A Logically Distributed Approach for Structuring Office Systems [1]

Panayiotis K. Chrysanthis
David Stemple
Krithi Ramamritham

Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003 [2]

Abstract

An object-oriented office model is presented. It uses the object taxonomy of Booch featuring object classes based on calling patterns. Our model is motivated by that of Woo and Lochovsky, but has a number of differences, especially in the definition of the object classes and in the treatment of nested objects. An office application, setting a meeting among office workers, is defined in terms the office model.

1 INTRODUCTION

Until recently, research on office systems has focused on modeling office work as a centralized environment which could often support only a single office worker. This focus was mainly due to the influence of database technology. *Logically centralized* databases were an attractive solution to the problem of locating and managing the information necessary for office problem solving. However, since logically centralized databases adopt the transaction-based model, they can only support *phase-sequencing* applications by which we mean logically sequencing the execution of programs and allowing intercommunication between these programs (transactions) only through committed data in the database. Applications within which the various modules need to communicate during the course of their execution cannot be supported by phase-sequencing the modules. Consequently, office models based on a logically centralized database cannot support complex, non-routine office work requiring cooperation between different office workers. Cooperation in processing information implies communication, and allows for inconsistency in the information seen by the cooperators. Such temporary inconsistencies arising in cooperative problem solving, presents a problem that needs to be addressed in any model of office work.

An alternative to logically centralized databases is the *logically distributed* approach. This approach avoids centralizing the office knowledge but instead distributes it among knowledge bases which are allowed to communicate. Communicating knowledge bases can capture the behavior of the manual office, which is, after all, a group of specialized workers who perform their jobs concurrently and asynchronously (independently) while interacting cooperatively to achieve a common task. Inconsistencies may develop as a result of the communication or independence of the workers. It is the responsibility of the cooperating office workers to take any necessary correcting actions to reconcile inconsistencies. Modeling cooperation, inconsistency, and reconciliation in a natural and efficient manner has been a goal that we have pursued in defining the model presented in this paper, but we do not deal directly with reconciliation in this paper.

Our motivation is to define an office model based on the logically distributed approach, which while being higher-level and hence easy to use, is also implementable and hence easy to realize. The proposed office model represents office work using objects that model entities and activities of the manual office. The model supports the whole spectrum of office work [9, 8] and uses the classification of objects presented by [2]. It has its origins in the object-oriented office model of Woo and Lochovsky [16], but differs in the definition of objects and their classes as well as in the treatment of nested objects. We show how the model maps easily into an implementable form via the use of the interconnection and object sharing model of the Gutenberg distributed operating system kernel [10, 3, 12].

[1] This material is based upon work supported in part by the N.S.F. under grants DCR-8403097, and DCR-8500332.

[2] E-mail address: panos@ccs3.cs.umass.edu, stemple@cs.umass.edu, krithi@nirvan.cs.umass.edu.

Gutenberg is a port-based, object-oriented distributed operating system kernel designed and implemented at the University of Massachusetts. It presents to its users an *object virtual machine* in which all interactions among objects are structured along abstract data type lines. Its purpose is to facilitate the development of systems made up of many distributed, cooperating modules.

In the following section, the office model is presented and related work is outlined. A quick overview of the Gutenberg system is given in section 3. Section 4 discusses how the proposed office model is effectively structured within the Gutenberg environment without compromising any of the model's semantics. An office application, namely the setting of meetings between office workers, is used to illustrate the idea. Finally, section 5 concludes with a summary and future plans.

2 AN OFFICE MODEL

2.1 Overview of the Model

The proposed office model is object-oriented, allowing the representation of office work in the model to conform to the structure of the manual office. *Objects* are instances of abstract data types that have a state and are characterized by the operations that may be performed on them. The model uses objects to represent both active and passive office entities that participate in the office activities. Report and payroll programs are examples of the active entities in an office that carry out office tasks. Passive entities are exemplified by file cabinets and calendars, though as we will see later, there are advantages to be gained in modeling some passive entities as active objects.

Our office model incorporates an object type hierarchy. The root of the hierarchy is the class of *Generic Office Objects* which have the ability to communicate with any other object through explicit message exchange. Messages do not have the same semantics as Smalltalk messages [7], i.e., they are not the means by which a method is invoked on an object. On the contrary, during the execution of an operation on an object, the invoker and the invokee may exchange any number of messages. The ability of objects to communicate by passing messages allows for cooperative office work, such as making an appointment, or locating an object that has certain information.

The rules for the behavior of an object reside in the specification of its operations, normally encoded in a programming language. The model captures the behavior of objects using a mechanism that is basically an abstraction of operation procedures, namely a set of *behavior specification rules.* These rules are part of the state of objects and thus can be subject to change, allowing for dynamic change of behavior.

As we will see later, Generic Office Objects correspond to Gutenberg objects, and thus, are directly implementable. This has the advantage that all the semantics of the application captured by the model are preserved while moving from the design level down to the implementation.

The Generic Office Objects class is further divided into four subclasses based on the objects' ability to invoke and be invoked by other objects or by users: *Actors, Agents, ActorAgents,* and *Servers.* An object's calling pattern reflects the object's functional capabilities, resource requirements, and position in the modelled organization structure.

Actors are used to model the interface between the user and the various objects in the system. Actors can be invoked by users but not by any other object, and may invoke operations on any object of other types. Consequently, communication between Actors and other objects must be initiated from the Actors' sites. Actors are the primary initiators and monitors of the system's performance of office work. They are not simply user interface managers, such as window managers.

At the other extreme, **Servers** are only invocable by other objects; they may not invoke any other object. They correspond to the passive entities of the office that serve as repositories of information. Of course, the question of when to consider that an object calls another is sensitive to the level of abstraction being discussed as well as to the context in which the components are being defined. For example, we will want to consider an object a server in some contexts even though it may call operating system objects to achieve some of its function. By structuring Servers as active databases [13] with alerters and triggers, it is possible for passive office entities to demonstrate some form of 'intelligence', for example, in resolving inconsistencies or triggering an automatic clerical task.

Agents serve to perform some operation (task) on behalf of another object and in turn may request operations on other objects. Agents expand the one-to-one communication topology to arbitrary topologies

appropriate to given problem-solving activities by interposing themselves between cooperating objects. The one-to-one topology of the basic message passing facility, although efficient, is restrictive even in some cases of exchange of information, as, for example, when it is not known where the needed information is stored. Thus, Agents are the means of supporting the office problem-solving activity. Furthermore, Agents facilitate the achieving of different views of stored information in the system.

ActorAgents are objects sharing the properties of both Actors and Agents. That is, they can be invoked by both users and other objects, and they can invoke operations on other objects. ActorAgents as opposed to a combination of Actors and Agents, can support communication between users in a straight forward way. Also, they can support the interface between the user and non-routine tasks (see below). Given that objects are activated on demand, they also have implications for the efficient management of resources.

2.2 Modeling Office Activities

In our model, an office task is an activity that is initiated by either an Actor or ActorAgent and may involve any number of Agents and Servers. If the Agents involved in an office task execute on behalf of a single Actor or ActorAgent, then the office task is classified as a *routine task*. On the other hand, if the Agents involved execute on behalf of more than one Actor, then the office task is known as a *non-routine task*.

Routine tasks can usually be expressed procedurally and can perform their task without the help of the user. Routine tasks correspond to the predefined transactions in logically centralized databases.

A typical example of a routine task is filling a form, such as a purchase-order. This task can be modeled as follows: An Actor is used to model the procedure of filling a form; whereas a Server models the folder where the forms are stored. A user fills out a form by selecting the appropriate Actor which communicates with the appropriate Server to get a blank form. The Server passes to the Actor the rules for filling out the form along with the form (schema). The Actor presents a spreadsheet-like interface to the user by using the local behavior specification rules in conjunction with the received ones. That is, when the user fills out a field of the form, the Actor uses this information to either derive and/or retrieve (possibly with the help of Agents) from other Servers the necessary information that fills out as many empty fields as possible. When the entire form is filled out, the Actor sends it back to the Server for storing. The Server certifies the form before storing it. During the certification process, the Server looks for inconsistencies and for potential application conflicts, and warns the Actor accordingly. An example of a potential application conflict is the existence of a second form with the same values filled within a given time period.

Non-Routine Tasks are the office tasks that are not algorithmic enough to be expressed as a single procedure and are required to synthesize their actions with the help of users and cooperation with other objects during their execution (synthesized transactions). A trivial example of a non-routine task is message exchange or a conference among Actors with the help of one or more Agents. Budgeting [16] and scheduling a meeting for a group of people are more complex, non-routine office tasks.

In the scheduling example, each office worker has his/her personal calendar for keeping his/her appointments. Calendars can be modeled as Servers. Worker's schedulers can be modeled as ActorAgents. A user makes an appointment with another user(s) by invoking the Scheduler ActorAgent. Then the ActorAgent enters into negotiation with the other users' ActorAgents to find an appropriate meeting time. During the negotiation, the cooperating ActorAgents consult the local calendar and seek the approval or help of the user. We return to this example in section 4 where we discuss its implementation within the Gutenberg environment.

2.3 Relation to Woo and Lochovsky Model

The model of Woo and Lochovsky (W&L) supports four classes of object types, namely, *Data Objects, Task Objects, Task Monitor Objects,* and *Agent Objects.* Data Objects which store the inactive information, and Task Objects which correspond to the office procedures, are associated with Task Monitor Objects which control their execution with the help of the user and consultation rules. Task Monitor Objects correspond to logical workstations in the office. Agent Objects facilitate the communication between objects residing in different Task Monitor Objects.

There are several subtle difference between our model and this model. The first relates to the status of objects. Specifically, in our model there is no notion of nested objects (objects residing within other objects), or of an object controlling the execution and access of another object. In the model of W&L, Task and Data Objects reside within a Task Monitor Object which controls their execution and their access by other objects inside or outside the domain of the Task Monitor Object.

In our model, all objects have the same status and independence except for that implied by the calling pattern allowed by the object's class. The reason for this is that in almost all office activities, there are objects that participate in more than one office task or cooperation. In such cases, in W&L's model, it is not always clear under which Task Monitor Object the sharable object should reside. In our model, objects accessible by more than one Actor or ActorAgent will typically be a Server or an Agent that accepts requests from more than one object. Furthermore, an object itself can be made responsible for handling any conflicts among concurrent requests; this responsibility does not need to be assigned to another object as is the case in the W&L model using Task Monitor Objects.

Another difference between the two models is that in our model there is no notion of stateless objects such as Task Objects. A stateless object of W&L is represented in our model in one of two ways. The first is as an operation exported by a Server or an Agent, in which case the combination of Data Object and its Task Object correspond to a Server or Agent. The second way of handling the W&L stateless object is as internal procedure calls of an ActorAgent in which case the Task Monitor Object with the Task Object corresponds to an ActorAgent.

A final difference between the two models under consideration is that we have extended the notion of behavior specification rules which are similar to the W&L consultation rules, by allowing them to be a property of Generic Office Object class, i.e., they are not limited to a specific object type class. Since behavior specification rules are allowed to be changed dynamically, this allows for any object in the system to change its behavior.

3 THE GUTENBERG OBJECT MODEL

The Gutenberg system evolved from a series of design decisions concerning the nature of communication and protection in distributed systems. In Gutenberg, a distributed system is a group of *objects* that execute asynchronously and concurrently, interacting cooperatively to perform a task. Objects are implemented using processes (independently schedulable units of computation) which are hidden from each others' views. Processes are also referred to as the object *managers*. Managers synchronize operations on their objects and are the only subjects able to directly manipulate the objects.

Gutenberg objects can communicate only through explicit message exchange over communication channels called *ports*. Objects do not share address spaces. Since objects are instances of abstract data types, interprocess communication in Gutenberg is always in terms of requests for abstract data type operations. While Gutenberg enforces an object-oriented view on all interprocess communication, it does not enforce this view upon the programs running in a single process. The organization of intraprocess communication depends on the programming language used to build the process program and can be object-oriented or not.

Ports between objects are established based on the need to provide or request a service. That is, Gutenberg has adopted the client/server model for basic interprocess communication in which the user of a port, called the client, sends a request for an operation to the port server, the object's manager, which then performs the operation and may send back a reply. Objects can simultaneously be clients and servers.

A port is established using *functional addressing*. A client creates a port by naming the service (the operation) it would like to request using the port rather than identifying the server object. The advantage of this strategy is that it supports service transparency, allowing for dynamic object re-implementation and/or relocation, an important property for distributed systems. The client object does not have to know the identity of the server object, or whether that object executes on a local or remote machine. The server object does not even have to be active prior to the creation of the port. Manager activation and deactivation (process creation and destruction) in Gutenberg is a side-effect of port operations. There are no primitives to activate or deactivate object managers explicitly. Furthermore, object manager interconnections can be dynamically changed by transferring ports over other ports. In order to restrict the use of ports to

the functionality for which they have been created, a port is typed with respect to its directionality and message contents.

The kernel enforces a port-based access control to objects. An object A can create a port for requesting an operation on an object B only if it has the *capability* to execute that operation. After port creation, the only check that needs to be made when the object A requests access to the remote object B via this port is whether that object A has the capability to access that port. This required check is done at the node in which the object A executes and so it is a local check.

The capabilities for creating ports are stored in the *Interconnection Schema*, a persistent, distributed object managed by the kernel. Thus, the Interconnection Schema expresses all the potential object interconnections achievable by programs running under the kernel's control. This means that the Interconnection Schema represents the organization of applications runnable under the Gutenberg kernel at any given time. Besides enforcing interconnection structure, the Interconnection Schema also supplies the kernel the information needed to locate, and if necessary instantiate, the server of a port. It also contains the definitions of object managers, that is, components that provide access to the code that implements the operations of objects, and the rules for activating the object managers. New managers are introduced into the system by storing their definitions in the Interconnection Schema.

The Interconnection Schema is not the only repository for capabilities in Gutenberg. There are also *transient* capabilities which persist only as long as an owning object is active, and these are stored in lists associated with manager processes. At any time, each object manager in the system is associated with a segment of the Interconnection Schema, designated as its *active directory*, and with its transient capabilities stored in its *capability list*, abbreviated *c-list*. Each object manager is associated with a single c-list which cannot be shared. Objects managed by the same manager can share their manager's c-list. Gutenberg supports dynamic access control by allowing objects to traverse the Interconnection Schema acquiring new capabilities, or by transferring capabilities over ports.

One of the novel features of the Gutenberg system is the manner in which it keeps track of object identities. Functional addressing implies that the identities of object managers are not usable in establishing connections between objects, and since in many cases an object's identity in Gutenberg is the same as the identity of its manager, object identification itself can be problematic. Thus, Gutenberg capabilities are unlike other systems' capabilities that contain the identities of the objects on which they allow operations. The reason for this difference is that Gutenberg capabilities are capabilities that allow operations on kernel objects, namely the entities in the interconnection schema and ports. These capabilities are used to create privileges for operating on user-defined objects such as those in our office model. The identification of user-defined objects is accomplished by using a capability unique to Gutenberg: the *cooperation class* [12]. Cooperation classes are generic capabilities that can be attached to distributed activities in order to provide the basis for coordinating cooperation. Communication among objects/managers that have no means of addressing each other explicitly is one of the major activities cooperation classes facilitate. This is the way they will be exploited in implementing the office model in Gutenberg.

4 An Example

We now turn to the problem of implementing our office model using the facilities of the Gutenberg kernel. We will address this problem by outlining the development of an application, namely, scheduling meetings, that features cooperation among asynchronous, autonomous entities.

4.1 Outline of the Application

We model the scheduling activity by defining three main objects, one modeling a worker scheduling his/her activities, another modeling his/her calendar, and the third modeling the scheduler for a group of workers. Other objects that would be needed in this application are not presented; these include a registry (agent or server) object that maintains the name of office workers. We first give an overview of these objects and how they interact to solve the problem of scheduling a meeting. After this, we give the specification rules for two of the scheduler object's operations. We then outline the mapping of the objects onto the Gutenberg object model.

The first object is the scheduler that controls the task of scheduling a meeting for a worker. The scheduler is an ActorAgent since it is invoked by the user and by other objects. The operations of the scheduler are

1. MakeAppointment - the operation that is invoked by the user when a meeting needs to be scheduled,
2. RequestAppointment - the operation that is invoked by other schedulers that are trying to schedule meetings for their users,
3. CancelAppointment - the operation that cancels meetings, invoked by the scheduler's user,
4. CallOffAppointment - the operation that cancels meetings, invoked by other schedulers.

Calendar objects are modeled as Servers and thus make no requests of other objects. Calendars allow the following operations to be requested of them:

1. GetSlots - returns the free slots that meet the restrictions supplied with the request,
2. ReserveSlots - sets a group of slots reserved (during a negotiation to settle on a time for a meeting),
3. FixSlots - sets slots as confirmed times for a meeting,
4. FreeSlots - frees slots from reserved or confirmed status,
5. FixPeriodicSlots - sets slots aside on a periodic basis (e.g., weekly),
6. FreePeriodicSlots - frees slots on a periodic basis,
7. TodaysMeetings - returns meetings for today,
8. NextMeeting - returns next meeting information.

The third object, the Scheduler for a group of workers is modeled as an Agent and has the same interface as the Scheduler of a single worker.

A user initiates the task of scheduling a meeting by invoking the MakeAppointment operation of his/her scheduler object with three input parameters: a set of users or a predicate which defines the users to be contacted, a set of restrictions such as date, time, length, priority, deadline, etc., and the purpose/agenda of the meeting. In the simplest case in which the appointment is between two users, the scheduler requests free slots that meet the restrictions from the initiator's calendar, and requests an appointment from the other user by making a RequestAppointment call to the other user's scheduler. The requesting scheduler initially sends its free slots along with the restrictions and the purpose of the meeting. Subsequently, the responding scheduler requests free slots which meet the restrictions from its local calendar, and enters into negotiation with the caller. If a decision is reached, the users' approval is requested if needed, and the meeting is confirmed by making the appropriate FixSlots request; otherwise, the help of the user is sought or the appointment is canceled.

In the case in which a scheduler is invoked with a set of users or a predicate, the scheduler, in turn, invokes a group scheduler object, named MeetingAgent, to establish the necessary communication and coordinate the negotiation process.

At any given time, a user may initiate any number of tasks trying to make an appointment with another user or a group of users, while at the same time other users may request an appointment with him/her.

Figures 1 and 2 give an object-oriented pseudo-code for the specification rules of the simplified versions of MakeAppointment and RequestAppointment operations of the scheduler object. In this form of pseudo-code, each statement has a stereotyped form, consisting mainly of an operation, an object name and a list of arguments. These are separated by prepositions that are appropriate to the operation names, e.g., from, to, with. The actual preposition used has no semantic significance. Operations have results in general and these can be assigned to variables or may be referred to by the function, ResultOf, that takes the operation name as its argument. The pseudo-code allows the specification of continuing reception of results from a single operation by using a form of a guarded command [5]. The keyword Dynamic is used to specify an asynchronous updating of a variable that has been updated in the body of an operation. The statement containing the variable assignment is flagged with [Dynamic]. The manner of the continuing update is specified after the body of an operation's code by a statement containing the condition under which the update takes place and the update assignment. This technique is used in specifying operation behavior that accommodates the reception of changing information that is typical in applications involving simultaneous, competing negotiations, such as scheduling meetings.

```
Operation MakeAppointment(Whom, Restrictions, Purpose): Appointment;

 If Not Singleton(Whom)
   Then
    GroupAppointment from MeetingAgent with (Whom, Restrictions, Purpose);
    ReturnExit ResultOf(GroupAppointment);
   Else
    Slots <- GetSlots from MyCalendar with Restrictions; [Dynamic]
    RequestAppointment from Whom with (User, Restrictions, Purpose, Slots);
    Match with (Slots, ResultOf(RequestAppointment));
    If Empty(ResultOf(Match))
      Then ConsultUser;
      Else
       Reserved <- ReserveSlots in MyCalendar with ResultOf(Match);
       If Reserved <> Success
         Then ConsultUser;
         Else
          RequestAppointment from whom with Reserved;
          If Empty(ResultOf(RequestAppointment))
            Then ConsultUser;
            Else
             Appt <- FixSlots in MyCalendar
                              with HeadOf(ResultOf(RequestAppointment);
             FreeSlots in MyCalendar with Difference(Reserved,{Appt});
             ReturnExit Appt;
 Dynamic:
      On New Result from GetSlots from MyCalendar with Restrictions:
         Slots <- ResultOf(GetSlots);
```

Figure 1: MakeAppointment Specification

```
Operation RequestAppointment(Whom, Restrictions, Purpose, When): Appointment;

 Slots <- GetSlots from MyCalendar with Restrictions; [Dynamic]
 Match with (Slots, When);
 If Empty(ResultOf(Match))
   Then ConsultUser;
   Else
    Return ResultOf(Match);
    Match with (Slots, ResultOf(Match))
    Reserved <- ReserveSlots in MyCalendar with ResultOf(Match);
    If Reserved <> Success
      Then ConsultUser;
      Else
       RequestAppointment from whom with Reserved;
       If Empty(Reserved)
         Then ConsultUser;
         Else
         Appt <- FixSlots in MyCalendar with HeadOf(ResultOf(RequestAppointment);
         FreeSlots in MyCalendar with Difference(Reserved,{Appt});
         ReturnExit Appt;
 Dynamic:
      On New Result from GetSlots from MyCalendar with Restrictions:
         Slots <- ResultOf(GetSlots);
```

Figure 2: RequestAppointment Specification

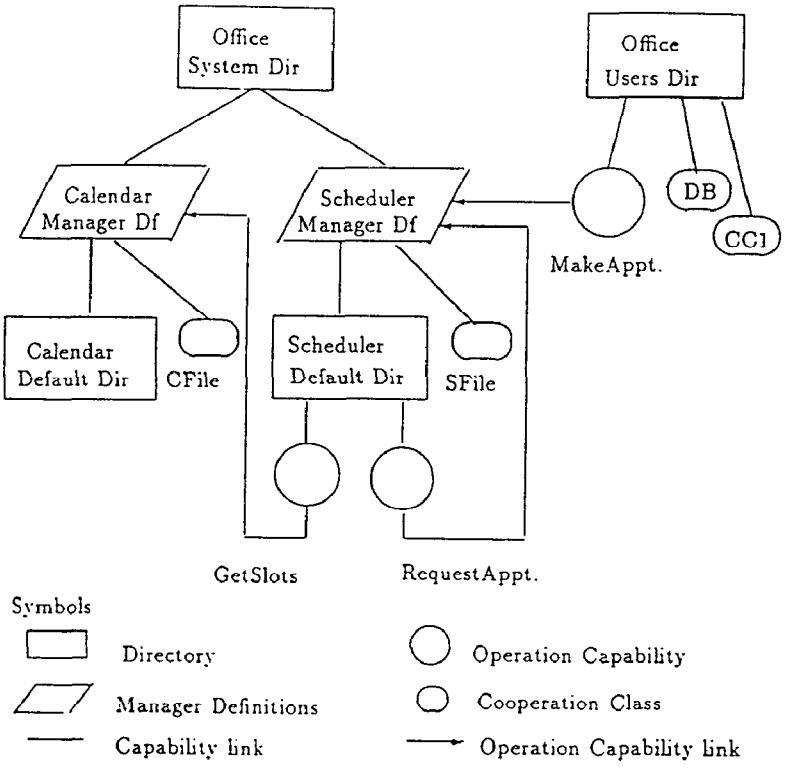

Figure 3: Segment of the Interconnection Schema of the Scheduling Application

This is a sample Interconnection Schema of a basic Scheduling system. The Office System directory contains the manager definitions of the Calendar and Scheduler objects created by the system developer. The directory of the office users contains capabilities that the users can use to invoke operations on a scheduler object. The Scheduler manager's default active directory also contains capabilities allowing the Scheduler objects to invoke operations on other Scheduler objects as well as on Calendar objects. The Calendar manager's default directory is empty, as it should be, since Calendar Objects as Servers cannot invoke operations on any other object.

4.2 Definition of the Application in Gutenberg

Now we will map this model onto Gutenberg facilities. Each office object is mapped onto a Gutenberg object. The Interconnection Schema shown in Figure 3 contains the interfaces that the scheduler and calendar objects present to their users. These interfaces are used to create ports to use in requesting the objects' operations and facilitating the negotiations. The pseudo-code in Figures 1 and 2 needs to be translated into a programming language for inclusion in a manager definition along with the manager's *default active directory* and *lifetype*[3]. The default active directory becomes the active directory of any object manager instantiated from this manager definition. The lifetype of an object manager indicates whether all the object instances of the type are managed by a single manager, or each object is managed by a different manager, corresponding to the Gutenberg lifetypes: *Class Conservative* and *Conservative*.

The kernel determines whether a newly created port is to be connected to a new object manager instantiated from a manager definition or to an already existing one, from the lifetype of the object manager and the cooperation class used at the port creation time. In the case of Creative lifetype, a new object manager is created for each new port created. This is useful in cases of shortlived objects owned, in effect, by the caller. Conservative lifetype, on the other extreme, creates a new object manager from a manager definition only if no other object manager for the object type is running on the system. Class-conservative allows new object managers to be instantiated selectively based on the cooperation class

[3]known also as *instantiation protocol*

capability supplied at port creation time. Class-conservative managers are the mechanism for implementing most office objects that need an identity (the cooperation class) that can be shared among other objects.

The lifetype of the Scheduler and Calendar objects is Class-Conservative, whereas the Registry is Conservative. By associating a cooperation class with each user in the system and using it to identify him/her while making an appointment, the proper Schedulers and Calendars are invoked.

4.3 Discussion

The two main issues in implementing the Scheduler and Calendar objects in Gutenberg are how to map these objects onto Gutenberg objects, and how to group, couple and uncouple them in an effective way while preserving the semantics of the design. Both issues need to be addressed in any application and have straightforward solutions within the Gutenberg environment.

Generic Office objects correspond to Gutenberg objects. The Gutenberg kernel does not interpret their types except to consult their lifetype when their managers need to be instantiated. Gutenberg allows office objects to be implemented in a way that it is most appropriate for the given application. For example, two Servers providing database facilities in the system can be implemented using different database models. Furthermore, it is possible to adopt type-specific concurrency control and recovery [11, 15, 1], improving the overall system performance. Existing tasks require simple interface modifications in order to be integrated into the system, not re-implementation.

The second issue deals with object interconnections. In general, objects are accessible by any other object. However, the organization structure (hierarchical or network) defines statically the object associations (who has the privilege to speak with whom). Gutenberg captures the organization structure in the Interconnection Schema while at the same time can capture any dynamic object associations by allowing port redirection. Consider the following example: A worker searching for information on a company usually calls the receptionist of the company who redirects the worker's call to the employee in the company who most probably will be in a position to give the information. That employee may immediately be able to give out the information or the information may require an approval from someone else in the company. In the latter case, the employee either further redirects the call to his/her supervisor, if the company policy allows it, or he/she needs to get the approval and accordingly release the information. If the approval takes some time, the employee could either ask the caller to call back or leave his/her telephone number. Any of these scenarios can be modeled by Gutenberg communication mechanisms.

The declarative form of the Interconnection Schema facilitates further the open-ended nature of an organization. That is, the Interconnection Schema can easily reflect changes to the structure of the organization. The arrival and departure of office workers correspond to changes to manager definitions; whereas responsibility redistribution corresponds to redistribution of capabilities to create ports.

The grouping of office activities into cooperations using the notion of cooperation class not only ensures proper object association, but it also provides an elegant and efficient way to capture the logical workstations in the office and facilitate their changes. By associating segments of the Interconnection Schema with cooperation classes and controlling their access using these cooperation classes, the redistribution of responsibilities involves a single capability, the cooperation class.

5 CONCLUSION

An object-oriented office model is presented. It uses the object taxonomy of Booch featuring object classes based on calling patterns. Our model is motivated by that of Woo and Lochovsky, but has a number of differences, discussed in section 2.3, especially in the definition of the object classes and in the treatment of nested objects. An office application, setting a meeting among office workers, is defined in terms the office model. We showed how it can be implemented in Gutenberg, an object-oriented distributed operating system.

We are in the process of building office applications along the lines outlined in this paper. Our agenda includes exploring the issues of inconsistencies arising during negotiations and their resolution through other negotiations, achieving reliability using Gutenberg atomicity features [14, 4] that we have not dealt with in this paper, and automatic or semi-automatic translation of the object-oriented specification language used

in the example into code for manager implementation on Gutenberg. Run-time code synthesis is another area that we would like to study in order to explore further the power of typed port redirection in achieving dynamic modification of behavior in office systems.

6 References

1. Badrinath, B. R., and, Ramamritham, K., 'Semantic-Based Concurrency Control: Beyond Commutative,' *Proceedings of the Third International Conference on Data Engineering*, February 1987.

2. Booch G., 'Object-Oriented Development, *Transactions on Software Engineering,* vol 12, no. 2, February 1986.

3. Chrysanthis, P. K., Ramamritham, K., Stemple, D. W., Vinter, S. T., 'The Gutenberg Operating System Kernel,' *Proceedings of the First ACM/IEEE Fall Join Computer Conference*, November, 1986.

4. Chrysanthis P. K., Ramamritham K., 'Capturing the Structure and Behavior of Complex Transactions,' *Proceedings of the Third Workshop on Large Grain Parallelism*, SEI, Carnegie Mellon University, Pittsburgh, October 1989.

5. Dijkstra E., 'Guarded Commands, Nondeterminancy and Formal Derivation of Programs,' *Communications of ACM,* vol. 18, no.8, August 1975.

6. Ellis C. A., and Nutt G. J., 'Office Information Systems and Computer Science,' *ACM Computing Surveys,* vol. 12, no.1, March 1980.

7. Goldberg, A., and Robson, D., 'Smalltalk-80: The Language and its Implementation,' Addison-Wesley Edition, 1983.

8. Lochovsky F. H.,'A knowledge-based approach to supporting office work,' *Data Engineering IEEE Computer Society,* vol. 16, no. 3, September 1983.

9. Panko, R. R., '38 offices: Analyzing needs in individual offices,' *ACM Transactions on Office Information Systems,* vol. 2, no.3, July 1984.

10. Ramamritham, K., Briggs, D., Stemple, D. W., Vinter, S. T., 'Privilege Transfer and Revocation in a Port-Based System,' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, May 1986.

11. Schwarz, P., and Spector, A., 'Synchronizing Shared Abstract Data Types,' *ACM Transactions on Computer Systems,* vol. 2, no. 3, August 1984.

12. Stemple, D. W., Vinter, S., Ramamritham, K., 'Functional Addressing in Gutenberg: Interprocess Communication Without Process Identifiers,' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 11, December 1986.

13. Stonebraker, M., Hanson, E., Hong, C. H., 'The Design of the POSTGRES Rules System,' *Proceedings of the Third International Conference on Data Engineering*, February 1987.

14. Vinter, S. T., Ramamritham, K., Stemple, D. W., 'Recoverable Communicating Actions,' *Proceedings of the fifth International Conference on Distributed Computing Systems*, May 1986.

15. Weihl W. E., 'Specification and Implementation of atomic data types,' Ph.D. Thesis, Tech. Rep. MIT/LCS/TR-314, MIT Laboratoty for Computer Science, March 1984.

16. Woo, C. C., and, Lochovsky F. H., 'Supporting Distributed Office Problem Solving in Organizations,' *ACM Transactions on Office Information Systems,* vol. 4, no. 3, July 1986.