An Open Data Management Architecture for Groupware Applications

Katsumi Kanasaki
Software Research Center
Ricoh Company, Ltd.
1-1-17, Koishikawa, Bunkyo, Tokyo 112, Japan

Tosiyasu L. Kunii
Department of Information Science
The University of Tokyo
7-3-1, Hongo, Bunkyo, Tokyo 113, Japan

Abstract
Object-oriented data management is applied to a system called SCOPE, which supports various groupware applications. It is shown with examples that the object-oriented paradigm is useful for representing forms and procedures in a groupware environment. In order to make SCOPE open to other systems, a communication layer is introduced under the object-oriented layer. Open systems also require easy schema evolution. The programming-by-example metaphor of spreadsheet systems is employed to easily define rules for objects. The process and module structure of SCOPE is also presented.

Keywords: object-oriented databases, office information systems, groupware, open systems, schema evolution, access-oriented programming, spreadsheet

1 Introduction
The importance of groupware or computer-supported cooperative work (CSCW) [1] has been increasingly emphasized in recent years. Data management becomes a key issue for groupware applications when the amount of communications builds up, or a recurring structure or pattern can be seen in the communications.

Both database management systems and groupware systems aim at information sharing. Research on groupware, however, reveals various aspects of data management that are different from traditional database management systems. For instance, new styles of access control and concurrency control are covered by [2].

This paper focuses on advantages and problems of object-oriented data management for groupware applications. The authors applied class hierarchy to electronic secretaries in a previous paper [3]. In the paper, office procedure was represented by relationships such as PART-OF and BEFORE. Subsequent experience reveals that flexible programming functions are required to represent office procedures. Hence, this paper employs an object-oriented programming paradigm.

Some advantages of object-oriented data management are well proven. The Lens system [4] is a groupware system that utilizes object-oriented data management to deal with both ill-structured and well-structured messages. The class hierarchy can be also utilized to absorb differences among views of different users [5]. This paper shows how to apply object-oriented data management to various forms used in cooperative work.

Existing object-oriented systems, however, have problems. In particular, object-oriented architecture often makes a system closed. For instance, objects cannot be created outside the system.

Groupware systems are relatively useless unless they can communicate with external agents. Hence, the systems must be open. Let us consider offices, which are open systems [6]. Offices must accept various forms that are not defined by themselves.

The SCOPE system proposed in this paper has a layer for communications, which is open to the other systems. An
Table 1: The fields of an instance of the class Calendar.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>April 4, 1991</td>
</tr>
<tr>
<td>Begin</td>
<td>9:00</td>
</tr>
<tr>
<td>End</td>
<td>18:00</td>
</tr>
<tr>
<td>Period</td>
<td>9:00</td>
</tr>
<tr>
<td>Event</td>
<td>DASFAA Symposium</td>
</tr>
<tr>
<td>Cancel</td>
<td></td>
</tr>
<tr>
<td>Notify</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2: The rules and the constraints for the class Calendar.

<table>
<thead>
<tr>
<th>Field</th>
<th>Rule</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td>Begin</td>
<td></td>
<td>TIME</td>
</tr>
<tr>
<td>End</td>
<td></td>
<td>TIME</td>
</tr>
<tr>
<td>Period</td>
<td>End - Begin</td>
<td>ACTION</td>
</tr>
<tr>
<td>Event</td>
<td></td>
<td>ACTION</td>
</tr>
<tr>
<td>Cancel</td>
<td>NOTIFY CREATOR AT TIME(Date, Begin)</td>
<td>ACTION</td>
</tr>
</tbody>
</table>

2 Three-Layer Data Modeling for Open Systems

ANSI/X3/SPARC [12] introduced three-layer data modeling to make conceptual schemata stable even if users' views and physical implementation change.

Let us introduce a similar three-layer model to make conceptual schemata useful even if users' tasks and surrounding situation change. These changes frequently occur in actual offices, because the offices are open systems.

ANSI/X3/SPARC assumed that a data administrator can design a conceptual schema at first. In an open environment however, nobody has complete information about the world. Hence, a stable conceptual schema cannot be designed.

The schema is, instead, gradually constructed and modified so that it is consistent with current occurrences of data. In this way, the conceptual schema always reflects the current understanding of the world.

2.1 Conceptual Layer

For the conceptual layer, an object-oriented data model is adopted, because object-oriented models are suitable for representing human understanding of the world.

Let us consider an example. The names and the values of the fields for an instance are shown in Table 1.

In Section 3, the process and module architecture of SCOPE is briefly presented. A client/server model using an open inter-process-communication protocol is employed so that SCOPE is open to other systems.
who created the instance checks the field above, an instance is sent to participants when the person of inconsistency due to concurrent execution is reduced by agent that is specified by the field identifies an agent, which is a user or a virtual user that corresponds to a specific role in a group. The possibility of inconsistency due to concurrent execution is reduced by specifying the agent that will fill the field.

The field that has a trigger action is internally a tri-state variable. The value of the field can be empty, Do, or Done. When the field is picked by a user, the value Do is assigned to the field. If time is specified as in the example above, the value Do is assigned when the time comes. The trigger action is then activated. When the execution of the trigger action is finished, the value Done is assigned to the field, and the user can see a check in the field.

In this way, any action is invoked as a trigger action; an access-oriented programming scheme [13] is employed. The action is executed when a field is updated. In other words, the value of the field depends on whether or not the action has been executed. Users can ascertain whether or not the action has been executed by investigating the value of the field.

A class, say Meeting Announcement, can be created as a subclass of Calendar. An instance of Meeting Announcement corresponds to a form that announces a meeting. The rules and the constraints shown in Table 3 are attached to the class.

<table>
<thead>
<tr>
<th>Field</th>
<th>Rule</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Creator INPUT</td>
<td>DATE</td>
</tr>
<tr>
<td>Begin</td>
<td>Creator INPUT</td>
<td>TIME</td>
</tr>
<tr>
<td>End</td>
<td>Creator INPUT</td>
<td>TIME</td>
</tr>
<tr>
<td>Period</td>
<td>End—Begin</td>
<td></td>
</tr>
<tr>
<td>Event</td>
<td>Creator INPUT</td>
<td></td>
</tr>
<tr>
<td>Place</td>
<td>Creator INPUT</td>
<td></td>
</tr>
<tr>
<td>Participants</td>
<td>Creator INPUT</td>
<td>AGENTS</td>
</tr>
<tr>
<td>Creator</td>
<td>CREATOR</td>
<td>AGENT</td>
</tr>
<tr>
<td>Announce</td>
<td>Creator SEND TO Participants</td>
<td>ACTION</td>
</tr>
<tr>
<td>Cancel</td>
<td></td>
<td>ACTION</td>
</tr>
<tr>
<td>Notify</td>
<td>NOTIFY Participants AT TIME(Date,Begin)</td>
<td>ACTION</td>
</tr>
</tbody>
</table>

Table 3: The rules and the constraints for the class Meeting Announcement, which is a subclass of Calendar. The rules define the structure of group communication.

The rule for the field Notify means that the user who created an instance is notified when the corresponding event occurs. Such an action that is attached to a field is called a trigger action for the field. The field is marked at the same time.

The field that has a trigger action is internally a tri-state variable. The value of the field can be empty, Do, or Done. When the field is picked by a user, the value Do is assigned to the field. If time is specified as in the example above, the value Do is assigned when the time comes. The trigger action is then activated. When the execution of the trigger action is finished, the value Done is assigned to the field, and the user can see a check in the field.

In this way, any action is invoked as a trigger action; an access-oriented programming scheme [13] is employed. The action is executed when a field is updated. In other words, the value of the field depends on whether or not the action has been executed. Users can ascertain whether or not the action has been executed by investigating the value of the field.

A class, say Meeting Announcement, can be created as a subclass of Calendar. An instance of Meeting Announcement corresponds to a form that announces a meeting. The rules and the constraints shown in Table 3 are attached to the class.

If the constraint AGENT is specified for a field, a value of the field identifies an agent, which is a user or a virtual user that corresponds to a specific role in a group.

The rule for a field specifies how the value of the field is determined. The rule CREATOR for the field Creator denotes that the field of an instance is filled by the name of the agent that creates the instance.

The rule Creator INPUT means that the field is filled by the agent that is specified by the field Creator. The possibility of inconsistency due to concurrent execution is reduced by specifying the agent that will fill the field.

SEND is another important keyword. In the example above, an instance is sent to participants when the person who created the instance checks the field Announce. Thus the fact that the meeting has been announced is recorded in the field.

It is assumed that each agent has its own copy of objects, but all agents share the same name space for classes. If an agent received an instance of an unknown class, the class is created.

As with other object-oriented programming systems, rules for Calendar are not used for instances of Meeting Announcement if the rules are overridden by rules for Meeting Announcement.

However, an instance of a class is, in theory, also an instance of its superclass. For example, an instance of Meeting Announcement is an instance of Calendar. It is useful that Calendar also contains meeting announcements. For example, overbooking can be found on Calendar.

Now, let us design the class Vacation Request, which represents a form to apply for a vacation. Its rules and constraints are shown in Table 4.

Vacation Request is a subclass of Calendar and also a subclass of Approval Procedure; multiple inheritance [14,15] is allowed. The class Approval Procedure shown in Table 5 is a collection of forms that must be approved by a supervisor.

First, an instance of Vacation Request is sent to a supervisor when an applicant checks the field Apply. The supervisor can see various kinds of forms that require approval as instances of Approval Procedure. When the supervisor approves the request by checking the field Approve, the instance is sent back to the applicant. The system recognizes that the received instance exists, and the instance is updated so that the applicant can see that the request has been approved.

In the Lens system, a message, which is not updated, is a fundamental object. In most cases, however, a message is related to another message. Relationship among the messages must be described so that readers understand the context of a message. In our model, related messages are combined into a form, and users can see how the form is updated by various agents that have various roles for the form.

As you can see from the examples above, class hierarchy gives various kinds of benefits. First of all, it makes class cre-
Table 4: The rules and the constraints for the class *Vacation Request*. The fields are inherited from the classes *Calendar* and *Approval Procedure*.

<table>
<thead>
<tr>
<th>Field</th>
<th>Rule</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Applicant INPUT</td>
<td>DATE</td>
</tr>
<tr>
<td>Begin</td>
<td>0:00</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td>24:00</td>
<td></td>
</tr>
<tr>
<td>Period</td>
<td>End—Begin</td>
<td></td>
</tr>
<tr>
<td>Event</td>
<td>CONCATENATE(Applicant, &quot;3 vacation&quot;)</td>
<td></td>
</tr>
<tr>
<td>Applicant</td>
<td>CREATOR</td>
<td></td>
</tr>
<tr>
<td>Supervisor</td>
<td>Applicant INPUT</td>
<td>AGENT</td>
</tr>
<tr>
<td>Apply</td>
<td>Applicant SEND TO Supervisor</td>
<td>ACTION</td>
</tr>
<tr>
<td>Approve</td>
<td>Supervisor SEND TO Applicant</td>
<td>ACTION</td>
</tr>
<tr>
<td>Reject</td>
<td>Supervisor SEND TO Applicant</td>
<td>ACTION</td>
</tr>
<tr>
<td>Comment</td>
<td>Supervisor INPUT</td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td>Reject</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: The rules and the constraints for the class *Approval Procedure*.

<table>
<thead>
<tr>
<th>Field</th>
<th>Rule</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant</td>
<td>CREATOR</td>
<td>AGENT</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Applicant INPUT</td>
<td>ACTION</td>
</tr>
<tr>
<td>Apply</td>
<td>Applicant SEND TO Supervisor</td>
<td>ACTION</td>
</tr>
<tr>
<td>Approve</td>
<td>Supervisor SEND TO Applicant</td>
<td>ACTION</td>
</tr>
<tr>
<td>Reject</td>
<td>Supervisor SEND TO Applicant</td>
<td>ACTION</td>
</tr>
<tr>
<td>Comment</td>
<td>Supervisor INPUT</td>
<td></td>
</tr>
</tbody>
</table>

2.2 Communication Layer

Object-oriented architecture often makes a system closed. Since objects must be created in the system, there is no straightforward way to import external objects. Users often have to write a program that reads data and creates objects. The program must not create a new object if the object exists, and the program must be written in an extensible manner, because the external environment is constantly changing. Such a program is hard to implement.

Unless such a program is written, an object once exported from the system cannot be re-imported as it was. When two object-oriented systems are connected through another system that is not object-oriented, the object-oriented systems cannot work as a single system.

Groupware systems must be open systems, because they are relatively useless unless they can communicate with the outside world. For instance, even if a system is used by all members of an office, the system must be able to communicate with the outside world, because the office communicates with the outside world. In a more practical situation, some members of the office may have been using another system. The systems, in this case, must be able to communicate each other.

This paper proposes an open layer under the object-oriented layer. Let us call the open layer the communication layer, because agents in and out of the system communicate through this layer.

The communication layer gives a uniform protocol that does not depend on whether or not a sender and/or receiver reside in the system.

An example of a message that is sent to announce a meet-
Class: Meeting Announcement
Object-ID: <SCOPE18459@astaire.src.ricoh.co.jp>
Date: September 3, 1990
Begin: 13:00
End: 15:00
Period: 2:00
Event: SCOPE design review
Place: 4th meeting room
Participants: takako, kuramoti, lihong, tunenori, kana
Creator: kana
Announce: Done
Cancel:
Notify:

Figure 1: A message that is sent to announce a meeting.

Class: Meeting Announcement
Superclass: Calendar
Rule: Date: Creator INPUT; DATE
Rule: Begin: Creator INPUT; TIME
Rule: End: Creator INPUT; TIME
Rule: Period: End-Begin;
Rule: Event: Creator INPUT;
Rule: Place: Creator INPUT;
Rule: Participants: Creator INPUT; AGENTS
Rule: Creator: CREATOR; AGENT
Rule: Announce: Creator SEND TO Participants; ACTION
Rule: Cancel: ; ACTION
Rule: Notify: NOTIFY Participants AT TIME(Date,Begin); ACTION

Figure 2: A message that creates the new class Meeting Announcement.

The message is shown in Figure 1.

The message is represented as plain text, so that the message can be edited with any text editor. The format of the message is consistent with the Internet mail format [16], because the format is familiar to mail system users.

Each line of the message consists of a field name, a colon, and a field value. Ordinary text at the end of the message is ignored by SCOPE, so that users can put comments and the usual mail body at the end.

The field Object-ID specifies an object that receives the message. An object-ID can be assigned out of the SCOPE system, so that the system is open to other systems.

If the object exists, in other words the agent that receives the message has the object, it is updated. The message contains the names of updated fields and the new values for the fields. If a trigger action is defined for an updated field, the action is activated.

If the specified object does not exist, the object is created. The fields of the object are initialized with the values specified in the message.

If the specified class does not exist, the class is created. In this case, the fields Superclass and Rule must be specified in a message as shown in Figure 2.

If a message has the field Template, the created object is not a real instance but a template of an instance. Trigger actions attached to the template are not executed. A new instance can be created by copying another instance or the template.

For example, a template for Meeting Announcement can be distributed to the users who may announce a meeting. A user who is going to announce a meeting creates a new instance of the meeting announcement by copying the template.

It must be observed that the format for a message can be also used to represent an object, because the message is represented in an access-oriented way. Objects can be stored in a file using the same format, so that the file can be edited with any text editor.

2.3 User-Interface Layer

Open systems cannot have complete information on the outside world. Open systems must always update their knowledge in the course of communications with the outside.

Consequently, new classes of objects must be frequently
created, and the definition of existing classes must be often modified. How can we make such schema evolution easy?

As presented above, class hierarchy helps to make class definition easier. The definition of superclasses can be reused for the class definition. Users, however, must write rules that are not defined in the superclasses.

In the course of dynamic schema evolution, we are going to create a new class when a new instance of the class must be created. At that time, we have a specific case that corresponds to the instance. It is easier to write rules for the specific case than rules for generic cases. In fact, we cannot write complete rules for generic cases, because we have incomplete information. When we encounter another case, the rules will be modified so that they can be applied to the both cases. The rules thus evolve.

Spreadsheet systems adopt the programming-by-example metaphor to cope with the same situation. When a rule is written and applied to an example on a worksheet, a user can see the result.

SCOPE employs the metaphor of spreadsheet systems. A layer for user interface, which looks like a spreadsheet, is constructed on top of the object-oriented layer.

Users can process data as if the users are using a normal spreadsheet system. For example, the values received from many users can be easily summed up. In contrast with spreadsheet systems, however, the data that are received from other agents must be inserted into a worksheet with no user interaction.

In SCOPE, a table of instances of a class can be mapped on an area of a worksheet so that received data automatically appears on the worksheet. The contents and the size of the table change when a message for the class is received.

An example of the table is shown in Figure 3. Each row corresponds to an instance or a template of instances. In addition to field values, the class name of each instance, which can belong to a subclass of the class that corresponds to the table, is shown in the row.

A new class and the table for the class can be created with a command. In this case, the names of superclasses and the location of the table on a worksheet must be specified. Additional fields can be created by inserting a column. A field name must be specified for each additional field. A new instance can be created by filling cells of a row.

The input for a cell can be a value as well as a rule. Users can thus write rules using an actual example. All instances of a class must have the same rules unless they are instances of a subclass. The rules are also used for the instances that are received from other agents. Users can create a template that has rules even if there is no instance.

A rule specified in a cell can refer a value in another cell. If only cells for the same instance are referred by the rule, the rule is attached to the class and sent to other agents when the instance is sent. If other cells are referred by the rule, the rule is local to the worksheet. The local rule is, however, copied when a new instance is inserted in the table. For example, a local rule that refers to a lookup table can find the name of the supervisor who will approve a vacation request.

Users can see the result of the rules, so that the users can feel sure that the rules are correct. However, if an instance was sent to other agents with the rules, the user who sent the instance cannot see the result of the rules.

SCOPE provides a simulation mode to cope with this problem. In the simulation mode, the instance is actually not sent to the other agents. The instance is, instead, sent to a temporary table that corresponds to the table of the receiver. The sender can interact with the temporary table.

When the class for a received instance does not exist, the class is created. In this case, the rules for the class are copied from a received message. The instance appears in the tables that correspond to superclasses of the created class. The name of the class can be seen in the tables. Users can then create a table for the class.

This scheme assumes that all agents use the same class for the same kind of matters. In real situation, however, different groups often use different classes for the same or very similar matters.

Rules on worksheets can be used to cope with this problem. Let us assume that two different groups use the classes A and B for the same matter. The classes can have slightly different sets of fields and rules. When an agent that uses A receives an instance of B, the instance can be converted into an instance of A.

The conversion rules are written in a special template called an import template. As in the example above, an import template for A is created, where fields of the tem-
plate refer fields of B. When an instance of B is received, an instance of A is created and inserted into the table that corresponds to A.

3 IPC-Based System Architecture for Open Systems

SCOPE must process received mail as well as user interaction. In some cases, more than one user may interact with the same worksheet. A client/server model is employed to handle the concurrent processing. Each agent has a server process, which is constantly running and accepts requests from client processes.

The communication layer described earlier provides the mail interface that allows other systems to communicate with SCOPE. Some systems, however, must interact with SCOPE in a more efficient manner than the mail interface. The client/server model provides an interface that is suitable for such systems, because the protocol of the inter-process communication (IPC) between the clients and the server is open.

Figure 4 shows the process and module structure of the SCOPE system.

The user interface is implemented as a client process that communicates with the SCOPE server process as well as the server process of the X window system. The user-interface process can update cells on a worksheet by sending a request to the server process.

In order to display a worksheet, the user-interface process first declares an area to be displayed. When cells in the area are updated, the server notifies the user-interface process. The user-interface process works in such an event-driven manner.

A worksheet manager in the server process knows how to map cells on worksheets into fields of objects. Requests from the user-interface process are converted here into requests on objects. Data on worksheets that are not mapped into objects are managed by the worksheet manager.

Objects are managed by an object manager in the server process. The object manager knows how objects are stored. Some data are stored in files that can be directly accessed by users. The format for messages used in the communication layer can be used to represent objects in the file. When such a file is updated, for example by a text editor, the object manager updates its internal state so that it is consistent with the file.

The worksheet manager and the object manager share a rule interpreter. Rules written on worksheets or objects are recognized by the rule interpreter.
Usual client processes communicate with the object manager. If a client process would like to control user interaction, the client can communicate with the worksheet manager in the same manner as the user interface process.

Client processes can work in the same event-driven manner as the user-interface process. For example, a client that produces graphical representation of data can immediately update the graph when the data are updated.

A mail receiving program is also implemented as a client process. A mail system invokes the client when mail is received. The client then communicates with the server, which creates or updates an object. When the server is going to send mail, the server invokes another client that sends the mail.

4 Conclusion

Object-oriented data management was applied to a system called SCOPE, which supports various groupware applications. In addition to class hierarchy, rules that are attached to objects were useful in representing forms and procedures in a groupware environment.

The SEND rules, which are used to send objects to other agents, were particularly important for cooperative work. Any user can know how objects of a class are processed by each agent, because the rules are attached to the objects.

A communication layer was introduced under the object-oriented conceptual layer. The communication layer provides an open interface that can be accessed from standard mail systems.

The format defined for the communication layer can be used to represent a message to an object as well as an object itself, due to the fact that an access-oriented programming scheme was employed. Objects created outside the SCOPE system can be thus imported into the system.

Class definition in open systems must be able to evolve, because the open systems cannot have complete information on the world that communicates with the system. The programming-by-example metaphor, which was borrowed from spreadsheet systems, was introduced for the schema evolution. Rules for a class can be designed by simulating the rules on actual instances.

The process and module structure of SCOPE was also presented. A client/server model was adopted so that more than one user and application program can concurrently access the system. Any system can communicate with SCOPE as a client because of the open protocol of the inter-process communication between a client and a server.

In this paper, it was assumed that a form consists of simple fields. In the real world, however, a form can contain tables and other structures. SCOPE can deal with such a form by assuming that each table has fixed number of entries. Although this solution is similar to the one used in the real world, more research is required.

It was also assumed that each agent has its own copy of objects, and all agents share the same name space for classes. These are other issues for future research.

Acknowledgements

The authors would like to thank Takako Hashimoto, Minoru Kuramochi, Hong Li, Tsunenori Ishioka, Hiroyasu Kurose, and Yukari Shirota for their helpful discussion.

References


