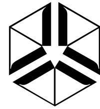


The United Nations
University

UNU-IIST

**International Institute for
Software Technology**

Automatic Transformation from Requirements Models to Executable Prototypes

Yining Wei, Xiaoshan Li, Zhiming Liu, Jifeng He

Oct, 2005

UNU-IIST Report No. 329

R

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

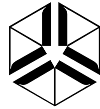
Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macau or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director



The United Nations
University

UNU-IIST

**International Institute for
Software Technology**

P.O. Box 3058
Macau

Automatic Transformation from Requirements Models to Executable Prototypes

Yining Wei, Xiaoshan Li, Zhiming Liu, Jifeng He

Abstract

As an joint effort between UNU-IIST and University of Macau, we are developing a tool for automatic prototype generation and analysis (AutoPA). In this paper, we present the initial version AutoPA1.0 that implements the transformations from UML system requirement models to executable prototypes. An UML system requirement consists of a *use-case model* and a *conceptual class model*. Each use case is either described as a pair of pre and post conditions in the context of the conceptual model or represented as an activity diagram drawn in MagicDraw9.5 by the user. AutoPA can transform an activity diagram into the Java code of the prototype for execution. For a use case specified in terms of its pre and post conditions, AuoPA1.0 first transforms the specification into a sequence of atomic actions to generate a corresponding activity diagram. The prototype of requirements model can be used to validate the requirements by checking the pre and post conditions of the use case operations and the system invariants. It helps to improve the understanding between customers and designers. We will use an example of a library system to illustrate the tool and its development.

Keywords: Prototype, System Requirements Model, Class Diagrams, Use-Case Model, UML

Yining Wei is a fellow at UNU/IIST. He is Master Candidate at Department of computer science, East China Normal University , Shanghai, China. His research interest are Object-Oriented analysis and design, Software Engineering, formal method, Requirement Engineering, formal method. E-mail: wyn@iist.unu.edu

Xiaoshan Li is an Associate Professor at the University of Macau. His research areas are interval temporal logic, formal specification and simulation of computer system, formal methods in system design and implementation. E-mail: xsl@umac.mo.

Zhiming Liu is a research fellow at UNU/IIST. His research interests include theory of computing system, including sound methods for specification, verification and refinement of fault-tolerant, real-time and concurrent systems, and formal techniques for OO development. His teaching interests are Communication and Concurrency, Concurrent and Distributed Programming, Internet Security, Software Engineering. Formal specification and Design of Computer Systems. E-mail: Z.Liu@iist.unu.edu

Jifeng He is a senior research fellow of UNU/IIST. He is also a professor of computer science at East China Normal University and Shanghai Jiao Tong University. His research interests include the Mathematical theory of programming and refined methods, design techniques for the mixed software and hardware systems. E-mail: hjf@iist.unu.edu.

Copyright © 2005 by UNU-IIST, Yining Wei, Xiaoshan Li, Zhiming Liu, Jifeng He

Contents

1 Introduction **1**

2 Architecture of AutoPA1.0 **2**

3 System Requirements Model **4**
3.1 Conceptual class model 5
3.2 States and state constraints 6
3.3 Use-case model 7
3.4 Activity diagrams of use cases 10

4 Automatic Generation of Primitive Actions **11**
4.1 Decomposing use cases into primitive actions 13
4.2 The algorithm for generating primitive actions from a precondition 14
4.3 Algorithm for generating primitive actions from a postcondition 15

5 Prototype Generation **16**
5.1 Generating declaration of prototype 19
5.2 Generating use-case handlers 19
5.3 Interface of prototype 20

6 Conclusion and Discussion **20**

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. [2, 3, 4, 5], and include

- 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,

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 [9]. 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 AutoPA1.0 demonstrates the executions of use cases in the use-case model of the application.

AutoPA1.0 for prototyping implements a transformation from a UML system requirements model to an executable prototype. It is based on a sound method [13, 14] that provides formal support to UML-based development. In [13, 14], 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*. It is represented as a UML class diagram in which no methods are designed in the classes. 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 [10]. A system requirements model is *consistent* if the conceptual class supports the realisation of all the use cases in the use-case model and all actions in the use cases preserves the state constraint of the conceptual model [14].

The conceptual class model and use-case model of a system requirements model are specified in UML1.4 [7] and produced by MagicDraw 9.5 and XMI1.2 [8]. 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

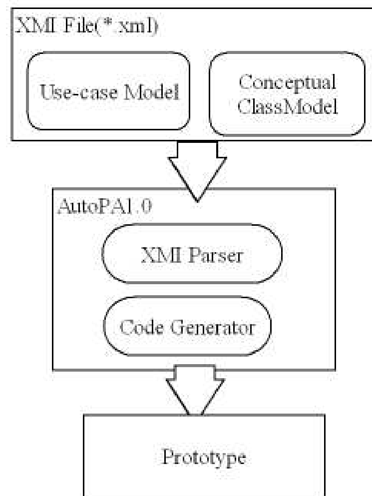


Figure 1: Architecture of AutoPA1.0

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, that is the *prototype of the project*. For the use cases with activity diagrams, the code generator can generate the source code directly from the primitive actions or included basic use cases in the activity diagram. After compiling the source code, the prototype can be executed for validating the use cases under the given conceptual class model and checking the consistency of the requirements model (See Figure 1).

The rest of this paper is organized as follows. Section 2 introduce the architecture of AutoPA1.0. Section 3 defines the use-case model and conceptual class model of a project. We also define the notion of system state. Section 4 presents two algorithms for that are used respectively for transforming pre and post conditions into primitive actions. Section 5 focuses on the generation and the execution of the prototype. The prototype is written in Java. Finally, Section 6 discusses the conclusions and further work.

2 Architecture of AutoPA1.0

AutoPA1.0 consists of two main parts. One is *XMI Parser*, the other is *Code Generator*. The use-case diagram of AutoPA1.0 is described in Figure2.

As shown in the use-case diagram, *XMI Parse* is used to parse the xml file generated by MagicDraw that conforms to UML1.4 and XMI1.2, it consists of *Actor Parse, Use-case Parse, Class Parse and Association Parse*:

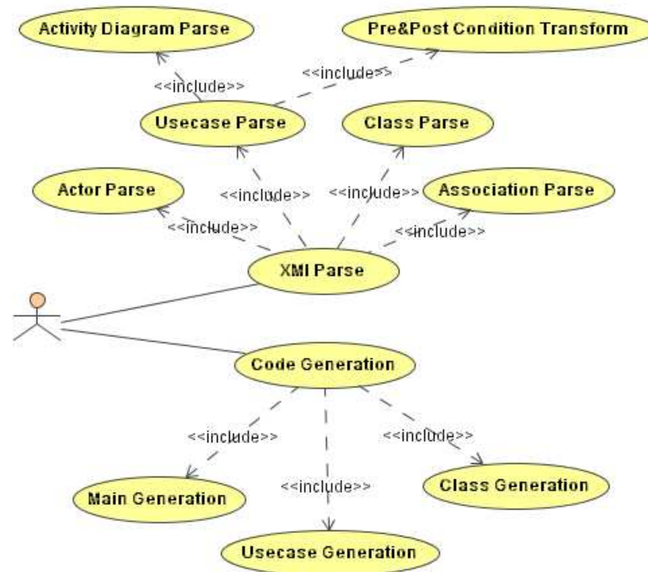


Figure 2: Use-case Diagram of AutoPA1.0

- Actor Parse is used to parse actors in the use-case diagram.
- Use-case Parse is used to parse use-cases in the use-case diagram. We can see that Use-case Parse include *Activity Diagram Parse* and *Pre&Post Condition Transform*. We use *Activity Diagram Parse* to parse the activity diagram in a use-case, and use *Pre&Post Condition Transform* to transform the pre and post condition of a use-case into a set of actions that form an activity diagram.
- Class Parse is used to parse classes in the class diagram. We will get names, types, and initial value of all the attributes of each class in the class diagram by the use-case.
- Association Parse is used to parse the associations between classes in class diagram, or actors and use-cases in use-case diagram.

Code Generation is used to generate the Java source files of a parsed project. It consists of Main Generation, Use-case Generation and Class Generation:

- Main Generation is used to Generate *Main.java* which is the entrance and the initial part of program.
- Use-case Generation is used to generate the Java class files for each use-case in use-case diagram. The generated class file will have method for executing the use-case.
- Class Generation is used to parse classes in the class diagram. Each generated class file will have the attributes corresponding to attributes described in the class diagram.

We design the class diagrams to implement AutoPA1.0, as shown in Figure 3,4,5.

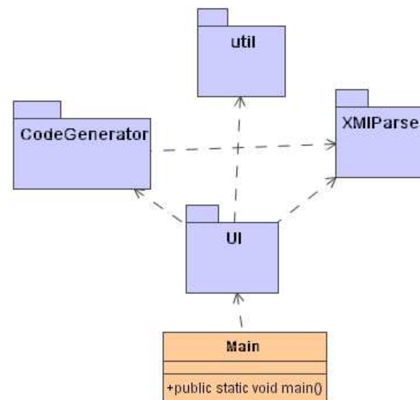


Figure 3: Class Diagram of AutoPA1.0

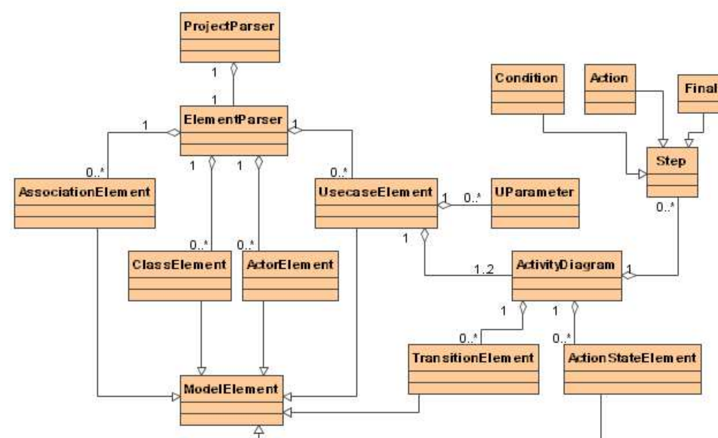


Figure 4: Class Diagram in package XMI Parser of AutoPA1.0

3 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) [9], requirement analysis mainly involves the creation and analysis of use-case model and *conceptual class model* [10, 18, 14].

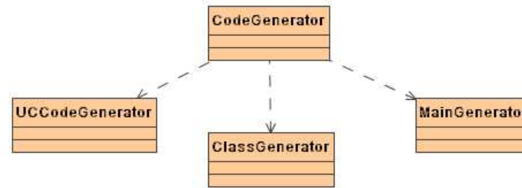


Figure 5: Class Diagram in package Code Generator of AutoPA1.0

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.

3.1 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 [14]. 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 finite 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 $\{< a_1 : \mathbf{T}_1 >, ..., < a_m : \mathbf{T}_m >\}$, where $\mathbf{T}_i$ stands for the type of attribute a_i. The type of an attribute is always primitive[1], e.g. **String**, **Boolean**, and **Integer**. As we do consider methods in a conceptual class model, we do not have distinguish private, protected and public attributes. We assume all attributes can 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

  $$\mathbf{A} :< \mathbf{C}_1, \mathbf{C}_2 >$$

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

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

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 $\mathbf{A} :< \mathbf{C}_1, \mathbf{C}_2 >$ by $Multi(\mathbf{A} :< \mathbf{C}_1, \mathbf{C}_2 >) = (M_1, M_2)$, 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.

3.2 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

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

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 $\mathbf{A} :< \mathbf{C_1}, \mathbf{C}_2 >$

$$A \subset C_1 \times C_2$$

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 constraint of an association A such that $Multi(\mathbf{A}) = (M_1, M_2)$ can be specified as a state constraint

$$\begin{array}{ll} & \forall o \in C_1 \bullet |\{o_1 \mid (o, o_1) \in A\}| \in M_2 \\ \wedge & \forall o \in C_2 \bullet |\{o_1 \mid (o_1, o) \in A\}| \in M_1 \end{array}$$

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

The 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 and remove from the library by librarians. Librarians also 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 6 and 7 are 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:

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

The multiplicities of the associations are shown in Figure 7.

3.3 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.

We define nine 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*,

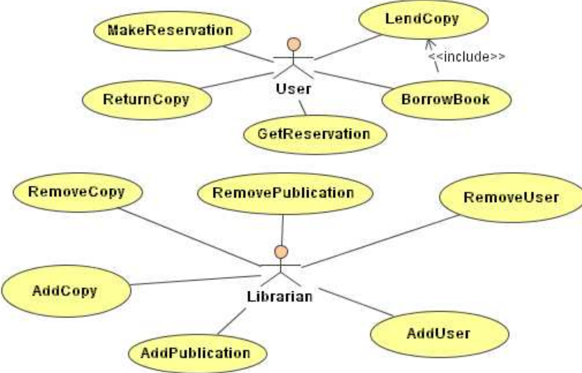


Figure 6: Use-case diagram of the library system

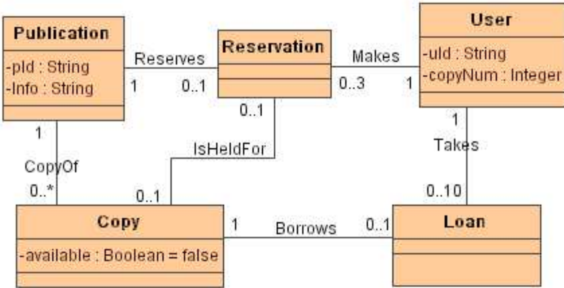


Figure 7: Conceptual class diagram of the library system

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 $\mathbf{A}$ 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 Bollean expression holds or not. The expression can only contain the attributes of existed objects and calculate without side effect, such as $user.copyNum < 10$.

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

$$\begin{array}{lll} op & \stackrel{def}{=} & \mathbf{pvar}\ x : \mathbf{T}_1; \mathbf{rvar}\ y : \mathbf{T}_2 \\ & & \mathbf{Pre} : p(v) \\ & & \mathbf{Post} : R(v, v') \end{array}$$

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 [16, 17].

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 [19] 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:

$$
\begin{array}{lcl}
LendCopy & \stackrel{def}{=} & \textbf{pvar}\ c : \textbf{Copy}, u : \textbf{User}; \\
\textbf{Pre} & : & copy \in Copy \wedge user \in User \\
 & \wedge & copy.available = true \\
 & \wedge & user.copyNum < 10 \\
\textbf{Post} & : & \exists loan : \textbf{Loan} \cdot loan \notin Loan \\
 & \wedge & Loan' = Loan \cup \{loan\} \\
 & \wedge & Borrows' = Borrows \cup \{< loan, copy >\} \\
 & \wedge & Takes' = Takes \cup \{< user, loan >\} \\
 & \wedge & copy.available' = false \\
 & \wedge & user.copyNum' = user.copyNum + 1
\end{array}
$$

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 8.

3.4 Activity diagrams of use cases

The model defined above is not rich enough for generating the prototype as it does not provide information about the flow of interacting events between the actors and the system when carrying out a use case. We specify such an event flow by a UML *activity diagram*. For simplicity, we do not consider concurrency, which will involve the notation of *join activities* in *activity diagram*. An activity diagram

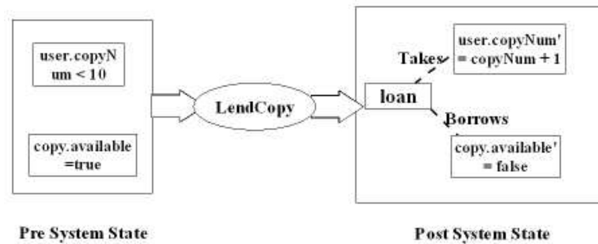


Figure 8: State change by use case LendCopy

is the just a control flow diagram consisting of an *initial state*, two *final states*, a numbers *action states*, a number of *decision states* and a number of *transitions* between two states (see Figure 9):

- The *initial state* denoted by a filled circle, represents the start point of the *activity diagram*,
- There are two *final* states, denoted by circled bulletins. One final state represents the normal termination and the other indicates an exceptional termination of the activity diagram.
- The action states are shown in the rectangles in the activity diagram and represent primitive actions. Each action is represented by its name and parameters, but sometimes we assign an unique identifer to each action rectangle. We can also add textual *comments* in UML comment boxes for the sake of readability.
- The conditional or decision states are shown by diamonds which are used for the selection of next transition *transition*. Each decision state is thus marked by a condition. The condition is either associated with a result of an action state which has a return value of a Boolean type, or choice to be made by an actor of the use case during the execution time.
- There are two kinds of transitions. One kind are transitions from action states to any other states. The other kind are conditional transitions from decision states.

4 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 AutoPA1.0 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.

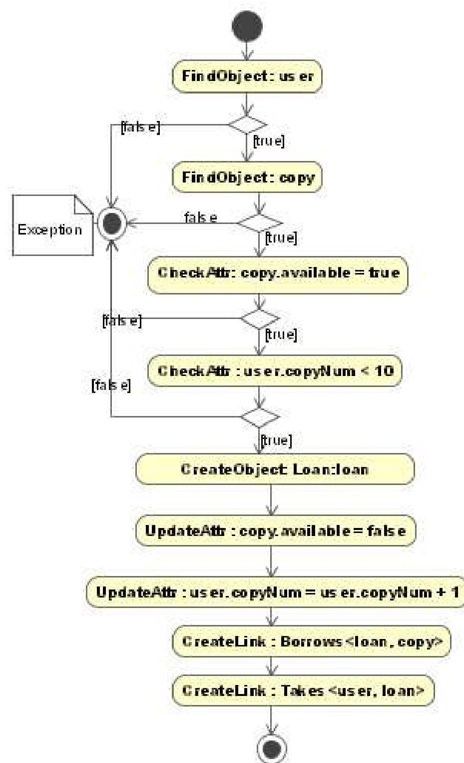


Figure 9: Activity diagram of use case LendCopy

4.1 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 $\mathbf{T}\ x$, where $\mathbf{T}$ is the type of parameter x. Parameters are separated by semicolon ";". For an example, the parameters of use case $LendCopy$ are declared as

$\mathbf{User}\ user;$
$\mathbf{Copy}\ copy;$

It is known that the current system sate 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 8).

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 $Takes < user, loan >$ and $Borrows < loan, copy >$ 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,
                copy.available = false,
                user.copyNum'= copy.copyNum+1,
                Loan loan;
  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 (see Figure 10). AutoPA1.0 can then generate two sequences of primitive actions from the specification to check the pre and post conditions. One sequence consists of query actions for checking the precondition, and the other consists of updating actions for checking the postcondition. We design two algorithms for generating these two sequences of actions, respectively, for a given declaration.

4.2 The algorithm for generating primitive actions from a precondition

A *pre-condition* may include:

- **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 $name :< obj_1, obj_2 >$ 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 algorithm in the Table 1 to generate actions for a pre-condition of a use case. Obviously, primitive actions for checking the pre-condition does not change the system state, also called query actions in [17].

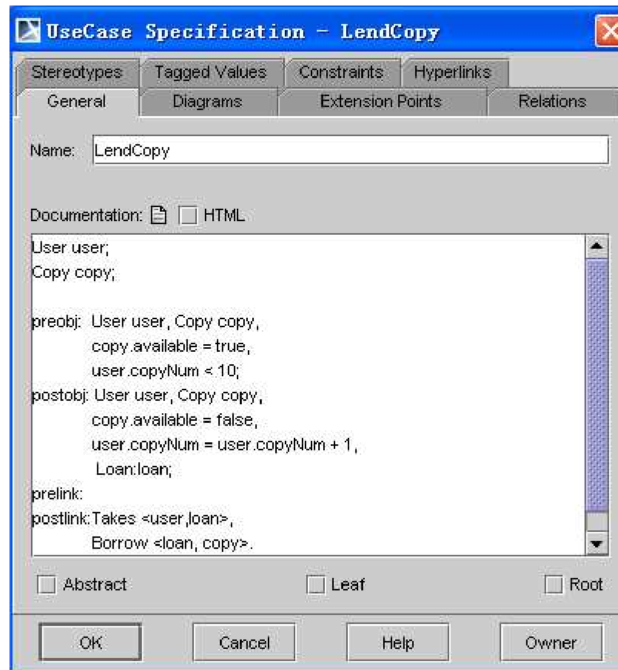


Figure 10: Specification of LendCopy in MagicDraw

4.3 Algorithm for generating primitive actions from a postcondition

A *postcondition* of a use case can be decomposed into the following subconditions [10], each can be satisfied by a primitive action:

- **Create object:** A new object obj of a class **C** is created in system state, i.e. obj $\in$ (*post-objects−pre-objects*).
- **Create link:** A new link $< obj_1, obj_2 >$ of an association **A** between two objects is created and added to the system state, i.e. $< obj_1, obj_2 > \in$ *post-links −pre-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 $< obj_1, obj_2 >$(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 algorithm in Table 2 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

Table 1: Generating actions from a precondition

Input: *pre-objects, post-objects,pre-links, post-links*
Output: *Vector*$<AtomicAction>actions$
Begin:
$Vector<AtomicAction>actions = new\ Vector();$

for $obj \in$ *pre-objects*
$actions.add(FindObject(obj) = true)$
end for

for $name :< obj1, obj2 > \in$ *pre-links*
$actions.add$(*FindLink(name, obj1, obj2) = true*)
end for

for *boolean-expresion* $\in$ *pre-objects*
$actions.add$(*ChecckAttr(boolean-expresion)*)
end for
End

deleting links and objects.

The two algorithms are used to generate the sequence of primitive actions for a use case **when** the pre and post conditions the use case are given in terms of the four sets: *pre-objects, post-objects, pre-links*, and *post-links*. The sequence of primitive actions generated is equivalent to an activity diagram of the use case and is used to by the code generator to directly generate the prototype. It would be difficult, however, for a complex use case be specified directly in terms of these sets. In this case, The AutoPA1.0 code generator can generate the prototype of a use case from its activity diagram drawn by using MagicDraw. For example, prototype in Figure 13 of generated directly for the use case *LendCopy* represented in Subsection A of this section also simulates its activity diagram in Figure 9. The code generator is discussed in the next section.

5 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

Table 2: Generating actions from a postcondition

```
Input: pre-objects, post-objects,pre-links, post-links
Output: Vector<AtomicAction>actions
Begin:
  Vector<AtomicAction>actions = new Vector();

for obj ∈ (post-objects − pre-objects)
  actions.add(CreateObject(obj))
end for

for name:< obj1, obj2 >∈ (post-links − pre-links )
   actions.add(CreateLink(name,obj1,obj2))
end for

for obj.attr = val-expresion ∈ post-objects
   actions.add(UpdateAttr(obj, attr, val-expresion))
end for

for name:< obj1, obj2 >∈ (pre-links − post-links)
   actions.add(RemoveLink(name,obj1,obj2))
end for

for obj ∈ (pre-objects − post-objects)
   actions.add(RemoveObject(obj))
end for
End
```

into a sequence of primitive actions. The prototype generated by AutoPA1.0 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 UML1.4 [7] and XMI1.2. [8]. *AutoPA1.0* 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 AutoPA1.0 can generate the prototype source code. The interface of AutoPA1.0 for parser and code generator are shown in Figure 11 and 12. The main design ideas of AutoPA1.0 are described as the following subsections.

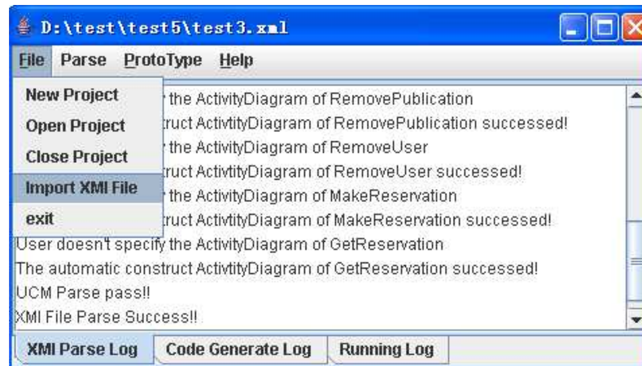


Figure 11: XMI parser of AutoPA1.0

Figure 12: Code generation of AutoPA1.0

5.1 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 constraints, which are all provided by the conceptual class model of the system.

- For each class $\mathbf{C}$ with attributes $Attr(\mathbf{C})$, AutoPA1.0 will 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 $\mathbf{A}$, AutoPA1.0 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 AutoPA1.0 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 $\mathbf{C}$, 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.

5.2 Generating use-case handlers

For each use case in the use-case model, AutoPA1.0 generates a use-case handler class. 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 this use case handler. For example, for use case $LendCopy$ in the library system, there is a use-case handler $LendCopyHandler$ with a method $lendcopy(Stringuid, Stringcid)$ in the library prototype.

As for the body of handler method, AutoPA1.0 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 generated source code for some non-executable requirement specification than cannot be automatic generated source code by AutoPA1.0.

5.3 Interface of prototype

The prototype interfaces of the library system generated by AutoPA1.0 are shown in Figure 13. 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. There are three colors for representing three kinds of action states: red for ”disabled”, $yellow$ for ”enabled” and $green$ for successful executed. 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 is consistent with the description of the use case modelled by an activity diagram or a pair of pre and post conditions.

The prototype also allows the user to check the multiplicity invariants on the system stable states (see Figure 14). ,

6 Conclusion and Discussion

Based on the formalization of UML requirements model [12, 13, 14], this paper extends the approach in [15] for generating a prototype automatically from a UML requirements model and implement it as a tool called AutoPA1.0.

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

Generally, more than 80 per cents of use cases of systems can be captured by the four sets of pre and post objects and links. Therefore, the automatic prototype generator is efficient and effective. People do not need to draw the activity diagram for each use case. For the library system case study, there are 7 use cases out of total 9 that can be automatic generated source code from the four-set algorithm of the pre and post objects and links sets. Use cases $MakeReservation$ and $ReturnCopy$ can be partially

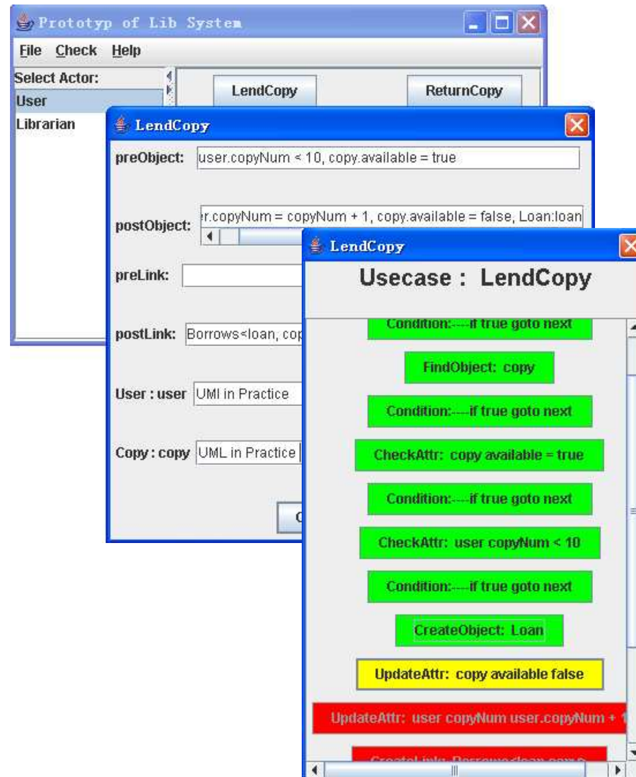


Figure 13: Use interface of AutoPA1.0

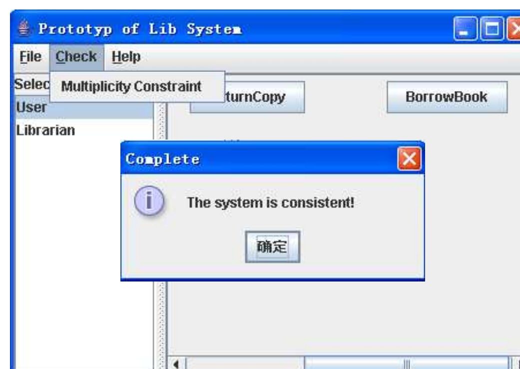


Figure 14: Check multiplicity invariants

generated by the method. We can only draw the partial activity diagrams of two use cases, and then use AutoPA1.0 to generate the library system prototype. With AutoPA1.0 support, system analysts can easily obtain rapid prototypes from system requirements models.

Comparing with the work in [15], this paper with AutoPA1.0 enhances the approach and old tool on the following aspects:

- AutoPA1.0 provides the mechanism to generate source code from activity diagrams. It can handle the complex use case with an interaction event flow (scenario), and including other basic use cases [1].
- The paper and tool can deal with more complex ”CheckAttr” and ”UpdateAttr” actions.
- AutoPA1.0 can directly accept the input of the system requirements model in *.xmi* file. This makes it close to practical application.
- The prototype interface becomes more visualized. The execution of use case can be demonstrated step by step, which is consistent with activity diagrams or system scenarios of event flows.
- AutoPA1.0 can check multiplicity constraints.

AutoPA1.0 is fully supported by the formal method developed in [12, 13, 14]. It is based on the simple set theory and predicate logic. The tool supports the practical use of the underlying formal theory. Based on the current version, we will extend the functionality of checking constraints. Thus, the new version of tool would be possibly applied to the consistency checking of requirements model.

Another future work is to extend the method to cover the pre condition with negation predicate, such as there does not exists a reservation on this publication. This case can be described by current algorithm with only four-set style because it just describes pre system state in positive.

Of course, if the tool can directly analyze the input pre and post conditions written in Object Constraint Language (OCL) or our defined formal language, and recognize the four sets of the pre and post objects and links, then the tool will become event more automated and make the contract based design [16, 17] even more applicable.

Finally, it should be mentioned that this method and prototyping tool even make it possible for these system analysts who cannot read and write formal specification in the pre and post conditions only if they can give the right four sets by analyzing the pre and post system states of use cases.

Acknowledgement: This work is partly supported by eMacao project funded by the Government of Macao and research project of University of Macau. We would like to thank Macao Government and Research Committee of University of Macau.

References

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

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

[3] 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, 20:825-832, 1994.

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

[5] I. Sommerville. *Software Engineering.* Addison-Wesley, 2000.

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

[7] *UML 1.4 Superstructure FTF convenience 2004.* "http://www.uml.org"

[8] *OMG-XML Metadata Interchange (XMI) Specification, v1.2 2002.*"http://www.omg.org/cgi-bin/doc?formal/2002-01-01".

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

[10] C. Larman. *Applying UML and Patterns.* Prentice-Hall International, 2001

[11] Z. Liu *Object-oriented software development in UML.* Technical Report UNU/IIST Report No.228, UNU/IIST, P.O.Box 3058, Macau, SAR, P.R. China, March 2001

[12] Z. Liu, X. Li, and J. He. *Using transition systems to unify uml Models.* In Proc. ICFEM2002, LNCS 2495, Shanghai, China. Spring, 2002.

[13] X. Li, Z. Liu, and J. He. *Formal and use-case driven requirement analysis in UML.* In Proc. COMPASAC01, pages 215-224, Illinois, USA, October 2001. IEEE Computer Society.

[14] Z. Liu, J. He, X. Li and Y. Chen. *A relational Model for Formal Object-Oriented Requirement Analysis in UML.* In Proc. ICFEM, LNCS 3308, November, 2003, Singapore.

[15] 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, Springer, December 22-24, 2004, Bhubaneswar, India.

[16] B. Meyer. *Object-oriented Software Construction(2nd Edition).* Prentice-Hall PTR, 1997

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

[18] D. D'Souza and A.C. Wills. *Objects, Components and Framework with UML: The Catalysis Approach.* Addison-Wesley, 1998

[19] Z. Mana and A. Pnueli. *The temporal framework for concurrent programs* In R.S. Boyer and J.S. Moore, editors, The Correctness Problem in Computer Science, pages 215-274. Academic Press, 1981.