.

Primitive member attributes may have a number of independent characteristics, each indicated by placing the appropriate keyword in the parentheses following the value class. If no characteristics are provided, the parentheses are not needed. These characteristics include:

common Designates a common attribute (*q.v.*). "common" and "class" are mutually exclusive.

class Designates a class attribute (*q.v.*). "class" and "common" are mutually exclusive.

unique Designates a unique attribute, indicating that each member of the class must have a different value for that attribute. If several attributes of an entity are unique, any one may serve to identify a particular member of the class. If a combination of attributes together provides a unique identifier for a member, those attributes should be defined as the second-level attributes of an hierarchical attribute(*q.v.*), whose top level would have the "unique" characteristic.

multiple Designates a multiple-valued attribute. The value of a default (single-valued) attribute is a member of the value class of the attribute, while the value of a multiple-valued attribute is a subclass of the value class. For example, the "Address" attribute of an MIT-EMPLOYEE has a single value taken from the class ADDRESSES; the "Role" attribute has as its value a subclass (of arbitrary size) of the class MIT-JOBS. This definition corresponds to a rule in the MIT environment that an employee

has one address, but may fill several roles (Teacher, Researcher, Committee member, *etc.*)

mandatory

Designates a mandatory attribute, one that cannot have the value "unknown".

restricted

Designates a restricted attribute, one that can only be changed by a specified person or persons. The form for a restricted attribute is

{(by <*person*>)}

For the special case of primitive type attributes, the value class must be one of the following (The value class of the type attribute "Role" of class MIT-EMPLOYEES (which is of built-in entity type EMPLOYEES) is used in the examples. Note that the definition of EMPLOYEES indicates that the value class of the type attribute "Role" is ROLES):

- the built-in class defined as the value class of the attribute in the type definition (*e.g.*, the value class of the "Role" attribute of MIT-EMPLOYEES could simply be ROLES).

- some derived class whose members are of the type of the value class of the attribute defined in the type definition, or some subclass of that class (*e.g.*, the value class of the "Role" attribute is the derived class MIT-JOBS, which is of type ROLES; it could also be any class derived from MIT-JOBS, or any other built-in class of type ROLES.)

A.3.3.4.2 Derived attributes

A derived attribute is defined by one of several types of derivation expressions that indicate how the value of the attribute is derived from other information in the environment. Such a derivation may take several forms:

- A derivation may be defined in terms of values of other attributes of the

entity. Such a specification may include a chain; the general form for such a specification is

attribute name: *chain*

For example, the value of the "Program" attribute of CONTRACT-CHANGE entities is defined as

```
Program: Contract.Program
```

indicating that the value is the same as the value of the "Program" attribute of the (GRANT/CONTRACT entity that is the value of the) "Contract" attribute of the CONTRACT-CHANGE.

- A derivation may use the class derivation mechanisms *restriction*, *merge-members*, *common-members*, or *missing-members* applied to classes or chains in the same way as derived classes are defined:

Attribute-name: `restrict` *chain* `where` *relation*
Attribute-name: `merge-members in` *chain*
Attribute-name: `common-members in` *chain*
Attribute-name: `missing-members in` *chain* `not in` *chain*

- A derivation may be an *inversion derivation.* An inversion on an attribute A of entity E indicates that the value of the inverted attribute for each member is the set of all entities E that have E as the value of their attribute A. The syntax of an inversion is

attribute name: `invert` *attribute name* `of` *chain*

For example, consider the class MIT-ACCOUNTS, having an attribute defined as

```
Source: invert Account of SPONSORED-RESEARCH-PROGRAMS
```

Then the value of the "Source" attribute of a given MIT-ACCOUNT, say the one identified by Account# ="12345", would be the set of all SPONSORED-RESEARCH-PROGRAMS that have "12345" as the value of their "Account" attribute.

- A derivation may be as a test of membership in a subset (*q.v.*). A membership attribute has a boolean value, whose value depends upon whether the entity is a member of some subset. The form for a *membership derivation* is

attribute name: `if exists in` *class name*

- A derivation may be defined as an ordering within the class. An ordering derivation is specified as

  attribute name: `order by [increasing` *or* `decreasing]` *chain*

  indicating that the value of the attribute is the number assigned to the member based upon an ordering upon the value of the chain. The value of such an attribute will of course change as entities are created and destroyed.

- A derivation may be defined as a *matching derivation.* The value of the derived attribute is the same as the value of a specified attribute of the entity that *matches* the current entity in a specified attribute. For example, consider a (hypothetical) attribute "OSP-Project-Nos" of FACULTY, representing the set of account numbers for all the sponsored research programs for which the faculty member is the principle investigator. Then the attribute derivation would be defined as

  ```
  OSP-Project-Nos:  Account of matching SRP.Pr-Inv
  ```

  The value of "OSP-Project-Nos" for a particular member of the FACULTY class is found by first finding the set of SPONSORED-RESEARCH-PROGRAMS (SRP) whose value of "Pr-Inv" is the FACULTY of concern; the value of "OSP-Project-Nos" is then the set of values of the "Account" attributes of those SPONSORED-RESEARCH-PROGRAMS.

- A derivation may be defined as a *calculation derivation*, indicating that some arithmetic calculation on one or more values or classes. The following are forms of possible calculation derivations:

  `maximum of` *chain*
  `minimum of` *chain*
  `average of` *chain*
  `sum of` *chain*
  `number of members in` *chain*
  chain `+` *chain*
  chain `-` *chain*
  chain `*` *chain* ***(multiplication)***
  chain `/` *chain*

A.3.3.5 Special Attribute Types

There are a number of attribute types used for special purposes. They are described in the following subsections.

A.3.3.5.1 Refers and Same Attributes

A refers attribute is used to indicate that the entity being described exists to store, format, or transmit information about some other entity. It is therefore used primarily with communication entities, which generally are of interest only insofar as they refer to some more abstract entity. This relationship is indicated by the use of the attribute "Refers"; the value of the attribute is the name of the class to which the communications pertains:

```
Refers: chain
```

A Same attribute is used to indicate that the named attributes are identical to (both name and value) those of the same name in the referred entity:

```
Same:  <attribute name>
```

An example of the use of these attributes is shown in the following class definition:

```
ACCOUNT-ACTION-NOTICE =  F001 is DOCUMENT
        where Refers: SPONSORED-RESEARCH-PROGRAM
        To: "Sponsored-accounting", "Keypunch", Pr-Inv, Dept/Lab.Head
        File: OSP-Master-File
        Same: Account#, Budget, Sponsor, Start-date, Title
        Date-issued: DATE
        Type: F001-TYPE
```

Thus, each member of the class ACCOUNT-ACTION-NOTICES refers to some member of the class SPONSORED-RESEARCH-PROGRAMS, and their attributes "Account#" have the same value.

A Same attribute may also specify that attributes of another attribute value may be copied:

Same as *attribute name*: <*attribute name*>

A.3.3.5.2 Dependent attributes

A dependent attribute is one whose derivation depends directly upon the value of another attribute. The syntax for a dependent attribute is

attribute name:
<if *attribute* = *value* then *derivation*>

The "Fiscal-Approval" attribute of a RESEARCH-PROPOSAL-SUMMARY entity includes a dependent attribute derivation.

A.3.3.6 Attribute Inheritance

Some attributes in OSL are *inherited.* In general, member attributes are inherited, while class attributes are not; common attributes are sometimes inherited. Of course, attributes characteristic of a derived class may be defined, but they always inherit some attributes from parent classes.

The type attributes (*q.v.*) of a derived class are inherited from the definition of the built-in entity type.

The attributes of a derived class are inherited from the parent(s) according to the following rules:

- For single-parent derivations, all the member and common attributes of the members of the parent class are attributes of members of the derived class.

- A merge-members class inherits all the member attributes that are shared by the parents. No common attributes are inherited.

- A common-members class inherits all the member attributes of all of its parents. No common attributes are inherited.

- A missing-members class inherits all the member and common attributes of its first parent. Thus, for a derivation

```
missing-members in A but not in B
```

all the member and common attributes of A would be inherited.

A.3.4 Defining Entity Instances

A class specification defines a homogeneous set of entities of a given type—the class's members. In some cases, it is desirable to define a single instance of some entity type, whether a member of some class or not. OSL provides several means for defining specific entity instances. One is via an explicit CREATE activity (*cf.*) in a procedure specification. The others are static mechanisms that are part of the environment specification.

A.3.4.1 Instances of Built-in Entity Types

The more general method of defining an instance follows the syntax of the definition of a class or entity type:

```
Define Instance of built-in entity type:
        <Type attribute name = value>
```

This syntax simply allows the definition of an instance of any kind of entity type; each type attribute is given a value, which must be of the value class of the attribute defined in the type definition. The instance is given a name, which must be unique among class and instance names; in this manner, instances that may not have unique attributes defined can be referenced.

Consider the following example:

```
Define Instance of ORGANIZATIONS
        Name = "MIT"
        Address = "77 Massachusetts Avenue
                  Cambridge, MA 02139"
        C-E-O = "President"
```

This defines an instance of type "organization" named "MIT." The instance definition provides a specific value for each (type) attribute of organizations: "Name" is given a value of class TEXT; "Address" of class ADDRESS; and "C-E-O" of class MIT-JOBS, which is a class of type INTERNAL-ROLES.

A.3.4.2 Defining Class Members

A similar syntax allows for the definition of a particular member of a class already defined in the environment:

```
Define Member of class name:
        <list of Attribute name = value>
```

In this case, a value is specified for each non-optional, and for any optional, primitive attribute of the entity. Each value must be a legal value of the attribute's specified value class. The value may be specified as an independent value:

class name(key values)

or as a derived value using any of the normal attribute derivations. (Note that the instance must be of an entity class; name class instances are defined by the class definition.)

A.3.4.3 Tabular Instance Definition

The final means of specifying instances is *tabular*, and is used primarily in defining environmental networks as well as logs and other inherently singular entities. The tabular format simply lists a few critical attributes as columns, and individual entities and are represented by rows of attribute values.

An an example, consider the following personnel hierarchy:

Name	Organizational-unit	Reports-to	Current-holder
President	Office-President	Chairman	P. Gray
Provost	Office-Provost	President	F. Low
VP-Fin-Ops	Office-VP-Fin-Ops	President	S. Cowen
Director-OSP	OSP	VP-Fin-Ops	G. Dummer
R-C	OSP	Director-OSP	several
A-D	OSP	Director-OSP	several

This is a tabular definition of several members of the class MIT-JOBS, including values for four key attributes.

A.3.5 Built-in Entity Types

This subsection defines all the built-in entity types. Each definition includes the attributes that are part of each entity, along with the class from which the values of the attribute may be drawn.

OSL includes in its built-in entities several broad categories of interest in a typical office environment. These include various kinds of organizational divisions, people, jobs, documents, *etc.*, as well as more abstract notions such as agreements, schedules, and so on. While it is hoped that the built-in entity types will adequately support most applications, it will sometimes be impossible to find one that "fits." In such a case, the generic type ENTITY can be used to define a built-in class. In the following, we will use the term "entity" to mean any entity type at all.

In the descriptions, the term "area" is used to designate the part of an organization being described in a given OSL specification. The area is the starting point for the organizational, external, and external context pieces of the environment, as described previously; it also provides the context in which several of the built-in entity types are defined.

The following subsections are arranged alphabetically, inasmuch as the relationships among the entity types do not admit of an obvious linear arrangement.

The format of an entity description includes an English description of the entity and its characteristics, followed by the set of built-in (type) attributes that are included in the type definition to provide for the characterization, in the following format:

Attribute name: *value class* *(characteristics)*

The *characteristics* are the same as those defined in Section A.3.3.4.1, plus the special characteristic "optional," which indicates that the attribute need not be used in every derived class of that type.

Note that each entity type has a "Name" attribute. For some types (Name labeled with "unique"), this attribute provides a unique identity for each such entity. For other types, each entity has a (not necessarily unique) name. A third category (Name labeled with "optional") covers those types for which individual entities are not normally named (such as documents); if the Name attribute is left out of a class of such type, the Name attribute is, by default, a class attribute with a value equal to the class name.

A.3.5.1 Account

An account is a record (*q.v.*) of transactions (*q.v.*). An account has a unique account number, and some role (*q.v.*) that is responsible for it.

```
Name: TEXT (optional)
Number: TEXT (unique)
Responsible: ROLE
```

A.3.5.2 Agreement

An agreement is a relationship that indicates mutual responsibilities among parties. The minimal characteristics for defining an agreement include the identification of all the parties, the period over which the agreement is to hold (start

and end dates), and the agreement itself. While in the general case this is just a text value, in any particular use it might be a hierarchical attribute describing various specific aspects of the agreement.

```
Agreement: TEXT
End: DATE
Name: TEXT (optional)
Party1: PARTY
Party2: PARTY
arbitrary number of <Partyn: PARTY>
Start: DATE
```

A.3.5.3 Appointment

An appointment is an agreement (*q.v.*) among one or more parties to meet at a given time, possibly involving specific physical facilities (rooms, equipment, *etc.*) It is often the case that an appointment is a piece of a larger structure, a schedule (*q.v.*); if so, the "Refers" attribute relates the appointment to the schedule.

```
Date: DATE
Facilities: ENTITY (multiple) (optional)
Length: INTERVAL (optional)
Name: TEXT (optional)
Party1: PARTY
arbitrary number of <Partyn: PARTY>
Refers: SCHEDULE (optional)
Time: TIME
```

A.3.5.4 Approval

Approval is a type that is used only as a primitive attribute value class specifier. An approval is a signature or other implementation of an authorization by some responsible person of the entity of which it is an attribute value. (It is most often used on documents and other communications.) An approval is created only by the execution of an APPROVE activity (see description in next section).

```
By: PERSON
Date: DATE
Signature: TEXT
```

A.3.5.5 Archive

An archive is a file (*q.v.*) that provides for long-term, long-access-time storage of entities or information about entities. The definition includes an attribute describing how long information is to be kept after entering the archive.

```
Destroy-after: INTERVAL (optional)
Name: TEXT
Refers: ENTITY
```

A.3.5.6 Communication

A communication is used to transmit information about an entity. (Specific kinds of communications include documents, memos, *etc*, which are defined separately; a "communication" type is used only when none of the more specific types is appropriate.) Since a communication always refers to another entity, that entity is specified in the "Refers" attribute. The office(s) or person(s) from whom the communication is expected to come and to whom it is to be sent can also be specified. If a reply is required, that communication is specified as the "Reply" attribute. Finally, the body of the communication may be specified as text, or with any other attributes that are required.

```
Body: TEXT (optional)
From: ORGANIZATIONAL-UNIT or ROLE (optional, multiple)
Name: TEXT (optional)
Refers: ENTITY
Reply: COMMUNICATION (optional)
To: ORGANIZATIONAL-UNIT or ROLE (optional, multiple)
```

A.3.5.7 Date

A date is a restriction of the TEXT name class that includes any legal date or day of week specification, *e.g.* `January 22` or `Thursday`

A.3.5.8 Document

A document is a formal, written communication (*e.g.*, a form).

```
Body: TEXT (optional)
From: ORGANIZATIONAL-UNIT or ROLE (multiple, optional)
Name: TEXT (optional)
Refers: ENTITY
Reply: COMMUNICATION (optional)
To: ORGANIZATIONAL-UNIT or ROLE (multiple, optional)
```

A.3.5.9 Employee

An employee is a person employed by some organization. The following attributes describe a minimal set of characteristics for an employee. Note that one person may hold many roles.

```
Company: ORGANIZATION
ID: TEXT (unique)
Name: TEXT
Office: ORGANIZATIONAL-UNIT (multiple)
Role: ROLE (multiple)
Supervisor: EMPLOYEE (multiple)
Title: TEXT (optional)
```

A.3.5.10 Evaluation

Evaluation is used only as the value class of a primitive attribute. An evaluation is the text that results from the execution of an EVALUATE activity. (Such an activity provides for someone to examine an entity (usually a document) and provide some comments—the evaluation—about his examination. See the next section for a more complete description.)

```
Name: TEXT
```

A.3.5.11 File

A file is a physical storage structure for a group of entities, the *constituents*. It includes an ordering attribute that serves to indicate by which attributes the

constituents are ordered within the file, and a time at which the constituents are removed form the file (to be archived or destroyed).

```
Name: TEXT (optional)
Constituents: ENTITY (multiple)
Ordering: <attribute name>
Remove: TIME
```

A.3.5.12 Internal-employee

An internal employee is an employee of the organization of which the area being described is a part. (See "Internal-organizational-unit"). The characteristics are the same as for an employee, except that the organization is not specified, since it is inherently a common attribute of known value.

```
ID: TEXT (unique)
Name: TEXT
Office: ORGANIZATIONAL-UNIT (multiple)
Role: ROLE (multiple)
Supervisor: INTERNAL-EMPLOYEE (multiple)
Title: TEXT (optional)
```

A.3.5.13 Internal-organizational-unit

Each OSL specification focuses on the functions carried out by one or more offices within an organization. That organization is described in the "organizational context" subsection of the environment description. An internal organizational unit is any subpart of that organization. (In Appendix I, the organization is MIT; in that context, an internal organizational unit is any subpart of MIT.)

An internal organizational unit is characterized by a name, a supervisor, and a parent, the unit(s) of which it is a direct part or to which it reports. (The "Parent" attribute provides a mechanism for expressing the relationships shown in an organizational chart.)

```
Name: TEXT
Parent: INTERNAL-ORGANIZATIONAL-UNIT (multiple)
Supervisor: INTERNAL-EMPLOYEE
```

A.3.5.14 Internal-role

An internal role is a role (*q.v.*) within the organization of which the area is a part. Except for the lack of the unneeded "Company" attribute, it is identical to a role.

```
Current-holder: EMPLOYEE (multiple)
Location: ORGANIZATIONAL-UNIT (multiple)
Name: TEXT
Reports-to: ROLE (multiple)
```

A.3.5.15 Log

A log is a structure whose purpose is to provide a record of events concerning some entity. The "Period" attribute may be used to indicate what period of time the log covers. The "Entry" attribute indicates what information is to be saved upon execution of a LOG activity (see next section). The "Entry.Copy" attribute takes a list of attribute names (attributes of the "Refers" entity), indicating which of those attribute values are to be recorded. Of course, other attributes may be added to the "Entry" definition for specific logs.

```
Entry:
        Copy: <attribute name> (optional)
        Date: DATE (optional)
        Time: TIME (optional)
Name: TEXT (optional)
Refers: ENTITY
Period: INTERVAL (optional)
```

A.3.5.16 Memo

A memo is a written, informal communication.

```
From: ROLE
Message: TEXT
Name: TEXT (optional)
Refers: ENTITY
Reply: COMMUNICATION (optional)
To: ORGANIZATIONAL-UNIT or ROLE (multiple)
```

A.3.5.17 Message

A message is an informal communication, often representing the result of a telephone call or other ephemeral action.

```
Date: DATE
From: ORGANIZATIONAL-UNIT or ROLE (multiple)
Message: TEXT
Name: TEXT (optional)
Refers: ENTITY
Reply: COMMUNICATION (optional)
Time: TIME
To: ORGANIZATIONAL-UNIT or ROLE (multiple)
```

A.3.5.18 Number

A number is a restriction of the TEXT name class that includes all legal numbers.

A.3.5.19 Organization

An organization is any business (in the legal (corporation, partnership, proprietorship) sense) with which the area deals (*e.g.*, MIT, IBM, Joe's Pizza). Each organization has a name, an address, and a person who is in charge.

```
Address: ADDRESS
C-e-o: EMPLOYEE
Name: TEXT
```

A.3.5.20 Organizational-unit

An organizational unit is any subpart of an organization. It is characterized by specifying the organization of which it is a part, its parent unit (see description of parent in "Internal-organizational-unit"), and the person in charge.

```
Name: TEXT
Organization: ORGANIZATION
Parent: ORGANIZATIONAL-UNIT
Supervisor: EMPLOYEE
```

A.3.5.21 Party

A party is any person or organization. This is a generic type used primarily for defining more specific parties.

```
Name: TEXT
```

A.3.5.22 Person

A person is a real person (as opposed to an organization).

```
Name: TEXT
```

A.3.5.23 Project

A project is a (partially ordered) set of tasks, with a project manager in charge. Each task has attributes that provide ordering information, and the project itself has "pointers" to the first and last tasks.

```
Elements: TASK (multiple)
First: TASK (multiple)
Last: TASK (multiple)
Manager: EMPLOYEE
Name: TEXT
```

A.3.5.24 Record

A record is an organizing structure for other, heterogeneous, entities, which are called *constituents.* It is used to gather together information about an entity. (As an example, consider the various messages, documents, *etc.* that make up a medical record, all referring to a particular person.)

```
Constituents: ENTITY (multiple)
Name: TEXT
Refers: ENTITY
```

A.3.5.25 Role

A role is a responsibility for performing a particular set of actions. A person may simultaneously have several roles, and several people may simultaneously or sequentially fill one role. (For example, an MIT faculty member might fill the roles of course lecturer, student advisor, committee chairman, and principal investigator; the role of course lecturer would have many simultaneous holders, though for a specific course the role of lecturer might have a single holder at any given time.) The characteristics of a role include its name, the organizational unit in which it is carried out, its current holders, and the role that is responsible for supervising it.

```
Company: ORGANIZATION
Current-holder: EMPLOYEE (multiple)
Location: ORGANIZATIONAL-UNIT (multiple)
Name: TEXT
Reports-to: ROLE (multiple)
```

A.3.5.26 Schedule

A schedule is a structure for allocating the time of some resource; it is an ordered set of appointments (*q.v.*). A schedule is normally set up for a particular interval (a day, week, month, *etc.*) and contains an arbitrary number of appointments.

```
Elements: APPOINTMENT (multiple)
Frame: INTERVAL (optional)
Name: TEXT
Resource: ENTITY (multiple)
Start-date: DATE (optional)
Start-time: TIME (optional)
```

A.3.5.27 Task

A task is a responsibility to perform some action(s). In some cases, a task is a part of a larger project (*q.v.*), in which case it follows one or more other tasks. There is at least one person who is responsible for accomplishing the task, and there may be specified starting and completion dates.

```
Completion: DATE (optional)
Description: TEXT
Follows: TASK (optional, multiple)
Name: TEXT
Precedes: TASK (optional, multiple)
Refers: PROJECT (optional)
Responsible: PARTY (multiple)
Start: DATE (optional)
```

A.3.5.28 Text

Text is a special class that is at the base of all name classes. It consists of all the characters in any standard alphabet (*e.g.*, ASCII].

Defining the value class of an attribute as TEXT indicates that it is uninterpreted information, rather than an entity of any sort.

7.4.0.1 Time

Time is a name class that is defined as any legal representation of a time specification; this be a specific time of day, *e.g.*, `10:00am` or an interval, *e.g.*, `3 weeks`

A.3.5.29 Transaction

A transaction is an event that represents some mutual action on the part of two or more parties, for which an account is kept. Examples in Appendix I include purchases and trips.

```
Account: ACCOUNT
Date: DATE
Party1: PARTY
Party2: PARTY
arbitrary number of <Partyn: PARTY>
```

Appendix B

Formal Syntax of OSL

This Appendix contains the formal grammar that defines OSL's syntax. It is described in a modified BNF form, with the following metasyntactic conventions:

- The left side of a production is separated from the right by a "←"

- Syntactic categories are capitalized, while literals are in lower case, with or without initial capitals. A few literals are in uppercase or otherwise ambiguous; they are enclosed in single quotation marks (' ')

- The first level of indentation in the syntax description is used to help separate the left and right sides of a production; all other indentation is in OSL.

- Symbols:

  { } means the enclosed item is optional.

  [ ] means one of the enclosed choices must appear; choices are separated by a semicolon (";"). (When used with "{ }" one of the choices may optionally appear.)

  < > means one or more of the enclosed can appear, separated by spaces with optional commas.

  << >> means one or more of the enclosed can appear, vertically appended.

  * * encloses a "meta-description" of a syntactic category, to explain it informally. (Not all context-sensitive descriptions are indicated in this manner, particularly some of the obvious ones as defined in the Language Reference Manual.)

```
SPECIFICATION ←
    ENVIRONMENT-PART
    OPERATIONAL-PART

OPERATIONAL-PART ←
    {<<FUNCTION>>}
    <<PROCEDURE>>

FUNCTION ←
    Function NAME
    {Structure:  CLASS-NAME}
    Resource:  CLASS-NAME
    Responsible:  CLASS-NAME *of a role or person*
    Initialization: INIT-SPEC
    {Structure initialization: INIT-SPEC}
    Required Reports Received: [<<REQ-REPORT-REC-SPEC>>; None]
    Required Reports Generated:  [<<REQ-REPORT-GEN-SPEC>>; None]
    Other Events:  [<<OTHER-EVENT-SPEC>>; None]
    Termination:  TERM-SPEC
    {Structure Termination: TERM-SPEC}
    Quantitative Information:
        F-QUANT-INFO

INIT-SPEC ←
    EVENT-SPEC: PROCEDURE-NAME

REQ-REPORT-REC-SPEC ←
    TIME: COMMUNICATION-NAME: PROCEDURE-NAME; PROCEDURE-NAME

COMMUNICATION-NAME ←
    CLASS-NAME

REQ-REPORT-GEN-SPEC ←
    TIME: COMMUNICATION-NAME: PROCEDURE-NAME

OTHER-EVENT-SPEC ←
    EVENT-NO: PROCEDURE-NAME

EVENT-NO ←
    NUMBER

TERM-SPEC ←
    EVENT-SPEC:  PROCEDURE-NAME
```

```
F-QUANT-INFO ←
    Number of resources:  NUMBER
    Number responsible:  NUMBER
    Number of personnel: NUMBER
    Initiation:  FREQUENCY
    Termination:  FREQUENCY
    {Other-events: <<EVENT-NO: FREQUENCY>>

FREQUENCY ←
    NUMBER/TIME

PROCEDURE ←
    Procedure PROCEDURE-NAME
    Focal Object:  OBJECT-CLASS-NAME
    Responsible:  ROLE-CLASS-NAME
    Main line:  PROCESS-SPEC
    {Timing Constraints:  TC-SPECS}
    {Quantitative Information:  P-QUANT-INFO}
    {Variations:  <<VARIATION-SPEC>>}
    {Exceptions:  EXCEPTION-SPECS}
    {Details:  <<DETAIL-SPEC>>}

PROCEDURE-NAME ←
    CONSTANT

OBJECT-CLASS-NAME ←
    CLASS-NAME

ROLE-CLASS-NAME ←
    CLASS-NAME

PROCESS-SPEC ←
    <<state STATE-NAME
        EVENT/STEP-SPEC>>

EVENT/STEP-SPEC ←
    Event.EVENT-NUMBER EVENT-SPEC
    Step.STEP-NUMBER
        STEP-SPEC >>

STATE-NAME ←
    CONSTANT

EVENT-NUMBER ←
    NUMBER

STEP-NUMBER ←
    NUMBER

TC-SPECS ←
    <<TIMING-CONSTRAINT>>
```

```
TIMING-CONSTRAINT ←
    CONSTRAINT-NUMBER. EVENT-NUMBER SCALAR-COMPARATOR EVENT-EXPRESSION

EVENT-EXPRESSION ←
    EVENT-NUMBER [+; -] TIME

CONSTRAINT-NUMBER ←
    NUMBER

P-QUANT-INFO ←
    Total time: TIME
    Number responsible:  NUMBER
    Number of personnel:  NUMBER
    Objects:  NUMBER
    {Exceptions:
        <<EXCEPTION-NUMBER:  FREQUENCY>>}
    {Variations:
        <<VARIATION-NUMBER:  PROBABILITY>>}
    {Branching:
        {<<STEP-NUMBER → STATE-NAME:  PROBABILITY>>}
        {<<STATE-NAME (EVENT-NUMBER): PROBABILITY>>} }

PROBABILITY ←
    .NUMBER

VARIATION-SPEC ←
    VARIATION-NUMBER. where ATTRIBUTE-EXPRESSION:
        {delete:
            {<<Event EVENT-NUMBER>>}
            {<<Step STEP-NUMBER>>}  }
        {add:
            <<EVENT/STEP-SPEC>> }
        {replace:
            {<<EVENT-SPEC>>}
            {<<STEP-SPEC>>} }

VARIATION-NUMBER ←
    NUMBER

EXCEPTION-SPECS ←
    {Timing constraint:
        <<TC-EXCEPTION>>}
    {Activity:
        <<ACTIVITY-NUMBER:
            <<EXCEPTION-NAME:  PROCESS-SPEC>> >> }
    {General:
        {Missing personnel:  PROCESS-SPEC}
        {Lost documents:  <<DOCUMENT-NAME:  PROCESS-SPEC>> }
        {Cancellation:  PROCESS-SPEC}
        {Backout:  PROCESS-SPEC}
        {Ad hoc: <<EXCEPTION-NAME: PROCESS>> } }
```

```
DOCUMENT-NAME ←
    CLASS-NAME

EXCEPTION-NAME ←
    CONSTANT  *0 or more defined with each activity*

DETAIL-SPEC ←
    <<ACTIVITY-NUMBER:  CONSTANT *any description of the activity*>>

EVENT-SPEC ←
    [EVENT; EVENT or EVENT-SPEC; (EVENT) and (EVENT-SPEC)]

EVENT ←
    [TRIGGER-EVENT; TIME-EVENT; ENVIRONMENT-EVENT;
    COMMUNICATIONS-EVENT; ACTIVITY-EVENT]

TRIGGER-EVENT ←
    by ROLE-CLASS-NAME

TIME-EVENT ←
    on [DATE; TIME [after; before] EVENT-NUMBER]

ENVIRONMENT-EVENT ←
    when [ATTRIBUTE-NAME = VALUE;
    ENTITY is [updated; created; deleted] ]

COMMUNICATION-EVENT ←
    [RECEIPT-EVENT; no RECEIPT-EVENT after TIME]

RECEIPT-EVENT ←
    receive [COMMUNICATION-NAME {[with ATTRIBUTE-NAME = VALUE;
    matching ENTITY on <ATTRIBUTE-NAME> ]};
    reply to COMMUNICATION-NAME]

COMMUNICATION-NAME ←
    CLASS-NAME

ACTIVITY-EVENT ←
    [complete; start] ACTIVITY-NUMBER

STEP-SPEC ←
    <<SUBSTEP-SPEC>>

SUBSTEP-SPEC ←
    [BRANCH-SPEC; ACTIVITY-SPEC]

BRANCH-SPEC ←
    [where ATTRIBUTE-EXPRESSION add
        <<ACTIVITY-SPEC>>
        {end in STATE-NAME};
    where ATTRIBUTE-EXPRESSION end in STATE-NAME]
```

```
ATTRIBUTE-EXPRESSION ←
    [ATTRIBUTE-NAME = VALUE;
    ATTRIBUTE-EXPRESSION or ATTRIBUTE-EXPRESSION;
    (ATTRIBUTE-EXPRESSION) and (ATTRIBUTE-EXPRESSION);
    ARITHMETIC-FUNCTION NUMBER-OPERATOR CHAIN]

ACTIVITY-SPEC ←
    ACTIVITY-NUMBER. ACTIVITY

ACTIVITY-NUMBER ←
    STEP-NUMBER.ACTIVITY-NO.

ACTIVITY-NO ←
    NUMBER{SMALL-LETTER}

SMALL-LETTER ←
    *single lowercase letter*

ACTIVITY ←
    {SUBJECT} BUILT-IN-ACTIVITY {MODIFIER} {SOURCE}

SUBJECT ←
    [CLASS-NAME; ENTITY]

BUILT-IN-ACTIVITY ←
    [CREATE; DELETE; SET; CALCULATE; REVISE; ARCHIVE; SEND; ADD;
    REMOVE; APPROVE; VERIFY; EVALUATE; NEGOTIATE; SELECT; ALLOCATE;
    GROUP; NOTIFY; RETRIGGER; INITIATE; TERMINATE; PERFORM; RETURN;
    REPEAT]

MODIFIER ←
    [matching on <ATTRIBUTE-NAME>; first; last; any; each]

SOURCE ←
    [using ENTITY; consulting [ROLE-CLASS-NAME; PARTY-CLASS-NAME] ]

CREATE ←
    Create CLASS-NAME

DELETE ←
    Delete ENTITY

SET ←
    Set ATTRIBUTE-NAME = VALUE

CALCULATE ←
    Calculate ATTRIBUTE-NAME {= *expression*}

REVISE ←
    Revise ATTRIBUTE-NAME
```

```
ARCHIVE ←
    Archive {ENTITY} {in ARCHIVE-CLASS-NAME}

SEND ←
    Send {(copy)} {ENTITY} {to <DESTINATION>}

DESTINATION ←
    PARTY-CLASS-NAME

ADD ←
    Add {ENTITY} to CLASS-NAME

REMOVE ←
    Remove {ENTITY} from CLASS-NAME

APPROVE ←
    Approve APPROVE-ATTRIBUTE-NAME {ENTITY} {by ROLE-CLASS-NAME}

APPROVE-ATTRIBUTE-NAME ←
    ATTRIBUTE-NAME

VERIFY ←
    Verify {<ATTRIBUTE-NAME>} {ENTITY}

EVALUATE ←
    Evaluate {<EVAL-ATTRIBUTE-NAME>} {ENTITY} {by ROLE-CLASS-NAME}

EVAL-ATTRIBUTE-NAME ←
    ATTRIBUTE-NAME

NEGOTIATE ←
    Negotiate ATTRIBUTE-NAME with
        < <ROLE-CLASS-NAME> <PARTY-CLASS-NAME> >

SELECT ←
    Select {NUMBER} CLASS-NAME {from CLASS-NAME}

ALLOCATE ←
    Allocate {NUMBER} from CLASS-NAME to <CLASS-NAME>

GROUP ←
    Group CLASS-NAME into <SUBSET-NAME>

SUBSET-NAME ←
    CLASS-NAME

NOTIFY ←
    Notify {ROLE-CLASS-NAME}

RETRIGGER ←
    Retrigger {CONSTRAINT-NUMBER} + TIME
```

```
INITIATE ←
    Initiate PROCEDURE-NAME

TERMINATE ←
    Terminate {PROCEDURE-NAME}

PERFORM ←
    Perform PROCEDURE-NAME

RETURN ←
    Return {ENTITY} to {PARTY-CLASS-NAME}

REPEAT ←
    Repeat {NUMBER}

ENTITY ←
    [CLASS-NAME; CHAIN; CLASS-NAME(ATTRIBUTE-NAME = UNIQUE-ID)]

ENVIRONMENT-PART ←
    IDENTIFICATIONS
    DEFINITIONS

IDENTIFICATIONS ←
    ORG-CONTEXT
    EXTERNAL-CONTEXT
    INTERNAL-CONTEXT

ORG-CONTEXT ←
            Organizational Context
    INSTANCE-DEF *of the organization of which the office is a part*
    ORG-HIERARCHY
    PERS-HIERARCHY
    <<CLASS-DESCRIPTION>> *describing the organization*

EXTERNAL-CONTEXT ←
            External Context
    <<CLASS-DESCRIPTION>> *external to the organization*

INTERNAL-CONTEXT ←
            Internal Context
    <<CLASS-DESCRIPTION>> *internal to office, except documents, etc*
            Documents and communications
    <<CLASS-NAME>>
            Names
    <<CLASS-NAME>>

CLASS-DESCRIPTION ←
    [CLASS-DESC; ALIAS-DESC]

CLASS-DESC ←
    CLASS-NAME is [CLASS-TYPE; CLASS-DERIVATION]
```

```
ALIAS-DESC ←
    CLASS-NAME = CLASS-NAME

CLASS-TYPE ←
    ENTITY-TYPE

ORG-HIERARCHY ←
    Name    Parent  Supervisor
    <<TAB-INSTANCE-DEFINITION>>

PERS-HIERARCHY ←
    Name    Organizational-unit Reports-to  Current-holder
    <<TAB-INSTANCE-DEFINITION>>

DEFINITIONS ←
    <<CLASS-SPEC>>

CLASS-SPEC ←
    [CLASS; NAME-CLASS]

CLASS ←
    CLASS-NAME {= CLASS-NAME} is CLASS-ORIGIN
    <<ATTRIBUTE-DEFINITION>>

CLASS-ORIGIN ←
    [CLASS-TYPE;
        CLASS-DERIVATION]

CLASS-NAME ←
    *string of capitals possibly including '-'*

NAME-CLASS ←
    CLASS-NAME is 'NAME'
    '{' <STRING> '}'

CLASS-DERIVATION ←
    [RESTRICT; SUBSET; COMMON-MEMBERS; MERGE-MEMBERS; MISSING-MEMBERS]

RESTRICT ←
    restrict CLASS-NAME {where RESTRICT-PREDICATE}

SUBSET ←
    subset of CLASS-NAME

COMMON-MEMBERS ←
    common members in <CLASS-NAME>

MERGE-MEMBERS ←
    merge members in <CLASS-NAME>
```

```
MISSING-MEMBERS ←
    missing members in CLASS-NAME but not in CLASS-NAME

RESTRICT-PREDICATE ←
    [SIMPLE-PREDICATE; (RESTRICT-PREDICATE); not RESTRICT-PREDICATE;
    RESTRICT-PREDICATE and RESTRICT-PREDICATE;
    RESTRICT-PREDICATE or RESTRICT-PREDICATE]

SIMPLE-PREDICATE ←
    [CHAIN SCALAR-COMPARATOR [CONSTANT; CHAIN];
    CHAIN SET-COMPARATOR [CONSTANT; CLASS-NAME; CHAIN];
    is a value of ATTRIBUTE-NAME of CLASS-NAME]

CHAIN ←
    [CHAIN-DEF; CLASS-NAME.CHAIN-DEF]

CHAIN-DEF ←
    [ATTRIBUTE-NAME; CHAIN.ATTRIBUTE-NAME]

SCALAR-COMPARATOR ←
    [EQUAL-COMPARATOR; >; ≥; <; ≤]

EQUAL-COMPARATOR ←
    [=; NOTEQUALS[]]

SET-COMPARATOR ←
    [is in; is not in; contains; does not contain]

CONSTANT ←
    [STRING; NUMBER]

STRING ←
    *a string constant*

NUMBER ←
    *a number constant*

PATTERN ←
    *a name class definition pattern*

ATTRIBUTE-DEFINITION ←
    [REGULAR-ATTRIBUTE-DEFINITION;
    REFERS; SAME; DEPENDENT-ATTRIBUTE]

REGULAR-ATTRIBUTE-DEFINITION ←
    ATTRIBUTE-NAME: {<ATTRIBUTE-NAME:>} VALUE-DETERMINANT

VALUE-DETERMINANT ←
    [PRIMITIVE-ATTRIBUTE-FEATURES; DERIVED-ATTRIBUTE-FEATURES]
```

```
ATTRIBUTE-NAME ←
    *string of lowercase letters beginning with a capital*

PRIMITIVE-ATTRIBUTE-FEATURES ←
    VALUE-CLASS {(CHARACTERISTICS)}

CHARACTERISTICS ←
    <multiple; common; class; unique; mandatory; by PERSON>

VALUE-CLASS ←
    CLASS-NAME

DERIVED-ATTRIBUTE-FEATURES ←
    [CHAIN; INTER-ATTRIBUTE-DERIVATION
    MEMBER-SPECIFIC-DERIVATION;
    CLASS-SPECIFIC-DERIVATION]

INTER-ATTRIBUTE-DERIVATION ←
    [common-members in <CHAIN>;
    merge-members in <CHAIN>;
    missing-members in CHAIN but not in CHAIN;
    CHAIN-EXPRESSION;
    [maximum, minimum, average, sum] of CHAIN;
    number of {unique} members in CHAIN;
    restrict ATTRIBUTE-NAME {where RESTRICT-PREDICATE}]

MEMBER-SPECIFIC-DERIVATION ←
    [invert ATTRIBUTE-NAME of CLASS-NAME;
    if exists in CLASS-NAME;
    order by [increasing; decreasing] <CHAIN>;
    ATTRIBUTE-NAME of matching CHAIN]

CLASS-SPECIFIC-DERIVATION ←
    [number of {unique} members in this class;
    [maximum; minimum; average; sum]
        of ATTRIBUTE-NAME over members of this class]

REFERS ←
    Refers: CHAIN

SAME ←
    [Same: <ATTRIBUTE-NAME>;
    Same as CHAIN: <ATTRIBUTE-NAME>]

DEPENDENT-ATTRIBUTE ←
    ATTRIBUTE-NAME:
        <<if <ATTRIBUTE-NAME> = CONSTANT then
            DERIVED-ATTRIBUTE-FEATURES>>

CHAIN-EXPRESSION ←
    [CHAIN; (CHAIN); CHAIN NUMBER-OPERATOR CHAIN]
```

```
NUMBER-OPERATOR ←
    [+; -; *; /; !]

INSTANCE-DEF ←
    [DEFINE-INSTANCE; TAB-INSTANCE-DEF]

DEFINE-INSTANCE ←
    Define Instance of CLASS-NAME
        <<ATTRIBUTE-INSTANCE-DEF>>

ATTRIBUTE-INSTANCE-DEF ←
    ATTRIBUTE-NAME:  ATTRIBUTE-VALUE

TAB-INSTANCE-DEF ←
    <ATTRIBUTE-VALUE>

ENTITY-TYPE ←
    *one of the following built-in entity types:
        ACCOUNT; AGREEMENT; APPOINTMENT; APPROVAL;
        ARCHIVE; COMMUNICATION; DATE; DOCUMENT; EMPLOYEE;
        EVALUATION; FILE; INTERNAL-EMPLOYEE;
        INTERNAL-ORGANIZATIONAL-UNIT; INTERNAL-ROLE; LOG;
        MEMO; MESSAGE; NUMBER; ORGANIZATION;
        ORGANIZATIONAL-UNIT; PARTY; PERSON; PROJECT;
        RECORD; ROLE; SCHEDULE; TASK; TEXT; TIME; TRANSACTION'
    each entity type requires the definition of certain attributes,
    as specified in the Reference Manual*
```

Appendix C

Admissions Office Case

Notes:

- *This description represents the product of an office analysis conducted without the benefit of OAM/OSL. It is for use only in the OAM/OSL training course.*

- *This example was produced from a case study performed in 1978; the Admissions Office has since undergone major changes in its information system support structure, and we make no claim to represent its current operations.*

The Admissions Office works relatively independently; formal communications within MIT are minimal and consist of the class size, received from the Chancellor, and a general report to the President, both annually. Other important interdepartmental communications are handled informally but regularly by weekly meetings of the Vice President for Administration and Personnel and the heads of the departments which report to him. These include Admissions, Student Financial Aid, and Career Planning and Placement (CPP), among others. The Admissions Office works closely with Financial Aid, and needs information from that office and from CPP in order to inform and counsel prospective students. Other important nodes include the Registrar, Freshman Advisory Committee, and the academic departments (for graduate admissions).

The functions of the Admissions Office include undergraduate recruitment, selection and admission; transfer student selection and admission; graduate and special student admission; foreign student admission and counseling; and

miscellaneous information source. We describe in detail two procedures: Undergraduate Admissions and Graduate Admissions. In the latter, the Admissions Office acts only as a central information switch for the Departments, which make the actual decisions.

The following are the major steps in the admissions procedure

- A preliminary application is sent to prospective students.

- The applicant returns to the Admissions Office (henceforth called "Office") a filled-in preliminary application card; receipt of this document initiates the applicant's file in the admissions system.

- Timely receipt of documents required to complete the application is controlled by the Office by generating appropriate letters.

- Completed applications are reviewed.

- Admission decisions are made on reviewed applications.

- Acceptance decisions from admitted students are received, allowing information on the incoming class to be transmitted to other Institute offices.

Currently, the Office uses a combination of paper and automated record-keeping procedures. The Office of Administrative Information Systems (OAIS) provides a computer-based file, the applicant system (called "the System" in the sequel), which serves as a central repository for information about the status of each prospective student's application; however, all information in the System is also kept on various paper records in the Office, so as to be available to queries. Insertions (of new applicants) are sent to OAIS in batches of 100; updates are batched weekly; both have a one or two day turnaround. The "being made up" file and the "earlier material" basket (see below) are artifacts of the batch processing; they serve as index and (temporary) storage, respectively, for information about applicants for whom an initial entry to the System is being made.

The Office keeps a "general file," which contains any correspondence about prospective students (*e.g.*, interview reports, references, *etc.*, although not simple requests for application materials from the student) who have not filed a preliminary application, and who therefore have not entered the admissions procedure *per se.*

The remainder of this section describes the details of the procedure.

Initial contact with the admissions procedure is made by one of three routes:

1. About 9000 people each year make initial personal contact with the Office by mail, phone or in person. In this case the prospect is sent (given) a brochure containing a Preliminary Application Card. No record is kept of this contact. If this initial contact comes late in the admissions season (*i.e.*, past the middle of September), a special Final Application packet is sent in response; this packet includes a Preliminary Application Card in a color different from that of the regular cards, so that when it is returned the Office can know that the Applicant already has his Final Application.

2. Office personnel visit high schools and return lists of students to the Office; brochures with Preliminary Application Cards are sent to these students. If such information is available, material appropriate to sex and minority status is also sent. The visits are made with output from the Office's School File, which contains names of all applicants from that high school in previous years, as well as results of their application; a printout of this file is made once per year for the recruiting trips.

3. The Office receives a mag tape from the Educational Testing Service each Spring, which contains the name, address, sex, and race of about 27000 prospects (the Search List). Brochures/Preliminary Applications are sent to each person on the list; additional information for women and minority prospects is sent as appropriate. The Search list is retained for several other mailings, but not used again in this procedure.

In all cases, if the student is appplying from a country other than the US or Canada, the standard card is not used; a Preliminary Application for Foreign Students is sent instead.

The next task is initiated by receipt of the Preliminary Application. If this is a foreign application, a special foreign "being-made-up" card (FBMUC) is created and placed in the foreign "being-made-up" file (BMUF). This application is supposed to arrive with a number of documents from the student's school, and the entire package is sent to the Foreign Student Office. This office makes a judgment as to whether the application is legitimate and acceptable, and if so, it is returned to the Office (possibly to the college transfer or graduate student admissions procedures, rather than this one), where it is handled in the same manner as US applications, except where noted.

US and Canadian students return the Preliminary Application Card. It is first checked for completeness; if it cannot be completed from information available, it is returned with a letter. A complete Card consists of two identical cards, labelled "B" and "R." The B card is placed in the (U.S.) BMUF. If there is any correspondence in the General File concerning this Applicant, it is removed from the file and placed in a basket on the Administrative Assistant's (AA) desk (this folder is now called "earlier material"). A check is also made in the College Board File (see below), to see whether test scores have been sent; if so, the College Board card is sent to OAIS for insertion into the Applicant's record when it is created. The R card is given to the appropriate section clerk ("Clerk"), who is reponsible for handling all materials for each Applicant in the (alphabetic) section. The Clerk enters the information from the card (which includes name & address, demographic and high school data) into the System; these "new preliminaries" (insertions to the System) are sent to OAIS whenever a batch of 100 is completed (once each day or two). The Clerk holds the R card in a temporary file.

In response to the new preliminary batch, OAIS creates a record for each Applicant, and returns a set of mailing labels, two each for the Applicant and the Educational Council member nearest him, who will conduct the interview (the

"Interviewer"). The Interviewer name is chosen by ZIP code by the system from its list of Interviewers, except for foreign students residing outside the US, whose Interviewer is determined manually by the Office. If the Applicant is not applying for the current year, or if he already has a Final Application, the Interviewer mailing label is placed on a postal card, which asks him to have an interview. Otherwise, he is sent a Final Application with the Interviewer label attached. In either case, the Interviewer is sent the R card and a report form. If the Applicant is applying for the current year and there is a record in his folder indicating that he has had an interview recently enough (since May of year *n* for admission in September, *n+1*), this is noted on the B card (in the BMUF) and the R card (to the Interviewer), and the Applicant is notified that another interview is not required by an additional sticker on the card or Final Application packet.

If the Applicant lives too far from the Interviewer, he is informed that his interview requirement is optional; if he lives in the Boston area, his interview is held in the Office, which is the address sent to him. In both cases, the R card is destroyed and no report form is sent. (Current plans call for the establishment of Interviewers in Boston. This will mean that the procedure will be the same for local Applicants, although they and any others will still have the option of an interview in the Office.)

Once each week, OAIS returns to the AA a set of documents for each "new preliminary" (applicant) entered during the week. The set includes two address labels, an E3 card, a green information card ("Green Card"), a master card ("Master"), and, if appropriate, a "minority" card. These documents contain the ID issued to the student by the system; this is a 12-character alphanumeric code created from the Applicant's name and birthdate. One of the labels goes to the Information Office, which uses it to mail a catalog to the Applicant. The AA sends the minority card to the Office staff member who deals with special projects. She then sends any "earlier material" from the basket along with the rest of the package to the appropriate Section Clerk.

At this time the Clerk removes the B card/FBMUC from the appropriate BMUF. For foreign students, the information on the Green Card is checked for accuracy against the E3, and then is placed in the Case Card File (CCF); for US students, the Green Card is filed in the CCF without checking.

The Clerk checks information on the E3 and Master against the B card, correcting the former two if necessary. The remaining address label is used to label a new file folder, into which goes the B card/FBMUC, correspondence, and the Interviewer report, if any; the folders are kept in the "Active File." The E3s and Masters are retained on the Clerk's desk, in separate files.

At this point, the Final Application is out, and further action can be initiated by the return of the Interviewer's report, or any form from the Final Application packet. The Master contains a checkoff for receipt of each of these forms, as well as one for completion of the interview. The appropriate area is marked on the card when a form is received, and the receipt is also noted for the System. The forms are placed in the Applicant's Active File folder. Specifics for processing these documents are as follows:

- Final application. This document is processed first by a clerk who looks for the application fee check, which is supposed to accompany the Final Application. If the check is there, an entry is made on the clerk's daily listing, noting the name and amount on the check. The Final Application form is stamped "fee received" and sent to the appropriate Clerk. The checks are taken each afternoon to the cashier's office. If there is no check with the Final Application, the clerk sends a letter to the Applicant requesting the fee, and attaches a copy of the letter to the Final Application, which then goes to the Clerk.

- The Clerk handles further processing of the Final Application, as well as all other forms. The ID is checked against the Master, with corrections made on the Master and E3 and to the System if required (the information on the Final Application is considered official). If the application is seriously incomplete, a letter is sent to the Applicant, with

a copy retained in the folder. Information from the Final Application form is entered to the System.

- Evaluation forms. Check ID, note receipt on Master, enter receipt and evaluator code to System.

- Secondary School Report. If a transcript does not accompany the report form, it is returned to the school with a letter requesting the transcript. If a transcript is received, then check ID, note receipt on Master, enter information from report and transcript into System.

- Interview report. Note receipt on Master and to System. If interview by Educational Council Member (rather than an Office staff member), read report for questions requiring answer. If so, send letter or telephone.

College Board reports are received on magnetic tape about once per month. (In the months following scheduled exams, there are about 4000 names on the tape; between exams the volume of data is much less.) The tape is sent to OAIS which enters the scores for all Applicants it has in the System, and returns to the Office a card containing name and birthdate for each student it could not find in the System. These cards are then manually checked against the CCF. Those which represent Applicants in the CCF (*i.e.*, whose ID numbers were different in the College Board and the MIT systems) are corrected and resubmitted to OAIS for updating the System (300-500 of the cards are matched manually). The remaining cards are filed in the College Board File. Three times each year, the College Board File is submitted to OAIS for rerunning, to catch those Applicants who have filed Preliminary Applications since their board scores have arrived, and whose scores were not located in the initial search.

Once each week during the admissions season, OAIS generates an Application Summary Report and an Applicant Action Card (AAC) for each Applicant whose application was completed (*i.e.*, all forms are now in) during the week. The Application Summary is a complete report of the Applicant's System file, and goes

to the Clerk, who pulls the E3 from the desk file and the folder from the Active File, arranges the folder in "review order," attaches the E3 and Application Summary Report to the folder, and places it in the "review file." The AAC is placed in the "out to review" file, to keep track of why the folder is not in the Active File.

Around the end of November, there is an Early Action review; the regular review period starts in January. The procedure is the same for both:

- Each day each member of the admissions staff or the Faculty Committe on Admissions is given a number of folders to review by the AA, who selects them at random from the Review File. The reviewer adds comments and a Personal Rating (a numerical rating, 5-10) to the E3 card and returns the folder to the AA. It is then placed in the second review file. The rating is on the front of the E3 and the summary on the back; the card is turned over so that the next reviewer will not see the previous reviewer's Personal Rating (until he marks his own, or chooses to look).

- The second review file is handled in the same manner, except that the AA ensures that if the first review was by a Faculty member, the second is by an Office staff member, and *vice versa*. Upon return from the second review, the AA looks at the two Personal Ratings; if they differ by more than one, she sends the folder to a staff member who has not seen it (or to a specific staff member if one of the reviewers so requested) for a third review.

- When the review of a folder is completed, the two or three Personal Ratings are entered to the System, along with the raters' initials; the E3 is removed from the folder and placed in the "Reviewed File"; the folder is returned to the Active File; and the AAC is moved to the "Roundup File."

At the end of November, OAIS returns a "laundry list" for each Applicant in the System who asked for Early Action on his Final Application Form. The List is a letter detailing the items missing from the Applicant's folder. A second Laundry List run is made in January for all "notified and complete" applications, defined as the

Applicants in the System who have submitted a Final Application Form, or for whom three other items from the Final Application packet have been received. Also at this time, a Seventh Semester Grade Report form is generated for each "notified and complete"; this is mailed to his school.

Admissions decisions are made at "Roundup" time; again, there is one Roundup in late November for Early Action, and the regular Roundup in late February. There is also a Roundup of waitlisted applications in April. Foreign Applicant Roundup is held about a week after the US procedure, except that there is no Early Action for foreign Applicants. The procedure is similar for all Roundups, although the results are slightly different:

- OAIS produces from the System file a set of E3 stickers for all Applicants in the Reviewed File (except at waitlist Roundup). There are two stickers produced, one with secondary school information, including grades, principal/guidance counselor recommendation (a numeric datum), and the Interviewer name. The second sticker contains the College Board results, and two numbers computed as a function of board scores and high school grades; these are the SI1 and SI2 (scholastic index) scores. The AA places the stickers on the E3 cards, separates the cards into minority and non-minority groups, and sends them to the Director (of Admissions) for action.

- The action procedure involves sorting the E3 cards in a physical array, indexed by the SI2 figure on one axis, and the highest Personal Rating on the other. The admissions staff then makes a decision on each Applicant, marks the decision on the card, and returns it to the AA. All actions are completed within two weeks for the regular (Feb.) Roundup.

- For Early Action, the decision is Admit, Hold, or Discourage. For the Feb. Roundup, the decision is Admit, Not admitted or Waitlist. At the waitlist Roundup, only Admit or Not admitted is allowed.

- For Admits, the AA checks for notes on the E3 card, and for missing grades or scores. If the card indicates that the chemistry/physics requirement has not been met, the folder is checked to be sure that the

card is correct. If evidence of meeting the requirement is in the folder, the correction is made on the E3 and to the System. If not, then a "provisional" note is added to the E3 card, which causes a special letter to be sent in place of the normal "admitted" letter.

- The AACs are taken from the Roundup File, and marked with the appropriate action. The AAC is used to mark the action on the Green Card, and is used for updating the System. AACs for Admits are kept in a temporary "waiting-for-reply" file; Holds and Discourages are returned to the Roundup File; Waitlists are moved to a "waitlist" file; and Not admitteds are destroyed.

- The E3 cards are used to generate the appropriate letter, copies of which go to the Applicant, his school, his Interviewer, and his folder. For admitted students, a reply form is sent with the letter. The E3 cards then go to the "admitted/no reply" file, the "not admitted" file, or the "waitlist" file, as appropriate. At Early Action, E3 cards for Hold and Discourage actions are returned to the Reviewed File.

If this is the regular (Feb.) Roundup, all Applicants in the Reviewed File who haven't completed the College Board exams are sent a letter detailing what test results are missing. After this Roundup, any Applicant who has filed a Final Application and paid his fee, but whose application is incomplete, is also sent a letter telling him what is missing.

All letters for the Feb. Roundup are sent out on the reply date, around March 24. (The Financial Aid Office sends its replies on April 15.)

The next activity is triggered by receipt of the reply form from admitted students; this is returned to indicate whether the Applicant accepts or refuses the admission. The AA maintains a tally and several statistical records on these returns as they come in; the tally is used by the Director to arrive at an estimate of the acceptance rate, and to make his decisions about the waitlist. The statistical records are retained in the AA's desk, available for future studies. The procedure for processing the reply forms for acceptance or refusal is:

- add to tally & stat. sheets

- send letter of acknowledgment (copy to folder)

- mark reply on AAC

- mark folder "cancelled"/"coming"

- mark E3 "cancelled"/"coming" with date & reason (if refused)

- if refused:
    * mark Green Card "reject"
    * send E3 to Financial Aid Ofc (then sent to Educational Council ofc)
    * add demographic information to reply form
    * on return of E3, place in "admitted/cancelled" file
    * send AAC to OAIS for System entry (not returned)

- if accepts:
    * place E3 in "admitted/coming" file
    * if foreign, assign MIT ID#
    * place memo sheet to Freshman Advisory Committee (standard info) in folder
    * send AAC to Financial Aid Ofc (then to Educational Council ofc)
    * send returned AAC to OAIS for System entry (not returned)

- send reply form to Office staff (statistical project)

- put returned reply form in folder

Alternatively, an admitted student may ask for deferred admission (*i.e.*, he wants to wait a year to enter). In this case, a letter is returned acknowledging the request and explaining the rules, and the AAC is marked "deferred" and sent to OAIS. All records (E3, Master, Green Card) are changed to indicate the new freshman admission year, and the E3 is moved to the "deferred" file, which is retained until the next year, when it is treated as an "admitted/coming" case.

Most of the reply forms are received by the May 1 due date. At this point, the Director uses the tally sheet to act on the waitlist, which typically numbers 300. These E3 cards are then assigned an Admit or Not admitted action, and processed accordingly (the letter sent to the Applicant is slightly different from the one sent to admits in earlier Roundups). The waitlist is processed by the end of May.

The Freshman Admitted List is generated from the System; it lists those who have accepted admission. The Registrar gets a weekly update on magtape, starting with the first week of admissions replies. In May, a hardcopy of the list goes to the Freshman Advisory Committee; after that, corrections are also kept manually by the AA and a copy is sent to FAC weekly.

In July, the folders for admitted/coming students are sent to the Freshman Advisory Committee. These are returned to the Office in September.

At the end of September, the Office clears out the cases of the current (freshman) class. All E3 cards are compared to their respective Green Card cards to ensure that the actions are recorded correctly on the Green Cards. Green Cards for admitted/registered students are archived separately from those for other Applicants. The rest are separated into "not admitted/not registered" and "incomplete" (application) archives, and the folders corresponding to these cards are stored. The E3 cards for Applicants not admitted are also archived. For admitted students, the E3 cards are sent to the Freshman Advisory Committee, which returns them in a day or two; they are added to the folders, which are then archived. The System file is used to produce a final statistical run, and then archived. All Masters are destroyed. All archives are kept for three years, then destroyed.

Cancellations of applications by the Applicant are handled in one of two ways, depending on the timing. If the notice is received before action (on that application), the procedure is:

- send acknowledgement letter

- mark AAC "withdrawn" and send to OAIS for System input

- mark Green Card "cancelled"

- mark Master "cancelled"

- mark E3 "cancelled", put in "withdrawn" file

If the cancellation is received after action, and the Applicant was waitlisted, the above procedure is followed, except that the E3 is placed in the "waitlist cancelled" file. If the cancellation is from an admitted student, it is treated as a refused admission; if from an Applicant who was not admitted, it is ignored.

An annual report is provided to the Director, containing statistics on total applications, and a statistical breakdown of the "admitted and coming" students. Typically, the Office receives 8500 preliminary applications, and 4500 final applications (for freshman admission). Of these, approximately 2000 are offerred admission (about 300 at Early Action) to achieve a typical Freshman class of 1000.

Database

Applicant System (computer-based, operated by OAIS)
contains state and summary of active application
record created when Preliminary Application returned
filed by ID# (= f(name, birthdate))
indexed by sex, race, ZIP
updated on receipt of any Final Appl. form or Interviewer report
updated by ETS College Board score tape or card
updated by "action"
updated by cancellation
updated by response to admission offer
each class archived after Sept freshman registration
Reports:

initial documents (new preliminaries)
E3 stickers
laundry list
7th semester grade rept
Applicant summary report & Action Card
Statistical reports
Freshman Admitted List

Auxiliary files:

Educational Council member list, indexed on ZIP
School file
By ZIP
all schools whose students have filed applications (cumulative), with data on each student

Search File

from ETS
Filed by ZIP
Indexed on sex, race

General File (GF)

by name
Contains correspondence re any Applicant who has not sent Prelim Appl

College Board File

Contains card with name & birthdate for all students who asked ETS to send scores to MIT, but who haven't filed an application

"Being-made-up" File (BMUF)

by name
Contains card for each student who has filed a Prelim Application, but for whom there is not yet a case card

Foreign BMUF

Contains foreign "Being-made-up" cards

"Case Card" File (CCF)

by name
Data=(name,addr,ID#,type of application (by color))
Contains card for each case (Applicant who has filed Prelim Application)
Approx 20000/yr (includes transfer & graduate applications)
After class is admitted, archived into one of four subfiles:
Admitted
Not admitted
Incomplete
Special Students
Archives kept 3 years, then destroyed

Folder Files
- Active File
  - Contains folders for each case, pre- and post-review
- Review file
  - Contains folders of complete applications for review
  - Subfile for 1st review filed randomly
  - Subfile for second review indexed on faculty/staff 1st review
  - Subfile for third review indexed on staff member seen
- Archives kept 3 years, then destroyed

AAC Files
- Out to review file
  - Contains AAC for each folder in review file
- Roundup file
  - Contains AAC for each case which has been completely reviewed
- Waiting-for-reply file
  - AACs for admitted students who have not replied to offer
- Waitlist file

E3 Card Files (E3 card is moved to indicate state of application)
- Reviewed file
- Admitted/no reply file
- Admitted/cancelled file
- Admitted/coming file
- Deferred file
- Waitlist file
- Waitlist/cancelled file
- Not admitted file
- Withdrawn file

Daily listing
- Listing of each A who has sent application fee

Tally list
- count of accept/refuse for admitted students
- statistics for sex, race, etc.
- kept by AA

References

1. Anthony, Robert N. *Planning and Control Systems: A Framework for Analysis.* Harvard University Graduate School of Business Administration, Boston, 1965.

2. Bach, Fred W. Analysis of Communications and Work Flow. In Carl Heyel, Ed., *Handbook of Modern Office Management and Administrative Services,* McGraw-Hill, New York, 1972, pp. 5-10 - 5-27.

3. Bair, James H. Productivity Assessment of Office Information Systems Technology. Trends and Applications: 1978, Distributed Processing, IEEE, 1978, pp. 12-24.

4. Barber, Gerry and Carl Hewitt. Towards the Development of Office Semantics. Draft, October 30, 1979.

5. Baumann, L. S. and R. D. Coop. Automated workflow control: A key to office productivity. AFIPS Conference Proceedings, Vol. 49, AFIPS Press, Arlington, Va., May, 1980, pp. 549-554.

6. Bracker, Lynne C. and Benn R. Konsynski. The OFFIS System - A Tool in Automated Office Design. 1981 Office Automation Conference Digest, AFIPS, March, 1981, pp. 417-419.

7. Canning, Richard G. Computer Message Systems. *EDP Analyzer 15,* 4 (April 1977).

8. Cook, Carolyn L. Streamlining office procedures--An analysis using the information control net model. AFIPS Conference Proceedings, Vol. 49, AFIPS Press, Arlington, Va., May, 1980, pp. 555-565.

9. Delehanty, George E. Office Automation and Occupation Structure: A Case Study of Five Insurance Companies. *Industrial Management Review 7* (Spring 1971), 99-108.

10. Ellis, Clarence A. and Gary J. Nutt. Office Information Systems and Computer Science. *Comput. Surv. 12,* 1 (March 1980), 27-60.

11. Goldstein, Ira P., and R. Bruce Roberts. NUDGE, A Knowledge-based Scheduling Program. Memo 405, MIT Artificial Intelligence Lab., Feb., 1977.

12. Good, Michael. Etude and the Folklore of User Interface Design. *SIGPLAN Notices 16* (June 1981), 34-43.

13. Gorry, G. A. and M. S. Scott Morton. A Framework for Management Information Systems. *Sloan Management Review 13*, 1 (Fall 1971), 55-70.

14. Hamill, B. J. and Steele. *Work Measurement in the Office.* Cambridge University Press, 1973.

15. Hammer, Michael, W. Gerry Howe, Vincent J. Kruskal and Irving Wladawsky. A Very High Level Programming Language for Data Processing Applications. *Comm. ACM 20*, 11 (Nov. 1977).

16. Hammer, Michael and Jay S. Kunin. Design Principles of an Office Specification Language. AFIPS Conference Proceedings, 1980 National Computer Conference, Vol. 49, AFIPS Press, Arlington, Va., May, 1980, pp. 541-547.

17. Hammer, Michael M. and Marvin A. Sirbu. What is Office Automation?. Proceedings of the National Computer Conference Office Automation Conference, AFIPS, March, 1980, pp. 37-49.

18. Hammer, Michael and Michael Zisman. Design and Implementation of Office Information Systems. Proc. NYU Symposium on Automated Office Systems, New York University Graduate School of Business Administration, May, 1979, pp. 13-24.

19. Heidorn, G. E. Natural Language Dialogue for Managing an On-line Calendar. Research Report RC 7447, IBM Thomas J. Watson Research Laboratory, Aug., 1978. Yorktown Heights, N.Y.

20. Hewitt, C. Viewing Control Structures as Patterns of Passing Messages. Memo 410, MIT Artificial Intelligence Lab., Dec., 1976.

21. Hoos, Ida R. *Automation in the Office.* Public Affairs Press, Washington, D.C., 1961.

22. IBM. The Time Automated Grid System (TAG): Sales and Systems Guide. Publication GY20-0358-1, IBM, May, 1971.

23. Ilson, Richard. Recent research in text processing. *Words 9*, 1 (June-July 1980), 32-34; 52-54.

24. Karkula, Steve. Information Flow in a Television Station. Memo OAM-002, MIT Lab. for Computer Science, Office Automation Group, Jan., 1979.

25. Keen, Peter G. W. and Michael S. Scott Morton. *Decision Support Systems: An Organizational Pespective.* Addison-Wesley, Reading, Mass., 1978.

26. Kunin, Jay S. Notes on a Letter of Credit Support System. Working Paper WP-001, MIT Lab. for Computer Science, Office Automation Group, Jan., 1979.

27. Kunin, Jay S. Notes on Insurance Company Procedures. Working Paper WP-002, MIT Lab. for Computer Science, Office Automation Group, Feb., 1979.

28. Kunin, Jay S. Analysis and Specification of Office Procedures. Ph.D. Th., Department of Electrical Engineering & Computer Science, MIT, Jan., 1982.

29. Kunin, Jay S. and Michael M. Hammer. Case Studies of Office Procedures: I. Memo OAM-001, MIT Lab. for Computer Science, Office Automation Group, Jan., 1979.

30. Ladd, Ivor and D. C. Tsichritzis. An office form flow model. AFIPS Conference Proceedings, Vol. 49, AFIPS Press, Arlington, Va., May, 1980, pp. 533-539.

31. Liskov, B. and V. Berzins. An Appraisal of Program Specifications. In P. Wegner, Ed., *Research Directions in Software Technology*, MIT Press, 1979.

32. Lynch, H. J. ADS: A Technique in System Documentation. *Database 1*, 1 (Spring 1969).

33. Matteis, Richard J. The new back office focuses on customer service. *Harvard Business Review* (March-April 1979).

34. Matthies, Leslie H. *The New Playscript Procedure: Management Tool for Action.* Office Publications, Inc., Stamford, Conn., 1977.

35. McLeod, Dennis. A Semantic Data Base Model and its Associated Structured User Interface. Technical Report TR-214, MIT Lab. for Computer Science, Aug., 1978.

36. Mintzberg, H. *The Nature of Managerial Work.* Harper & Row, 1973.

37. Myer, T. H. and D. W. Dodds. Notes on the Development of Message Technology. Proc. Berkeley Workshop on Distributed Data and Computer Networks, May, 1976.

38. National Bureau of Standards. Guidance on Requirements Analysis for Office Automation Systems. Tech. Rep. NBS Special Publication 500-72, U.S. Department of Commerce, Dec., 1980.

39. Ness, D. Office Automation Project: Personal Scheduler Version 2-1C(6). Department of Decision Sciences, Wharton School, University of Pennsylvania, July, 1975.

40. Newell, M., W. Newman and B. Sheil. A Design for the OfficeTalk Language. Internal Memo , Xerox PARC, Nov., 1977.

41. Newman, W. Studies of Office Procedures and Information Flow. Office Research Group, Xerox Palo Alto Research Center, May, 1976.

42. Propst, R. The office - A facility based on change. Herman Miller, Inc. Zeeland, Mich., ,1968.

43. Ruth, G. R. Protosystem I: An Automatic Programming System Prototype. Technical Memo TM-72, MIT Lab. for Computer Science, July, 1976.

44. Schoichet, Sandor. Case Studies of Office Procedures: The Office of Sponsored Research. Memo OAM-022, MIT Lab. for Computer Science, Office Automation Group, Oct., 1980.

45. Shepard, Jon M. *Automation and Alienation.* MIT Press, 1971.

46. Sirbu, M. A Regional Sales Office. Working Paper WP-008, MIT Lab. for Computer Science, Office Automation Group, July, 1979.

47. Sirbu, Marvin A., James W. Driscoll, Robert Alloway, Willam Harper, Moshen Khalil and Michael Hammer. Office Automation: A Comparison of In-House Studies. MIT Center for Policy Alternatives, May, 1980.

48. Sirbu, Marvin, Sandor Schoichet, Jay Kunin, and Michael Hammer. OAM: An Office Analysis Methodology. Memo OAM-016, MIT Lab. for Computer Science, Office Automation Group, Oct., 1980. Revised Jan. 1981.

49. Suchman, Lucy A. Office Procedures as Practical Action. Proceedings of the Fifth International Conference on Computer Communications, North-Holland, Oct., 1980, pp. 355-360.

50. Suchman, L. and E. Wynn. Procedures and Problems in the Office Environment. Xerox Advanced Systems Department, April, 1979.

51. Teichrow, D. Problem Statement Analyzer: Requirements for the PSA. In *Systems Analysis Techniques*, John Wiley & Sons, 1974.

52. U.S. Department of Labor, Bureau of Labor Statistics. *Adjustments to the Introduction of Office Automation.* U.S. Government Printing Office, Washington, D.C., 1960.

53. Wegner, P., ed. *Research Directions in Software Technology.* MIT Press, 1979.

54. Wynn, Eleanor H. Office Conversation as an Information Medium. Ph.D. Th., Department of Anthropology, University of California, Berkeley, May, 1979.

55. Zarmer, Craig L. and Jay S. Kunin. Case Studies of Office Procedures: The Industrial Liaison Office. Memo OAM-020, MIT Lab. for Computer Science, Office Automation Group, Oct., 1980.

56. Zarmer, Craig and Sandor Schoichet. Case Studies of Office Procedures: The Work Control Center. Memo OAM-013, MIT Lab. for Computer Science, Office Automation Group, July, 1980.

57. Zisman, M. D. Representation, Specification and Automation of Office Procedures. Ph.D. Th., The Wharton School, University of Pennsylvania, 1977.

58. Zisman, M. D. Office Automation: Revolution or Evolution. *Sloan Management Review 19*, 3 (June 1978), 1-16.

59. Zloof, Moshe M. Query by Example. AFIPS Conference Proceedings, Vol. 44, AFIPS Press, Montvale, N. J., 1975, pp. 431-438.

60. Zloof, M. M. and S. P. de Jong. The System for Business Automation (SBA): Programming Language. *Comm. ACM 20*, 6 (June 1977).