Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 207 (2008) 17–32
www.elsevier.com/locate/entcs

Prototyping System Requirements Model

Xiaoshan Li

Faculty of Science and Technology, University of Macau, Macao SAR, China

Zhiming Liu

International Institute for Software Technology
United Nations University, Macao SAR, China

Abstract

In this paper, we present a tool for automatic prototype generation and analysis (AutoPA2.0) that implements the transformations from UML system requirements models to executable prototypes with the function of checking multiplicity invariants. A UML system requirements model consists of a *use-case model* and a *conceptual class model*. Generally, a use case is either described as a system operation which can be defined as a pair of pre and post conditions in the context of the conceptual class model, or described as a sequence of such system operations. AutoPA2.0 can transform the descriptions of use cases into an executable prototype in Java. The execution of each use case is a sequence of basic atomic actions which first check the pre-condition and then enforce the post-condition of the corresponding use case. It helps to improve the understanding between customers and designers. A simple library system is used to explain the method, and illustrate the feasibility of tool as well as its development.

Keywords: Prototype, System Requirements Model, UML.

1 Introduction

At the beginning of a software project, software engineers always find it difficult to capture the right requirements of the software system. The difficulty is due to the gap between customers and designers in their understanding of the system and its requirements. Prototyping is an efficient and effective way to closing this gap and validating the customers' requirements. The general purposes of building a prototype are now well understood, e.g. [18,11,4,19], and include

- to validate the requirements by demonstrating a prototype to the customers,

[1] Email: xsl@umac.mo, lzm@iist.unu.edu,
This work is partially supported by the HTTS project of Macao Science and Technology Development Fund, and 863 project 2006AA01Z165 and NSFC-60673114

doi:10.1016/j.entcs.2008.03.083

- to ensure the correct understanding of the requirements by that the designers and implementors,
- to cope with changing requirements better,
- to be used for testing.

Ideally, a prototype should cover two aspects of the system being developed: the requirements of the application and the architecture of the software. The Rational Unified Process (RUP) is now widely used in practical software projects. RUP is UML-based and use-case driven [7]. A use-case model represents the use cases and describe the important and critical functionalities and their relations. The prototype of an application generated by AutoPA2.0 demonstrates the executions of use cases in the use-case model of the application.

AutoPA2.0 for prototyping implements a transformation from a UML system requirements model to an executable prototype. It is based on modeling method presented in [9,13] that provides formal support to UML-based development. In [9,13], we define a *system requirements model* as a pair of a *conceptual class model* and a *use-case model*. The conceptual class model represents the domain concepts as *classes* and their relations as *associations*. Unlike a class in a design class model, a class in a conceptual class model does not have methods. The conceptual class model also has a predicate called the *state constraint* that specifies the allowed states of the system. This conceptual model determines the static structure of domain that is to be realised by a software structure. The use-case model describes the business processes and their dependency relations that are to be realized as computation processes in the software system. A number of objects associated in the class diagram are to be jointly involved in carrying out or realising a use case. The effect of a use case can be generally decomposed into a number of *atomic* or *primitive actions* which are creating a new object, updating the attribute of an existing object, creating a new link between two objects (i.e. an instance of an association), deleting/destroying an existing object, and removing an existing link [8]. A system requirements model is *consistent* if the conceptual class supports the realisation of all use cases in the use-case model and all actions in use cases preserve the state constraint of the conceptual model [13].

The conceptual class model and use-case model of a system requirements model are specified in UML2.0 and produced by MagicDraw9.5 and XMI1.2. The prototype tool consists of an *XMI parser*, a *code generator* and a graphical user interface. The *XMI parser* parses the *.xml* file of the conceptual class model and use-case model of a project. It transforms each UML metadata in the *.xml* file into the corresponding *Java* classes according to the transformation that we will define. From the file produced by the XMI parser, the code generator decomposes each use case declared with its pre and post conditions into a sequence of primitive actions, and then generates an executable source code in Java. The prototype can be executed for validating the use cases under the given conceptual class model and checking the consistency of the requirements model.

The rest of this paper is organized as follows. Section 2 defines the use-case model and conceptual class model of a project. We also define the notion of system state. Section 3 presents an algorithm that is used for transforming pre and post conditions into primitive actions. Section 4 focuses on the generation and the execution of the prototype. The prototype is written in Java. Finally, Section 5 discusses the conclusions and further work.

2 System Requirements Model

Requirements capture, analysis, validation and modelling are the main technical activities in the early stage of a software development project. For a cycle of the Rational Unified Process (RUP) [7], requirement analysis mainly involves the creation and analysis of use-case model and *conceptual class model* [8,13].

A system requirements model consists of a conceptual class model and a use-case model. The use-case model consists of a set of *use-case diagrams*. Each use case in a use-case diagram represents a functional service of the system that is to be used by a *specific actor* and satisfies a requirement specified in terms of a pair of pre and post conditions. The conceptual model is a class diagram that describes the application domain in terms of *classes* (also called concepts) and *associations* between these classes. A class represents a set of conceptual objects and an association determines how the *objects* in the associated classes are related. Classes may have *attributes* whose values determine the properties of the objects of the class.

2.1 Conceptual class model

A *conceptual class model* is a pair $CM = (\mathcal{D}, \mathcal{I})$, where $\mathcal{D}$ is a *class diagram* and $\mathcal{I}$ is a state constraint written as a predicate of attributes and associations [13]. The conceptual class diagram $\mathcal{D}$ identifies the classes and their associations. It defines the environment in which the use cases are to be operated. In our model, a conceptual class diagram $\mathcal{D}$ consists of following parts:

- **Class set:** We use $\mathcal{CN}$ to denote the set of classes in the diagram. We use bold capital letters to represent classes and types.
- **Attributes of classes:** for each class $\mathbf{C} \in \mathcal{CN}$, we use $Attr(\mathbf{C})$ to denote the set of the attributes of class $\mathbf{C}$ in the form of $\{\langle a_1 : \mathbf{T}_1\rangle, ..., \langle a_m : \mathbf{T}_m\rangle\}$, where $\mathbf{T}_i$ stands for the type of attribute a_i. The type of an attribute is always primitive [2], e.g. **String**, **Boolean**, and **Integer**. As we do not consider methods in a conceptual class model, we do not have distinguish private, protected and public attributes. We assume all attributes of a class can be inherited by its subclasses and the specification of a use case can refer to any attributes of the relevant classes.
- **Association set:** We use $\mathcal{AN}$ to represent the set of associations. Each association has a name and two roles which are the classes associated by the association

  $\mathbf{A} : \langle \mathbf{C}_1, \mathbf{C}_2 \rangle$

[2] Associations are used to model attributes whose types are classes in programming languages.

where $\mathbf{A}$ is the name of association and $\mathbf{C}_1, \mathbf{C}_2 \in \mathcal{CN}$ are the roles.

A role $\mathbf{C}_i$ of an association has a multiplicity which is a set of integers. We denote the multiplicities of the roles of $\mathbf{A} : \langle \mathbf{C}_1, \mathbf{C}_2 \rangle$ by $Multi(\mathbf{A} : \langle \mathbf{C}_1, \mathbf{C}_2 \rangle) = (M_1, M_2)$, where M_1 and M_2 are sets of integers. When M_1 (or M_2) forms an interval of integers from l to u, we use $l..u$ to denote this set; and when M_1 is a singleton $\{n\}$, we use n to denote the set. And here u and n would be *many* "*", which denotes the infinite.

2.2 States and state constraints

An object of a class has an *identity* and a state which assigns values to the attributes of the class of the object. Let $\mathcal{O}(\mathbf{C})$ denote the set of all possible objects of class $\mathbf{C}$. For each class $\mathbf{C}$ in the class diagram $\mathcal{D}$, we use the capital letter (not bold) C to represent the variable which records the current existing objects of class $\mathbf{C}$ in the system. The type of C is the powerset $\mathbb{P}(\mathcal{O}(\mathbf{C}))$. Let $CVar$ be the set of all class variables of a class diagram

$$CVar \stackrel{def}{=} \{C \mid \mathbf{C} \in \mathcal{CN}\}$$

Similarly, for an association $\mathbf{A} \in \mathcal{AN}$, we use A to denote the variable which records the current existing links between objects associated by $\mathbf{A}$, and let

$$AVar \stackrel{def}{=} \{A \mid \mathbf{A} : \langle \mathbf{C_1}, \mathbf{C}_2 \rangle \in \mathcal{AN}\}$$

The type of A is the powerset $\mathbb{P}(\mathcal{O}(\mathbf{C}_1) \times \mathcal{O}(\mathbf{C}_2))$. For a class diagram $\mathcal{D}$, a *state* or and *object diagram* S of $\mathcal{D}$ is a well-typed mapping from the variables $CVar \cup AVar$ to their object space such that for each association $\mathbf{A} : \langle \mathbf{C_1}, \mathbf{C}_2 \rangle$

$$A \subset C_1 \times C_2$$

meaning that existing links only link existing objects.

Also, for each $C \in CVar$ and each attribute $a \in Attr(\mathbf{C})$, $S[C].a$ is the attribute value of object $S[C]$. Therefore, S also maps the attribute variable $C.a$ to a value. Let Var be the set

$$Var \stackrel{def}{=} CVar \cup AVar \cup \{C.a \mid C \in CVar \wedge a \in Attr(\mathbf{C})\}$$

A *state constraint* is a predicate over Var whose truth value can be defined over the state space (the set of all object diagrams) of $\mathcal{D}$. The multiplicity invariants of an association A such that $Multi(\mathbf{A}) = (M_1, M_2)$ can be specified as a state constraint

$$\forall o \in C_1 \bullet |\{o_1 \mid \langle o, o_1 \rangle \in A\}| \in M_2 \wedge \forall o \in C_2 \bullet |\{o_1 \mid \langle o_1, o \rangle \in A\}| \in M_1$$

where $|\cdot|$ is the function that returns the number of elements of a set.

Library system example

The system provides the services for a library of a university. Librarians maintain a catalogue of publications which are available for lending to users. There may be many copies of the same publication. Publications and copies may be added to

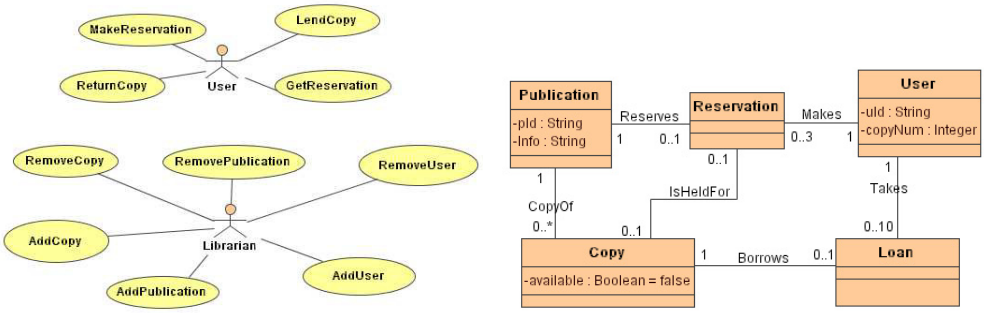

Fig. 1. Use case diagram and conceptual class diagram of the library system

and remove from the library by librarians. Librarians also handles registrations of users, they can add a new user and delete a registered user. When a copy has been borrowed by a user, it is on loan and becomes unavailable for lending to other users. A user can borrow no more than 10 copies, and can make a reservation on a publication when no copy is currently available. When a copy is held for a reservation, is can only be borrowed by the user who made the reservation. The reservation is deleted once the user of the reservation gets a copy of his reserved publication. A user cannot make more than one reservation on the same publication and cannot make more than 3 reservations totally. When a copy is returned, it will be held for a reservation if its publication is reserved and no copy is held for the reservation, otherwise it becomes available to users again and the corresponding loan is deleted. Figure 1 is the use case diagram and conceptual class diagram for the library system, respectively.

The conceptual class model for the library application can be defined as follows:

$$
\begin{aligned}
\mathcal{CN} &= \{\mathbf{Publication}, \mathbf{User}, \mathbf{Copy}, \mathbf{Reservation}, \mathbf{Loan}\} \\
Attr(\mathbf{User}) &= \{\langle uid : \mathbf{String}\rangle, \langle copyNum : \mathbf{Integer}\rangle\} \\
Attr(\mathbf{Copy}) &= \{\langle cid : \mathbf{String}\rangle, \langle available{:}\mathbf{Boolean}\rangle\} \\
\mathcal{AN} &= \{\mathbf{CopyOf}{:}\langle \mathbf{Copy},\mathbf{Publication}\rangle, \\
&\quad \mathbf{Borrows}{:}\langle \mathbf{Loan},\mathbf{Copy}\rangle,\ \mathbf{Takes}{:}\langle \mathbf{User},\mathbf{Loan}\rangle, \\
&\quad \mathbf{Makes}{:}\langle \mathbf{User},\mathbf{Reservation}\rangle, \\
&\quad \mathbf{Reserves}{:}\langle \mathbf{Reservation},\mathbf{Publication}\rangle, \\
&\quad \mathbf{IsHeldFor}{:}\langle \mathbf{Copy},\mathbf{Reservation}\rangle\}
\end{aligned}
$$

2.3 Use-case model

Informally, a use-case model consists of a use-case diagram and an textual description of each use case in the use-case diagram. Each use case provides services to one or more actors which can be any entity external to the system. Actors interact with the system by calling the *system operations* of a use case to request a service of the

system. The execution of a system operation in a use case is either the execution of a number of primitive actions that change the system state or an invocation of another use case.

For simplicity, we do not consider the case when more than one use case mutually invoke each other. The use-case model thus describes the overall functional requirements of the system, e.g., the use case diagram of library system is shown in Figure 1.

Now we use a *canonical form* to describe a use case as the following form [12]:

$$op \stackrel{def}{=} \textbf{pvar}\ x : \mathbf{T}_1; \textbf{rvar}\ y : \mathbf{T}_2; \quad \textbf{Pre} : p(v) \quad \textbf{Post} : R(v, v')$$

where **pvar** and **rvar** declare the input parameters and the result parameters. This specification is also used in the method of *design by contract* in [15,16].

The precondition $p(v)$ and postcondition $R(v, v')$ use variables v in $Var \cup x$ that are declared in the conceptual class diagrams, as well as the input parameters x to describe the pre-state of the operation and the primed version of $Var \cup y$ to describe the pos-state of the operation. Following the method given in [12], we use the model of a transition system [14] to combine the conceptual class model and the use-case model together to obtain a formal model of the system requirements. It is a tuple $(VAR, \mathcal{I}, \Theta, \mathcal{P})$, where

- VAR is the set of variables Var defined from the class diagram plus the input and output parameters.
- $\mathcal{P}$ is the set of atomic actions of the use cases. Each action specified with a precondition $p(v)$ and a postcondition $R(v, v')$ is defined as a *design* $p(v) \vdash \mathcal{R}(v, v')$ in originally proposed in Hoare and He's UTP [6], where
  · the *pre-condition* $p(v)$ must be true before the successful execution of the action.
  · the *post-condition* $\mathcal{R}(v, v')$ must be true after the execution of the action.
- Θ is a predicate over Var that defines the initial state of the system.
- $\mathcal{I}$ is a predicate over Var, called invariant. It is the state constraint in our model. It has to be true at initial state and preserved by each action in $\mathcal{P}$.

For example, use case *LendCopy* is about how the library can lend a copy of a publication to a user. Obviously, a user *user* and a copy *copy* are participants in this action, and a loan *loan* should be created for *user* and *copy*. This use case can be formally specified as follows:

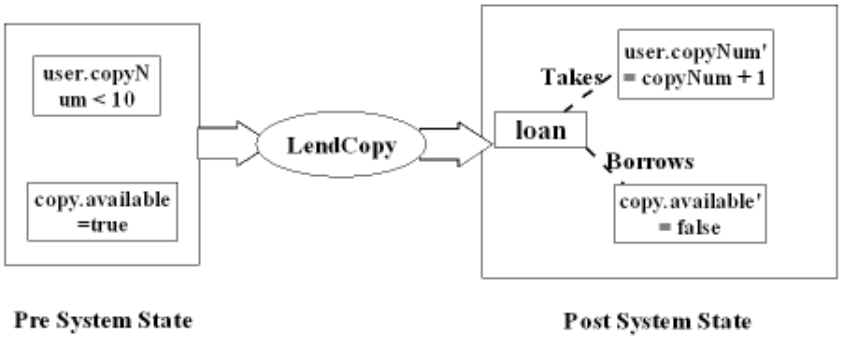

Fig. 2. State change by use case LendCopy

$$LendCopy \stackrel{def}{=} \textbf{pvar } copy : \textbf{Copy}, user : \textbf{User};$$

$$\textbf{Pre} \ : \ copy \in Copy \wedge user \in User$$
$$\wedge \ copy.available = true \wedge user.copyNum < 10$$

$$\textbf{Post} \ : \ \exists loan : \textbf{Loan} \cdot loan \notin Loan$$
$$\wedge \ Loan' = Loan \cup \{loan\}$$
$$\wedge \ Borrows' = Borrows \cup \{\langle loan, copy\rangle\}$$
$$\wedge \ Takes' = Takes \cup \{\langle user, loan\rangle\}$$
$$\wedge \ copy.available' = false$$
$$\wedge \ user.copyNum' = user.copyNum + 1$$

The precondition says that *copy* and *user* are known by the system, *copy* is available, and the total number of copies that user lends from the library are less than 10 at current pre system state. And the post condition asserts that first a new *loan* is created to record the loan of *copy* and *user*, i.e., two links are added to associations $Borrows$ and $Takes$, and *copy* becomes unavailable, and the attribute $copyNum$ of *user* will increase one at post system state. This can be shown informally in Figure 2.

We define eight primitive actions that will be used for implementing any system operations. They can be classified into five categories: *Create an Instance* including objects and links, *Delete an Instance*, *Update a Property* (or attribute) of an object, *Find an Instance*, and *Check a Property* of an object. The first three kinds of operations change the system state, while the last two kinds do not. These actions are described below in terms of the possible changes that they causes in the system state and the relation between the input and output parameters:

- *CreateObject*: Create a new object of a given class C and add it to the set C.
- *CreateLink*: Create an instance of a given association **A** and add it to the set A.

- *RemoveObject*: Remove a specified object of a given class **C** from C. Notice when this is done the links associating this object to other objects are all removed.
- *RemoveLink*: Remove a given link from an association A.
- *UpdateAttr*: Change an attribute of an object into a given value (or a non-side effect value expression).
- *FindObject*: Given an object identifier and a class name **C**, check whether an object with the identity exists. This action returns $true$ if the object is found and $false$ for otherwise. We can also find an object by an association which may contain the object. In general, we can find all objects in C that satisfy a provided property.
- *FindLink*: Check whether a link between two given objects is in a given association A. For example, $FindLink(CopyOf{:}\langle *, pub\rangle)$ is used to check whether there exists a link that contains the object *pub* in the association *CopyOf*. This action returns $true$ if there exists, and otherwise returns $false$.
- *CheckAttr*: Check whether a Boolean expression holds or not. The expression can only contain the attributes of existed objects and calculate without side effect, such as
  $user.copyNum < 10$.

3 Automatic Generation of Primitive Actions

As mentioned in the last section, we may formalize a use case by a pair of pre and post conditions. Then we transform the pre and post condition specification into a sequence of primitive actions like action state in an activity diagram. These primitive actions can be transformed into source code. In this section, we discuss how use case is represented in AutoPA2.0 in terms of its parameters and the pre and post conditions of a use case, and then give two generating algorithms to decompose them into primitive actions.

3.1 Decomposing use cases into primitive actions

Parameters of a use case is presented before the pre and post conditions. Each parameter is declared in the form **T** x, where **T** is the type of parameter x. Parameters are separated by semicolon ";". For example, the parameters of use case *LendCopy* are declared as **User** *user*; **Copy** *copy*.

It is known that the current system state is a state over $CVar \cup AVar$. For example, the execution of use case *LendCopy* changes a state that satisfies the precondition to a state that satisfies the postcondition (see Figure 2).

A use-case generally involves only a few objects and associations in the system, we can thus capture the objects and associations that participate in the use case. We can also identify the objects and associations which are created or deleted by a use case by observing pre system state and post system state of the use case. Therefore, the precondition of a use case can be checked by checking the existence

of some objects and links, non-existence of some objects and links, and properties of attributes of the existing objects. The postcondition of a use case that changes system states can be decomposed into the conjunction of creation of new objects, creation of new links, deleting existing objects, deleting existing links, and update values of attributes of existing objects. A query use cases only returns a truth value according to the existing objects and links as well as the values of the attributes of the existing objects. In summary, the truth value of pre and post conditions can be determined from the following four variables whose types are sets of instances. These variables will be used for implementing the decomposition of the pre and post conditions of a use case.

- *pre-objects*: is the set that records the objects which should exist in before the execution of the use case. When we declare value of *pre-objects*, we also specify the values of attributes of objects in this set. The elements of *pre-objects* can only appear as parameters of use cases.
- *post-objects*: is the set that is assumed to contain the objects which should exist after the execution of the use case. Values of attributes of objects should be declared when declaring *post-object*. The objects in *post-objects* can either be parameters of the use case or be an object newly created in the execution of an action.
- *pre-links*: is similar to *pre-objects*, but records the links that exist in the system state before the execution of the use case. The two objects associated be such a link can only be parameters of a use case.
- *post-links*: is similar to *post-objects*, but it records the links between two objects which will exist in the system state after the execution of the use case. The two linked objects can either be parameters of the use case or newly created by the execution of the action.

Consider use case *LendCopy* in the library application as an example. This use case has two parameters *user* and *copy*. The precondition of the use case states that the *user* and *copy* exist, *copy.available* is *true*, and *user.copyNum* is less than 10 in *pre-objects*. The post condition says that a new *loan* is created, *copy.available* becomes *false*, *user.copyNum* increases 1 in *post-objects*, and the links $Takes\ \langle user, loan\rangle$ and $Borrows\ \langle loan, copy\rangle$ are established in *post-links*:

```
Usecase LendCopy {
  User user; Copy copy;
  pre-objects:  User user, Copy copy,
                copy.available = true, user.copyNum < 10;
  post-objects: User user, Copy copy, Loan loan,
                copy.available = false,
                user.copyNum'= user.copyNum+1;
  pre-links:    ;
  post-links:   Takes <user,loan>, Borrow <loan, copy>; }
```

The specification of a use case in this form can be typed in the documentation window in MagicDraw interface based on the system analyst's understanding to the formal specification of use case. AutoPA2.0 can then generate a sequence of primitive actions with two parts from the pre and post conditions. The first part of the sequence consists of query actions for checking the precondition, and the

Table 1
Generating actions from a pair of pre and post conditions

Input: *pre-objects, post-objects,pre-links, post-links*

Output: *Vector<AtomicAction>actions*

Begin: $Vector\langle AtomicAction\rangle actions = new\ Vector();$

for $obj \in$ *pre-objects* **do**

$\{actions.add(FindObject(obj) = true)\};$

for $name : \langle obj1, obj2\rangle \in$ *pre-links* **do**

$\{actions.add(FindLink(name,\ obj1,\ obj2) = true)\}$;

for *boolean-expresion* $\in$ *pre-objects* **do**

$\{actions.add(CheckAttr(boolean\text{-}expresion))\}$;

for $obj \in (post\text{-}objects\ -\ pre\text{-}objects)$ **do**

$\{actions.add(CreateObject(obj))\ \}$;

for $name{:}\langle obj1, obj2\rangle \in (post\text{-}links\ -\ pre\text{-}links\ )$ **do**

$\{actions.add(CreateLink(name,obj1,obj2))\}$;

for $obj.attr = val\text{-}expresion \in post\text{-}objects$ **do**

$\{\ actions.add(UpdateAttr(obj,\ attr,\ val\text{-}expresion))\ \};$

for $name{:}\langle obj1, obj2\rangle \in (pre\text{-}links\ -\ post\text{-}links)$ **do**

$\{\ actions.add(RemoveLink(name,obj1,obj2))\ \}$;

for $obj \in (pre\text{-}objects\ -\ post\text{-}objects)$ **do**

$\{\ actions.add(RemoveObject(obj))\}$;

End

second part consists of the actions for enforcing the postcondition to be satisfied. We present the abstract algorithm for generating the sequence of actions in the Table 1.

3.2 Algorithm for generating primitive actions from pre and post conditions

A *pre-condition* of a use case can be decomposed and checked by the following three kinds of primitive actions:

- **Find object:** check whether each object *obj* in *pre-objects* with a given class as its type and an identifer exists in the pre system state.
- **Find link:** check whether a given link $name : \langle obj_1, obj_2\rangle$ in *pre-links* between two objects exists in the pre system state.

- **Check condition:** check whether a Boolean expression which contains the attributes of existing objects, holds at the pre system state.

We use the first three **for-do** loops of the algorithm in Table 1 to generate primitive actions for a pre-condition of a use case. Obviously, primitive actions for checking the pre-condition do not change the system state, also called query actions in [16].

A *post-condition* of a use case can be decomposed into subconditions [8], each can be satisfied by the following five kinds of primitive actions:

- **Create object:** A new object *obj* of a class **C** is created in system state, i.e. $obj \in (post\text{-}objects - pre\text{-}objects)$.
- **Create link:** A new link $\langle obj_1, obj_2 \rangle$ of an association **A** between two objects is created and added to the system state, i.e. $\langle obj_1, obj_2 \rangle \in (post\text{-}links - pre\text{-}links)$.
- **Update attribute:** An attribute of an existing objects *obj.attribute* in *post-objects* is updated in post system state.
- **Remove link:** An old link $\langle obj_1, obj_2 \rangle$ in *pre-links* was removed from the system state, because it is not in *post-links*.
- **Remove object:** An old object *obj* (in *pre-objects*) of a class **C** was removed from the system state, because it is not in *post-objects*.

We use the remaind part of algorithm with five **for-do** loops in Table 1 to generate actions from a given postcondition. The order of these actions should follow the order of first creation of objects and links, then update attributes, and finally remove links and objects.

The algorithm is used to generate the sequence of primitive actions for a use case when its pre and post conditions can be abstracted in terms of the four sets: *pre-objects, post-objects, pre-links*, and *post-links*. The sequence of primitive actions is equivalent to the execution of the use case. For example, the prototype of the library system can be generated automatically from AutoPA2.0 tool shown in Figure 3. The execution of *LendCopy* is simulated as a sequence of primitive actions in Figure 4. The code generator is discussed in the next section.

4 Prototype Generation

Recall that we use the variables $Var = CVar \cup AVar$ to represent the system state of the prototype. A system requirements model is a pair of a conceptual class model and a use-case model.

To generate a prototype of a system requirements model, we need to construct a corresponding *use case instance* or *execution* for each input of the parameters. During the execution of the use case with the input parameters, it interacts with its actors to perform a sequence of system operations as specified by the use case. A system operation is either a primitive action or an invocation of another use case. For the latter, the system operation will be instantiated as a use case instance which is further decomposed into a sequence of primitive actions. The prototype generated

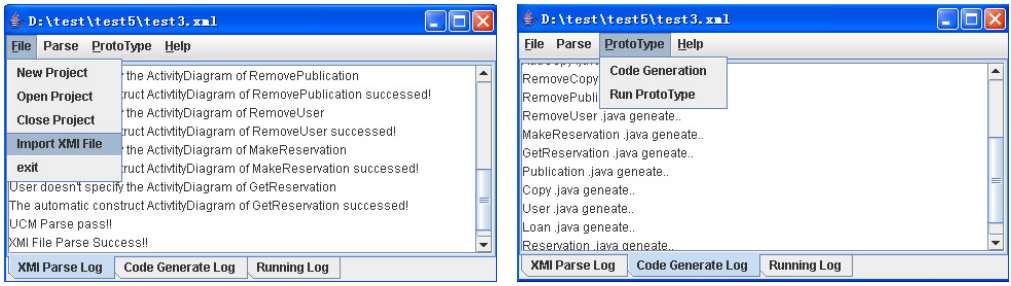

Fig. 3. XMI parser and code generator of AutoPA2.0

by AutoPA2.0 must demonstrate these interactions for each use case execution.

The conceptual class model and use-case model are visually drawn by MagicDraw9.5 and stored as a *.xml* file. The *.xml* conforms to UML2.0 and XMI1.2. *AutoPA2.0* takes the *.xml* file as the input and parses it. Therefore, all the information of the system requirements model can be obtained automatically, such as class and association name sets, the pre-post objects and links sets for some use cases, as well as action names and transition orders in activity diagrams for the other use cases. Based on the information, the code generator of AutoPA2.0 can generate the prototype source code. The interface of AutoPA2.0 for parser and code generator are shown in Figure 4. The main design ideas of AutoPA2.0 are described as the following subsections.

4.1 Generating declaration of prototype

From the parsed result of *.xmi* file of a given application, we can get the corresponding information of class set $\mathcal{CN}$, attributes of classes $Attr(\mathbf{C})$ for each $\mathbf{C}$, and $\mathcal{AN}$ as well as multiplicity invariants, which are all provided by the conceptual class model of the system.

- For each class $\mathbf{C}$ with attributes $Attr(\mathbf{C})$, AutoPA2.0 generates a corresponding conceptual class with the same class name and attributes in the prototype. The primitive actions of "CreateObject" and "RemoveObject" are corresponding to the object creation and destruction of the class in the prototype.
- For each class $\mathbf{C}$ and association $\mathbf{A}$, AutoPA2.0 also introduces the corresponding global variables $Cset$ and $Aset$ for recording the existing objects and links of class $\mathbf{C}$ and association $\mathbf{A}$, i.e., variables $Var = CVar \cup AVar$. However, $Cset$ just stores the identities of existing objects at the current system state. Similarly, $Aset$ only stores the pairs of associated two object identities. Here, the identities just the key words of objects which are unique for the searching reasons. The primitive actions of "FindObject", "Findlink", "CreateLink",and "RemoveLink" are equivalent to the corresponding operations on the variables.
- The global variables $Cset$ and $Aset$ are generally initialized to be empty sets. However, we can also design a initial function to AutoPA2.0 for directly setting a prototype system into any particular system state. This will be convenient for validating system different requirements by running the prototype.

- For each attribute a of class **C**, we also introduce two generic methods "geta()" and "seta(v)". They are useful for realizing the primitive actions of "CheckAttr" and "UpdateAttr".

From the conceptual class model, we can obtain the declaration part of the system prototype.

4.2 Generating use-case handlers

For each use case, AutoPA2.0 generates a use-case handler class [13]. Each use-case handler has a method with the same name and parameters as the use case. An invocation of a use case is equivalent to the invocation of the method of use-case handler. For example, for use case *LendCopy* in library system, there is a use-case handler *LendCopyHandler* with a method $lendcopy(String\ uid, String\ cid)$ in the library prototype.

As for the body of handler method, AutoPA2.0 uses the automatic generation algorithms in Section 3 to transform the specification of use case in four-set style into a sequence of primitive actions. The primitive actions can be easily realized as 8 global methods of main class in the prototype. For each kind of primitive action, we can define the corresponding global method. For example, "CreateObject" action can be defined as follows:

```
public void creatobject(ClassName, cid){ ...
  new ClassName(cid);
  ClassNameset = ClassNameSet union {cid}; ... }
```

After generating *Java source code*, we compile the code and run it. If necessary, we can modify the corresponding parts in the generated source code for some non-executable requirement specification than cannot be automatically generated source code by AutoPA2.0.

4.3 Interface of prototype

The prototype interfaces of the library system generated by AutoPA2.0 are shown in Figure 3. People can first choose an actor by clicking mouse on main window, and then click one of enabled use cases for the actor. Once a use case is clicked, its description window will pop out, and then people confirm for running the use case. A window with a sequence of buttons will pop out. And each button represents a primitive action or other basic use case. The *gray* button denotes for successful "executed", and *white* for "enabled" and *black* for "disabled". A path of going through the buttons demonstrates a corresponding instance execution (scenario) of the use case.

Through the prototype, people can easily understand requirements and also validate whether the interactions between actors and system are consistent with the description of the use case modelled by a pair of pre and post conditions or a sequence system operations.

AutoPA2.0 has an extending function for checking the multiplicity invariants

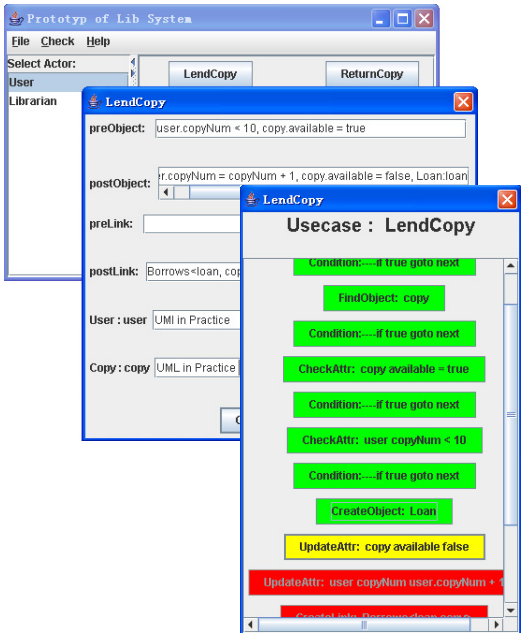

Fig. 4. Prototype interface of library system

on the system stable states. The function is implemented by checking the number of links in the variables of $AVar$. For example, the multiplicity of **Takes** from class **User** to class **Loan** in Figure 1 (any user cannot borrow more than 10 copies, $\forall u : \mathbf{User} \cdot u \in User \wedge |Takes(u)| \leq 10$), can be checked by calculating the number of links in the association variable $Takes$ for any user u in $User$. Because of the limitation of pages, here we omit the details on the algorithm of checking multiplicity invariants.

5 Conclusion and Discussion

Based on the formalization of UML requirements model [12,9,13], this paper extends the approach in [10] for generating a prototype automatically from a UML requirements model and implements it as a tool called AutoPA.

The key idea is to map the formal specification of a use case defined in pre and post conditions in the context of conceptual class model to a sequence of executable primitive actions on system state global variables Var.

As for some complex use cases, we need to draw their corresponding system sequence diagrams or activity diagrams. And it is enough to refine use cases into the system operations which can be captured by the four sets. After then, the system prototype can be generated automatically by using AutoPA2.0 tool. For example, there are total 10 use cases in the library system case study, which can be all handled by AutoPA2.0 without drawing system operation sequence diagrams. As for use case $MakeReservation$, the library should be in the state of no reservation on the publication *pub*, before a user can make a reservation for a publi-

cation. Therefore, there is a negative condition that is $\neg\exists r : \mathbf{Reservation} \cdot (r \in Reservation \wedge \langle r, pub\rangle \in Reserves)$ in the pre condition of $MakeReservation$. However, this r cannot appear in the sets *pre-objects* or *post-objects*. This case also happens similarly to use cases $ReturnCopy$ and $RemoveUser$. However, because we introduce attribute $reservationNum$ into class $Publication$ for denoting the number of links of association $Reserves$, like $copyNum$ in class $User$. Thus, it is possible to capture the use cases by the four sets just by adding $pub.reservationNum = 0$ and $user.copyNum = 0$ into their pre-conditions.

Based on the initial work in [10], this paper with AutoPA2.0 enhances the approach and improves the tool on the following aspects:

- AutoPA2.0 can handle a complex use case that has sequences of system operations represented by an activity diagram. A use case can also include other use cases [3] represented by the UML stereotype $\langle\langle include\rangle\rangle$.
- Complex "CheckAttr" and "UpdateAttr" actions are included for checking the conditions about attributes of pre-objects in precondition and updating the values of attributes of post-objects in postcondition respectively.
- AutoPA2.0 can be plugged in a *MagicDraw* CASE tool so that the a requirement model constructed with the CASE tool can be used to generate a prototype by AutoPA2.0.
- Compared to the earlier versions, the AutoPa2.0 have more visual features in its interface. The execution of use case can be demonstrated step by step.
- AutoPA2.0 can now check invariants that are imposed by multiplicities of associations in UML class diagrams.

AutoPA2.0 is founded on the formal semantic model developed in [12,9,13]. The model is based on the simple set theory and predicate logic. The popular formal method based prototyping tools, such as [17,5], generate prototypes from models of detailed designs, including UML Sequence Diagrams, State Diagrams, and Live Sequence Charts. The advantage of AutoPA2.0 is that it directly generates an prototype from an executable model of requirements.

With AutoPA2.0, the model of requirements is specified by the primitive operations on the four sets. This is generally applicable to information systems. Such a system usually has a large number of use cases with operations for managing a database of conceptual entities. This approach is applied to the use cases of *Online License Application* of the e-government government project (known as *eMacau*: http://www.egov.iist.unu.edu/). However, for more complicated software systems, we would need a high level formal language for specifying the model of requirements. We are working on the translation of the formal specification of requirements in rCOS [1,2] into the operations on the four sets in order to automate prototyping formal requirements specifications written in rCOS. This idea equally applies if the requirements specification is written OCL (Object Constraint Language). With this enhancement to AutoPA2.0, we hope to improve the practical applicability of the techniques of formal model driven development [2] with the support of *design by contract contract* [15,16].

Acknowledgement: We would like to thank the reviewers for their helpful comments. Thanks also go to Mr. Percy Loi, Yining Wei, and Jicong Liu for their hard work on implementing this prototype generator tool.

References

[1] X. Chen, J. He, Z. Liu and N. Zhan. *A Model of Component-Based Programming* . in Proc. FSEN07 LNCS4767, pp191-206, Springer, 2007.

[2] Z. Chen, Z. Liu, V. Stolz, A.P. Ravn and N. Zhan. *Refinement and Verification in Component-Based Model Driven Design*. Technical Report 388, UNU-IIST, P.O. Box 3058, Macau, http://www.iist.unu.edu, 2007.

[3] A. Cockburn. *Writing Effective Use Cases*. Person Education, 2001.

[4] J. Coplien. *A Development Process Generative Pattern Language*. AT&T, 1995.

[5] D. Harel and R. Marelly. *Come, Let's Play, Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[6] C.A.R. Hoare and J. He *Unifying theories of programming*. Prentice-Hall, 1998.

[7] I. Jacobson, G. Booch and J. Rumbaugh *The Unified Software Development Process* . Addison-Wesley, 1999.

[8] C. Larman. *Applying UML and Patterns (3rd ed.)*. Prentice-Hall, 2005.

[9] X. Li, Z. Liu, and J. He. *Formal and use-case driven requirement analysis in UML*. In Proc. COMPSAC01, pp215-224, IEEE Computer Society, 2001.

[10] X. Li, Z. Liu, J. He and Q. Long. *Generating a Prototype From a UML Model of System Requirement*. In Proc. ICDCIT 2004, LNCS 3347, pp255-265, Springer, 2004.

[11] H. Lichter, M.S chnerer-Hufschmidt, and H. Zullighoven. *Prototyping in Industrial Software Projects-Bridging the Gap between Theory and Practice*. IEEE Transcactions on Software Engineering, vol20, pp825-832, 1994.

[12] Z. Liu, X. Li, and J. He. *Using transition systems to unify uml Models*. In Proc. ICFEM2002, LNCS 2495, pp535-547, Springer, 2002.

[13] Z. Liu, J. He, X. Li and Y. Chen. *A relational Model for Formal Object-Oriented Requirement Analysis in UML*. In Proc. ICFEM, LNCS 2885, pp640-664, Springer, 2003.

[14] Z. Mana and A. Pnueli. *The temporal framework for concurrent programs* In R.S. Boyer, ed., The Correctness Problem in Computer Science, pp215-274. Academic Press, 1981.

[15] B. Meyer. *Object-oriented Software Construction(2nd Ed.)*. Prentice-Hall, 1997.

[16] R. Mitcheel and J. McKim. *Design by Conctract by Example*. Addison-Wesley, 2002.

[17] R. Plosch. *Contracts, Scenarios and Prototypes: An Integrated Approach to High Quality Software*. Springer, 2004.

[18] M.F. Smith. *Software Prototyping:Adoption,Practice and Management*. McGraw-Hill,1991.

[19] I. Sommerville. *Software Engineering (7th ed.)*. Addison-Wesley, 2004.