Vol. 36 No. 6 Transactions of Information Processing Society of Japan June 1995

Regular Paper

Specification of Subsystems in Object-Oriented Design

HIROTAKA SAKAI†

In the object-oriented design of complex applications, it is essential to specify subsystems as manageable units of information systems. In accordance with the concept of contracts, partnerships, and business rules, we propose a formal specification of subsystems under the name of "subsystem schemes". We also discuss two closely related issues: expression of the responsibilities of subsystems and delegation of these responsibilities to other subsystems. The concept of abstract subsystems (also called frameworks) is significant as regards reuse of the specification of subsystems. As a preliminary step toward establishing a general method for the design of abstract subsystems, we create a specification of the behavior of an abstract business entity class, taking as a typical example an abstract subsystem of the resource-requesting-and-providing type.

1. Introduction

Recently, a number of object-oriented methods for developing information systems have been introduced and widely accepted. These methods are the result of efforts to establish object-oriented software engineering for reusable, extensible, and robust information systems, and to propose a variety of techniques such as structural, dynamic, and functional modeling as well as state transitions and process modeling methods[1)~7)].

In most of these methods, object-oriented development of information systems consists of three phases: analysis, design, and implementation. The purpose of the analysis phase is to provide a description of the user's problem, or the so-called "real-world" domain. In the description, the user's needs must be identified precisely and correctly in an understandable way. In the design phase, various concepts of the objects that constitute the information system are extracted and specifications of these concepts are created. In this phase, software engineers analyze issues such as what objects exist, what structural and behavioral characteristics they have, and what constraints they should obey to preserve the integrity of the system, and then prepare specifications of the concepts in appropriate forms. The implementation phase is for constructing an executable application system from the objects specified in the design phase, using specific languages and object-oriented database systems.

† Department of Industrial and Systems Engineering, Faculty of Science and Engineering, Chuo University

In the design phase of the development of complex applications, it is essential to divide the system into manageable units called subsystems[2),8)~10)]. A subsystem is usually identified by the service it provides. A service is a group of related functions that share some common purpose, such as entering orders, making seat reservations, and handling graphic user interfaces[2)]. In the analysis phase, a subsystem is identified corresponding to the service it provides. In the design phase, a subsystem is modeled as a conceptual situation in which many objects collaborate with each other to perform certain tasks called "responsibilities". It is particularly important to give a precise formal specification of a subsystem in the design phase. Subsystems are finally implemented as manageable components of the information system.

In the object-oriented methods proposed so far, this concept has been introduced under names such as "ensembles", "layers", "clusters", and "subsystems".

Various approaches to subsystem design have also been proposed, including Design by Contract[11)], Responsibility-Driven Design[9)], Object-Oriented Analysis and Top-down Software Development[12)], the Law of Demeter[13)], and Use Cases[10)]. However, it is still an open problem how to formally specify the subsystem concept, including its behavioral characteristics.

The most important issue in specifying subsystems is the conceptualization of business rules[14)]. Since a subsystem is taken as an object to which the job of providing a service is assigned as a set of responsibilities, it is nat-

ural to consider a business rule to be a kernel of the subsystem concept. In accordance with this idea, we propose a formal specification of a subsystem in the design phase under the name of the subsystem scheme.

Subsystems are specified as objects that collaborate with each other through interfaces called "contracts" in application environments. We define this collaboration as the delegation concept of subsystems. The delegation of responsibilities to other subsystems provides a basis for designing interaction between subsystems.

One of the main advantages of object-oriented design is that it supports software reuse. In the long run, the reuse of design is more important than the reuse of code[15]. Reusability of design is accomplished by developing abstract subsystems (also called frameworks). An abstract subsystem is a collection of abstract and concrete classes and a specification of their collaboration. Some examples are the Model/View/Controller of Smalltalk-80 and ET^{++} for user interface subsystems.

We can apply the proposed specification technique to the design of abstract subsystems in more general application areas. However, it is necessary to extract specification patterns from similar concrete subsystems, which requires a lot of work. We consider that the key to designing an abstract subsystem is specification of the behavior of the business entity class. As a preliminary step toward establishing a general design method, we create a specification of the behavior of the abstract business entity class, taking as a typical example an abstract subsystem of the resource-requesting-and-providing type. The technique can be straightforwardly applied to specification of the behavior of abstract business entity classes in general abstract subsystems.

The uniform specification of schemes of objects proposed in this article provides a stable basis for object-oriented design of information systems.

The paper consists of the following sections. In Section 2, we define the basic object scheme, centered on the life cycle concept of objects, and introduce techniques for representing life cycles by using state classes. Section 3 explains the notion of behavior relationships as inter-life-cycle constraints between objects. In Section 4, we give a formal description of the subsystem scheme, with detailed examples. Taking as typical examples subsystems of the resource-requesting-and-providing type in Section 5, we create a specification of the behavior of an abstract business entity class as well as its application to designing the behavior of concrete business entity classes. Section 6 concludes the paper with a summary and remarks on problems yet to be investigated.

2. Descriptions of Objects

2.1 Basic Object Scheme

For the purpose of establishing an object-oriented approach to information systems, we take Mylopoulos's standpoint[16], that anything about which a statement can be made is an object. Object-orientation is a way of thinking that requires the subject to be distinctly identified and to obey the locality principle; that is, information about the subject should be grouped in a capsule.

A population of objects having common structural and behavioral characteristics are categorized as belonging to the same class. These characteristics are described in the basic object scheme (or simply the scheme) of the class, and each object in the class is said to be an instance of the scheme. For notational convenience, we use capital initial letters for classes and small letters for objects in classes.

The scheme of a class X, denoted B-Scheme(X), is of the form

$$\text{B-Scheme}(X) = (\text{Action}, \text{Memory}, \text{Life Cycle});$$

(1) Action is a set of symbols representing processes defined for the objects. It is the interface of objects in the class X with other objects.

(2) Memory, which is the internal structure of objects, is a set of pairs of the form (Attribute: Domain), where Attribute is a structural property of an object and Domain is a set of objects in a certain class. Each attribute takes an element of the associated domain as its value. We ignore, for simplicity, the static constraints concerning the values of attributes. Actions that operate on an object share its attribute values as the internal memory.

(3) Life Cycle expresses the behavioral constraints of objects, and is described as a pre-ordered (i.e., reflexive and transitive) set of states. A state u is a symbol that is interpreted as a milestone in the object life cycle. Certain actions that determine the pre-order relations

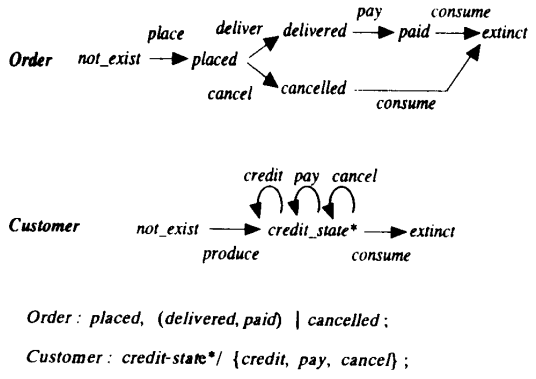

Fig. 1 Life cycles of the classes Order and Customer.

are associated with each state u. We call them life-cycle actions. Each life-cycle action t associated with u represents a process that causes, if activated, a state transition from the pre-state u to the post-state v. The action t has a single post-state v defined in the life cycle with the meanings that t can be activated only if the object is in the state u. As a special kind of state, we consider a state that is neither a pre-state nor a post-state of any state. We call it the void state.

2.2 Descriptive Notations of Life Cycles

We denote a life cycle by a sequence of forms $\{u/t\}$, where u is a state and t is the life-cycle action that causes transition from u to its post-state. We often omit the action part for simplicity. **Figure 1** shows life cycles of the classes *Order and Customer* illustrated in the graph. The initial state *not-exist* ("the object does not exist yet") and the final state *extinct* ("the object has disappeared") are omitted in both classes. In the life cycle of *Order*, the notation $(delivered, paid)|cancelled$ indicates a possible selective occurrence of either of two sequences delimited by a symbol '|'. In the life cycle of *Customer*, the notation $credit\text{-}state^*/\{credit, pay, cancel\}$ indicates possible repetition of state transitions from *credit-state* to itself zero or more times through any one of the actions *credit*, *pay*, and *cancel*. We call states of this kind repeated states.

The life cycle has a granularity in its representation. A state u may be further decomposed into a local life cycle that is in turn a pre-ordered set of more refined states.

2.3 Generalization Concept

The generalization concept, which takes account of the life cycles of objects, is defined as follows.

Suppose we have schemes of class X and Y: B-Scheme(X) = (Action-X, Memory-X, Life-Cycle-X) and B-Scheme(Y) = (Action-Y, Memory-Y, Life-Cycle-Y). If there is a function $h = (h_A, h_M, h_L)$ from B-Scheme(X) to B-Scheme(Y) with the following properties (1) through (3), we say that B-Scheme(X) is the generalization of B-Scheme(Y) and that Y is the subclass of X (or X is the superclass of Y) w.r.t. the generalization function h.

(1) h_A is an inclusion function from Action-X to Action-Y; that is, Action-$X \subseteq$ Action-Y and $h_A(t) = t$ for each t in Action-X.

(2) For each attribute-domain pair $(A : D)$ in Memory-X, we have an attribute-domain pair in Memory-Y $h_M((A : D)) = (A : D')$ such that $D' \subseteq D$.

(3) h_L is an inclusion function from Life-Cycle-X to Life-Cycle-Y; that is, Life-Cycle-$X \subseteq$ Life-Cycle-Y and $h_L(u) = u$ for each state u in Life-Cycle-X. Under this mapping, $h_L(u)$ may be further refined in Life-Cycle-Y.

The superclass-subclass relationship is also called the *is-a* relationship, and makes up the *is-a* hierarchy of classes.

2.4 State Classes as Diagrammatic Representations of Life Cycles

Using the notion of "State" as a design pattern of objects[17], we represent the life cycle diagrammatically. **Figure 2** shows an object diagram using the OMT[2]-like notations, with rectangles, triangles, and diamonds representing classes, *is-a* relationships, and references to objects, respectively.

We model the life cycle of a class, say *Order*, by a set of classes: *Order-State* and its subclasses *Placed*, *Delivered*, *Paid*, and *Cancelled*. We call them the state classes. *Order-State* is an abstract class (i.e., a class that produces no objects). Each of its subclasses has exactly one object, called the state object, that represents a state of the life cycle of *Order*. The value of the attribute *state* of each object *order* refers to a state object that reflects the current state of *order*. Each object *order* delegates the activities of its life-cycle actions to the state objects. The state object of the class *Placed*, say, has life-cycle actions such as *deliver* and *cancel* that perform appropriate processes necessary for state transitions of *order*, and change the value of *state* to state objects reflecting the

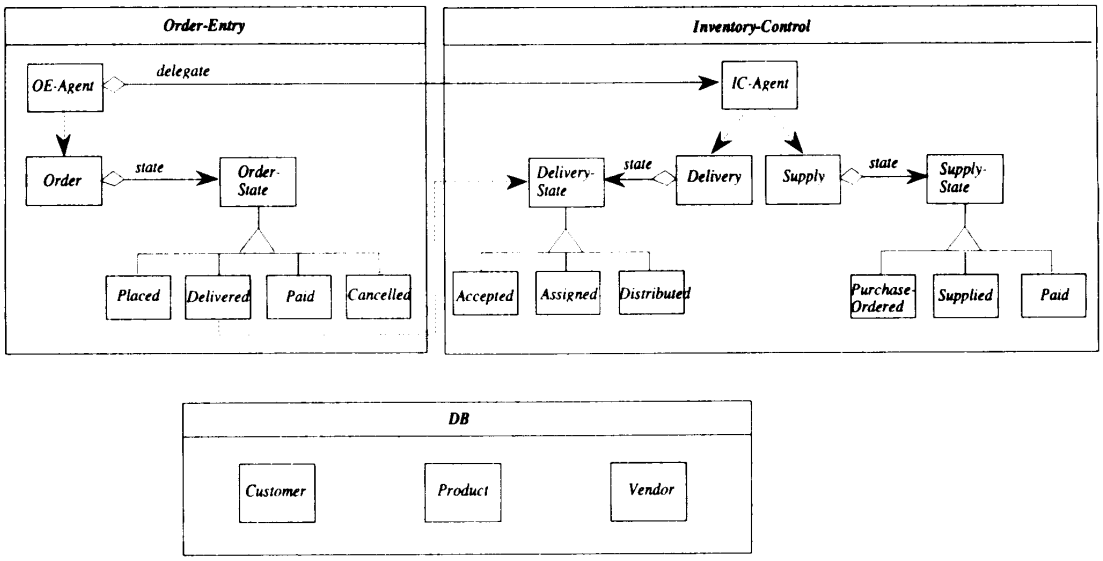

Fig. 2 State classes associated with business-entity classes in the subsystems Order-Entry and Inventory-Control.

post-states.

In the diagrammatic representation, the pre-order relations of states are not explicitly illustrated, but are hidden in the behavior of state objects. Corresponding to a void state, we assume a dummy state class that has no effective actions.

The notion of state classes, though not indispensable for representing life cycles, is useful for representing life-cycle patterns of objects having *is-a* relationships. For example, the refinement of a state is represented by *is-a* hierarchies of state classes. In abstract subsystems design, which will be discussed in Section 5, the behaviors of abstract classes and their subclasses are also represented by *is-a* hierarchies of state classes.

3. Behavior Relationships as Inter-Life-Cycle Constraints

In any state transition, an object should obey the defined pre-order in the life cycle. In this sense, the life cycle defines constraints on the behavior of a single object.

In general application environments, an object does not exist in isolation, but behaves in collaboration with objects of the same or different classes. In addition to the life cycle constraints defined in the scheme, it is necessary to maintain consistent relationships between life cycles of collaborating objects. As inter-life-cycle constraints, we define the concept of behavior relationships, denoted as follows:

(a) μ state$(x, u)\{/t\}$ {and p}
$\rightarrow$ μ state(y, v) $\{/s\}$;

The notation μ state(x, u) expresses "a state transition of the object x to the state u." The symbol p is a Boolean expression with attributes as variables; it may or may not appear. This form expresses: "If the transition of the object x to the state u occurs and the Boolean expression p, if any, is true, then the transition of the object y to the state v should also occur." If the states u and/or v are repeated states, we attach for clarity the life-cycle actions that bring about the transitions denoted as μ state$(x, u)/t$ and/or μ state$(y, v)/s$.

The general form of behavior relationships is written as in (b), where each σ_i denotes μ state$(x_i, u_i)\{/t_i\}$ or p_i, and τ_j denotes μ state$(y_j, v_j)\{/s_j\}$.

(b) σ_1 and σ_2 and ... and σ_k
$\rightarrow$ τ_1 and τ_2 and ... and τ_m;

4. Schematic Representation of Collaborating Objects

4.1 Subsystem Scheme

A subsystem is a situation in which objects of various classes collaborate for a certain business purpose, namely, to perform given responsibilities. We call participating objects partners. A subsystem is itself a subject about which statements can be made regarding topics such as responsibilities, collaborating objects, and behavior relationships between objects. Therefore, a subsystem could be thought of as an

object. However, a subsystem is required to have more knowledge or intelligence than instances of basic object schemes. To describe the characteristics of a subsystem,we must take account of contracts concerning responsibilities, roles of partners, and behavior relationships between partners. For this reason, we define the subsystem scheme by extracting the concepts of contracts, partnerships, and business rules that are not described in the basic object scheme.

The subsystem scheme of a subsystem S, denoted S-Scheme(S), is of the form

S-Scheme(S)
= (Contract, Partnership, Business-Rule);

Since a subsystem scheme is considered to have exactly one instance, we use, for simplicity, the same symbol S for the subsystem that is an instance of S-Scheme(S).

(1) Contract is the interface with other subsystems and is a set of forms (Responsibility:Client), where Responsibility and Client are lists of responsibilities and subsystems, respectively. A responsibility is a symbol representing a certain task. Contract denotes that the subsystem S contracted the specified client subsystems for the specified responsibilities.

In response to a request from client subsystems to perform a responsibility, the subsystem S brings about certain state transitions of partners. The set of states of related partners that should be brought about to perform a responsibility could be thought of as a specification of the responsibility. We call it the realization of the responsibility.

(2) Partnership is a set of pairs of the form (Role:Partner), where Role is a symbol representing a certain role and Partner is a list of names of classes. Partnership is the internal structure of a subsystem with the meanings that objects of the classes in Partner play the given role in the subsystem S.

In ordinary subsystems, we assume three basic roles: Agent, Business-Entity, and Resource. There is exactly one class of the role Agent in a subsystem, and its only object, the agent, behaves as a representative as well as a coordinator of the subsystem. The request for a responsibility is directed to the agent by the agent of the client subsystem. The agent then coordinates the necessary state transitions of related partners to realize the responsibility.

Each class of the role Business-Entity models a type of business in the real world, and its object represents a business unit. There may be more than one class having the role Business-Entity in a subsystem.

Each class of the role Resource models a type of resource that is used to perform a certain business function. Resource objects are shared by Business-Entity objects of various subsystems. We assume a special subsystem *DB* consisting of partner classes that are shared Resource classes, and treat each Resource class in *DB* as being also a partner in subsystems that necessitate it.

(3) Business-Rule consists of the behavioral constraints of a subsystem, and is specified as a collection of life cycles of partners and behavior relationships that should be maintained between partners. Business-Rule determines the behavioral characteristics of the subsystem. The heart of Business-Rule is life cycles and behavior relationships concerning Business-Entity classes. The life cycle of a Business-Entity class describes the business process steps, and the behavior relationships between Business-Entity objects and Resource objects express business rules to maintain the consistency of system states.

4.2 Subsystems *Order-Entry* and *Inventory-Control*

[Subsystem Schemes]

As examples of subsystems, we consider *Order-Entry* and *Inventory-Control*, together with the subsystem *DB*, as shown in Fig. 2. Since the objects of *Customer*, *Product*, and *Vendor* in *DB* are shared by the two subsystems, we treat these objects as partners of *Order-Entry* and *Inventory-Control as well.* **Figure 3** shows the subsystem schemes of *Order-Entry* and *Inventory-Control.*

In *Order-Entry*, OE-Contract defines responsibilities concerning the order entry business to be requested from the client subsystem *OE-Graphic-User-Interface.* To carry out the responsibilities, *OE-agent* coordinates the activities of objects of Business-Entity class *Order* and related Resource classes. *Inventory-Control* has two groups of responsibilities that concern two Business-Entity classes, *Delivery* and *Supply*, respectively. IC-Deliver is the responsibility for assigning and distributing products and is subdivided into three responsibilities. The second group of responsibilities is for purchasing products to supply stocks of products. The business rules of two subsystems define the way in which objects of Business-Entity

S-Scheme (*Order-Entry*) = (OE-Contract, OE-Partnership, OE-Business-Rule) ;

[OE-Contract]

{OE-Place, OE-Deliver, OE-Pay, OE-Cancel} : *OE-Graphic-User-Interface* ;

[OE-Partnership]

(Agent : *OE-Agent*) , (Business-Entity : *Order*) , (Resource : *Customer*) ;

"*Customer* is a class of *DB*."

[OE-Business-Rule]

Life Cycles of Partner Classes

Order : *placed*, (*delivered*, *paid*) | *cancelled* ;

Customer : *credit-state**/ {*credit*, *pay*, *cancel*} ;

Behavior Relationships

(OE_1) μ state (*order*, *placed*) → μ state (*customer*, *credit-state*) /*credit* ;

(OE_2) μ state (*order*, *paid*) → μ state (*customer*, *credit-state*) /*pay* ;

(OE_3) μ state (*order*, *cancelled*) → μ state (*customer*, *credit-state*) /*cancel* ;

(a) Subsystem Scheme of *Order-Entry*.

S-Scheme (*Inventory-Control*) = (IC-Contract, IC-Partnership, IC-Business-Rule) ;

[IC-Contract]

{ IC-Deliver = (IC-Accept, IC-Assign, IC-Distribute)} : *Order-Entry* ;

{IC-Purchase, IC-Supply, IC-Pay} : *IC-Graphic-User-Interface* ;

[IC-Partnership]

(Agent : *IC-Agent*) , (Business-Entity : *Delivery*, *Supply*) , (Resource : *Product*, *Vendor*) ;

[IC-Business-Rule]

Life Cycles of Partner Schemes

Delivery : *accepted*, *assigned*, *distributed* ;

Supply : *purchase-ordered*, *paid* ;

Product : *stock-state**/ {*assign*, *deliver*, *supply*} ;

Vendor : *credit-state**/ {*credit*, *pay*} ;

Behavior Relationships

(IC_1) μ state (*delivery*, *accepted*) and enough-quantity (*delivery*) → μ state (*delivery*, *assigned*) ;

(IC_2) μ state (*delivery*, *accepted*) and not enough-quantity (*delivery*) → μ state (*supply*, *purchase-ordered*) ;

(IC_3) μ state (*delivery*, *assigned*) → μ state (*product*, *stock-state*) /*assign* ;

(IC_4) μ state (*delivery*, *distributed*) → μ state (*product*, *stock-state*) /*deliver* ;

(IC_5) μ state (*supply*, *perchase-ordered*) → μ state (*vendor*, *credit-state*) /*credit* ;

(IC_6) μ state (*supply*, *supplied*) → μ state (*product*, *stock-state*) /*supply* ;

(IC_7) μ state (*supply*, *paid*) → μ state (*vendor*, *credit-state*) /*pay* ;

(b) Subsystem Scheme of *Inventory-Control*.

Fig. 3 Subsystem schemes of Order-Entry and Inventory-Control.

classes should be processed.

[Realization of Responsibilities]

We define the state expression of the realization of a responsibility r, denoted $R(r)$, as a set of states that a business entity and its related partners should reach to accomplish r. The realization is straightforwardly derived from

[OER_1] R (OE-Place) = {state (*order*, *placed*) , state (*customer*, *credit-state*) /*credit*} ;

[OER_2] R (OE-Deliver) = {⇒ R (IC-Deliver) , state (*order*, *delivered*)} ;

[OER_3] R (OE-Pay) = {state (*order*, *paid*) , state (*customer*, *credit-state*) /*pay*} ;

[OER_4] R (OE-Cancel) = {state (*order*, *cancelled*) , state (*customer*, *credit-state*) /*cancel*} ;

[ICR_1] R (IC-Deliver) = (R (IC-Accept) , R (IC-Assign) , R (IC-Distribute))

"This realization is broken down into the three realizations [$ICR_{1,1}$] ~ [$ICR_{1,3}$] ."

[$ICR_{1,1}$] R (IC-Accept) = {state (*delivery*, *accepted*)} ;

[$ICR_{1,2}$] R (IC-Assign) = {state (*delivery*, *assigned*) , state (*product*, *stock-state*) /*assign*} ;

[$ICR_{1,3}$] R (IC-Distribute) = {state (*delivery*, *distributed*) , state (*product*, *stock-state*) /*deliver*} ;

[ICR_2] R (IC-Purchase) = {state (*supply*, *purchase-ordered*) , state (*vendor*, *credit-state*) /*credit*} ;

[ICR_3] R (IC-Supply) = {state (*supply*, *supplied*) , state (*product*, *stock-state*) /*supply*} ;

[ICR_4] R (IC-Pay) = {state (*supply*, *paid*) , state (*vendor*, *credit-state*) /*pay*} ;

Fig. 4 State Expression for the realization of responsibilities.

the business rule. In **Fig. 4**, [OER_1] through [OER_4] (except for OER_2) and [ICR_1] through [ICR_4] describe state expressions for realization of the responsibilities of *Order-Entry* and *Inventory-Control*.

To carry out a responsibility r, the agent of the subsystem performs all the activities necessary to bring about the states specified in the state expression of $R(r)$, activating appropriate actions of partners. To carry out a responsibility, say OE-Pay, *OE-agent* activates the life-cycle action *pay* of a Business-Entity object *order* and changes its state. *OE-agent* also updates the state *credit-state* of the related Resource object *customer* according to the behavior relationships (OE_2). If we model the life cycle by state classes, the life-cycle actions of Business-Entity objects are in turn delegated to the state objects.

Instead of performing all of these activities, the agent could delegate activities for updating Resource objects to Business-Entity objects. The life-cycle actions of Business-Entity objects are then extended, not only to change their own states, but also to update the states of related Resource objects. In the state class representation of life cycles, these extensions should be reflected in the actions of state objects.

[Delegation of Responsibilities]

The responsibility OE-Deliver of *Order-Entry*, since it concerns the complicated inventory control business processes, is usually delegated to the subsystem *Inventory-Control*. In general, the delegation of a whole or a part of a responsibility r of a subsystem S to a responsi-

bility r' of another subsystem S' is thought of as a collaboration of subsystems, and is specified by using the realization concept. Through the delegation, a whole or a part of the state expression of R(r) is not realized in the subsystem S (i.e., actual state transitions do not occur in S), but is replaced with the realization R(r'). We denote this replacement $\Rightarrow$ R(r').

For example, the state expression [OER_2] in Fig. 4 indicates that a part of the responsibility OE-Deliver is delegated to *Inventory-Control*, where the responsibility IC-Deliver is realized. The delegation $\Rightarrow$ R(IC-Deliver) is accomplished by directing the request from *OE-agent* to *IC-agent* through the reference attribute *delegate*. *IC-agent* carries out this responsibility by producing *delivery*, an object of *Delivery*, and bringing about necessary state transitions of this object together with its related Resource objects as specified in [ICR_1] and [$ICR_{1,1}$] through [$ICR_{1,3}$]. *IC-agent* notifies *OE-agent* of the completion of the realization. *OE-agent* interprets this notification that the state transition to *delivered* of *order* has been completed. We can model this delegation as the replacement of the state subclass *Delivered* of *Order-State* by the state subclasses of *Delivery-State*, and represent it by the dotted line in Fig. 2.

5. Specification of Business-Entity Classes in the Abstract Subsystem of Type RRP

One of the main advantages of object-oriented design is that it supports software reuse. In the long run, the reuse of design is more important than the reuse of code. Reusability of design is accomplished by developing abstract subsystems (also called frameworks). An abstract subsystem is a collection of abstract and concrete classes and a specification of their collaboration. We can apply the proposed specification technique to the design of abstract subsystems in general application areas.

As we have seen in the subsystems *Order-Entry* and *Inventory-Control*, the role Business-Entity and the related business-rules characterize the behavior of subsystems. Since the processing of business units in the real world is mapped to the Business-Rule in the subsystem scheme, we consider the life cycles of Business-Entity classes to be a kernel of subsystem design. As a preliminary step toward establishing a general design method for abstract subsystems, we have created a specification of the behavior of an abstract Business-Entity class, taking as a typical example an abstract subsystem of the resource-requesting-and-providing type (type RRP). Concrete subsystems of this type are seen in many existing real systems for tasks such as airline reservation, hotel reservation, order entry, and inventory control that involve the handling of requests for resources such as seats, rooms, and products. Although we take subsystems of type RRP as an example, the technique is straightforwardly applicable to specification of the behavior of abstract business entity classes in general abstract subsystems.

In an abstract subsystem of type RRP, the life-cycle pattern of the abstract Business-Entity class is developed by analyzing general rules of resource-requesting-and-providing businesses. We consider the abstract class *RRP-Business* that has the life-cycle pattern represented by the state class *RRP-State*, as shown in **Fig. 5**. Each subclass of *RRP-State* is also an abstract class and expresses the following business steps:

Not-Exist:	the business does not exist yet.
Negotiated:	the requester requests resources and the provider accepts the request; that is, the business is contracted between the requester and the provider.
Provided:	the resources (goods or money) are provided to the requester.
Received:	the resources (money or goods) from the requester are received in return.
Cancelled:	the business is cancelled.

If we design Business-Entity classes such as *Order*, *Delivery*, and *Supply* in subsystems of type RRP as concrete subclasses of *RRP-Business*, the life-cycle pattern *RRP-State* provides effective means for reusability of the specification of life cycles of these classes. Figure 5 shows the mappings of *RRP-State* to each of the state classes *Order-State*, *Delivery-State*, and *Supply-State*. We refer to these state classes as application state classes derived from the life-cycle pattern *RRP-State*.

In applying the life-cycle pattern to application state classes, we assume the following design principles:

(1) Each state class in the life-cycle pattern corresponds to one application state class (including dummy state classes). Appli-

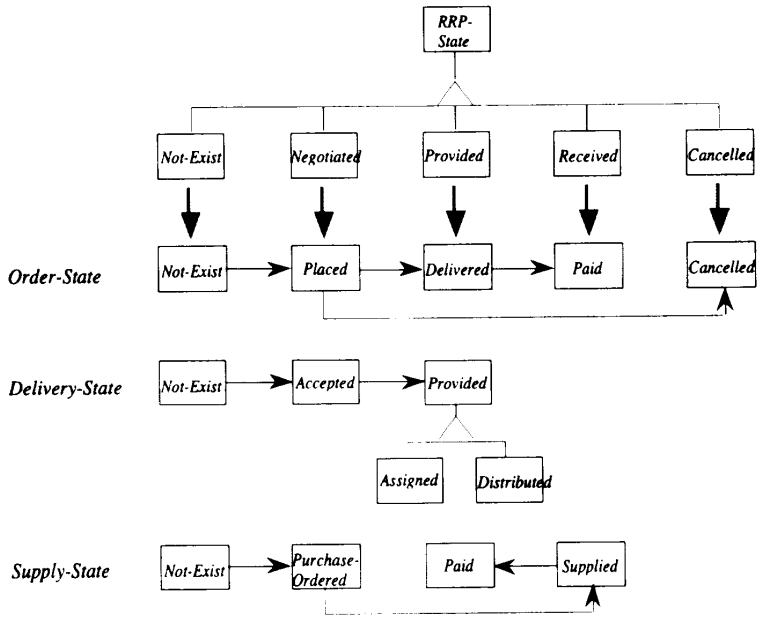

Fig. 5 The life-cycles pattern RRP-state and its application state classes.

cation state classes may be renamed.

(2) Application state classes may further be refined into subclasses (e.g., the state class *Provided* of *Delivery-State* is refined into the state classes *Assigned* and *Distributed*).

(3) In each application state class, the life-cycle actions should be defined according to the business rules to reflect the pre-order relations, behavior relationships, and delegation of states transitions. They should perform one of the following actions:

(a) Change the state to the post-state by itself or delegate this activity to another agent. Furthermore, change the states of related Resource objects. This activity depends on the design decision.

(b) Do nothing. This means that no action should be taken in this state.

The life-cycle actions of subclasses of *Order-State*, for example, are described below. Actions that do not appear in the subclass do nothing.

Not-Exist place Update the *credit-state* of *customer* by activating the action *credit*;
Set the attribute *state* of *order* to the state object of *Placed*;

Placed deliver Delegate IC-Deliver to IC-*agent*;
Set the attribute *state* of *order* to the state object of *Delivered*;

cancel Update the *credit-state* of *customer* by activating the action *cancel*;
Set the attribute *state* of *order* to the state object of *Cancelled*;

Delivered pay Update the *credit-state* of *customer* by activating the action *pay*;
Set the attribute *state* of *order* to the state object of *Paid*;

6. Conclusion

As a basis for object-oriented design of complex applications, we have proposed a formal specification of subsystems that covers the following areas:

(1) Formal specification of subsystems as subsystem schemes, taking the business rule as a kernel.

(2) Realization of responsibilities derived from the business rule.

(3) Interaction between subsystems in accordance with the concept of delegation of responsibilities.

(4) Extraction of the life-cycle pattern of the

abstract Business-Entity class in an abstract subsystem of type RRP and its application to the derivation of application state classes.

As generally recognized by practitioners, "An important characteristic of object-oriented development is that the analysis, design, and implementation phases adopt similar models, although each phase has different emphasis. This enables smooth transition between different phases"[7]. It is expected that the unified view of objects including subsystems will enhance this advantage by allowing extensive use of essential concepts of object-orientation such as encapsulation of internal design, generalization, aggregation, and reuse of specification for subsystems.

Among many problems yet to be investigated, we are especially interested in pursuing research on the following:

(1) Analysis of various types of subsystems that involve complicated contracts, partnerships, and business rules.

(2) Development of a formal specification of *is-a* hierarchies of classes spreading across various subsystems as partner classes.

(3) Object-oriented database design methods for integrating basic object schemes and subsystem schemes.

References

1) Booch, G.: *Object Oriented Design with Applications*, The Benjamin/Cummings Publishing Company (1991).

2) Rumbaugh, J., et al.: *Object-Oriented Modeling and Design*, Prentice-Hall (1991).

3) Sakai, H.: A Method for Contract Design and Delegation in Object Behavior Modeling, *IEICE Trans. Information and Systems*, Vol.E76-D, No.6, pp.646–655 (1993).

4) Shlaer, S. and Mellor, S.: *Object Lifecycles: Modeling The World in States*, Yourdon Press (1992).

5) Shlaer, S. and Mellor, S.: *Object-Oriented System Analysis: Modeling The World in Data*, Yourdon Press (1992).

6) Jacobson, I., et al.: *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley (1992).

7) Aksit, M. and Bergmans, L.: Obstacles in Object-Oriented Software Development, *Proc. OOPSLA '92*, pp.341–358 (1992).

8) Rubin, K.S. and Goldberg, A.: Object Behavior Analysis, *Comm. ACM*, Vol.35, No.9, pp.48–62 (1992).

9) Wirfs-Brock, R., et al.: *Designing Object-Oriented Software*, Prentice-Hall (1990).

10) Jacobson, I., et al.: Using Contracts and Use Cases to Build Pluggable Architectures, *Journal of Object-Oriented Programming*, Vol.8, No.2, pp.18–24 (1995).

11) Meyer, B.: *Object-Oriented Software Construction*, Prentice-Hall (1988).

12) De Champeaux, D.: Object-Oriented Analysis and Topdown Software Development, *European Conference on Object-Oriented Programming*, pp.360–375 (1991).

13) Lieberherr, K. and Holland, F.: Assuring Good Style for Object-Oriented Programs, *IEEE Software*, pp.38–48 (1989).

14) Loucopoulos, P., et al.: Business Rules Modelling: Conceptual Modelling and Object-Oriented Specifications, *Object-Oriented Approach in Information Systems*, pp.323–342, North-Holland (1991).

15) Wirfs-Brock, R. and Johnson, R.: Surveying Current Research in Object-Oriented Design, *Comm. ACM*, Vol.33, No.9, pp.104–124 (1990).

16) Mylopoulos, J.: Object-Oriented and Knowledge Representation, *Object-Oriented Databases: Analysis, Design, & Construction (DS-4)*, pp.23-37, North-Holland (1990).

17) Gamma, E., et al.: *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994).

(Received July 31, 1995)
(Accepted February 7, 1996)

Hirotaka Sakai is a professor of Department of Industrial and Systems Engineering, Faculty of Science and Engineering, Chuo University. His research contributions have been primarily in the field of data engineering, in particular, semantic data models, database design, and object-oriented software engineering. Sakai received Dr. of Engineering in 1982 from Kyoto University. Until 1983, he has been in Hitachi, Ltd., where he was engaged in developments of software products. He also served as a chairman of special interest groups: Database Systems of IPSJ and Data Engineering of IEICE. He is the author or coauthor of several books in database design and object-oriented design.