ELSEVIER

Information and Software Technology 38 (1996) 419–434

INFORMATION AND SOFTWARE TECHNOLOGY

An executable specification language for specification understanding in object-oriented specification reuse

Shih-Chien Chou, Jen-Yen Chen*, Chyan-Goei Chung

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, 30050 Taiwan

Received 26 January 1995; revised 23 August 1995; accepted 21 September 1995

Abstract

System analysis time can be reduced through specification reuse which, however, requires specification understanding. This paper presents an object-oriented executable specification language which reduces understanding time through executing specifications. In addition to being executable, the specification language hides as many classes as possible within subsystems, and explicitly specifies relationships between specification components. This facilitates specification modification. Moreover, the language explicitly specifies interface parameters of specification components. This facilitates specification composition.

Keywords: Specification reuse; Specification behavior; Executable specification language

1. Introduction

Software reuse can tremendously improve software development productivity and software quality [1–3]. Currently, many program code reuse techniques [4–11] are available. With these techniques, however, software developers must specify a system's specification and design document before they can reuse program code. To enhance the power of software reuse, some techniques for reusing design documents [12–17], and for reusing specifications [19–21] have been developed. Techniques for reusing specifications, for reusing design documents, and for reusing program code can be integrated to support a reuse-based software development paradigm which will substantially reduce software development time.

The object-oriented (OO) software development approach enhances software reusability through information hiding (encapsulation), modularity, abstraction, inheritance, and so on [22]. Reusability is thus one of the most important promises of the OO approach [23–25]. Despite the fact that many OO software reuse techniques have been developed [8–9,16,24,26–27], few successful techniques for reusing OO specifications are available. This prompted us to work on an OO specification reuse technique.

A specification primarily specifies functions, data, and behavior of a software system [28], where the system behavior is shown by the system's state changes. Since executing a system's functions will change its states, a system's behavior can be exhibited by executing its functions. Functions and data are thus major components in a specification. According to this, specifications with functions and data similar to those of an intended system are reusable. On the other hand, specifications with similar functions but different data may still be reusable. This reuse can be accomplished by changing data. However, specifications with different functions cannot be reused. In this sense, specifications with functions similar to those of the intended system are reusable. As the execution of a specification's functions exhibits the specification's behavior, specifications with similar functions have similar behaviors, and specifications with totally different behaviors have no similar functions. Therefore, specifications with behaviors similar to that of an intended system may have functions similar to those of the system and hence may be reusable. Such specifications are thus considered candidates for reuse.

Existent specifications are retrieved from a repository for reuse. In a retrieval, there may be many specifications retrieved, where some are reusable and others not. An

* Corresponding author. Phone: 886-35-712121 Ext. 54726 Fax: 886-35-724176 email:jychen@csie.nctu.edu.tw.

analyst should understand the retrieved specifications before he or she can identify the reusable ones. Normally, the analyst browses and reads the retrieved specifications in detail for the understanding. To save the understanding time, the retrieved specifications which are unreusable should be neither browsed nor read. They should be filtered out before the browsing and reading. As specifications with behaviors similar to that of an intended system are considered candidates for reuse, the retrieved ones with behaviors dissimilar to that of the intended system can be filtered out. Behaviors of the retrieved specifications should thus be understood for the filtering.

Normally, a system's behavior can be understood by observing the system's state changes. Understanding a system's behaviour by observing all its state changes, however, may be difficult, because the system's states may be complex. Since behavior understanding in our technique is just for filtering out unreusable specifications, understanding detailed behaviors is not neccessary. Thus, in this paper, 'behavior' refers to the input/output data transformations and object state changes during execution of a specification's functions. In understanding this behavior, an executable specification language is needed. This paper presents a specification language for the understanding.

In addition to being executable, our specification language facilitates modification and composition of specifications, because specifications may need to be modified before being reused, and the reused specifications should be properly composed to form a specification for the intended system. In the remainder of this paper, the process and issues of our OO specification reuse technique are described first. Next, the model of the executable specification language and the language itself are respectively described. Then, an example is used to illustrate the usage of the language. Finally, conclusions are given.

2. Process and issues of our OO specification reuse technique

Our OO specification reuse technique is based on the following considerations:

(1) Specifications with behaviors similar to that of an intended system are considered candidates for reuse.
    As mentioned above, systems with similar behaviors may have similar functions. Specifications with behaviors similar to that of an intended system may thus be reusable. Accordingly, in a specification retrieval, retrieved specifications with dissimilar behaviors are not reusable and hence should be filtered out. Note that in this paper, a *specification's behavior* denotes the behavior of the system specified by the specification.

(2) In addition to classes, subspecifications, or even entire specifications, are considered candidates for reuse.
    Most OO software reuse techniques [8–9, 24,26] reuse only classes, which are usually primitive units in software systems. Applying such techniques is often time-consuming because they tend to compose existent classes to form subsystems and then compose subsystems to form a software system. To remedy that, our technique reuses classes as well as subspecifications, or even entire specifications. As an OO specification or subspecification is composed of several classes and their relationships, reusing it corresponds to reusing all those classes and relationships. As classes in the same (sub)specification seem closely related and related information tends to be reused together [23], reusing (sub)specifications is more efficient than reusing classes alone. Moreover, reusing relationships among classes reduces the time for structuring classes, because classes should be structured by their relationships. According to the above description, reusing classes as well as (sub)specifications makes our technique more time-saving than those that reuse classes alone.

(3) A software system may be specified by reusing various existent (sub)specifications.
    Normally, a software system can be partitioned into several subsystems. There is a possibility that some subsystems of a software system are similar to those of some existent (sub)specifications, and other subsystems are similar to those of other existent (sub)-specifications. A software system may thus be specified by reusing various existent (sub)specifications.

Existent OO specifications should be classified and stored in a repository before being retrieved for reuse. They can be classified by facets [29], features [30], semantic networks [31] and so on. According to the considerations given above, our OO specification reuse process is outlined below.

(1) When an analyst wants to specify a (sub)system (or a class), he or she describes its requirements in a query. The query format should be the same as those for classifying specifications. For example, if specifications are classified by semantic networks, the query should be a semantic network.
(2) A reuse support tool retrieves (sub)specifications (or classes) from the repository according to the query. The retrieved ones are *candidate reusable (sub)specifications (or classes)*.
(3) The analyst executes the retrieved candidates and filters out those with behaviors dissimilar to that of the intended (sub)system (or class). He or she then browses and reads the details of the other candidates and selects some for reuse. Normally, those with the most similar behaviors are selected. If the selected

ones do not exactly fit the intended (sub)system (or class), they should be modified.

(4) If no candidates are retrieved in step (2) or no retrieved candidates are selected for reuse, the intended (sub)system (or class) cannot reuse any existent (sub)specification (or class). The analyst should thus specify the (sub)system (or class) from scratch.

(5) After specifying all subsystems and classes of the intended system, the analyst composes the reused (sub)specifications and classes into a specification for the intended system

According to the process outlined above, major issues for our OO specification reuse technique are:

(1) classification and retrieval of specifications;
(2) understanding of specification behaviors; and
(3) modification and composition of specifications.

The classification and retrieval technique for our OO specification reuse technique has been described elsewhere [32]. This paper presents an OO executable specification language which facilitates the understanding of specification behaviors and facilitates the modification and composition of specifications.

3. Model of the specification language

The specification language is based on an object-oriented analysis (OOA) model. The following subsections respectively describe the design philosophy of the model, the model itself, and a specification example represented in the model.

3.1. Design philosophy of the model

A software system responds to a set of *stimuli, or events*, which occur in the external environment or in the internal of the system [33,34]. An external event can be a user command, an interrupt from another system, etc. For example, a library system responds to such external event as 'A borrower wants to borrow a book'. An internal event may occur periodically inside the system. For example, a supermarket system may periodically check and print the stock levels of its selling items. The response of an event corresponds to a part of the system's behavior. For example, suppose that a library system responds to the event 'A borrower wants to borrow a book' by lending the book to the borrower. Then, one will know that the library system will lend books to borrowers, which is a part of the system's behavior. In this sense, a system's behavior can be observed by checking its event responses. A system's event responses are referred to as its *system functions* in this paper. Under this circumstance, executing a system's system functions exhibits the system's behavior. As existent specifications with behaviors similar to that of an intended system are considered candidates for reuse, the reusability of a specification can be determined by checking its system functions.

Major components in an OO specification are classes and their relationships [22]. Classes are domain-oriented. They encapsulate attributes and operations, and instantiate objects that exhibit specific behaviors. A class operation is referred to as a *class service* in this paper. Class services can be separated into two types: (1) query services (Qservices), which retrieve attribute values from objects, and (2) state transition services (STservices), which change object states. This separation will reduce the couplings among class services and hence makes classes easier to modify. The relationships among classes include association, inheritance, aggregation, using (invocation) relationships, and so on [22].

In an OO specification, objects instantiated from classes and the instantiated objects' class services are used by system functions, which are outside the classes. When a system function is executed, some objects are triggered. The triggered object(s) may trigger still other object(s), and the triggering continues until the system function is completed. For example, when executing the system function 'Borrow_a_book', the library system triggers the object 'Book' to change its status from 'In_library' to 'Borrowed', and triggers the object 'Borrower' to increase his or her borrowed amount. The triggered objects will execute their class services. This in turn will accomplish a system function. Therefore, in an OO specification, system functions can be identified by tracing class services and invocation relationships among the services.

Some OOA methods [35] do not explicitly specify system functions in specifications. Anyone who wants to understand the behaviors of such specifications may need to trace class services and invocation relationships among the services in order to identify and check system functions, because a specification's behavior is exhibited by executing its system functions. Behaviors of such OO specifications are thus not easy to understand. On the other hand, if an OO specification explicitly specifies system functions, and describes how the system functions are accomplished by class services, the specification's behavior would be easier to understand. Since our specification language must facilitate the understanding of specification behaviors, it should be based on an OOA model that explicitly specifies system functions. The next subsection describes our OOA model. Only the model is described, but not the OOA process.

3.2. The OOA model

Our OOA model is composed of two sub-models: (1) the class sub-model that specifies classes and relationships among classes, and (2) the function sub-model

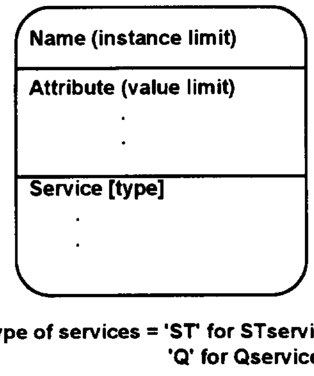


Fig. 1. Notation for a class.

that specifies system functions. In the class sub-model, a class encapsulates attributes and services, and instantiates objects that exhibit specific behavior. Class services are separated into Qservices and STservices. Relationships among classes specified in our model include association, inheritance, aggregation, and invocation (using) relationships.

Fig. 1 shows the notation for representing a class which is divided into three fields. The first field specifies the name and instance limit of the class, where the instance limit limits the number of objects instantiated from the class. The second field specifies class attributes and their value limits. And the last field specifies class services. Each class service is associated with a mini-spec that describes its detailed operations. To prevent information overload, class service mini-specs are not shown in the notation, but described in the specification document instead. As for the relationships among classes, they are represented by the notations shown in Fig. 2. Classes and their relationships constitute the class sub-model of our OOA model. Fig. 3 illustrates an example of a class sub-model for a simplified library system which will be described shortly.

In the function sub-model, system functions are explicitly specified. Each system function is associated with a mini-spec that describes its detailed operations. Since system functions are accomplished by invoking class services, their mini-specs are described by class service invocations which are structured by these control structures: sequences, selections, and iterations. For example, the mini-spec of the system function 'Borrow_a_book' can be described in pseudo code as shown in Fig. 4.

When there are quite a few system functions, the list of system functions is so long that it may jeopardize readability of the specification. Thus, several system functions should be grouped into a subsystem. System functions which invoke services of the same classes are candidates to be grouped together. System functions are not specified alone in the function sub-model. Instead, they are listed in the subsystems to which they belong. System functions in a subsystem will invoke classes for services (i.e. invoke their class services). The invoked classes show the relationships between the system functions and the classes. Those classes are thus listed in the subsystem to which those system functions belong. Fig. 5

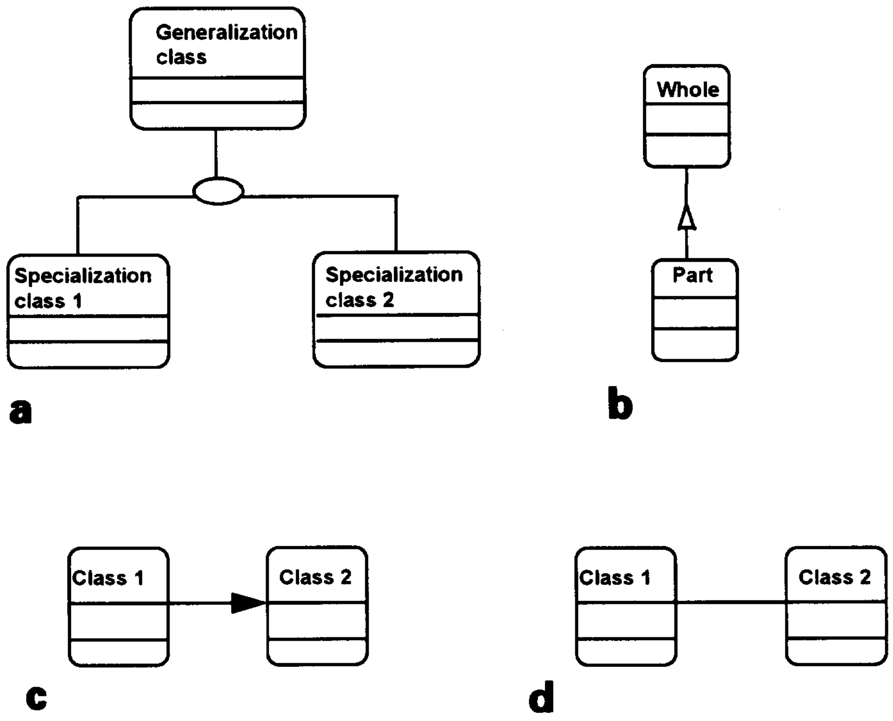


Fig. 2. Notations for relationships among classes: (a) Notation for an inheritance relationship; (b) Notation for an aggregation relationship; (c) Notation for an invocation relationship; (d) Notation for an association relationship.

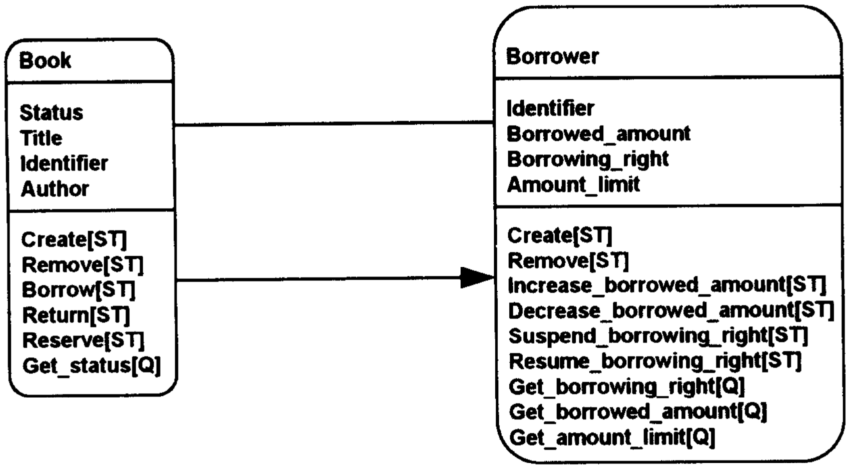


Fig. 3. Class sub-model for a simplified library system.

shows the notation for representing a subsystem which is divided into three fields. The first field specifies the subsystem name. The second field specifies classes invoked by the subsystem (actually, they are invoked by the subsystem's system functions). And the last field specifies the subsystem's system functions. Again, to prevent information overload, system function mini-specs are not shown in the notation, but described in the specification document instead.

After grouping system functions into subsystems, further grouping of those subsystems into even larger subsystems may be needed if there are many subsystems. Thus, the grouping activity will result in a hierarchy structure, called a *system-subsystem hierarchy* in this paper. The grouping ends when there is only one node (i.e. the system) in the root of the hierarchy. The hierarchy constitutes the function sub-model of our OOA model. Fig. 6 illustrates an example of a function sub-model for a simplified library system which will be described shortly.

In a system-subsystem hierarchy, the system or some of its subsystems may not directly contain system functions and classes (e.g. the root node in Fig. 6), because they are obtained by grouping subsystems. In this case, the (sub)systems indirectly contain all the system functions and classes of their subsystems. For example, in Fig. 6, the root node (i.e. the library system) indirectly contains all the classes and system functions that are in its two subsystems.

The system-subsystem hierarchy (i.e. the function sub-model) is used as a structure for large-scale model partitioning [36], and hence is a guidance for reading or checking the specification. When someone reads a specification, he or she first browses through its system-subsystem hierarchy to identify system functions. Then he or she checks each system function mini-spec and each class invoked by the system function.

Note that in Fig. 6, a class with the annotation '[P]' means it is a private class for the subsystem. That is, the class is invoked by no subsystems other than that subsystem. Conversely, a class with the annotation '[S]' means it is shared (or invoked) by several subsystems. These annotations are used for information hiding, which will be described in the next subsection.

3.3. Information hiding in the model

Modification is necessary when the reused (sub)specifications (or classes) do not exactly fit the intended (sub)system (or class). As most software engineers agree, information hiding is a good feature to improve modifiability [37]. In our technique, classes, subsystems and

```
IF the book is in library, the borrower's borrowing right is 'YES',
and the borrower's borrowed amount is not over limit THEN

    Invoke the service 'Borrow' of the class 'Book' to
        borrow the book to the borrower.

    Invoke the service 'Increase_borrowed_amount' of the class 'Borrower' to
        increase the borrower's borrowed amount.

ENDIF
```

Fig. 4. Pseudo code for the mini-spec of the system function 'Borrow_a_book'.

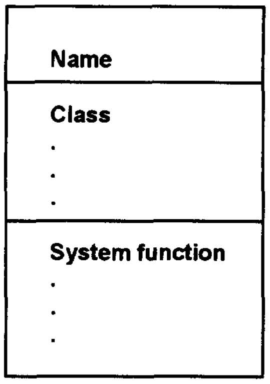


Fig. 5. Notation for a (sub)system.

even entire system specifications are subjected to reuse, classes and subsystem specifications (including system specifications) should thus possess the information hiding feature.

Classes in our model possess information hiding feature, but not subsystems. We thus rearrange subsystems' contents, hoping that they will possess this feature. Basic considerations for this rearrangement are to: (1) hide the classes invoked by a subsystem within it, and (2) access those classes by means of the subsystem's system functions. However, this rearrangement will fail when several subsystems invoke the same classes, because the shared classes cannot be hidden within any subsystem. For example, in Fig. 6, the class 'Borrower' are invoked by both the subsystems 'Book_management' and 'Borrower_management'. To remedy that, we loosen the requirements for information hiding in subsystems so that classes can be either hidden within subsystems or shared by several subsystems. If a class is invoked by only one subsystem, the class is hidden. Conversely, if a classes is invoked by several subsystems, it is considered shared and cannot be hidden. With this rearrangement, subsystems can possess partial information hiding feature, and hence improve their modifiability.

3.4. An example

A simplified library system is used as an example here. Its functional requirements are described in brief below:

A library system should manage both book and borrower status. Books in the library can be borrowed by borrowers. Borrowed books can be returned. If necessary, books can be reserved. Reserved books cannot be borrowed. When a borrower borrows books, his or her borrowed amount should be increased by the number borrowed. On the other hand, when the borrower returns books, the borrowed amount should be decreased. Borrowed amounts are limited. New books can be added and obsolete books can be discarded.

A borrower's borrowing right can be suspended. A suspended right can be resumed. New borrowers can be added and current borrowers can be removed.

Fig. 3 shows the class sub-model of the system's specification where two classes, 'Book' and 'Borrower', are specified. An association relationship between those classes connects a borrower and the books he or she

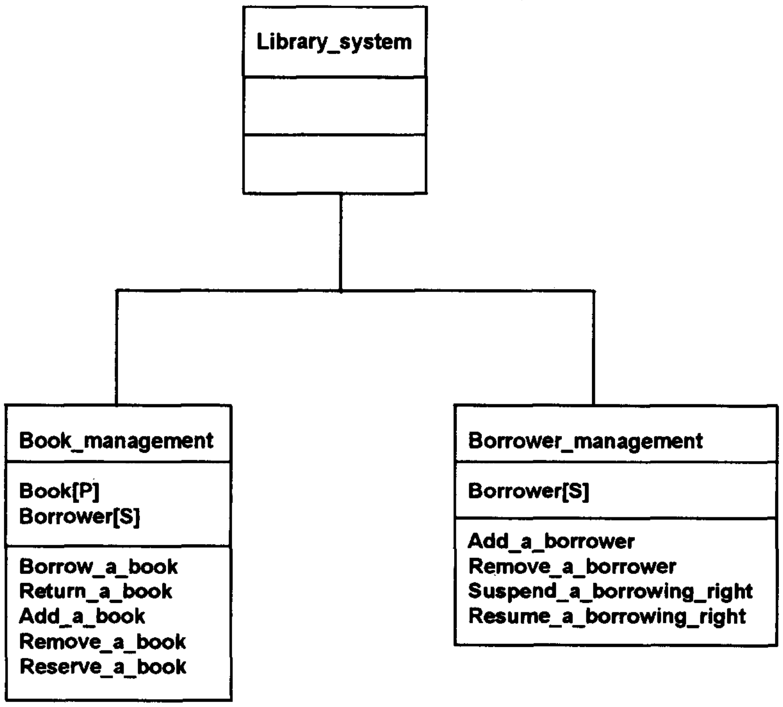


Fig. 6. Function sub-model for a simplified library system.

borrows. Moreover, there is an invocation relationship between the two classes, because when books are borrowed or returned by a borrower, the borrower's borrowed amount should be increased or decreased. That is, the services 'Borrow' and 'Return' of the class 'Book' invoke the services 'Increase_borrowed_amount' and 'Decrease_borrowed_amount' of the class 'Borrower', respectively.

Fig. 6 shows the function sub-model of the system specification. Here we partition the library system into two subsystems: one is 'Book_management' and the other is 'Borrower_management'. The former subsystem's system functions are: 'Borrow_a_book', 'Return_a_book', and so on. They are accomplished by invoking services of the classes 'Book' and 'Borrower'. For example, the system function 'Borrow_a_book' is accomplished by invoking the service 'Borrow' of the class 'Book' and the service 'Increase_borrowed_amount' of the class 'Borrower'. The latter subsystem's system functions are: 'Add_a_borrower', 'Remove_a_borrower', and so on. They are accomplished by invoking services of the class 'Borrower'. Each system function is associated with a mini-spec. For example, the mini-spec of the system function 'Borrow_a_book' is shown in Fig. 4. Note that in Fig. 6, the library system indirectly contains all the system functions and classes of its two subsystems.

4. The specification language

This specification language provides statements to specify (sub)systems (including their system functions) and classes in an OO specification. A (sub)system or class specified in this language is partitioned into two parts: (1) its declaration which primarily specifies interfaces, and (2) its body which specifies specification details. In the following subsections, basic considerations of this language are first described. Then, (sub)system specifications and class specifications are respectively described. Furthermore, Appendix A specifies the simplified library system in this language.

4.1. Basic considerations of the language

The specification language is based on the following considerations:

(1) It should facilitate specification behavior understanding.
As mentioned above, a specification's behavior can be understood by executing its system functions. System functions are thus executable units in our specification language. When executing a specification, an analyst follows the specification's system-subsystem hierarchy to locate a subsystem for the execution. The analyst then executes system functions of the subsystem. If the subsystem contains other subsystem(s), system functions of the subsystem(s) should also be executed, because the subsystem indirectly contains all system functions of its subsystem(s). After system functions of all the specification's subsystems are executed, the specification has been executed.

(2) It should facilitate specification modification.
Our OOA model allows subsystems to hide as many classes as possible so that modifiability of specifications can be improved. Our specification language further enhances this modifiability by providing statements to explicitly declare the relationships among classes and for declaring the classes invoked by subsystems. With this declarations, the dependency relationships among classes and those between system functions and classes can be constructed. From these dependency relationships, one can identify the system functions and classes that should be modified accordingly when some classes are modified. For example, in Fig. 3, the class 'Book' depends on the class 'Borrower', because the former's services invokes the latter's. Thus, when the latter is modified, the former may need to be modified.

(3) It should facilitate specification composition.
Composing software components into software systems can be facilitated by clear component interfaces. Composing (sub)specifications is not an exception. This language thus provides statements for explicitly declaring the interfaces of class services and those of system functions in order to facilitate specification composition.

4.2. (Sub)system specifications

As shown in Fig. 7, a (sub)system's specification starts with a 'SUBSYSTEM' statement which specifies its name. Its declaration and body are then respectively specified. The declaration delcares classes hidden in the (sub)system, shared classes, the (sub)system's subsystems and system functions, and interfaces of the system functions. The body specifies the detailed operations of the system functions.

In a (sub)system declaration (see Fig. 8), the following statements are used:

(1) 'PRIVATE_CLASSES' statement for declaring classes hidden in the (sub)system.
(2) 'SHARED_CLASSES' statement for declaring shared classes.
(3) 'DOMINATED_SUBSYSTEMS' statement.
This statement declares the (sub)system's subsystems, hence it can be used to declare the system-subsystem hierarchy of a specification.

```
SUBSYSTEM subsystem_name

    SUBSYSTEM_DECLARATION
        /* subsystem declaration is here */
    END_SUBSYSTEM_DECLARATION

    SUBSYSTEM_BODY
        /* subsystem body is here */
    END_SUBSYSTEM_BODY

END_SUBSYSTEM
```

Fig. 7. (Sub)system specification template.

(4) ‘SYSTEM_FUNCTIONS’ statement.
This statement is used to declare the (sub) system's system functions and their input/ output parameters. It thus facilitates specification composition.
Parameter types are also declared here.

A (sub)system's body (see Fig. 9) specifies detailed operations of the (sub)system's system functions. Referring to Fig. 4, system functions' detailed operations are described by class service invocations, which are structured by these control structures: sequences, selections, and iterations. Thus, service invocation statements, condition statements, and loop statements are needed for specifying system functions. Major statements for specifying system functions are described below:

(1) Service invoking statements.
A service invoking statement has the following syntax:
class_name.service_name(parameter, ..., parameter);
If the invoked service is an STservice (state transition service), no values will be returned. If the invoked service is a Qservice (query service), attribute values will be returned by means of the parameters.

A service invoking statement invokes a *class* service. Since a class may instantiate many objects, invoking the class's services may affect more than one of its objects. For example, suppose that the class ‘Book’ has instantiated two objects with the same title ‘Software engineering’. And, the service ‘Borrow’ of the class ‘Book’ lends books with a certain title to a borrower. Then, if a borrower wants to borrow a book with the title ‘Software engineering’, the service ‘Borrow’ will lend both books to the borrower. However, the invocation of a class service should normally affect only one of the class's objects. Thus, a class should have attribute(s) that can be used as a key so that the class's objects can be uniquely identified by their key values. Such attribute(s) are called *key attribute(s)*. Key attributes should be passed as parameters to class services so that invoking a class service will affect only one of the class's objects.

```
SUBSYSTEM_DECLARATION

    PRIVATE_CLASSES class_name, class_name, ... ;
    SHARED_CLASSES class_name, class_name, ... ;
    DOMINATED_SUBSYSTEMS subsystem_name, ... ;
    SYSTEM_FUNCTIONS {
        system_function_name ( parameter[I or O or IO]:type, ... );
         .
         .
         .
    }

END_SUBSYSTEM_DECLARATION
```

Fig. 8. (Sub)system declaration.

(2) Condition statement.
A condition statement has one of the following syntax:

IF condition THEN statements ELSE statements ENDIF
IF condition THEN statements ENDIF

where ‘statements’ is a sequence of statements.

(3) Loop statement.
A loop statement has the following syntax:

WHILE condition DO statements ENDDO

(4) Object retrieving statement.
Sometimes system functions must access objects. For example, to create/dissolve aggregation relationships among objects (see below), system functions must access objects. Objects must be retrieved before being accessed. Their key attributes are used to retrieve them. The following statement is used to retrieve an object from a class:

RETRIEVE object_name FROM class_name WITH_KEY key_attributes;

The retrieved object is assigned to the variable ‘object_name’, which can then be used to access the object.

(5) Relationship creating/dissolving statement.
During runtime, an object of a component class can exist without being attached to an aggregated object. An aggregated object can also be created without its component objects. Moreover, an object can exist without associating with any other objects. Therefore, aggregation and association relationships are dynamic relationships during runtime, hence statements are needed for creating and dissolving these relationships. An aggregation relationship between two objects is created by this statement:

object_1 PART_OF object_2;

where ‘object_1’ and ‘object_2’ are objects retrieved by the object retrieving statement as described above.

An association relationship between two objects is

```
SUBSYSTEM_BODY

    system_function_name ( parameter, ... )
    {
        statement;
        statement;
            .
            .
            .
    }

    system_function_name ( parameter, ... )
    {
            .
            .
            .
    }
    .
    .
    .
END_SUBSYSTEM_BODY
```

Fig. 9. (Sub)system body.

created by the statement

object_1 ASSOCIATION_OF object_2;

A relationship between two objects are dissolved by this statement

DISCONNECT object_1, object_2;

4.3. Class specifications

As shown in Fig. 10, a class's specification starts with a 'CLASS' statement that specifies its name and instance limit. Its declaration and body are then respectively specified. The declaration declares class attributes, class services, class service interfaces, and the relationships between the class and other classes. The body specifies the detailed operations of the class's STservices. Qservices do not need to be specified in the body, because the only operation of such services is to retrieve attribute values.

In a class declaration (see Fig. 11), the following statements are used:

(1) 'SUPER_CLASSES' statement.
This statement declares the class's super-classes. It can thus be used to declare inheritance relationships among classes.
(2) 'PART_CLASSES' statement
This statement declares the class's part classes. It can thus be used to declare aggregation relationships among classes. Cardinalities of those relationships are also declared. For example, in the class declaration of the class 'Car', the statement 'PART_CLASSES Wheel(0:4)' declares that the class 'Wheel' is a part class of the class 'Car', and a car can have at most four wheels.
(3) 'ASSOCIATED_CLASSES' statement.
This statement is used to declare association relationhips among classes. Cardinalities of such relationships are also declared. For example, in the class declaration of the class 'Book', the statement 'ASSOCIATED_CLASSES Borrower (0:1, 0:10)' declares that there is an association relationship between the classes 'Book' and 'Borrower', and a borrower can borrow at most 10 books and a book can be borrowed by a borrower at a time.
(4) 'INVOKED_CLASSES' statement.
This statement declares the classes whose services are invoked by the class being specified. It can thus be used to declare invocation relationships among classes.
(5) 'ATTRIBUTES' statement.
This statement is used to declare class attributes, their value limits, and their types. For example, the statement 'ATTRIBUTES Status ('Y','N'):CHAR' declares that the class being specified has the attribute 'Status' with the type 'CHAR', and the attribute value must be either 'Y' or 'N'.
(6) 'KEY_ATTRIBUTES' statement for declaring the class's key attributes.
(7) 'PROVIDED_STSERVICES' statement.
This statement is used to declare the class's STservices and their parameters (interfaces). Key attributes should be included in the parameters to uniquely identify objects. Parameter types should also be declared. All the parameters are for input, because STservices return no values.
(8) 'PROVIDED_QSERVICES' statement.
This statement is used to declare the class's Qservices and their parameters. Parameter types should also be declared. Key attributes should be included in the parameters. They are solely for input. Other parameters are for retrieving attribute values. They are for output.
(9) 'REQUIRED_SERVICES' statement for declaring class services invoked by the class being specified.

The last three statements are used to explicitly specify class interfaces, hence facilitate specification composition.

A class body (see Fig. 12) specifies detailed operations of the class's STservices. STservices of a class change states of the class's objects. They are thus specified according to the objects' state transitions. That is, the detailed operations of an STservice is composed of several state transitions. A state transition has this syntax:

```
current_state:
    statements;
```

where 'current_state' denotes the state of the class's objects whose states will be changed after the STservice is executed, and 'statements' is a sequence of statements that specifies the operations of the state transition. When an STservice of a class is executed, states of the class's

```
CLASS class_name(instance_limit)

    CLASS_DECLARATION
        /* class declaration is here */
    END_CLASS_DECLARATION

    CLASS_BODY
        /* class body is here */
    END_CLASS_BODY

END_CLASS
```

Fig. 10. Class specification template.

objects are checked. The object with the same state as that specified in a state transition's current state (i.e. 'current_state' above) are first identified. The statements specified in the state transition are then executed to change the identified object's state.

The current state in a state transition (i.e. 'current_state' as described above) takes the form:

⟨expression_1,expression_2, ...⟩

where the values of 'expression_1', 'expression_2', etc. correspond to the values of the first, second, etc. attributes of the object, respectively. In general, the values of all attributes constitute an object state. However, in some STservices, not all attributes are needed to specify the state transitions. The 'USED_ATTRIBUTES' statement is used to define the attributes used in an STservice. Attributes that can be used by a class's STservices include attributes of its own and those of its superclasses. To ensure that the invocation of a class service will change the state of only one of the class's objects, key attributes should be included in the used attributes.

There is a special current state used in STservice specifications: '⟨ ⟩'. It means null state. That is, the object is not existent. This state is used to specify the STservices that create or remove an object.

A state transition is composed of a state change operation and some other operations (e.g. operations to invoke other services). Thus, to specify state transitions in STservices, all the statements for specifying system functions mentioned above can be used. Moreover, this language provides a state change statement for STservices. It has the following syntax:

NEW_STATE_IS new_state;

where 'new_state' takes the same form as 'current_state' described above.

5. Usage of the specification language

The process to specify a car rental system by reusing existent specifications is used to illustrate the usage of the specification language. Functional requirements of the

```
CLASS_DECLARATION

    SUPER_CLASSES class_name, class_name, ... ;
    PART_CLASSES class_name(cardinality), class_name(cardinality), ... ;
    ASSOCIATED_CLASSES class_name(cardinality), class_name(cardinality), ... ;
    INVOKED_CLASSES class_name, class_name,...;
    ATTRIBUTES attribute_name(value_limit): type, attribute_name(value_limit): type, ... ;
    KEY_ATTRIBUTES attribute_name,attribute_name,...;

    PROVIDED_STSERVICES {
        STservice_name(parameter: type, parameter: type, ...);
        STservice_name( ... )
            .
            .
            .
    }

    PROVIDED_QSERVICES {
        Qservice_name(parameter[I or O]: type, parameter[I or O]: type, ...);
        Qservice_name( ... )
            .
            .
            .
    }

    REQUIRED_SERVICES {
        class_name.service_name, class_name.service_name,...;
    }

END_CLASS_DECLARATION
```

Fig. 11. Class declaration.

```
CLASS_BODY

    STservice_name(parameter, ...)
    {
        USED_ATTRIBUTES: attribute_name, attribute_name, ...;

        current_state_1:
              statements;

        current_state_2:
              statements;

        .
        .
        .
    }

    STservice_name(parameter,...)
    {
        .
        .
        .
    }
    .
    .
    .

END_CLASS_BODY
```

Fig. 12. Class body.

car rental system are described in brief below:

> The car rental system manages both car and customer status. Cars in the system can be rented by customers. Rented cars can be returned. Moreover, cars can be sold. When a customer rents cars, his or her rental amount should be increased by the number rented. On the other hand, when the customer returns cars, the rental amount should be decreased. New cars can be added and obsolete cars can be discarded.
>
> A customer's renting right can be suspended. A suspended right can be resumed. New customers can be added and current customers can be removed. The system also manages its employee's status, such as salaries and work times. New employees can be added and current employees can be removed.

In the reuse process, suppose that after specification retrieval, the library system specification (see Figs. 3 and 6, and Appendix A) and some other specifications are retrieved as candidate reusable specifications. For these candidates, the analyst executes them to observe their behaviors so that the unreusable ones can be filtered out. When executing a specification, the analyst first follows the system-subsystem hierarchy to select a subsystem. He or she then executes the subsystem's system functions. When executing a system function, the analyst first keys in its input data, then observes the output data and object state changes after the execution. Fig. 13 shows the execution of the library system specification (as specified in Appendix A) where the subsystem 'Book_management' is selected to execute. The execution of the subsystem's system function 'Borrow_a_book' is shown in the figure. After the execution of this system function, the state of the object 'Book' with identifier '1001' is changed from '⟨1001, "In_library"⟩' to '⟨1001, "Borrowed"⟩' and the state of the object 'Borrower' with identifier '1001' is changed from '⟨1001,0⟩' to '⟨1001,1⟩'. After observing the output data and state changes due to the execution of all the specification's system functions, the analyst can understand the specification's behavior to a certain degree of detail.

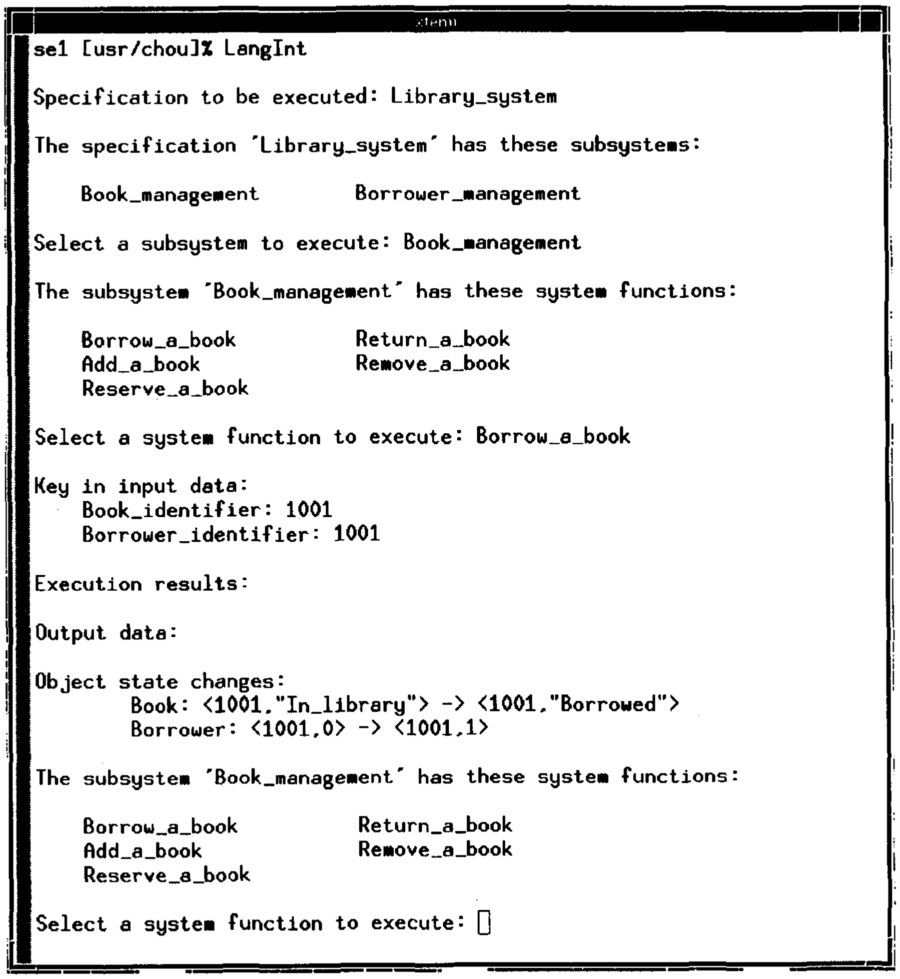
After understanding the behaviors of the retrieved candidates and filtering out the unreusable ones, the analyst reads the other candidates and selects some for reuse. If the reused ones do not exactly fit the intended system, they should be modified. For example, suppose that the library system specification is reused for the car rental system. Then, the following modifications should be performed: (1) the class 'Borrower' of the library system should be renamed to be 'Customer', (2) the class 'Book' should be renamed to be 'Car', (3) attributes of the reused classes should be modified, for example, the attribute 'Author' should not be an attribute of the class 'Car' and hence should be deleted from the reused class 'Book', (4) services of the reused classes should be modified, for example, the service 'Sell' should be a service of the class 'Car' and hence should be added, and (5) system functions should be modified, for example, a new system function 'Sell_a_car' should be added.

After specifying all subsystems of the car rental system, the analyst composes the subsystems to form a complete specification for the system. The composition can be accomplished by defining relationships among classes and constructing the system-subsystem hierarchy. For example, suppose that the analyst reuses the library system and an employee management subsystem of a particular specification to specify the car rental system, where the employee management subsystem contains the class 'Employee'. Then, the specification of the car rental system is depicted in Figs. 14 and 15, where the former figure shows the class sub-model and the latter shows the function sub-model.

6. Conclusions

Techniques for specification reuse, design document reuse, and program code reuse can be integrated to support a reuse-based software development paradigm, which can tremendously improve software development productivity and software quality. Object-oriented development further fosters software reuse.

In specification reuse, candidate reusable specifications are retrieved from a repository, where some are reusable and others not. The unreusable ones should be filtered out before an analyst reads the candidates to



```
sel [usr/chou]% LangInt

Specification to be executed: Library_system

The specification 'Library_system' has these subsystems:

    Book_management         Borrower_management

Select a subsystem to execute: Book_management

The subsystem 'Book_management' has these system functions:

    Borrow_a_book           Return_a_book
    Add_a_book              Remove_a_book
    Reserve_a_book

Select a system function to execute: Borrow_a_book

Key in input data:
    Book_identifier: 1001
    Borrower_identifier: 1001

Execution results:

Output data:

Object state changes:
        Book: <1001,"In_library"> -> <1001,"Borrowed">
        Borrower: <1001,0> -> <1001,1>

The subsystem 'Book_management' has these system functions:

    Borrow_a_book           Return_a_book
    Add_a_book              Remove_a_book
    Reserve_a_book

Select a system function to execute: []
```

Fig. 13. Execution of the library system specification.

understand them. This filtering can be accomplished by observing the candidates' behaviors, because specifications with behaviors dissimilar to that of the intended system cannot be reused and hence can be filtered out. A specification's behavior can be easily observed by executing the specification. To make specifications executable, an executable specification language is needed. This paper presents an executable specification language for specification behavior understanding in OO specification reuse. Since a specification's behavior can be understood by executing its system functions, this language is based on an OOA model that, in addition to classes, explicitly specifies system functions, which are further grouped into subsystems.

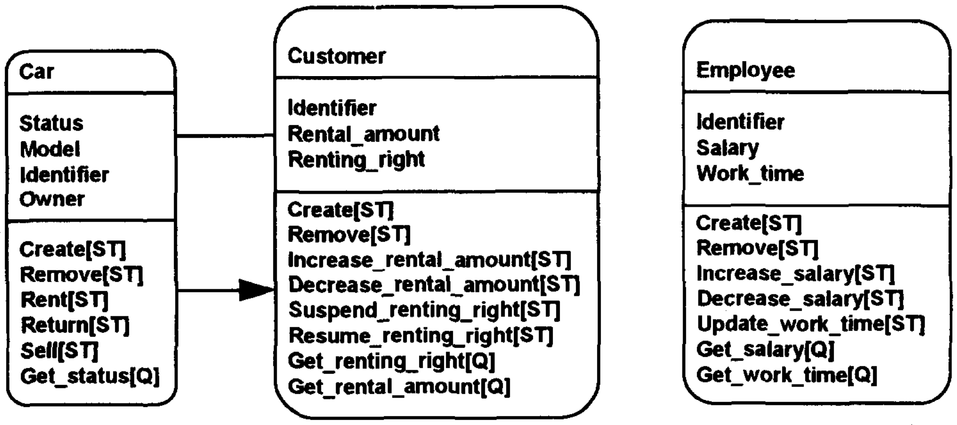


Fig. 14. Class sub-model for a car rental system.

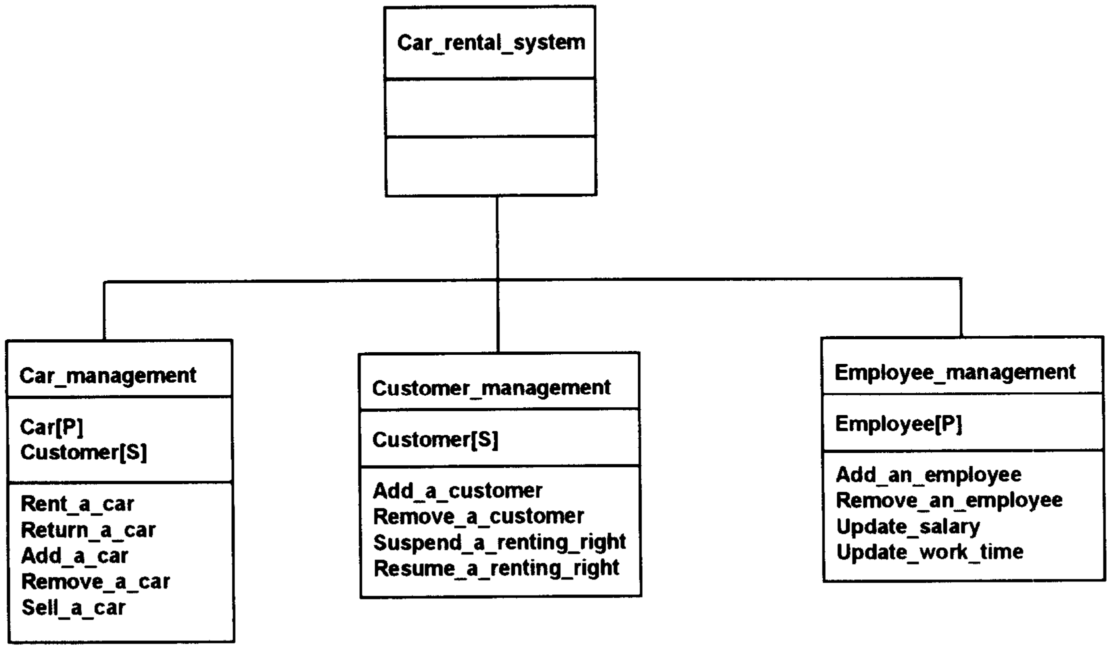


Fig. 15. Function sub-model for a car rental system.

This language offers the following features:

(1) **It facilitates specification behavior understanding.** This language provides executable statements for specifying system functions. A specification's behavior can thus be understood by executing its system functions. Since system functions will invoke class services during execution, this language also provide executable statements for specifying class services.
(2) **It facilitates specification modification.** To facilitate specification modification, this language hides as many classes as possible within subsystems so that they possess partial information hiding feature. In addition, this language provides statements for explicitly declaring the relationships among classes and for declaring the classes invoked by subsystems. This allows the language interpreter to construct dependency relationships both among classes and between system functions and classes. From the dependency relationships, one can identify the system functions and classes that should be modified accordingly when some classes are modified.
(3) **It facilitates specification composition.** Composing software components into software systems can be facilitated by clear component interfaces. Composing(sub)specifications is not an exception. This language thus provides statements for explicitly declaring the interfaces of class services and those of system functions to facilitate specification composition.

Acknowledgment

This research is supported by the National Science Council, ROC, under project number NSC81-0408-1009-542.

Appendix A

The simplified library system's specification written in the proposed specification language is listed below.

```
SUBSYSTEM Library_system
  SUBSYSTEM_DECLARATION
    DOMINATED_SUBSYSTEMS
    Book_management, Borrower_management;
  END_SUBSYSTEM_DECLARATION
END_SUBSYSTEM

SUBSYSTEM Book_management
  SUBSYSTEM_DECLARATION
    PRIVATE_CLASSES Book;
    SHARED_CLASSES Borrower;
    SYSTEM_FUNCTIONS {
      Borrow_a_book
      (Book_identifier[I]:INTEGER,
      Borrower_identifier[I]:INTEGER);
      Return_a_book
      (Book_identifier[I]:INTEGER,
      Borrower_identifier[I]:INTEGER);
      Reserve_a_book(Identifier[I]:INTEGER);
      Add_a_book(Identifier[I]:INTEGER,
      Status[I]:STRING, Title[I]:STRING,
      Author[I]:STRING);
      Remove_a_book(Identifier[I]:INTEGER);
    }
  END_SUBSYSTEM_DECLARATION
  SUBSYSTEM_BODY
    Borrow_a_book(Book_identifier,
    Borrower_identifier) {
      Borrower.Get_borrowed_amount
      (Borrower_identifier,amount);
      Borrower.Get_amount_limit
      (Borrower_identifier,limit);
      Borrower.Get_borrowing_right
      (Borrower_identifier,right);
      Book.Get_status(Book_identifier,status);
      IF((status = = "In_library") AND
      (amount < limit) AND (right = = 'Y'))THEN
        Book.Borrow(Book_identifier,
        Borrower_identifier);
        RETRIEVE book FROM Book
        WITH_KEY Book_identifier;
        RETRIEVE    borrower    FROM    Borrower
        WITH_KEY Borrower_identifier;
        book ASSOCIATION_OF borrower;
      ENDIF
    }

    Return_a_book(Book_identifier,
    Borrower_identifier) {
      Book. Return(Book_identifier,
      Borrower_identifier);
      RETRIEVE book FROM Book
      WITH_KEY Book_identifier;
      RETRIEVE    borrower    FROM    Borrower
      WITH_KEY Borrower_identifier;
      DISCONNECT book,borrower;
    }

    Reserve_a_book(Identifier) {
      Book.Reserve(Identifier);
    }

    Add_a_book(Identifier,Status,Title,Author) {
      Book.Create(Identifier,Status,Title,Author);
    }

    Remove_a_book(Identifier) {
      Book.Remove(Identifier);
    }
  END_SUBSYSTEM_BODY
END_SUBSYSTEM

SUBSYSTEM Borrower_management
  SUBSYSTEM_DECLARATION
    SHARED_CLASSSES Borrower;
    SYSTEM_FUNCTIONS {
      Add_a_borrower(Identifier[I]:INTEGER, Borro-
      wed_amount[I]:INTEGER,
        Borrowing_right[I]:CHAR,
        Amount_limit[I]:INTEGER);
      Remove_a_borrower
      (Identifier[I]:INTEGER);
      Suspend_a_borrowing_right
      (Borrower_identifier[I]:INTEGER);
      Resume_a_borrowing_right
      (Borrower_identifier[I]:INTEGER);
    }
  END_SUBSYSTEM_DECLARATION

  SUBSYSTEM_BODY
    Add_a_borrower
    (Identifier,Borrowed_amount,
    Borrowing_right,Amount_limit) {
      Borrower.Create(Identifier,
      Borrowed_amount, Borrowing_right,
      Amount_limit);
    }

    Remove_a_borrower(Identifier) {
      Borrower.Remove(Identifier);
    }

    Suspend_a_borrowing_right
    (Borrower_identifier) {
      Borrower.Suspend_borrowing_right
    (Borrower_identifier);
    }

    Resume_a_borrowing_right
    (Borrower_identifier) {
      Borrower.Resume_borrowing_right
    (Borrower_identifier);
    }
  END_SUBSYSTEM_BODY
END-SUBSYSTEM

CLASS Book
  CLASS_DECLARATION
    ASSOCIATED_CLASSES
    Borrower(0:1,0:10);
    ATTRIBUTES Identifier:INTEGER,
    Status:STRING,Title:STRING,
    Author:STRING;
    KEY_ATTRIBUTES Identifier;
    PROVIDED_STSERVICES {
      Create(Id:INTEGER,Sta:STRING,
      Tit:STRING,Aut:STRING);
      Remove(Id:INTEGER);
      Borrower(Book_id:INTEGER,
      Borrower_id:INTEGER);
      Return(Book_id:INTEGER,
      Borrower_id:INTEGER);
      Reserve(Id:INTEGER);
    }
    PROVIDED_QSERVICES {
      Get_status(Identifier[I]:INTEGER,
      Status[O];STRING);
    }
    REQUIRED_SERVICES {
      Borrower.Increase_borrowed_amount,
      Borrower.Decrease_borrowed_amount;
    }
  END_CLASS_DECLARATION

  CLASS_BODY
    Create(Id,Sta,Tit,Aut) {
      USED_ATTRIBUTES Identifier,Status,
      Title,Author;
      <>:
        NEW_STATE_IS⟨Id,Sta,Tit,Aut⟩;
    }
    Remove(Id) {
      USED_ATTRIBUTES Identifier;
      ⟨Id⟩:
        NEW_STATE_IS<>;
    }

    Borrow(Book_id,Borrower_id) {
      USED_ATTRIBUTES Identifier,Status;
      ⟨Book_id,Status⟩:
       Borrower.Increase_borrowed_amount
       (Borrower_id);
       NEW_STATE_IS⟨Book_id,"Borrowed"⟩;
    }

  Return(Book_id,Borrower_id) {
    USED_ATTRIBUTES Identifier,Status;
    ⟨Book_id,Status⟩:
     Borrower.Decrease_borrowed_amount
     (Borrower_id);
     NEW_STATE_IS⟨Book_id,"In_library"⟩;
    }

  Reserve(Id) {
    USED_ATTRIBUTES Identifier,Status;
    ⟨Id,Status⟩:NEW_STATE_IS
     ⟨Id,"Reserved"⟩;
    }
  END_CLASS_BODY
END_CLASS

CLASS Borrower
  CLASS_DECLARATION
    ASSOCIATED_CLASSES Book(0:10,0:1);
    ATTRIBUTES Identifier:INTEGER,
    Borrowed_amount:INTEGER,
        Borrowing_right:CHAR,
        Amount_limit:INTEGER;
    KEY_ATTRIBUTES Identifier;
    PROVIDED_STSERVICES {
      Create(Id:INTEGER,B_amount:INTEGER,
      B_right:CHAR,A_limit:INTEGER);
      Remove(Id:INTEGER);
      Increase_borrowed_amount
      (B_id:INTEGER);
      Decrease_borrowed_amount
      (B_id:INTEGER);
      Suspend_borrowing_right
      (B_id:INTEGER);
      Resume_borrowing_right
      (B_id:INTEGER);
    }
    PROVIDED_QSERVICES {
      Get_borrowed_amount
      (Identifier[I]:INTEGER,
      Borrowed_amount[O]:INTEGER);
      Get_borrowing_right
      (Identifier[I]:INTEGER,
      Borrowing_right[O]:CHAR);
      Get_amount_limit(Identifier[I]:INTEGER,
      Amount_limit[O];INTEGER);
    }
  END_CLASS_DECLARATION

  CLASS_BODY
    Create(Id,B_amount,B_right,A_limit) {
      USED_ATTRIBUTES Identifier,
      Borrowed_amount,Borrowing_right,
      Amount_limit;
      <>:
        NEW_STATE_IS
        ⟨Id,B_amount,B_right,A_limit⟩;
    }

    Remove(Id) {
      USED_ATTRIBUTES Identifier;
      ⟨Id⟩:
       NEW_STATE_IS<>;
    }

    Increase_borrowed_amount(B_id) {
      USED_ATTRIBUTES Identifier,
      Borrowed_amount;
      ⟨B_id,Borrowed_amount⟩:
       NEW_STATE_IS
       ⟨B_id,Borrowed_amount+1⟩;
    }

    Decrease_borrowed_amount(B_id) {
      USED_ATTRIBUTES Identifier,
      Borrowed_amount;
      ⟨B_id,Borrowed_amount⟩:
       NEW_STATE_IS
       ⟨B_id,Borrowed_amount-1⟩;
    }

    Suspend_borrowing_right(B_id) {
      USED_ATTRIBUTES Identifier,
      Borrowing_right;
```

```
        ⟨B_id,Borrowing_right⟩;
        NEW_STATE_IS⟨B_id,'N'⟩;
      }

      Resume_borrowing_right(B_id) {
        USED_ATTRIBUTES Identifier,
        Borrowing_right;
        ⟨B_id,Borrowing_right⟩:
        NEW_STATE_IS⟨B_id,'Y'⟩;
      }
    END_CLASS_BODY
  END_CLASS
```

References

[1] T. Biggerstaff and C. Richter, Reusability framework assessment, and directions, IEEE Software, 4 (March 1987) 41–49.
[2] V.R. Basili and H.D. Rombach, Support for comprehensive reuse, Software Eng. J., 6 (1991) 303–316.
[3] W.C. Lim, Effects of reuse on quality, productivity, and economics, IEEE Software, 11 (September 1994) 23–30.
[4] G. Fischer, S. Henninger and D. Redmiles, Cognitive tools for locating and comprehending software objects for reuse, in Proc. 13th Int. Conf. on Soft. Eng., 1991, pp. 318–328.
[5] B.A. Burton, R.W. Aragon, S.A. Bailey, K.D. Koehler and L.A. Mayes, The reusable software library, IEEE Software, 4 (July 1987) 25–33.
[6] M. Lenz, H.A. Schmid and P.F. Wolf, Software reuse through building blocks, IEEE Software, 4 (July 1987) 34–42.
[7] D.-J. Chen and P.J. Lee, On the study of software reuse using reusable C++ components, J. Syst. Software, 20 (1993) 19–36.
[8] B.M. Kennedy, Design for object-oriented reuse in the OATH library, J. Object-Oriented Progr., 5 (July/August 1992) 51–57.
[9] P. Johnson and C. Rees, Reusability through fine-grain inheritance, Software Pract. Experi., 22 (1992) 1049–1068.
[10] A. Podgurski and L. Pierce, Retrieving reusable software by sampling behavior, ACM Trans. Softw. Eng. Methodol., 2 (1993) 286–303.
[11] S. Henninger Using iterative refinement to find reusable software, IEEE Software, 11 (September 1994) 48–59.
[12] M.D. Lubars, The IDeA design environment, in Proc. 11th Int. Conf. on Soft. Eng., 1989, pp. 23–32.
[13] M.D. Lubars and M.T. Harandi, Knowledge-based software design using design schemas, in Proc. 9th Int. Conf. on Soft. Eng., 1987, pp. 253–262.
[14] S. Katz, C.A. Richter and K.-S. The, PARIS: a system for reusing partially interpreted schemas, in Proc. 9th Int. Conf. on Soft. Eng., 1987, pp. 377–385.
[15] G. Arango, E. Schoen and R. Pettengill, A process for consolidating and reusing design knowledge, in Proc. 15th Int. Conf. on Soft. Eng., 1993, pp. 231–242.
[16] D.J. Chen and D.T.K. Chen, An experimental study of using reusable software design frameworks to achieve software reuse, J. Object-Oriented Progr., 7 (1994) 56–67.
[17] S. Khajenoori, D.G. Linton and C.A. Morris, Enhancing software reusability through effective use of the essential modelling approach, Inf. and Soft. Technol., 36 (1994) 495–501.
[18] N. Maiden, Analogy as a paradigm for specification reuse, Software Eng. J., 6 (1991) 3–15.
[19] A. Finkelstein, Re-use of formatted requirements specifications, Software Eng. J., 3 (1988) 186–197.
[20] N.A.M. Maiden, Saving reuse from the noose: reuse of analogous specifications through human involvement in reuse process, Inf. and Soft. Technol., 33 (1991) 780–790.
[21] N.A. Maiden and A.G. Sutcliffe, Exploiting reusable specifications through analogy, Comm. ACM, 35 (1992) 55–64.
[22] G. Booch, Object Oriented Analysis and Design with Applications, 2nd edn., Benjamin/Cummings, 1994.
[23] B. Meyer, Object-Oriented Software Construction, Prentice-Hall, Englewood Cliffs, NJ, 1988.
[24] B. Meyer, Reusability: the case for object-oriented design, IEEE Software, 4 (March 1987) 50–64.
[25] J.A. Lewis, S.M. Henry, D.G. Kafura and R.S. Schulman, An empirical study of the object-oriented paradigm and software reuse, in Proc. OOPSAL'91, 1991, pp. 184–196.
[26] R. Helm and Y.S. Maarek, Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries, in Proc. OOPSAL'91, 1991, pp. 47–61.
[27] P. Coad, Object-oriented patterns, Comm. ACM, 35 (1992) 152–159.
[28] R.S. Pressman, Software Engineering: A Practitioner's Approach, 3rd edn, McGraw-Hill, 1992.
[29] R.P. Diaz and P. Freeman, Classifying software for reusability, IEEE Software, 4 (January 1987) 6–16.
[30] E. Ostertag, J. Hendler, R.P. Diaz and C. Braun, Computing similarity in a reuse library system: an AI-based approach, ACM Trans. Soft. Eng. Methodol., 1 (1992) 205–228.
[31] P. Devanbu, R.J. Brachman, P.G. Selfridge and B.W. Ballard, LaSSIE: a knowledge-based software information system, in Proc. 12th Int. Conf. on Soft. Eng., 1990, pp. 249–261.
[32] S.-C. Chou and J.-Y. Chen, A behavior-based classification and retrieval technique for object-oriented specification reuse, submitted to Software Pract. Exper. for publication.
[33] S.M. McMenamin and J.F. Palmer, Essential Systems Analysis, Yourdon Press, New York, 1984.
[34] E. Yourdon, Modern Structured Analysis, Prentice-Hall, Englewood Cliffs, NJ, 1989.
[35] P. Coad and E. Yourdon, Object-Oriented Analysis, 2nd edn, Prentice-Hall, Englewood Cliffs, NJ, 1991.
[36] R.G, Fichman and C.F. Kemerer, Object-oriented and conventional analysis and design methodologies, comparison and critique, IEEE Comput., 25 (1992) 22–39.
[37] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Comm. ACM, 15 (1972) 1053–1058.