Conceptual Design of Active Object-Oriented Database Applications Using Multi-level Diagrams

Mauricio J. V. Silva * C. Robert Carlson

Department of Computer Science
Illinois Institute of Technology
10 West 31st Street, Chicago, IL 60616
email: silvmau@charlie.acc.iit.edu, cscarlson@minna.iit.edu

Abstract

Several active object-oriented database systems have been developed to address the needs of applications with complex requirements and time execution constraints (e.g. computer integrated manufacturing). However, no comprehensive and integrated modeling approach has been described for conceptually modeling active object-oriented database applications.

This paper deals with these issues by extending the research of object-oriented methods with an integrated approach, called A/OODBMT (Active Object-Oriented Database Modeling Technique), which integrates and extends the Object Modeling Technique (OMT) method for conceptually designing active object-oriented database applications.

A/OODBMT models database applications by defining and integrating four new types of models, namely the nested object model (NOM), the behavior model (BM), the nested rule model (NRM), and the nested event model (NEM). The nested object model extends the OMT object model by adding nesting capabilities, and by providing a better abstraction mechanism for developing database applications in multi-level diagrams. Moreover, the nested object model adds rules to classes to specify their active behavior. The behavior model combines the dynamic and the functional modeling techniques proposed in the OMT method. In addition, the behavioral model represents database transactions through transaction diagrams. The nested rule model supports a comprehensive set of rules and visually defines the rules and their interactions using multi-level diagrams. The nested event model supports a comprehensive set of events and visually represents them in the context of rules.

1 Introduction

One of the major problems with the development of new and emerging database applications (e.g. computer integrated manufacturing) is that rules describing the policy of an organization are hard coded and dispersed all over different programs. This approach leads to applications that are harder to validate and difficult to maintain [Tsal91, Rasm95]. Moreover, as part of their semantics these applications need to be

* Supported by the Brazilian Government Agency - CAPES

active, i.e, continually monitoring changes to the database state and reacting by executing an appropriate action without user or application intervention[Chak93].

Active Object-Oriented Database Systems (AOODBSs) [Daya88, Geha91, Buch92, Gatz92, Anwa93, Kapp94] try to solve these problems by providing event driven behavior necessary for implementing time critical reactions, and by integrating rules with the database. A rule is composed of three components: an event, a condition and an action. The rule monitors the database, which only executes the action of the rule when the event occurs and the condition is evaluated to "true" [Carl83, Daya88]. Rules are used to declaratively specify all the control aspects of an application and are easy to manage because of their explicit specification.

While there has been considerable research and development of AOODBs [Kapp95, Buch95], little attention has been paid to defining an integrated modeling technique for conceptually modeling AOODBs applications.

Object-Oriented methods and modeling techniques [Mona92, Hutt94] can be useful for modeling the object-oriented schema of an active object-oriented database application. However, missing from these approaches is an integration of their models with rules and the capability to model database related features such as database transactions.

This paper adds to the research on object-oriented methods by developing an integrated approach, called A/OODBMT (Active Object-Oriented Database Modeling Technique), which integrates and extends the Object Modeling Technique (OMT) method [Rumb91] for conceptually designing active object-oriented database applications.

We chose to integrate and extend the OMT method because it is one of the most popular methods for analysis and design of object-oriented software development, and was developed specifically for modeling and reasoning about complex applications [Rumb91, Thur94]. Moreover, because OMT uses object-oriented models, it is capable of modeling the real world of interest more naturally than other models which do not consider the behavior of the application[Rumb91, Mart95].

A/OODBMT models database applications by defining and integrating four new types of models, namely the nested object model (NOM), the behavior model (BM), the nested rule model (NRM), and the nested event model (NEM). The nested object model extends the object model originally proposed in [Rumb91] by adding nesting capabilities proposed in [Carl89], and by providing a better abstraction mechanism for developing database applications in multi-level diagrams. Moreover, the nested object model adds rules to classes to specify their active behavior. The behavior model combines the dynamic model and the functional model originally proposed in [Rumb91]. In addition, the behavioral model represents database transactions through transaction diagrams. The nested rule model supports a comprehensive set of rules, and visually define rules and their interactions in multi-level diagrams. The nested event model supports a comprehensive set of events, and visually represents them using multi-level diagrams.

In A/OODBMT, each model describes one aspect of the system but contains references to the other models(see Figure 1). The nested object model contains descriptions of the classes and objects in the system that the behavioral model operates on.

The operations in the nested object model are described in the behavior model. The behavior model also uses operation events defined in the nested event model to describe the control aspects of the objects. Rules referenced in the nested object model are defined in nested rule model, and are executed in the context of the database transactions defined in the behavioral model. Rules specified in the nested rule model are triggered by events defined in nested event model.

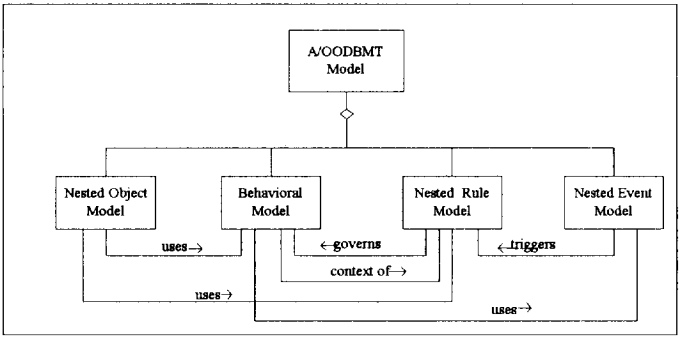

Fig. 1. Relationships among A/OODBMT Models

The remainder of this paper is organized as follows. Section 2 describes previous research and compares it to this approach. Section 3 describes the steps followed during conceptual design of active database applications using A/OODBMT models, and illustrates the application of these steps by modeling a library database system. Finally, Section 4 concludes the paper.

2 Related Work

The purpose of this section is to compare existing approaches to our approach. We analyze these approaches in the context of five modeling issues involved in the development of active object-oriented database applications. These include modeling objects, modeling passive object behavior (operations), modeling transactions, modeling rules (active behavior), and modeling events.

Current object-oriented methods have been used to model the objects of a system. However, their representation of objects employs "flat-diagrams", which can lead to very complicated diagrams if the objects have many associations [Carl89]. The Nested Object Model in A/OODBMT has been defined to deal with this problem. It provides a multi-level representation approach to the definition of the objects in a system, including their attributes, methods and rules.

In the object-oriented paradigm, the functionality of a system is achieved by the interaction and execution of object methods. Most object-oriented methods have used state diagrams to model object operations[Shal92, Jaco92, Rumb91]. In particular, [Rumb91] has used Harel state diagrams [Hare88], since it models the operations in multi-level diagrams. Moreover, [Rumb91] has used data-flow diagrams to model the

functionality of the whole database application. From the data-flow diagram, object operations are extracted. Like [Rumb91], we use Harel state diagrams with data-flow diagrams. The difference is that instead of representing the data flow diagram for the whole system, we create a data-flow diagram for each operation and its activities.

The interaction of the objects in a system defines a transaction. Moreover, the active behavior of the systems (i.e. rules) is only executed within the context of a transaction. Although current object oriented methods (e.g. [Rumb91, Jaco92, Booc94, Cole94]) model the interaction of objects in a system, they do not model database transactions. Missing from their representation are the database commands(e.g. transaction begin, abort, commit) used to define transactions and subtransactions. Our approach is to extend the modeling of object interactions with database transaction commands.

Existing visual approaches to rule specification include [Bich94, Mart95, Tsal91, Shen92, Grah94, Pras94]. All of these approaches support event /condition /action (ECA) rules. [Shen92], [Grah94], [Pras94], and [Mart95] specify production rules, pre/ post condition of operations. However, only a few approaches (e.g., [Tsal91, Pras94]), support exception, and contingency rules. Only [Grah94] provides support for grouping rules, by allowing a rule to be defined as the disjunction of simple rules. But, none of the approaches support the grouping of exclusive rules representing one abstract rule. None of the approaches define the semantics for rule overriding. Only petri-net like approaches [Tsal91, Pras94] show the interconnection of rules in "flat diagrams", which can lead to highly connected diagrams that are almost impossible to comprehend. Our approach is to integrate previous approaches and extend them by visually representing a comprehensive set of rules and their interactions including composite rules in a multi-level model, called the nested rule model. We also provide the semantics for overriding rules and support a wealth of coupling modes for rule execution [Bran93].

The modeling of events is crucial to active object-oriented database systems, since they determine when rules will be evaluated. Existing active object-oriented database systems model events textually, which is not an appropriate level for designing applications. Moreover, existing visual approaches (e.g. [Mart95, Bich94, Gatz94]) do not support a comprehensive set of both simple and composite events. Further, existing approaches do not represent events at multiple levels of abstraction using multi-level diagramming techniques. Our approach deals with these problems by supporting a comprehensive set of events and by providing a high-level graphical representation of events in multi-level diagrams.

3 Active Database Conceptual Design using A/OODBMT Models

In this section, we give the steps followed during conceptual design for the modeling of an active database application. We illustrate the application of these steps by modeling a library database system. The requirements of the library database system are described in Figure 2. From these requirements, we build the A/OODBMT models and show their integration. Only the essential points of the A/OODBMT models

will be discussed in the forthcoming subsections. The detailed description of the application of A/OODBMT can be found in [Silv95a].

The steps defined for conceptual database design in A/OODBMT are:

(i) Design the static structure of object/classes using the nested object model.

(ii) Design the passive behavior (methods) of object/classes and the database transactions using the behavior model.

(iii) Design the active behavior (rules) of object/classes using the nested rule model.

(iv) Design the events that appear in rules using the nested event model.

Note that, although we describe the steps above in a sequential order, the actual process of modeling an application is iterative. For instance, after building the behavioral model and identifying the operations of a class, we may have to go back and add them to the nested object model. Further, rules identified in the nested rule model for a specific object/class, have their name added to the rule part of the corresponding object/class in the nested object model.

In the following subsections we describe the nested object model, the behavioral model, the nested rule model, and the nested event model in turn.

3.1 Nested Object Modeling

The nested object model (NOM) is used to represent the static aspects of applications and is based on the object model [Rumb91] enhanced with nesting capabilities [Carl89] and rules. The major concepts found in NOM include class, object, relationships, attributes, operations, rules, and complex objects. A detailed description of NOM can be found in [Silv95a].

NOM enhances the object model by allowing objects/classes and associations to be abstracted as either simple or complex. Complex classes and associations can be expanded into sub-diagrams, where a more detailed specification is provided. This allows one to describe the static aspects of the application in multi-level diagrams, which facilitate the comprehension of the model. Without nesting the diagrams, very large object models resemble circuit diagrams rather than comprehensible structures [Carl89]. Further, NOM adds rules to the definition of a class and its instances (objects) to specify their declarative behavior. Only the names of the rules are placed inside an object/class as the actual definition of the rule is shown in the nested rule model.

The nested object model is built in several steps described below. We illustrate each step by applying it to the library database system example. The complete nested object model of the library database example is described in Figure 3. Note that in

Figure 3, we have also included the method and the rule part of the classes which will be discussed later in this article.

(1) The library system is composed of books and members.

(2) The library is opened from 10 a.m. - 10 p.m. (Mondays to Fridays) and from 10 a.m. - 6p.m. in the weekends.

(3) A member is a person and is characterized by a name, a ss# and an address. There are two types of members: students, and faculty.

(4) A book is characterized by a title, an author, a date of publishing and a publisher.

(5) A person may check out books only if he/she is a member of the library, otherwise an invalid checkout request message is sent to the person. Only available books can be checked out.

(6) All members may check out books for 2 weeks.

(7) When a person checks out a book, he/she must provide the title of the book and his/her ss#.

(8) When a book is checked out, a due date for returning the book is set.

(9) Books are expected to be returned by the due date. If the book is returned after the due date, it is considered to be overdue.

(10) When a person returns a book , if it is overdue a fine of 10 cents per day is charged to the member for each book not returned on time. A faculty member will receive only warnings for the first 5 overdue books. After that, the faculty member will start paying fines for overdue books.

(11) A notice is mailed to a member if he/she has a book that has been overdue for seven consecutive days .

Fig. 2. Library Database System Example

(i) Identifying the Classes and Attributes in the System

A class is a description of a group of related objects with similar behavior, semantics, and relationships [Rumb91]. In NOM, a class is composed of three parts: attributes, operations, and rules. It is depicted as a four part box, with the name of the class on the top part, a list of attributes with optional types on the second part, a list of operations with optional arguments and return value types on the third part, and a list of rule names on the fourth part. The attribute, operation, and rule sections of the class box can be omitted to reduce the detail of a visual specification.

From the library database requirements, we identify six major classes: *Person*, *Member*, *Student Member*, *Faculty Member*, *Library*, and *Book*. From line (3) we

define a class *Person* with attributes name, *ss#* and *address*. From (10), we define a class *Member* with an attribute *fine*. From line (4) and (9) we define a class *Book* with attributes *title, author, type, date of publishing, publisher, duedate, and status.*

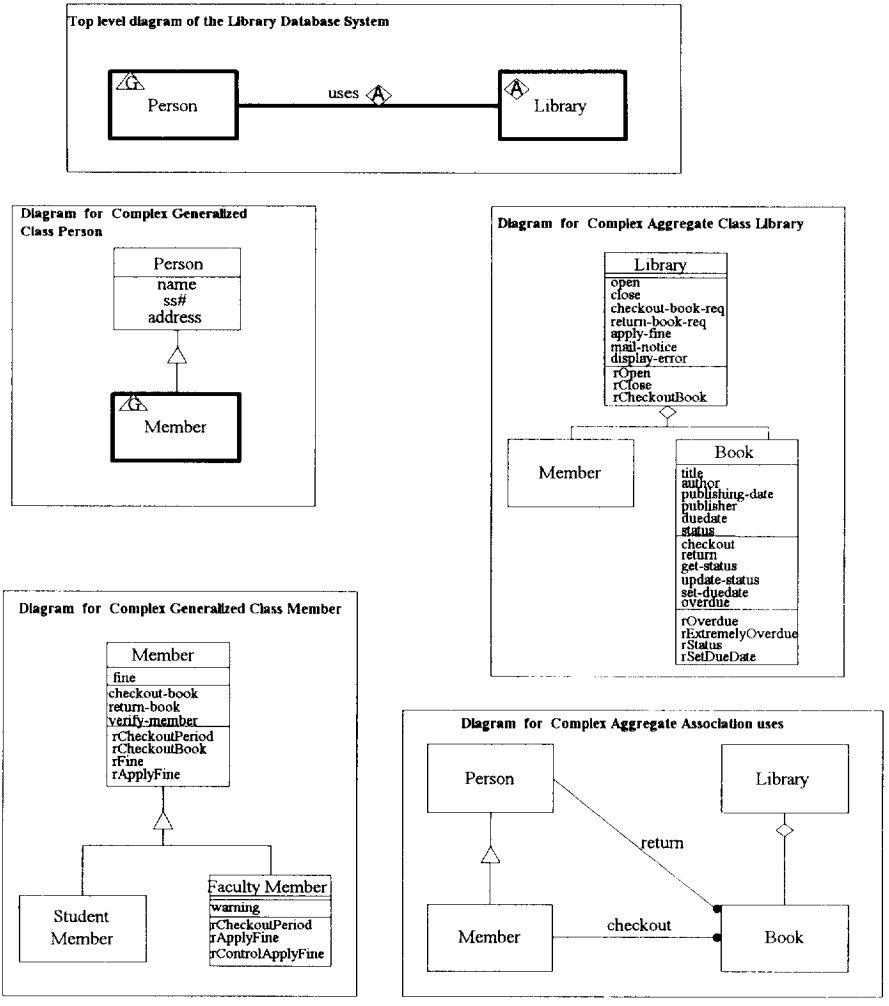

Fig. 3. Nested Object Model of the Library System

(ii) Identifying the Relationships in the System

A relationship is a characteristic that specifies a mapping from one object to another [Hutt94]. In NOM, there are three types of relationships, namely association, aggregation and generalization [Rumb91].

From line (1) of the library database example, we identify that class *Library* is an aggregation of classes *Member* and *Book*. From line (3), we identify that class *Person* is a generalization of class *Member* and that class *Member* is a generalization of classes *Student Member, and Faculty Member*. From line (5), we identify a *check-out* association from class *Member* and *Book*. From line (10), we identify an association *return* from class *Person* to class *Book*.

(iii) Identifying Complex Classes

A complex class is a virtual class. It is used to visually improve the understanding of NOM. Expanding a complex class yields a lower level diagram, which defines the elements of the complex class.

There are two types of complex classes, namely complex aggregate classes and complex generalized classes. A complex aggregate class is always related to a class, which has an aggregate relationship (e.g. class *Library* in Figure 3). A complex aggregate class represents a higher abstraction of the related aggregate class. It has the same name as the related aggregate class and is depicted as a bold class box with a diamond with the letter A (aggregation) inside, placed on the upper left corner of the box. Expanding a complex aggregate class yields a subdiagram showing the aggregation relationship of the related aggregate class and its components. A complex generalized class is always related to a class which has a generalization relationship referred to as a general class (e.g. class *Person* in Figure 3). A complex generalized class represents a higher abstraction of the general class. It has the same name as the related general class and is depicted as a bold class box with a triangle with the letter G (generalization) inside, placed on the upper left corner of the box. Expanding a complex generalized class yields a lower level diagram, showing the generalization relationship of the related general class and its components.

By using the aggregation and generalization relationships derived from step (ii), we identify that class *Person* and *Member* are complex generalized classes and that class *Library* is a complex aggregate class.

(iv) Identifying Complex Associations

A complex association is a visual construct to improve the understanding of NOM. The purpose of a complex association is to consolidate all the associations related to the pairs of classes or their subclasses and/or subcomponents. Expanding a complex association between two classes yields a lower level diagram, defining all the associations between the classes and their subcomponents .

There are two types of complex associations in NOM, namely complex aggregate associations and complex generalized associations. A complex aggregate association is formed from a "serial" path between the participating classes. It is depicted by a bold line connecting the related classes with the name of the association followed by a diamond with the letter A (aggregation) inside. A complex generalized association is formed from a set of "parallel" associations between two classes. It is depicted by

a bold line connecting the related classes with a name of the association followed by a triangle with the letter G (generalization) inside.

By using the associations derived from step (ii), we identify a complex aggregate association *uses* from class *Person* to class *Library*. The complex aggregate association *uses* is composed of the associations *checkout* and *return*.

3.2 Behavioral Modeling

The behavioral model represents the temporal and transformational aspects of a system. It combines and integrates the dynamic (state diagrams) and functional model (data-flow diagrams) originally proposed in [Rumb91], adding database transaction capabilities. Thus, the behavioral model is composed of three diagrams: state diagrams, data-flow diagrams and transaction diagrams.

State diagrams are used to describe the life history of objects of a particular class. It specifies and implements the control of objects, identifying object operations and activities. Data-flow diagrams are used to describe the operations of an object/class. Transaction diagrams define the database transactions of the system, describing the sequence of communications between objects.

The behavioral model is built in several steps described below. We analyze the library database requirements and build the state diagram, data-flow diagram and transaction diagram for some of the classes identified in the nested object model.

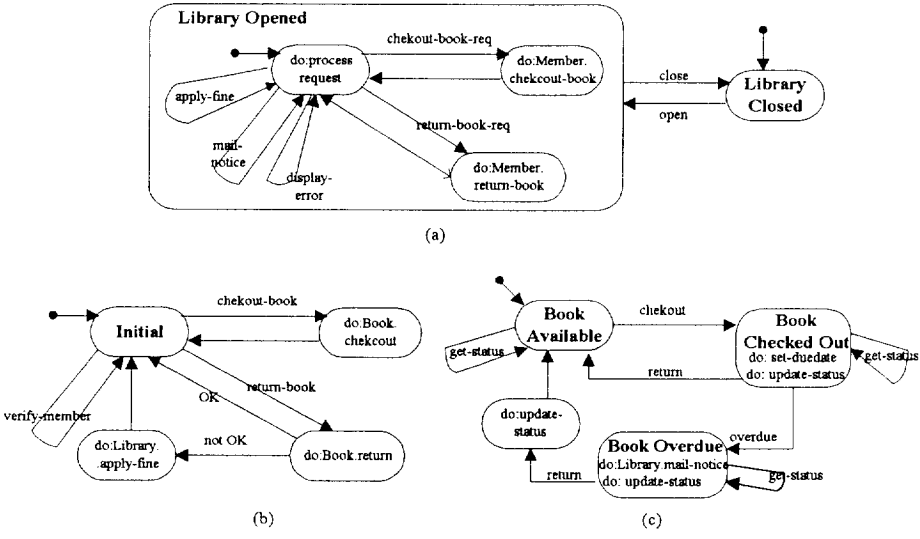

Fig. 4. State Diagrams for the Library System

(i) Building State Diagrams

In the behavioral model, each class will have a state diagram. A state diagram represents the life history of objects in a class and shows the sequence of operations that takes objects into several different states. It is a graph whose nodes are states of an object and whose arcs are transitions between states caused by events applicable to that object. The graphical notation used for a state diagram is the Harel statecharts [Hare88].

The state diagrams derived from the library system are described in Figure 4. We define state diagrams only for the main classes of the library system, namely *Library, Member, and Book.* Class *Library* will have the following operations: *open, close, checkout-book-req, return-book-req, apply-fine, mail-notice, and display-error* (see Figure 4(a)). Class *Member* will have the following operation: *checkout-book, return-book, and verify-member* (see Figure 4(b)). Class *Book* will have the following operations: *checkout, return, overdue, set-duedate, get-status, and update-status* (see Figure 4(c)).

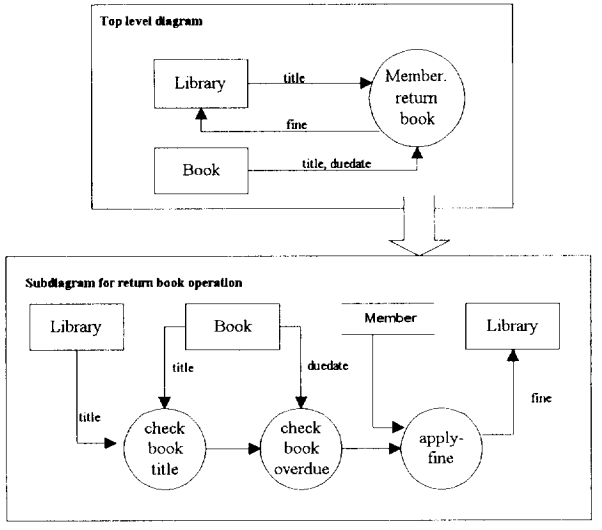

Fig. 5. Data-flow Diagram for return-book Operation

(ii) Building Data-flow Diagrams

In the behavioral model, a data-flow diagram (DFD) is used to describe the operation of an object/class identified in the state diagram. It is represented as a graph whose nodes are processes (transform data), actors (produce and consume data) and datastores (store data), and whose arcs are data-flows (move data) and control-flows (control process evaluation). The graphical notation used for a DFD is based on the notation proposed by [Dema79].

The level 0 of a DFD represents the operation of a class as a single process. Each operation has a set of activities described in a lower level diagram and are also represented as a process. Some activities may need to access the state of the class, which is represented as a dataflow from the datastore with the name of the class to the activity process. Moreover, some activities may use operations defined in other classes. In this case, these other classes are represented as actors with a dataflow connecting the activity process.

Since the operations derived from the state diagram of the library database example are very simple, we only show for illustrative purposes the data-flow diagram for the *return-book* operation in class Member (see Figure 5).

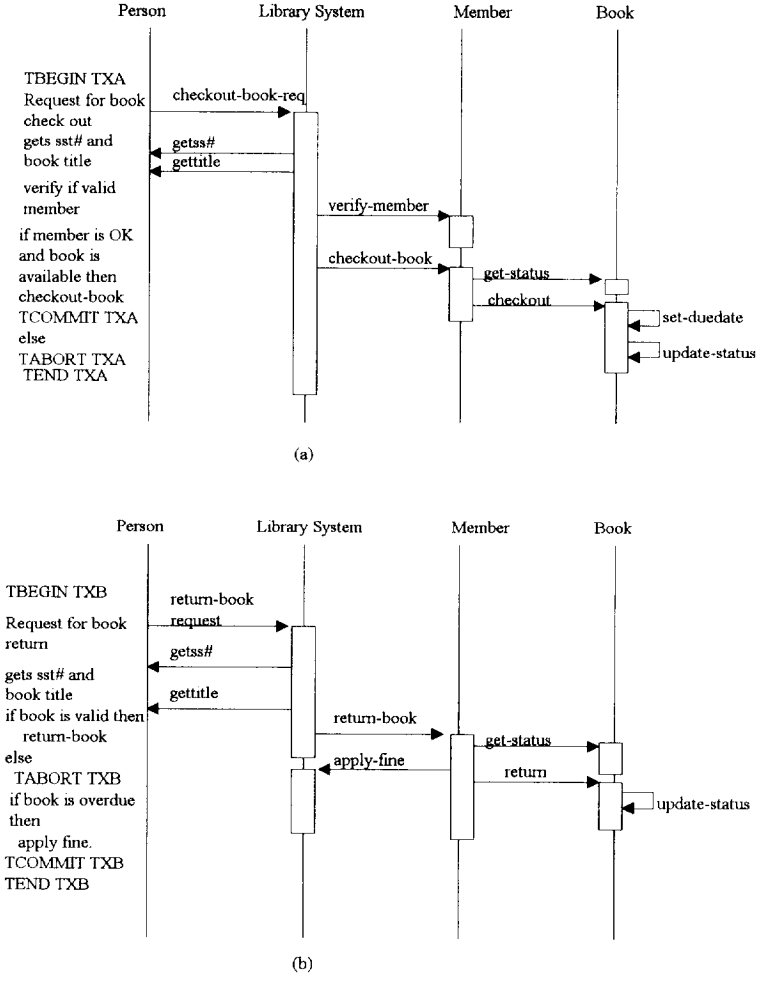

Fig. 6. Describing the Checkout and Return Book Transactions

(iii) Building Database Transactions

In the behavioral model, database transactions are described by transaction diagrams. Transaction diagrams extend interaction diagrams [Jaco92], with transaction commands, i.e. TBEGIN, TEND, TABORT, TCOMMIT. Each transaction defined has a name and represents a unit of work describing the interaction between the objects in the system.

In the transaction diagram, each object/class participating in the transaction is represented by a column drawn as a vertical line. The interactions between the objects are shown as operation invocations. Each operation of a class involved in the transaction is shown as a rectangle in its class column. The object originating the transaction is represented as the left most column and is called an "external object" to the system. To the left of the external object column we describe the transaction in natural language (pseudocode). The pseudocode describes what is happening in a particular part of the transaction (class operations).

From the library database example, we identify two major transactions in the system. The first one checks out books from the library and is described in Figure 6(a). The second returns books to the library and is described in Figure 6(b).

3.3 Nested Rule Modeling

The nested rule model (NRM) is a high-level graphical approach for conceptually designing rules in active object-oriented databases. NRM models a comprehensive set of rules, using two types of diagrams - nested rule diagrams (NRDs) and rule interaction diagrams (RIDs).

In NRD, a rule is depicted by an ellipse and has a name and a type (simple or composite), which is represented inside the rule icon separated by a line (see Figure 7). NRDs visually represent both simple and composite rules by using two types of abstraction techniques, embedding and nesting. Simple rules may be used to constrain the structure of objects and may also govern the object's behavior through dynamic rules, such as event-condition action or exception rules. Composite rules enable the designer to express complex object behavior by applying a set of constructors to simple rules and previously defined composite rules. Further, NRDs define the semantics of rule inheritance and overriding, and describe how the coupling mode of rules can be specified.

RIDs visually represent the interactions between rules, using multi-level diagrams. Multi-level diagrams are used to avoid the complexity of flat diagrams, where complex interconnection of rules may become impossible to comprehend because of the number of connecting lines. Both diagrams (NRD and RID) may be used together, so that the database designer can have a multi-level view of the rules defined for an object and at the same time visualize their interactions.

Below, we describe the steps used to build the NRM of an application. We use the library database example to illustrate the application of each step.

(i) Identifying Static Simple Rules

Static rules define constraints on the structure of a class that must always hold. These constraints are specified in terms of classes, objects, attributes and associations. Since static rules are always true and are defined in the context of an object/class, they are represented by placing an invariant condition within the context object/class all embedded within the rule icon (see Figure 7).

In NRD, static rules can be classified into the following constraint rules: attribute constraints, attribute domain constraints, mandatory/optional attribute constraints, attribute cardinality constraints, population type constraints, association cardinality constraints, existence dependence constraints, relational constraints, and uniqueness constraints. Below we describe each of the static rule types. Examples of constraint rules not illustrated in this paper can be found in [Silv95a].

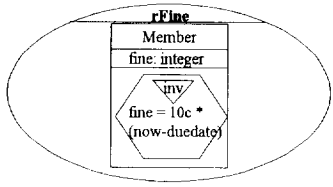

Fig. 7. Attribute Constraint for Class Member

a) Attribute Constraint: It is a constraint imposed on the attributes of a class or an object. An attribute constraint can also represent constraints that span multiple classes/objects. In this case, we attach the attribute constraint to a control object/class.

From the library example line (10) - "When a person returns a book , if it is overdue a fine of 10 cents per day is charged to the member for each book not returned on time.", we derive an invariant attribute constraint rule *rFine* in class *Member* (see Figure 7) .

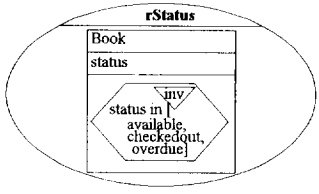

Fig. 8. Domain Constraint for Class Book

b) Attribute Domain Constraint: It is a constraint imposed on the domain of object/classes attributes. It is described by an enumeration if the domain is restricted to a list of discrete elements, or it is described by an interval in the case of well ordered domains. In NRD, domain constraints are represented in a textual form placed inside an invariant condition icon. The internal BNF syntax for the definition

of a domain constraint is: *[(<class_name>|<object_name>).]<attribute_name> in [<list_of_elements>| <interval>]*

Based on the library example, a book can be in states available, checkedout or overdue. To model these different states, we define a domain constraint rule *rStatus* in class *Book* (see Figure 8) . *rStatus* is specified with attribute *status* restricted to a list of elements *(available, checkedout, and overdue).*

c) Mandatory/Optional Attribute Constraint: It is a constraint that specifies that the attributes of a class or object must always have a value (mandatory) or may not have a value (optional).

d) Attribute Cardinality Constraint: It is a constraint that specifies the minimum and maximum occurrences of a multivalued attribute.

e) Population Type Constraint: It is a constraint that limits the number of objects in a class.

f) Association Cardinality Constraint: It is a constraint that limits the number of associations between two objects/classes.

g) Existence Dependent Constraint: It is a constraint that an object cannot exist without being associated with another object.

h) Relational Constraint: It is a constraint that an object must maintain correspondence (inverse) with another object. That is, if an object is updated, the related inverse object must also be updated.

i) Uniqueness Constraint: It is a constraint that determines that every object of a class has a unique value for an attribute.

(ii) Identifying Dynamic Simple Rules in NRD

Dynamic rules monitor the way an object's processes may execute and relate to one another and how objects respond to specific events and exceptions. In NRD, dynamic rules include event-condition-action rules, exception rules, contingency rules, precondition rules, postcondition rules, and production rules. In this article, only the dynamic rules related to the library database example will be illustrated. Examples of the dynamic rules not illustrated in this paper can be found in [Silv95a].

a) Event-Condition-Action Rule: The Event-Condition-Action (ECA) rule defined by [Daya88] was originally used by [Carl83] to extend relational databases with active capabilities. The ECA rule enables the database to monitor a situation represented by an event and one or more conditions and execute the corresponding actions when the event occurs and the conditions are evaluated to true. In NRD, a named event is only referenced in the rule using an event icon (a parallelogram). The actual

description of the event is represented using the nested event diagram, which is described in the next section. A condition determines if an action can be executed and is represented by a condition icon (a hexagon). A condition that is always "true", can be omitted. An action is always represented as a process which is applicable to a specific object. An action is implemented as a method in the object and can actually trigger the execution of other rules. In addition, the action part can be used to control the reasoning of production rules. The complete ECA rule is depicted by an event icon connected with an arrow to the condition icon and which in turn is connected to the action icon with an arrow. Below we show examples of ECA rules for the classes *Library, Member, and Book*. ECA rules of class *Member* that are overridden by class *Faculty Member* are described later in this article.

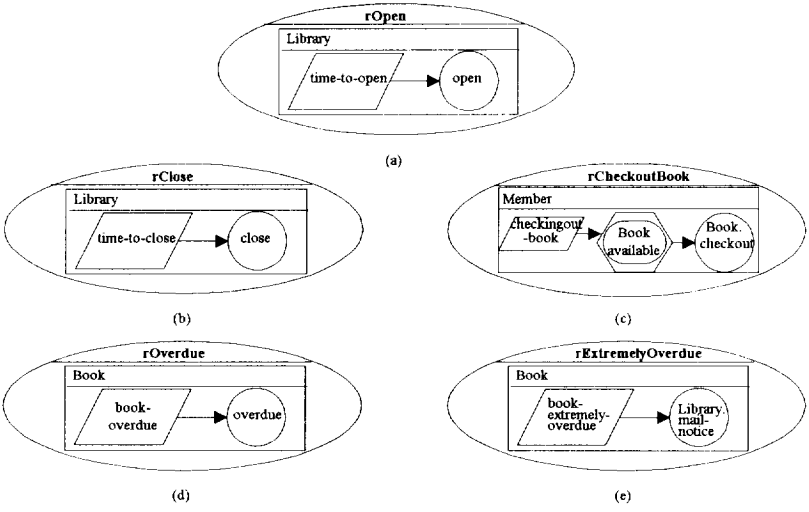

Fig. 9. ECA Rules for the Library System

- ECA rules for class Library: From line(2) we identify rules *rOpen and rClose* for opening and closing the library. *rOpen* has an event *time-to-open* and an action which is the operation *open* (see Figure 9(a)). *rClose* has an event *time-to-close* and an action which is the operation *close* (see Figure 9(b)).

- ECA rules for class Member: From line(5) we identify a rule *rCheckoutBook* for checking out a book. *rCheckoutBook* has an event *checkingout-book*, a condition testing if the book is in state *available*, and an action *checkout* applied to the book (see Figure 9(c)).

- ECA rules for class Book: From line(9) we identify a rule *rOverdue* for returning an overdue book. *rOverdue* has an event *book-overdue*, which determines when the operation *overdue* is executed (see Figure 9(d)). From line (11), we identify a rule *rExtremelyOverdue* for a person who keeps an overdue book more than seven

consecutive days. *rExtremelyOverdue* has an event *book-extremely-overdue*, and an action which is the operation *mail-notice* in class *Library* (see Figure 9(e)).

b) Exception Rule: An exception rule is a special case of an ECA rule. It specifies an event, a condition, a main action and a series of exception actions. However, only one action will be executed when the event occurs and the condition is evaluated. Each action will have a triggering value attached to the end of the incoming arrow from the condition. An action is executed if its triggering value matches the value returned by the condition evaluation. If no match is found, the main action is executed. The main action is defined by a straight arrow from the condition. Each exception action is represented as a branch of the straight arrow. Below we show an example of an exception action in which a condition returns two values: TRUE (T) or FALSE (F).

From the library example line (5) - "A person may check out books only if it is a member of the library, otherwise an invalid checkout request message is sent to the person.", we define an exception rule *rCheckoutBook* in the context of the class *Library* (see Figure 10). *rCheckoutBook* has an event *checkout-book-requested,* a condition using the *verify-member* method to check if the person is a member, a main action which is an operation *checkout-book* in class Member with a triggering value *T* (TRUE), and an exception action *display-error* with a triggering value *F* (FALSE).

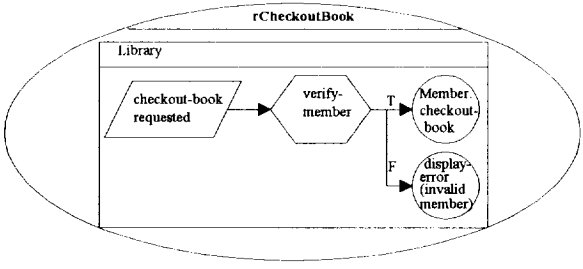

Fig. 10. Exception Rule for class Library

c) Contingency Rule: Contingency rules are specified for actions that are time-constrained. If these actions cannot be carried out within a pre-specified time, alternative actions need to be executed. In NRD, contingency rules are represented by modifying an ECA rule by defining an action to be constrained by a condition that specifies its timing constraints.

d) Operation Precondition Rule: An operation precondition rule expresses those constraints under which an operation will be allowed to be performed.

e) Operation Postcondition Rule: Operation Postcondition rules are constraints that guarantee the results of an operation. From the library database example, we define

a postcondition rule named *rSetDueDate (duedate = now + 2 weeks)* in class *Book*, which must hold after executing the operation *set-duedate* (see Figure 11).

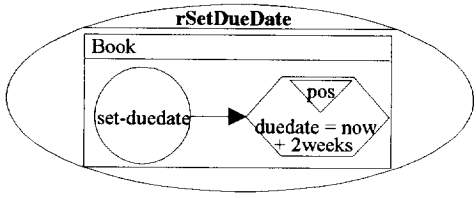

Fig. 11. Postcondition rule rSetDueDate

f) Production Rule: A production rule specifies the policies or conditions for inferring or computing facts from other facts. It is composed of a condition and an action and can be used for representing heuristic knowledge. Production rules are controlled by ECA rules, i.e. production rules will only be evaluated when triggered by an ECA rule.

(iii) Identifying Composite Rules in NRD

Composite rules enable the database designer to structurally abstract rules. They are particularly useful when the number of rules becomes very large and very difficult to comprehend. In this case, the rules are divided and grouped into related rules (components) represented by a composite rule.

There are two possible ways to represent a composite rule. The first approach, called nesting, is to define the composite rule and its components at different levels of abstraction through multi-level rule diagrams. The composite rule is represented in a diagram at level i, and can be expanded to a subdiagram at level i + 1, where its components are represented. At the higher level diagram, a composite rule is drawn as a bold ellipse with the composite rule name and a keyword representing the composite rule type within an upside down triangle (see Figure 12(a)). The keyword is used to inform the type of subdiagram that will be shown to the designer, once the composite rule is expanded. The second approach, called embedding, is to define all the component rules at the same level of abstraction, by enclosing all the component rules inside the rule icon (see Figure 12(b)). By placing component rules within the composite rule, we avoid using edges to describe the composite rule, which reduces the complexity of the diagram.

Both approaches may be used by the designer to represent different composite rules. When there are too many subdiagrams for a composite rule one should use the second approach (embedding), and when there are too many levels of embedded icons one should use the first approach (nesting). It is up to the designer to decide how to combine the two approaches.

In NRD, composite rules are classified into the following rules: disjunction rules and exclusive rules. A disjunction rule represents a set of rules, called member rules, which may be applicable to an object/class. A disjunction rule is successfully applied

to an object, when one of its member rules has been successfully executed (i.e. committed). An exclusive rule R represents a set of rules, which cannot be applicable to an object/class at the same time. An exclusive rule is successfully applied to an object, when only one of its rules has successfully executed (i.e. committed), but the other rules have failed (i.e. aborted or deactivated).

For example, let us consider that we change the requirements of the library system to allow a faculty member to checkout books for a period of one month if he/she has no books overdue, but reduce the checkout period to three weeks after the fifth overdue book.

To model this example, we assume that rules *rCheckout1Month* (representing a checkout period of one month), and *rCheckout3Weeks* (representing a checkout period of three weeks) have already been defined. Then, we define a more general rule *rCheckoutPeriod* in class Faculty Member as a composite exclusive rule of *rCheckout1Month* and *rCheckout3Weeks* (see Figure 12).

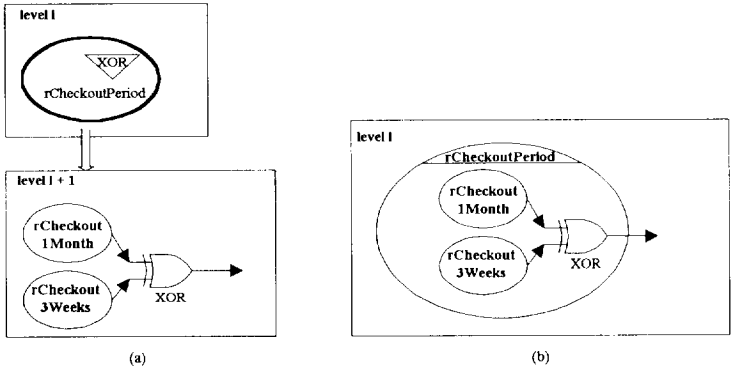

Fig. 12. Composite Exclusive Rule rCheckoutPeriod

(iv) Identifying the Coupling Mode of Rules in NRD

In AOODBSs, rules are triggered within a database transaction (i.e., triggering transaction) and are executed according to their coupling modes to the transaction. The coupling modes of a rule determine at what point in the triggering transaction the rule will be evaluated and determine whether the rule will be executed as a separate top level transaction or will be executed as a subtransaction of the triggering transaction.

In NRD, we support the comprehensive collection of coupling modes defined in [Bran93]. These coupling modes include immediate, deferred, separate, parallel causally dependent, sequential causally dependent, and exclusive causally dependent. Coupling modes are represented by keywords identifying the coupling mode type and are depicted by attaching the keyword to the connections between an event and a condition as well as a condition and an action. ECA rules without an explicit representation of their coupling modes are assumed to have an immediate mode. Below,

for each coupling we list its name, its keyword name within parenthesis, and give a full description of its meaning.

- Immediate (*imm*): The rule is evaluated immediately after its event is detected and is executed as a subtransaction of the triggering transaction.

- Deferred (*def*): The rule is evaluated immediately before the triggering transaction commits and is executed as a subtransaction of the triggering transaction.

- Separated (*sep*): The rule is evaluated immediately after its event is detected and is executed as a separate top level transaction independent of the triggering transaction.

- Parallel Causally Dependent (*pcd*): The rule is evaluated immediately after its event is detected and is executed as a separate top level transaction with commit and abort dependency with the triggering transaction. That is, the rule may execute in parallel to the triggering transaction as a top level transaction, but may not commit until the triggering transaction commits and must abort if the triggering transaction aborts.

- Sequential Causally Dependent (*scd*): The rule is evaluated immediately after its event is detected and is executed as a separate top level transaction only after the triggering transaction has committed.

- Exclusive Causally Dependent (*ecd*): The rule is evaluated immediately after its event is detected and is executed as a separate top level transaction in parallel with the triggering transaction, but it only commits if the triggering transaction has aborted. This type of coupling mode is used for contingency rules.

Figure 13 shows a rule *rExtremelyOverdue* in class *Book,* where the rule is evaluated *immediately*, but the action of *mailing a notice* is executed as a *separate* transaction.

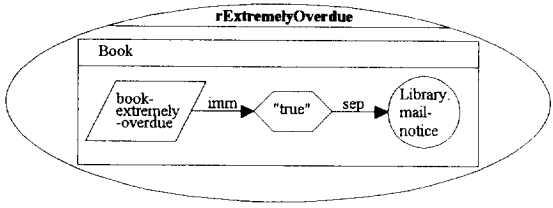

Fig. 13. Rule with an Immediate/Detached Coupling Mode

(v) Identifying the Inheritance and Overriding of Rules in NRD

Rules attached to a class are automatically inherited by each subclass. Like methods, rules can be overridden. Overriding a rule means that the subclass has attached to it a modification or a refinement of the rule. There are two main reasons the designer may want to override a rule: to specify a rule that is the same as the inherited rule, except it adds some behavior usually affecting new attributes of the subclass, or to tighten the specification of a rule by tightening the type of the arguments used in the expressions.

In NRD, a rule can only be overridden by another rule of the same type and the same name. Below we give the definition of the semantics for overriding rules in the context of a class A and a subclass B.

Static Rules:

A static rule R1 with an invariant condition C1 in class A is overridden by a rule in subclass B, if and only if the rule has the same name R1, but a different invariant condition.

Dynamic Rules:

a) ECA Rule Overriding: An ECA rule R1 with an event E1, a condition C1, and action A1 in class A is overridden by a rule in subclass B, if and only if the rule has the same name R1, the same event E1 but either a different condition or action.

b) Exception Rule Overriding: An exception rule R1 with an event E1, a condition C1, an action A1 and an exception action A2 is overridden by a rule in subclass B, if and only if the rule has the same name R1, the same event E1 but with either a different condition, action, or exception action.

c) Contingency Rule Overriding: A contingency rule R1 with an event E1, a condition C1, an action A1, a pre-specified execution time T1 and an alternate action A2 is overridden by a rule in subclass B, if and only if the rule has the same name R1, the same event E1 but either a different condition, action, pre-specified time, or alternate action.

d) Operation Precondition Rule Overriding: An operation precondition rule R1 with precondition P1 and action A1 is overridden by a rule in subclass B, if and only if the rule has the same name R1 with a different precondition, but the same action A1.

e) Operation Postcondition Rule Overriding: An operation postcondition rule R1 with an action A1 and postcondition P1 is overridden by a rule in subclass B, if and only if the rule has the same name R1 with the same action A1, but different postcondition.

f) Production Rule Overriding: A production rule R1 with condition C1 and action A1 is overridden by a rule in subclass B, if and only if the rule has the same name R1 with the same condition C1 but different action.

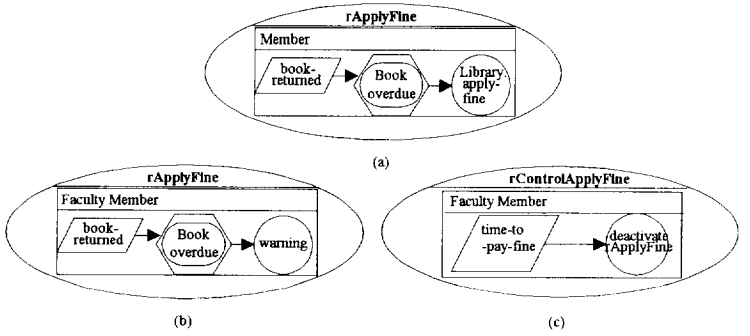

Fig 14. Rule Inheritance and Overriding for classes Member and Faculty Member

Not all inherited rules may be suitable for a subclass [Dill93, Kapp94]. To model this case in NRD, we override a rule that is not suitable for the subclass and falsify its condition and nullify its actions. Thus, a rule will still be propagated to the subclasses, but it will never be executed, unless it is overridden again.

Moreover, there are situations where the database designer wants to deactivate a rule, stopping the inheritance of a rule. This is particularly useful for experimenting with "what-if" scenarios, analyzing the impact of different rules on the system behavior [Buch95, Kapp94, Dill93]. In NRD, the activation/ deactivation of a rule is represented by an action using the keywords *activate/deactivate* followed by the name of the rule. A deactivated rule is represented with a gray background. Below we illustrate the overriding of rules in the library database system.

- Example of Rule Overriding: From the library example line(10) - "When a person returns a book , if it is overdue a fine of 10 cents per day is charged to the member for each book not returned on time. A faculty member will receive only warnings for the first 5 overdue books. After that, the faculty member will start paying fines for overdue books.", we define a rule *rApplyFine*, which is first defined in class *Member*, and then redefined in class *Faculty Member*. In class *Member*, *rApplyFine* is defined with an event *book-returned*, a condition that the book is *overdue*, and an action which is the operation *apply-fine* in class *Library* (see Figure 14(a)). In class *Faculty Member*, *rApplyFine* overrides the rule in class *Member* with a different action, which is an operation *warning* (see Figure 14(b)). Also, in order to deactivate the overriding of rule *rApplyFine*, we define another rule *rControlApplyFine*, which is executed after the *Faculty Member* has received *five warnings*. *rControlApplyFine* has an event *time-to-pay-fine*, and an action *deactivate rApplyFine* (see Figure 14(c)).

(vi) Identifying the Interconnection of Rules in RID

Rule interaction diagrams (RIDs) visually show the interdependence between rules. It is a very useful diagram to show the database designer, the cascading of rules, and how they relate to each other. The interdependence between rules is based on the action of each rule. If the action of a rule R1, causes an event which triggers the evaluation of another rule R2, then R1 is a triggering rule for R2.

In a tightly coupled environment where rules are highly interdependent, any flat diagram representing the interconnection of these rules is likely to be highly connected. In the case of many connected rules, a flat diagram can become almost impossible to comprehend. An effective solution to this problem is to use multi-level diagrams [Carl89] to represent the interconnection of rules.

RID uses multi-level diagrams to show the interdependence of rules. To support the definition of multi-level diagrams, RID defines a new type of rule object, called a rule connector. A rule connector is an object in the RID which contains the same name of a rule defined in the NRD. It is depicted by a dotted ellipse with the related rule name inside (see Figure 15) and it is used to depict the triggering rules of a rule.

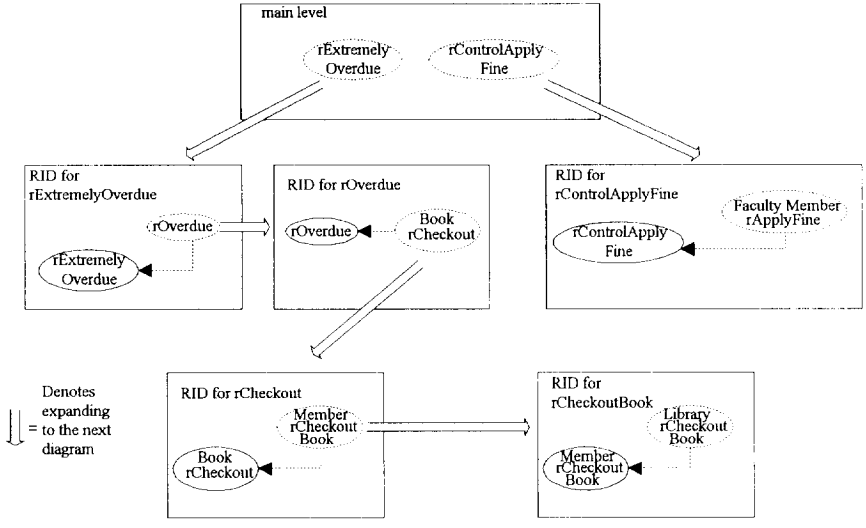

Fig. 15. Rule Interaction Diagrams for the Library System

RIDs are organized hierarchically. At the main level (diagram level zero) all the rules that do not trigger the execution of another rule are represented as rule connector objects. Moreover, each rule will have a diagram and will be referred to as the "main rule". The main rule will be represented with an ellipse, if simple, and a bold ellipse, if composite, with its name inside. All the triggering rules of the main rule will be represented in the diagram as rule connector objects and will be linked to the main rule with dotted arrows. The navigation between rule interaction diagrams is done by expanding a rule connector.

In NRM, RIDs and NRDs are used in an integrated way. The integration between these diagrams is achieved by allowing the main rules in RID to be expanded into NRDs. This allows a multi-level view of the rules defined for an object, showing at different levels of abstraction the visual definition and interaction of rules.

From selected rules defined in the library database system, we derive the following interactions described in Figure 15. Rule *rExtremelyOverdue* in class Book is triggered by rule *rOverdue*, which is in turn triggered by rule *rCheckout*. Rule *rCheckout* is triggered by rule *rCheckoutBook* in class Member, which is in turn triggered by rule *rCheckoutBook* in class Library. Rule *rControlApplyFine* is triggered by rule *rApplyFine* in class *Faculty Member*.

3.4 Nested Event Modeling

The Nested Event Model (NEM) visually models the events referenced in rules using a multi-level diagram, called the Nested Event Diagram (NED). NED models a comprehensive set of events, integrating the event types present in existing active object-oriented database systems. NED is composed of primitive (simple) and composite (complex) events. Primitive events correspond to elementary occurrences, and composite events correspond to events that are formed by applying a set of constructors to primitive and composite events. Below we use the nested event model to describe the events used in the rules defined for classes *Library, Member, Faculty Member and Book*. Event types not illustrated in this article can be found in [Silv95a].

(i) Identifying Primitive Events

A primitive event describes a point in time specified by an occurrence in the database (method execution events, and transaction events), temporal events, and explicit events.

a) Method Execution Event: In AOODBs, methods implement an operation for a specific class. The method executes when an object receives a message with the name of the method. The execution of a method gives rise to two events: an event which occurs immediately before the method is executed and an event immediately after it has executed [Geha92]. The parameters of a method can be later used in conditions specified in rules.

In NED, a method execution event is related to a particular class or to a particular object, only if a particular class name or object name is given. The BNF syntax for a method event is: *(before|after) [(<class_name>|<object_name>).]<method_name>*

Note that the class name can be omitted when specifying a method execution event. Below we represent the method execution events used by the rules in the library database example.

- Events for class Library: The event *checkout-book-requested* was referenced in rule *rCheckoutBook* (see Figure 10) and occurs after a request for checking out a

book is made. It is modeled as a method execution event which occurs *after* operation *checkout-book-req* is executed (see Figure 16(a)).

- Events for class Member: The event *book-returned* was referenced in rule *rApplyFine* (see Figure 14(a)) and occurs after a book is returned. It is modeled as a method execution event which occurs *after* the operation *return-book* is executed (see Figure 16(b)). The event *checkingout-book* was referenced in rule *rCheckoutBook* (see Figure 9(c)) and occurs before a book is checked out. It is modeled as a method execution event which occurs *before* the operation *checkout-book* is executed (see Figure 16(c)).

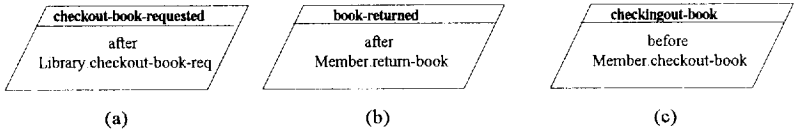

Fig. 16. Method Execution Events for classes Library and Member

b) Transaction Event: We treat transaction events as a special case of a method execution event. Transaction events are defined after or before transaction operations are executed (e.g. after tCommit). A transaction operation can be considered as a method applied to each object involved in the transaction.

c) Temporal Event: Temporal events are defined as an explicit point in time. They can be divided into two categories: absolute temporal events, and periodic temporal events.

Absolute Temporal Event: An absolute temporal event is specified with an absolute value of time [Daya88]. It is depicted by showing the year, month, week, day and time within a box inside an event icon.

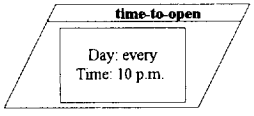

Fig. 17. Representation of Event "time-to-open"

Periodic Temporal Event: A periodic temporal event is an event that periodically reappear in time [Daya88]. Periodic temporal events are defined like absolute temporal events where some of the data fields can be omitted, meaning that the omitted data field matches *any* valid value for that field. In addition, the data field which represents the periodicity has the keyword *every* attached to it. The temporal event occurs periodically at every point in time that matches the partial specification. From the library database example, we represent the event *time-to-open,* referenced in rule

rOpen (see Figure 9(a)) as a periodic event which occurs every day at 10 a.m. (see Figure 17).

d) Explicit Event: An explicit (also called external or abstract) event is defined by the database designer and is explicitly signaled inside applications of the database system [Chak93]. This event may have parameters which are supplied at the time it is signaled to the AOODBS. An explicit event is depicted inside of an event icon with the following BNF syntax: *EXPLICIT <event_name> [(<parameters>)]*

(ii) Identifying Composite Events

The primitive events defined above are only able to represent elementary events. However, there are many applications that need to model composite (complex) events, which are composed of primitive and also previously defined complex events. Composite events are defined by applying event constructors to previously defined events, called component events, and occurs at the point of occurrence of the last event that was needed to make it happen [Geha92]. Like NRM, NEM uses two types of abstraction techniques, nesting (see Figure 20(a)) and embedding (see Figure 20(b)), to visually represent composite events. It is up to the designer to decide how to combine the two approaches.

In NED, composite events are classified into the following events: conjunction event, disjunction event, monitoring interval event, relative temporal event, closure event, history event, every-nth event, negative event, and sequence event.

a) Conjunction Event: The conjunction of events E1 and E2 occurs when both E1 and E2 have occurred, regardless of order[Gatz92].

b) Disjunction Event: The disjunction of two events E1 and E2, occurs when E1 occurs or E2 occurs [Gatz92]. Figure 18 represents the disjunction of two monitoring events *close-weekdays* and *close-weekends.*

c) Monitoring Interval Event: A monitoring interval event occurs when an event E happens anytime in an interval I and some condition C holds during the interval [Gatz92].

An interval I is specified by a starting and ending point in time and is depicted by a bar. The starting point and ending point of an interval can be defined by the occurrences of two events. A condition C is always associated with an interval. It is depicted within the bar interval. If there are no conditions related to the interval we do not represent the condition icon. An arrow with a flash below the time interval denotes the point in time of the occurrence of the monitoring interval event.

If an interval has a starting or ending point defined by an absolute or periodic temporal event, the interval is represented with the textual description of the date and time of the temporal event placed below its left and right end. Below we show a monitoring interval events in the library database example.

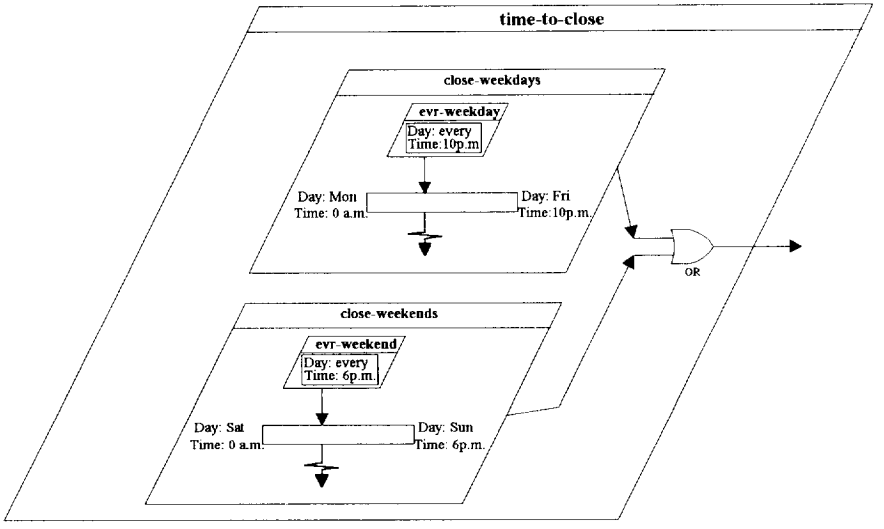

Fig. 18. Representation of Event "time-to-close"

- Events for class Library: The event *time-to-close* was referenced in rule *rClose* (see Figure 9(b)) and occurs every weekday at 10 p.m., or Saturdays and Sundays at 6 p.m. It is modeled as a composite event, which represents the disjunction of two monitoring events, called *close-weekdays* and *close-weekends* (see Figure 18). The monitoring temporal event *close-weekdays* occurs only if a periodic event *evr-weekday* occurs at 10 p.m. between Monday and Friday. The monitoring temporal event *close-weekend* occurs only if a periodic event *evr-weekend* occurs at 6 p.m. between Saturday and Sunday.

d) Relative Temporal Event: A relative temporal event [Daya88] is a special case of a monitoring interval event. It corresponds to a specific point in time in relation to a triggering event E. A relative temporal event occurs after a triggering event E has occurred and a time interval I has elapsed. The triggering event E can be any event specified in NED.

- Events for Class Book: The event *book-overdue* was referenced in rule *rOverdue* (see Figure 9(d)) and occurs at the book's duedate after it had been checked out. It is modeled as a relative temporal event caused after the execution of the operation *checkout*, with an interval defined by the *book's duedate* (see Figure 19(a)). The event *book-extremely-overdue* was referenced in rule *rExtremelyOverdue* (see Figure 9(e)) and occurs when a book has been overdue for seven days. It is modeled as a relative temporal event caused after the execution of the operation *overdue*, with an interval of *seven days*, subject to a constraint that the book is in state *overdue*(see Figure 19(b)).

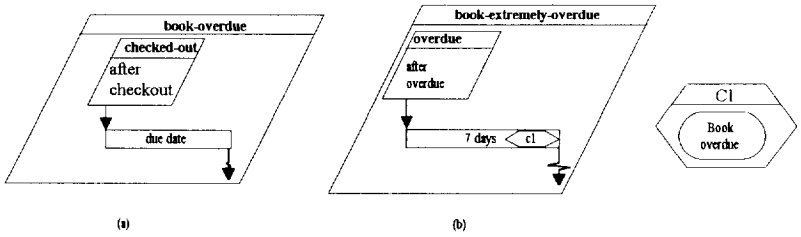

Fig. 19. Representation of Events "book-overdue" and "book-extremely-overdue"

e) Closure Event: A closure event signals only the first occurrence of an event E, even if the event E continues to occur [Gatz92]. The closure event is denoted by placing an '*' to the "out arrow" of the event (e.g. see event *checked-out* in Figure 20).

f) History Event: In some applications an event may repeatedly occur. In such a case, a history event designates a specific occurrence as the triggering event [Gatz92]. The history event is denoted by placing an occurrence identifier to the "out arrow" of the event (e.g. see event *warning* in Figure 20).

g) Every-nth Event: A Every-nth event is used to describe events that occur periodically [Geha92]. It is defined similarly to a history event by placing the keyword *every* before the occurrence number.

h) Negative Event: A negative event is specified by applying an negative constructor to an event. A negative constructor applied to an event E occurs only if event E did not occur [Gatz92]. The non-occurrence of an event is depicted by a cross over the event icon.

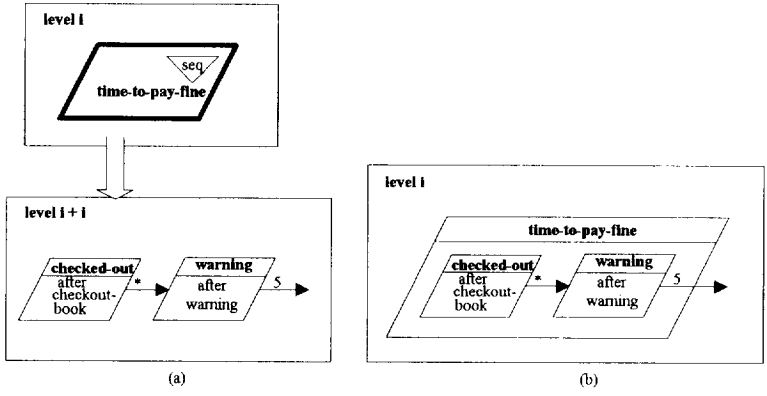

Fig. 20. Sequence Event "time-to-pay-fine" in class Faculty Member

i) Sequence Event: A sequence of two events E1 and E2, occurs when E2 occurs, provided E1 has already occurred [Geha92]. The time of occurrence of E1 is guaranteed to be less than E2.

- Events for class Faculty Member: The event *time-to-pay-fine* was referenced in rule *rControlApplyFine* (see Figure 14(c)). It is modeled as a sequence event which occurs after the method *warning* has been executed five times, since the first time a book had been *checked-out*. In Figure 20, we show the two possible ways (nesting, and embedding) to depict the sequence event *time-to-pay-fine*.

4 Conclusion

In this paper we proposed an integrated approach to active object-oriented database conceptual design, called A/OODBMT, based on several modeling techniques namely, the nested object model (NOM), the behavioral model (BM), the nested rule model (NRM), and the nested event model (NEM).

The nested object model is based on the object model originally defined in [Rumb91]. It extends the object model with nesting capabilities[Carl89], and provides abstraction mechanisms for developing database applications using multi-level diagrams. Moreover, it extends the object model to include rules to the definition of objects to specify the active behavior of objects. The behavioral model is based on the dynamic and functional models proposed in [Rumb91]. It integrates and extends these models to define the operations of objects. Moreover, the behavioral model enhances interaction diagrams [Jaco92] to describe the database transactions of the system. The nested rule model is based on a comprehensive survey [Silv95a] of rule specification approaches. It integrates and extends these approaches by supporting a comprehensive set of rules and by visually representing the rules and their interactions using multi-level diagrams. The nested event model is based on a comprehensive survey [Silv95a] of event specification approaches. It integrates and extends these approaches by supporting a comprehensive set of events and by visually representing events using multi-level diagrams in the context of rules.

The conceptual design of active object-oriented database applications is defined by integrating all the A/OODBMT models. Therefore, the significance of A/OODBMT is that:

(1) It provides a multi-level representation approach to the definition of the objects in a system, including their attributes, operations and rules, based on a new object model, called nested object model.

(2) It describes rules to be encapsulated within objects, localizing the modification of rules.

(3) It provides the modeling of database transactions in a conceptual level, so that the context of rule evaluation is understood.

(4) It provides a high-level graphical representation of active behavior through the specification of rules using multi-level diagrams. A very comprehensive set of rules is supported, and the modeling of the their coupling modes, inheritance, overriding and interactions is described.

(5) It provides a high-level graphical representation of active behavior through the modeling of events using multi-level diagrams. A very comprehensive set of events is supported and events are defined in the context of rules .

(6) It provides a complete description of how the different models used to represent the database application are integrated at higher-levels of abstraction in multi-level diagrams.

Therefore, we believe the presented approach is well suited for the design of active object-oriented database application, because of its semantic richness and the ability to deal with the complexity of many object/classes, rules and their interaction, and events. We are currently developing a CASE tool that will, not only support the graphical notation proposed in this paper, but also support the automatic code generation of the active object-oriented database schema.

References

[Anwa93] Anwar, E., Mangis, L., and Chakravarthy, S., "A New Perspective On Rule Support For Object-Oriented Databases" In *Proc. of the 1993 ACM SIGMOD Int'l Conference on Management of Data*, June 1993, pp. 99-108.

[Bich94] Bichler, P., and Schrefl, M., "Active Object-Oriented Database Design Using Active Object/Behavior Diagrams," In *Proceedings of the Fourth International Workshop on Research Issues in Data Engineering*, IEEE Comp. Soc. Press, Los Alamitos, CA, USA, 1994.

[Booc94] Booch, G., *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, 1994.

[Bran93] Branding, H., Buchmann, A. P., Kudrass, T., and Zimmermann, J., "Rules in an Open System: The REACH rule system," In Paton, N., and Williams, M. (eds.), *Rules in Database Systems*, Workshops in Computing, Springer-Verlag, 1993, pp. 111-126.

[Buch92] Buchmann, A. P., Branding, H., Kudrass, T., and Zimmermann, J. "REACH: a REal-time, ACtive and Heterogeneous mediator systems," In *IEEE Bulletin of the Technical Committee on Data Engineering*, Vol. 15, No. (1-4), December 1992.

[Buch95] Buchmann, A. P., A., Zimmermann, Blakeley, J. A., and Wells, D. L., "Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions," In *Proceedings of the 11th International Conference on Data Engineering*, 1995.

[Carl83] Carlson, C. R. and Arora, A. K., "UPM: A Formal Tool for Expressing Database Update Semantics", In *Proceedings of the Third International Conference on Entity-Relationship*, North Holland, NY, 1983, 517-526.

[Carl89] Carlson, C. R., and Ji, W., " The Nested Entity-Relationship Model," *In the 8th International Conference on Entity-Relationship Approach*, October 1989.

[Chak93] Chakravarthy, S. and Mishra, D., "Snoop: An expressive event specification language for active databases," Technical-Report UF-CIS-TR-93-007, University of Florida, March 1993.

[Cole94] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., Jeremaes, P., *Object-Oriented Development: the Fusion Method*, Prentice Hall, 1994.

[Daya88] Dayal, U., "Active Database Management Systems," In *Proceedings 3rd International Conference on Data Knowledge Bases*, Jerusalem, Israel, June 1988.

[Dema79] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, 1979.

[Dill93] Dillon, T., and Tan, P. L., *Object-Oriented Conceptual Modeling*, Prentice Hall, 1993.

[Gatz92] Gatziu, S., and Dittrich, K., "SAMOS: An active object-oriented database system," In *IEEE Bulletin of the Technical Committee on Data Engineering*, Vol. 15, No. (1-4), December 1992.

[Gatz94] Gatziu, S. and Dittrich, K., "Detecting composite events in active database systems using Petri nets," In *Proceedings of the Fourth International Workshop on Research Issues in Data Engineering*, IEEE Comp. Soc. Press, Los Alamitos, CA, USA, 1994.

[Geha91] Gehani, N. H., and Jagadish, H. V., "Ode as an Active Database: Constraints and Triggers," In *Proceedings of the 17th International Conference on Very Large Databases*, Barcelona, September 1991.

[Geha92] Gehani, N. H., Jagadish, H. V., and Shmueli, O., "Event Specification in an Active Object-Oriented Database," In *Proc. of the 1992 ACM SIGMOD Int'l Conf. on Management of Data*, CA, June 1992, pp. 81-90.

[Grah94] Graham, I., *Migrating to Object Technology*, Addison-Wesley, 1994.

[Hare88] Harel, D., "On Visual Formalisms," *Communications of the ACM*, Vol. 31, No. 5, May 1988, pp. 514-530.

[Hutt94] Hutt, Andrew. T. F., "Object Analysis and Design: comparison of methods," John Wiley & Sons Inc., 1994.

[Jaco92] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[Kapp94] Kappel, G., Rausch-Schott, S., Retschitzegger, W., and Vieweg, S., "TriGS: Making a passive object-oriented database system active," *Journal of Object-Oriented Programming*, June/July 1994, pp. 40-51.

[Kapp95] Kappel, G., Rausch-Schott, S., Retschitzegger, W., Tjoa, A., Vieweg, S., and Wagner, R., "Active Object-Oriented Database Systems for CIM Applications," In Marik, V. (ed.), *CIM-Textbook (TEMPUS-Project)*, Springer LNCS, (in print), 1995.

[Mart95] Martin, J., and Odell, J., *Object-Oriented Methods: a foundation*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[Mona92] Monarchi, D. E., and Puhr, G. I., "A Research Typology for Object-Oriented Analysis and Design," *Communications of the ACM*, Vol. 35, No. 9, September 1992, pp. 35-47.

[Pras94] Prasad, B., Perraju, T., Uma, G., and Umarani, P., "An Expert System Shell for Aerospace Applications," *IEEE Expert*, August 1994, pp. 56-64.

[Rasm95] Rasmus, D. W., "Ruling classes: The heart of knowledge-based systems," In *Journal of Object-Oriented Programming*, Vol. 5, No. 4, July/August 1995, pp. 41-43.

[Rumb91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-oriented modeling and design*, Prentice Hall, EngleWood Cliffs, 1991.

[Shla92] Shlaer, S. and Mellor, S. J., *Object Lifecycles : modeling the World in States*, Prentice Hall, 1992.

[Shen92] Sheng, O. R. L., and Wei, C., "Object-Oriented Modeling and Design of Coupled Knowledge-base/ Database Systems," *IEEE 8th International Conference on Data Engineering*, 1992, pp. 98-105.

[Silv95a] Silva, M. J. V., *A/OODBMT, an Active Object-Oriented Database Modeling Technique*, Ph.D. Thesis, Illinois Institute of Technology, 1995.

[Silv95b] Silva, M. J. V., and Carlson, C. R., "MOODD, a Method for Object-Oriented Database Design," *Data & Knowledge Engineering Journal*, Elsevier Science Publishers, Vol. 17, No. 2, November 1995.

[Thur94] Thuraisingham, B. and Schafer, A., "RT-OMT: A Real-Time Object-Modeling Technique for Designing Real-Time Database Applications," In *Proceedings of the IEEE Workshop on Real-Time Applications*, IEEE Comp. Soc. Press, Los Alamitos, CA, USA, 1994.

[Tsal91] Tsalgatidou, A., and Loucopoulos, P., "An Object-Oriented Rule-Based Approach to the Dynamic Modelling of Information Systems," In Sol, H. G., and Van , K. M. H. (eds.), *Dynamic Modelling of Information Systems*, North-Holland, Elsevier-Publications, 1991., pp. 165-188.