Innovations Syst Softw Eng (2008) 4:241–248
DOI 10.1007/s11334-008-0062-3

ORIGINAL PAPER

Validation of requirement models by automatic prototyping

Dan Li · Xiaoshan Li · Jicong Liu · Zhiming Liu

Received: 1 July 2008 / Accepted: 26 July 2008 / Published online: 12 August 2008

Abstract Prototyping is an efficient and effective way to understand and validate system requirements at the early stage of software development. In this paper, we present an approach for transforming UML system requirement models with OCL specifications into executable prototypes with the function of checking multiplicity and invariant constraints. Generally, a use case in UML can be described as a sequence of system operations. A system operation can be formally defined by a pair of preconditions and postconditions specified using OCL in the context of the conceptual class model. By analyzing the semantics of the preconditions and postconditions, the execution of the operation can be prototyped as a sequence of primitive actions which first check the precondition, and then enforce the postcondition by transferring the system from a pre-state to a post-state step by step. The primitive actions are basic manipulations of the system state (an object diagram), including *find* objects and links, *create* and *remove* objects and links, and *check* and *set* attribute values. Based on this approach, we have developed a tool of automatic prototype generation and analysis: *AutoPA3.0*.

Keywords Requirement validation · Prototype · Code generation · UML · OCL

D. Li · J. Liu · Z. Liu
The United Nations University,
International Institute for Software Technology,
P. O. Box 3058, Macau, China
e-mail: lidan@iist.unu.edu

J. Liu
e-mail: liujic@iist.unu.edu

Z. Liu
e-mail: lzm@iist.unu.edu

D. Li · X. Li (✉)
Faculty of Science and Technology,
University of Macau, Macau, China
e-mail: xsl@umac.mo

D. Li
Guizhou Academy of Sciences, 550001 Guiyang, China

1 Introduction

At the beginning of software development, capturing and defining the system requirements are always difficult tasks for software engineers. 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 close this gap and validate the customer's requirements. The general purposes of building a prototype are now well understood, e.g. [9,17,18]:

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

However, the development of a prototype for a system takes much effort and time; thus it is desirable if a prototype can be automatically generated from the requirements specification. It is well known that a model of requirements can hardly be entirely executable. It is thus a challenging and important problem to propose a way of modeling the requirements that allows us to explore the executable parts of a model for generating a prototype without the need for the detailed design.

In [6,10], we formally defined the UML model of requirements and proposed an approach to use case driven requirements modeling and analysis. Based on that work, in this paper we present a tool that we have developed for automatic prototype generation and analysis: *AutoPA3.0*. AutoPA3.0 can transform a UML model of requirements to an executable prototype. In [6,10], we define a *system requirements model* by 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 diagram together with annotation assertions (called comments informally in UML) imposes constraints on the allowable states of the system. This conceptual class model determines the static structure of domain that is to be realized 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 conceptual class diagram are to be jointly involved in carrying out or realizing a use case. The effect of a use case can be generally decomposed into a number of *primitive actions* which are finding object or link, checking attribute values, creating a new object or link, updating attribute values, and removing an existing object or link [5]. A system requirements model is *consistent* if the conceptual class diagram supports the realization of all use cases in the use case model and all executions of use cases preserve the state constraint of the conceptual class model [10].

The Object Constraint Language (OCL) [2] as a part of the UML 2.0 standard is a language allowing the specification of formal constraints on UML models. It is designed as a formal language to express side-effect-free constraints precisely within UML models. Thus, OCL is a natural choice for the specification of annotation assertions and preconditions and postconditions. OCL is based on mathematical set theory and predicate logic, and it has a formal mathematical semantics. Instead of using mathematical symbols, OCL uses plain ascii words that express the same concepts. OCL is also expressive enough for the specification of state constraints and preconditions and postconditions of use case operations.

The prototype generator takes as its inputs a conceptual class model in the form of a UML class diagram and use cases specified in OCL, and generates an executable program as the prototype of the system. In this paper, we focus on the discussion about how to transform use cases specified in OCL into a sequence of *primitive actions*. The key problem is how to analyze the operational semantics of OCL specifications, and then translate them into a well-ordered sequence of primitive actions. A set of templates and rules are presented for the algorithm of prototype generation. For a given OCL expression, it can be parsed to an abstract syntax tree (AST). Each fragment of the OCL expression, i.e., subexpression, corresponds to a sub-tree of the AST. Applying the given templates and rules to the fragments of OCL expressions, one then translates their implicit operational semantics to the corresponding primitive actions.

The UML/OCL model can be prepared by a general UML CASE tool, such as MagicDraw, and exported as an XMI2.1 file. The prototype tool first parses the XMI file into an instance of UML/OCL metamodel, and then transforms each use case into a sequence of primitive actions. Other necessary functions of the prototype tool, such as generating extra primitive actions by post processing, generating declarations of a prototype, generating use case handlers and implementing the GUI interface of a prototype, have already implemented in old versions' development of AutoPA and previous work [7,8]. Here we omit them due to the page limitation.

The rest of this paper is organized as follows. Section 2 briefly introduces the use case model and the conceptual class model of the system requirements model. And then Sect. 3 focuses on how to transform OCL expressions to primitive actions by the given four templates and ten rules. Section 4 presents the system prototype with AutoPA3.0 for the library case study. Some interesting ideas of tool implementation are briefly discussed in Sect. 5. Finally, Sect. 6 concludes the paper and discusses some further work.

2 System requirements model

A system requirements model consists of a *conceptual class model* and a *use case model* [10]. The conceptual class model is a class diagram that describes the application domain in terms of classes (also called concept classes, because no methods are contained in them) and associations between these classes. A class represents a set of conceptual objects and an association determines how the objects are related. Classes may have attributes whose values determine the properties of the objects of the class. The use case model includes a set of use case diagrams. Each use case in a use case diagram represents a required functional service that the system is expected to provide for certain kinds of users called actors. A use case can be defined as a sequence of system operations or included basic use cases by a system sequence diagram or an activity diagram. Such a system operation or an activity can be specified formally as a pair of preconditions and postconditions.

By decomposing the preconditions and postconditions of the system operations, a use case can be transformed into a sequence of *primitive actions*. Then these actions can be transformed into Java code. Thus, a prototype of the system requirements model can be used for demonstrating the execution of each use case as a sequence of primitive actions to the end-users for requirements validation.

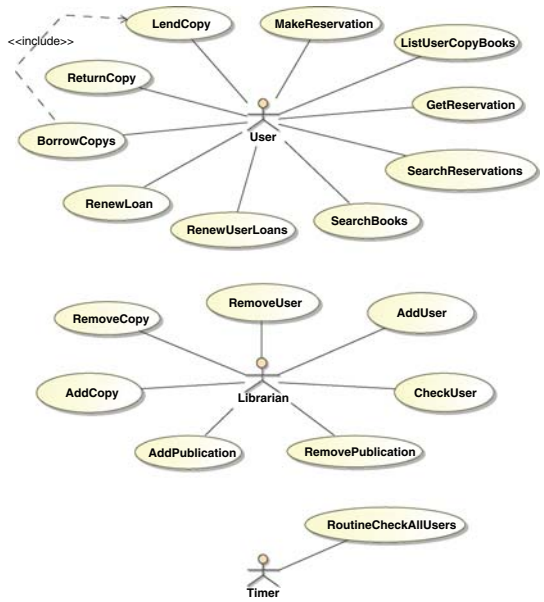


Fig. 1 Use case diagram of a library system

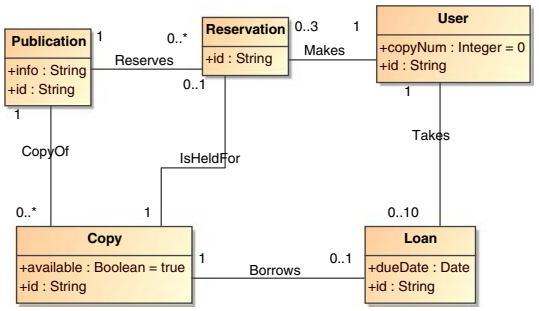


Fig. 2 Conceptual class diagram of a library system

Here we use a library system as a case study to illustrate the feasibility of our approach as well as the efficiency of tool AutoPA3.0. There are some quite complex use cases in the system, such as *MakeReservation*, *RenewUserLoans*, and *BorrowCopys*. The use case diagram and conceptual class diagram are shown in Figs. 1 and 2, respectively.

The following two sub-sections briefly introduce the conceptual class model and use case model formally using the formal method we proposed. For more details, please refer to [4,10,11]. The work makes us understand the formal semantics of UML models precisely, as well as the semantics of OCL. Therefore, it is easy for us to translate previous rCOS specification of the library system into OCL.

2.1 Conceptual class model

A *conceptual class model* is defined as $CM = (\mathcal{D}, \mathcal{I})$, where $\mathcal{D}$ is a class diagram and $\mathcal{I}$ is a state constraint written as a predicate of attributes and associations [10]. The conceptual class diagram $\mathcal{D}$ identifies the classes and their associations, and consists of following three parts:

- $\mathcal{CN}$: A set of classes in the diagram.
- *Attr*: Attributes of class. For each class $\mathbf{C} \in \mathcal{CN}$, $Attr(\mathbf{C})$ represents the set of the attributes of class $\mathbf{C}$ in the form of $\{\langle a_1 : \mathbf{T}_1\rangle, \ldots, \langle a_m : \mathbf{T}_m\rangle\}$, where $\mathbf{T}_i$ is the type of attribute a_i. The type of an attribute is always primitive, such as **String**, **Boolean**, and **Integer**. We assume that a use case can access any attribute of the relevant classes.
- $\mathcal{AN}$: A set of associations. An association takes the form of $\mathbf{A} : \langle \mathbf{C}_1, \mathbf{C}_2\rangle$, where $\mathbf{A}$ is the identifier of the association and $\mathbf{C}_1, \mathbf{C}_2 \in \mathcal{CN}$ are the roles which are the two classes associated by the association. A role $\mathbf{C}_i$ has a multiplicity which is denoted as a set of integers.

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

$$\text{CVar} \triangleq \{C \mid \mathbf{C} \in \mathcal{CN}\}$$

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

$$\text{AVar} \triangleq \{A \mid \mathbf{A} : \langle \mathbf{C_1}, \mathbf{C}_2\rangle \in \mathcal{AN}\}$$

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 $\mathbf{A} : \langle \mathbf{C_1}, \mathbf{C}_2\rangle$

$A \subset C_1 \times C_2$ means that existing links only link existing objects.

Also, for each $C \in$ CVar and each attribute $a \in \text{Attr}(\mathbf{C})$, $S[C].a$ 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

$$\text{Var} \triangleq \text{CVar} \cup \text{AVar} \cup \{C.a \mid C \in \text{CVar} \wedge a \in \text{Attr}(\mathbf{C})\}$$

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 $\text{Multi}(\mathbf{A}) = (M_1, M_2)$, where M_1 (or M_2) forms an interval of integers such as "$0 \ldots 1$", "$1 \ldots n$", etc., can be specified as a state constraint

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

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

For example, the conceptual class model of the library system shown in Fig. 2 can be partly defined as follows.

$\mathcal{CN}$ ={ **Publication, User, Copy, Reservation, Loan**}
Attr(**User**) = { ⟨ *id* : **String** ⟩, ⟨ *copyNum* : **Integer** ⟩ }
Attr(**Copy**) = { ⟨ *id* : **String** ⟩, ⟨ *available* : **Boolean** ⟩}
$\mathcal{AN}$ = { **CopyOf** : ⟨ **Copy**, **Publication** ⟩,
Borrows : ⟨ **Loan**, **Copy** ⟩,
Takes : ⟨ **Copy**, **Publication** ⟩,
IsHeldFor : ⟨ **Copy**, **Reservation** ⟩,
Reserves : ⟨ **Publication**, **Reservation** ⟩,
Makes : ⟨ **User**, **Reservation** ⟩ }

2.2 Use case model

Informally, a use case model consists of a use case diagram and a textual description which defines the behavior of each use case. Formally, a system operation of a use case can be defined as a *canonical form* [11]:

$op \triangleq$ **pvar** x : $\mathbf{T}_x$; **rvar** y : $\mathbf{T}_y$
Pre : $p(v)$ **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*. 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 post-state of the operation. The precondition $p(v)$ must be true before the successful execution of the operation and the postcondition $R(v, v')$ must be true after the execution of the operation. The preconditions and postconditions are described using a first-order logic-based formal notation.

We can define the specification of use case *AddCopy* (*copy*: **Copy**, *pub*: **Publication**) in rCOS [4] specification as follows, whose function is to add a new copy of a publication to the library.

AddCopy $\triangleq$ **pvar** *copy* : **Copy**, *pub* : **Publication**;
Pre : $copy \notin Copy \wedge pub \in Publication$
Post : $Copy' = Copy \cup \{copy\}$
$\wedge\ copy.available' = \texttt{true}$
$\wedge\ CopyOf' = CopyOf \cup \{\langle copy, pub\rangle\}$

And we can write the equivalent formal definition of use case *AddCopy* in OCL as follows.

context *Usecase*::*AddCopy*(*copy*: **Copy**, *pub*: **Publication**)
Pre: *Publication* -> includes(*pub*)
and *Copy* -> excludes(*copy*)
Post: *copy*.oclIsNew()
and *copy*.available = true
and *copy.CopyOf* = *pub*

3 Transform OCL expressions to primitive actions

OCL has proven to be a very versatile constraint language that can be used for different purposes in different domains [12]. In order to enhance the capacity and usability of our prototype AutoPA tool, OCL is used to specify the system *constraints* and *preconditions, postconditions* of system operations in UML requirement models. In this section, we focus on how to transform OCL expressions into *primitive actions*.

Based on Octopus [15], an Eclipse-based BSD-licensed tool supporting UML class models enriched with OCL 2.0, an OCL parser and semantic analyzer are used in our tool to build an internal representation, known as abstract syntax trees (ASTs), of OCL expressions. ASTs contain the concepts referring to both the syntax of expressions and its conceptual model.

For example, the precondition of use case *AddCopy* in Sect. 2.2 states that the object *pub* must be in the collection set of *publication* and *copy* must be a new one. Its corresponding AST is shown in Fig. 3. And the AST of the postcondition is in Fig. 4.

Transforming the OCL specification of a system operation into a sequence of primitive actions should start from the AST of the precondition first, and then to the AST of the postcondition. By analyzing the executable parts of OCL, we find that there are four kinds of templates (**T1**,…,**T4**, shown as follows) in their corresponding fragments of AST. The algorithm first tries to match each fragment of AST with the given four templates. And we present ten rules (**R1**,…,**R10**) for the ten primitive actions as follows. Templates and rules are derived from two aspects of OCL expression: syntax and semantics. If a rule can be applied to an AST fragment of OCL expression, then the primitive action can be obtained for the corresponding OCL sub-expression.

3.1 Find object action

Primitive action *FindObject(ObjId,Classifier)* is to check whether a given instance object with identifier *ObjId* of class

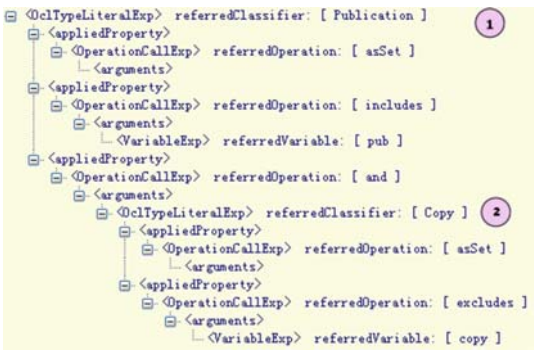


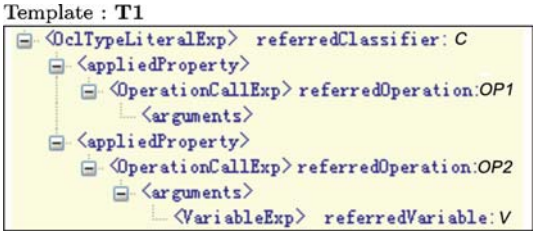
Fig. 3 Precondition of use case *AddCopy*

Fig. 4 Postcondition of use case *AddCopy*

Classifier exists in the current system state. If the object does not exist in the system, it means that the corresponding precondition will be false, and the prototype should go to exceptional handling.

For an AST fragment *fg* of OCL expression, *T(fg)* denotes its corresponding template, could be one of **T1**,…,**T4**. For example, if *T(fg)* = **T1** and the contents of the fragment parameters also satisfy the given conditions in rule **R1**, then the corresponding primitive action of the fragment *fg* is *FindObject(ObjId,Classifier)*.

Template : **T1**



$$T(fg) = \mathbf{T1} \wedge \texttt{OP1} \in \{asSet,\ asBag,\ asSequence\}$$

$$\mathbf{R1} \triangleq \frac{\wedge \texttt{OP2} \in \{includes,\ notEmpty,\ one\}}{FindObject(V, C)}$$

That is to say, **R1** can be used to generate the *FindObject(ObjId,Classifier)* action from the OCL expression like *Classifier-> includes(ObjId)*. Similarly, we can simply give the other nine rules as follows.

3.2 Find no-object action

Primitive action *FindNoObject*(*ObjId,Classifier*) checks whether a given object *ObjId* of class *Classifier* does not exist in the current system state.

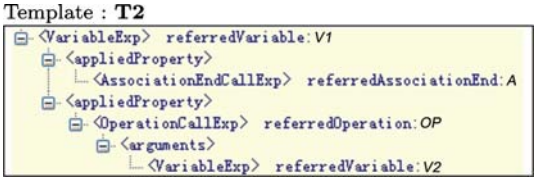
$$T(fg) = \mathbf{T1} \wedge \texttt{OP2} \in \{isEmpty, excludes\}$$

$$\mathbf{R2} \triangleq \frac{\wedge \texttt{OP1} \in \{asSet, asBag, asSequence\}}{FindNoObject(V,C)}$$

3.3 Find link action

Primitive action *FindLink(AsName, SObjId, TObjId)* is to check whether there exists a given link of the association *AsName* from object with identifier *SObjId* to object with identifier *TObjId*.

Template : **T2**



$$\mathbf{R3} \triangleq \frac{T(fg) = \mathbf{T2} \wedge \texttt{OP} \in \{=, includes, notEmpty\}}{FindLink(A,V1,V2)}$$

3.4 Find no-link action

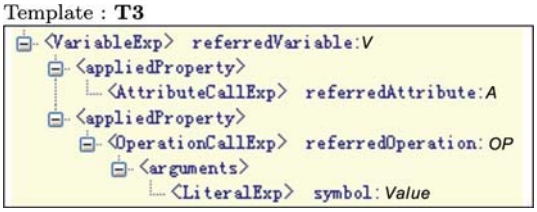
Primitive action *FindNoLink(AsName,SObjId,TObjId)* is to check whether a given link of the association *AsName* from object with identifier *SObjId* to object with identifier *TObjId* does not exist.

$$\mathbf{R4} \triangleq \frac{T(fg) = \mathbf{T2} \wedge \texttt{OP} \in \{isEmpty, excludes\}}{FindNoLink(A,V1,V2)}$$

3.5 Check attribute values action

Primitive action *CheckAttr(ObjId,AtName,Operator,Value)* is to check whether the value of attribute *AtName* of the object *ObjId* satisfies the relation with the given *operation* with the *value*

Template : **T3**

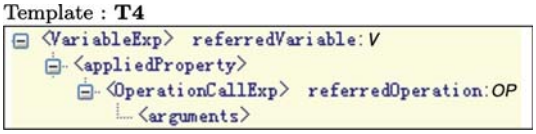


$$\mathbf{R5} \triangleq \frac{T(fg) = \mathbf{T3} \wedge \texttt{OP} \in \{=, >, <, >=, <=\}}{CheckAttr(V,A,OP,Value)}$$

3.6 Create object action

Primitive action *CreateObject(ObjId,Classifier)* is to create a new object of class *classifier* with the identifier *ObjId*.

Template : **T4**



$$\textbf{R6} \triangleq \frac{T(fg) = \textbf{T4} \wedge \texttt{OP} \in \{oclIsNew, including\}}{CreateObject(V,\ V.\text{Type})}$$

3.7 Create link action

Primitive action *CreateLink(AsName, SObjId, TObjId)* is to create a new link of association *AsName* between two objects with identifiers *SObjId* and *TObjId*.

$$\textbf{R7} \triangleq \frac{T(fg) = \textbf{T2} \wedge \texttt{OP} \in \{=, including\}}{CreateLink(A,V1,V2)}$$

3.8 Set attribute value action

Primitive action *SetAttr(ObjId,AtName,Value)* is equivalent to setting the attribute *AtName* with the given *Value*.

$$\textbf{R8} \triangleq \frac{T(fg) = \textbf{T3} \wedge \texttt{OP} \in \{=\}}{SetAttr(V,A,Value)}$$

3.9 Remove link action

Primitive action *RemoveLink(AsName,SObjId,TObjId)* is to remove the given link of association *AsName* between two given objects with identifiers *SObjId* and *TObjId*.

$$\textbf{R9} \triangleq \frac{T(fg) = \textbf{T2} \wedge \texttt{OP} \in \{isEmpty, excludes\}}{RemoveLink(A,V1,V2)}$$

3.10 Remove object action

Primitive action *RemoveObject(ObjId,Classifier)* is to remove a given object with identifier *ObjId* of class *Classifier*.

$$\textbf{R10} \triangleq \frac{T(fg) = \textbf{T1} \wedge \texttt{OP2} \in \{isEmpty, excludes\} \quad \wedge \texttt{OP1} \in \{asSet, asBag, asSequence\}}{RemoveObject(V,C)}$$

3.11 Example of use case *AddCopy*

Let us consider the OCL specification of use case *AddCopy* in Sect. 2. After parsing and semantic analysis, the corresponding ASTs of its preconditions and postconditions are shown in Fig. 3 and Fig. 4, respectively.

For the AST of precondition in Fig. 3, the generator first starts from the root node ①, and finds the template **T1** matched. The primitive action *FindObject(pub,"Publication)* is first generated by applying rule **R1**. And then rule **R2** can apply to the next node ②, *FindNoObject(copy,"Copy")* can be obtained.

Similarly, for the AST of the postcondition in Fig. 4, the generator starts from node ① to nodes ② and ③. The rules **R6**, **R8** and **R7** can be applied, respectively. Three corresponding primitive actions: *CreatObject*, *SetAttr*, and *CreatLink* can be generated. Therefore, for use case *AddCopy*, its corresponding sequence of *primitive actions* can be generated as follows:

1. *FindObject(pub,Publication)*;
2. *FindNoObject(copy,Copy)*;
3. *CreateObject(copy,Copy)*;
4. *SetAttr(copy,available,true)*;
5. *CreateLink(CopyOf,copy,pub)*;

4 System prototype with AutoPA3.0

A system prototype should demonstrate the interactions between actors and system for each use case. A system operation is formalized by a pair of preconditions and postconditions. And each use case is defined as a sequence of system operations. From the semantics of the use case, we can transform the specification of preconditions and postconditions into a sequence of *primitive actions*. Then, the execution of the sequence of primitive actions is equivalent to the execution of the use case [11].

An algorithm was developed in AutoPA 2.0 to generate the sequence of primitive actions for a use case [7]. The behavior of checking a precondition can be translated into a sequence of primitive actions, which belong to the following five kinds of query actions on system state: *FindObject*, *FindNoObject*, *FindLink*, *FindNoLink* and *CheckAttr*. These five kinds of actions are side-effect-free queries that don't change the system state. Similarly, the postcondition can be translated into a sequence of primitive actions, which belong to the following five kinds of actions: *CreatObject*, *CreatLink*, *SetAttr*, *RemoveLink*, *RemoveObject*. However, they are not side-effect-free primitive actions, whose execution will change system states. Therefore, with the additional three more steps: generating declaration from *conceptual class model*, generating use case handlers and generating GUI interface, a prototype of the system requirements model can be generated automatically in Java.

For the case study of the library system, the prototype interface is shown in Fig. 5, and the interface and prototyping execution of use case *AddCopy* are shown in Fig. 6 and Fig. 7. In addition, the prototype also has the functionality of checking multiplicities and invariants on system states automatically, shown in Fig. 7.

5 Tool implementation

As an extension of AutoPA2.0, we have improved the tool by adapting the techniques of the Octopus tool [15] for translating from OCL expressions into *primitive actions* of the prototype.

Our implementation follows the typical structure of a translator, consisting of four processes: lexical analysis, parsing, semantic analysis and translation.

The lexical analyzer and parser generate an AST from the input XMI file generated by general UML CASE tool, such as *MagicDraw*; syntactic errors are reported by these processes. The next process, semantic analysis, requires the input of an AST and the conceptual class model to which OCL expressions are attached. Semantic analysis will find static semantic errors. Finally, we provide an algorithm for transforming the result of semantic analysis into the primitive actions.

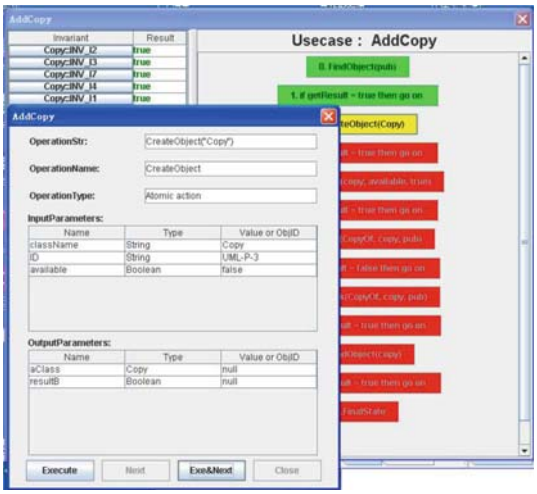


Fig. 7 Prototyping execution of use case *AddCopy*

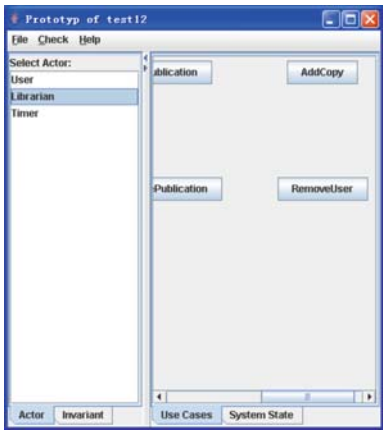


Fig. 5 Prototype interface of library system

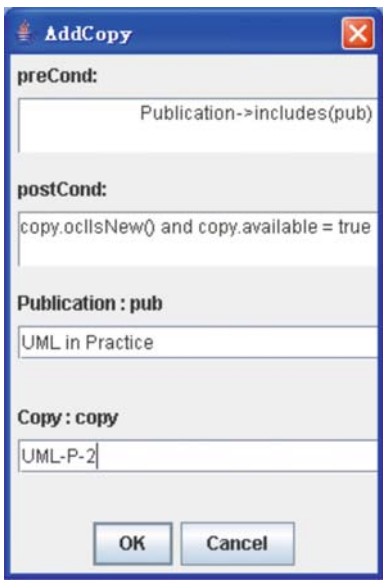


Fig. 6 Interface of use case *AddCopy*

Based on AutoPA2.0, we mainly designed an OCL interpreter in the new version, and plugged it into AutoPA2.0. And we also improved the functionality for dealing with complex use cases with iteration and choice decision control as well as the constraints checking.

6 Conclusion and future work

In this paper, we mainly discussed how to analyze and translate OCL expressions to the primitive actions. With our tool support, all use cases of the library system can be generated to the executable prototype automatically. Therefore, validation of requirement models by automatic prototyping becomes possible. And the tool also provides the functionality for checking the system multiplicities and invariants, which are specified OCL.

The main contribution of AutoPA3.0 compared with AutoPA2.0 [7], is that OCL is introduced for specifying preconditions and postconditions of system operations as well as constraints, rather than using *four sets*: (*preObjects*, *preLinks*, *postObjects*, and *postLinks*), which had to provided to AutoPA2.0 tool by translating from the formal specification manually. Thus, AutoPA3.0 becomes more automatic and makes contract-based design [13,14] even more applicable. Actually the expressive ability of OCL is also larger than the expressiveness of *Four Sets* in AutoPA2.0. In addition, for complex use cases, AutoPA3.0 becomes more powerful than before because the control information among primitive actions can be obtained from their system sequence diagrams or activity diagrams.

Compared with other prototype tools, the advantage of AutoPA is that it can transfer the executable parts of require-

ments model directly, rather than using the design models like design sequence diagrams and state diagrams, or live sequence charts in [16,3]. Tool USE [19] with many papers [1] mainly focuses on validating UML models with OCL constraints by testing and animation on the given system states (object diagrams). However, AutoPA is mainly designed for generating prototypes automatically, although some functions like tool USE can also be provided by our tool, such as checking constraints on the given system states.

At present some OCL expressions with constructs like *TupleLiteralExp* and *MessageExp*, can not be handled in our current tool. Actually, lots of use cases of information database systems are executable, because their behaviors can be decomposed or defined by the primitive actions. However, the implicit style equation about several attributes like $obj1.x'^4 + obj2.y'^4 = obj3.z'^4$ is obviously not executable. The complex algorithms of special problems like *tax calculation* should be provided explicitly in system design models. For some non-executable parts, software designers can develop manually, and then integrate them with the generated prototype from the executable parts of the system.

Our future work is to improve the generator algorithm to cover even larger executable set of OCL, mainly for post-conditions. Since the formal syntax of the rCOS specification language [4] is now issued, we will extend our work to support rCOS and system automated testing. Meanwhile, we will apply the tool to more case studies as well as practical systems. Afterwards, tool AutoPA will be put onto the Internet for the public.

Acknowledgments Many thanks to the referees and Mr. Chris George for their valuable comments. This work is supported by the projects of HighQSoftD and HTTS of Macao Science and Technology Development Fund, and NSF of China No. 90718014.

References

1. Gogolla M, Büttner F, Richters M (2007) Use: a uml-based specification environment for validating uml and ocl. Sci. Comput. Program 69(1-3):27–34
2. Object Management Group Object constraint language specification, http://www.omg.org/docs/ptc/03-10-14.pdf
3. Harel D, Marelly R (2003) Come, let's play, scenario-based programming using LSCs and the play-engine. Springer, Heidelberg
4. He J, Li X, Liu Z (2006) rcos: a refinement calculus of object systems. Theor. Comput. Sci 365(1–2):109–142
5. Larman C (2001) Applying UML and patterns. Prentice-Hall International, Englewood Cliffs
6. Li X, Liu Z, He J (2001) Formal and use-case driven requirement analysis in UML. In: COMPSAC01. IEEE Computer Society, Illinois, pp 215–224
7. Li X, Liu Z (2008) Prototyping system requirements model. Electr. Notes Theor. Comput. Sci 207:17–32
8. Li X, Liu Z, He J, Long Q (2004) Generating a prototype from a UML model of system requirements. In: Ghosh RK, Mohanty H (eds) ICDCIT, LNCS 3347. Springer, Heidelberg, pp 255–265
9. Lichter H, Schneider-Hufschmidt M, Zullighoven H (1994) Prototyping in industrial software projects—bridging the gap between theory and practice. IEEE Trans Softw Eng 20:825–832
10. Liu Z, He J, Li X, Chen Y (2003) A relational model for formal object-oriented requirement analysis in uml. In: Dong JS, Woodcock J (eds) ICFEM, LNCS 2885. Springer, Heidelberg, pp 641–664
11. Liu Z, Li X, He J (2002) Using transition systems to unify uml models. In: George C, Miao H (eds) ICFEM, LNCS 2495. Springer, Heidelberg, pp 535–547
12. Markovic S, Baar T (2006) An ocl semantics specified with qvt. In: Nierstrasz O (ed) Models, LNCS 4199. Springer, Heidelberg, pp 661–675
13. Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice Hall PTR, Englewood Cliffs
14. Mitchell R, McKim J (2002) Design by conctract by example. Addison-Wesley, Reading
15. Objecten K Octopus: Ocl tool for precise uml specifications. http://octopus.sourceforge.net/
16. Plosch R (2004) Contracts, scenarios and prototypes: an integrated approach to high quality software. Springer, Heidelberg
17. Smith MF (1991) Software prototyping: adoption, pratice and management. McGraw-Hill, New York
18. Sommerville I (2004) Software engineering, 7th edn. Addison-Wesley, Reading
19. USE A uml-based specification environment. http://www.db.informatik.uni-bremen.de/projects/use/