AutoPA: Automatic Prototyping from Requirements

Xiaoshan Li, Zhiming Liu, Martin Shäf and Ling Yin

July 2010
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 Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People’s Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao 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, Macao 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
AutoPA: Automatic Prototyping from Requirements

Xiaoshan Li, Zhiming Liu, Martin Shäf and Ling Yin

Abstract

We present AutoPA, a tool to analyze and validate the consistency and functional correctness of use case designs. The tool directly generates an executable prototype from the requirements. The requirements are captured from different views of the application. Each view is constructed as UML diagram annotated with OCL specifications. Based on a formal semantics, the tool is implemented so that both syntactic and semantic consistency among the provided views can be guaranteed. Afterwards the requirements are analyzed and translated into an executable prototype, allowing the user to interactively validate the functional properties of the requirements model. We illustrate the benefits of the tool using a real-world sized example.

Keywords: Formal Semantics, Requirements Models, Prototyping, Validation
Xiaoshan Li is an associate professor of University of Macau. He research interest is in formal specification and verification. \texttt{xsl@umac.mo}

Zhiming Liu is a senior research fellow at UNU-IIST. He is the leader of the rCOS (“Refinement of Component and Object Systems”) research group, working on foundations, advances and applications in information engineering, especially on component-based model driven software design. \texttt{Z.Liu@iist.unu.edu}

Martin Shäf is a postdoctoral research fellow of the rCOS group at UNU-IIST. His research interest is in the area of program verification, static analysis and runtime verification in particular. \texttt{shaef@iist.unu.edu}

Ying Yin is a UNU-IIST from East China Normal University where she is a master student in computer science. Her subject of study is formal method based tool development. \texttt{yl@iist.unu.edu}

The work was supported by the projects HTTS funded by Macau Science and Technology Development Fund, NSFC 90718014, 973 program 2009CB320702, and STCSM 08510700300.
Contents

1 Introduction ............................................ 1
2 The Library System ..................................... 1
3 Modeling of requirements ............................... 2
   3.1 UML models of requirements ....................... 2
   3.2 Use case model ................................... 3
4 Design of AutoPA ....................................... 6
   4.1 Implementation of OCL expressions ............... 8
5 Prototype of the Library System ....................... 10
   5.1 Generating a prototype ............................ 11
   5.2 Execution of generated prototype ................ 11
6 Conclusion and Discussion ............................ 14
1 Introduction

In the use case driven incremental development process specifying a system, e.g. by using UML [3, 11] and the Rational Unified Development Process (RUP) [7, 9], starts with identifying its use cases. Any further step in the process relies on these use cases. Therefore it is crucial that the use cases are consistent.

We propose the use of a formal semantics for use cases which is based on our earlier work on formal semantics of UML models of requirements [11, 12]. A model of requirements of an application is defined as a consistent set of models of different views: use case diagrams, conceptual class model, use-case sequence diagrams and data functionality of the use case operations specified as pre- and post-conditions. We extend the set of models with use case activity diagrams that specify synchronization of parallel use cases (cf. Section 3.2).

In this paper, we present AutoPA. A tool to automatically generate an executable prototype from a requirements model to check the functional properties and semantic consistency of use cases.

AutoPA provides a rich user interface to design the requirements model of the system but also can work on input from other case tools. The user generates a set of use cases, conceptual class diagrams, and use case activity diagrams. Requirements regarding the data functionality of the system can be provided as OCL specification. The consistency of these views of the system w.r.t. our formal semantics is verified automatically (see Section 3). As a result, we obtain a model of the requirements which has a precisely defined semantics. This model is then automatically translated to an executable prototype (see Section 4). The user can now use the prototype to interactively validate the functional requirements of the system.

We illustrate the functioning of AutoPA on a real-world sized example of a library system (see Section 5) and show how AutoPA can improve the use case identification process.

2 The Library System

Throughout this paper we illustrate the functioning of AutoPA using a running example. Consider a Library System. It maintains a catalogue of three kinds publications: Book, Periodical, and AudioVisual. Periodicals can only be read in the library, while other kind of items can be borrowed. Publications are organized according to subjects for statistic analysis, and each publication can have multiple copies. Users need to be registered with systems, and different policies apply to different types of users, students and teachers. In this paper, we only consider the functionalities handling loans, returns and reservations of users. A class diagram for these functionalities is given in Fig. 1. Invariants specified in OCL can be attached to specify the business rules.
Modeling of requirements

3 Modeling of requirements

We follow the use-case driven approach to capture requirements that is proposed in RUP \[7\], which is described in \([10, 4]\) more systematically. Informally the process of requirements understanding, capture and analysis is the following:

1. Identify the business processes as use cases, including actors, use cases and relations between use cases. Write the description in terms of interactions between actors and use cases, and among use cases, and construct the use case diagram.
2. Identify interaction events and input output data of the system for defining the use case operations and form the activity diagrams.
3. Study and formulate the data functionalities of the use case operations and write their specification in terms of pre- and post-conditions.
4. The previous steps also lead to identification of data types, domain concepts and objects. Based on this, classes and data types are defined, as well as their associations and constraints, and they are represented as a class diagram, called the *conceptual class model* (see Definition the next subsection).

3.1 UML models of requirements

Based on the above discussion, we formalize our definition of a model of the requirements.
Definition 1 (Model of requirements) A model of requirements $\mathcal{M} = (\mathcal{U}, \mathcal{C})$ consists of a use case model $\mathcal{U}$ and a conceptual class model $\mathcal{C}$.

Although in the process of requirements capture, identification of use cases may start first, their static functionality and dynamic behavior can only be defined after the data types and classes of objects are defined.

A precise definition of the conceptual class model $\mathcal{C}$ is given in our earlier work [11, 12], that we do not repeat here. Informally, the conceptual class model can be represented by a UML class diagram and a set of OCL specifications. The OCL specifications specify constraints on objects and their relations, for example “a Copy that is held for a Reservation must be a Copy of the Publication that the Reservation reserves”. Such a class model is called conceptual because its classes does not have methods, and associations do not have direction. Thus, it represents the concepts and their relations in the application domain.

For simplicity, the association name is used as the role name for an end of the association. For example, if $p$ is a Publication instance, $p.\text{reserves}$ is the Reservation instance that reserves $p$; symmetrically if $r$ is an instance of Reservation $r.\text{reserves}$ denotes the publication that $r$ reserves.

At the level of requirement modeling, a state is the set of existing objects and the relations between them at a distinct point of time during the execution. Thus, the conceptual class model defines the possible state space of system, and a state can be represented as a UML object diagram object diagrams. All the object diagrams have to satisfy the invariants of the class model specified in OCL. In later stages of design and implementation, these objects can be refined and implemented by by much more objects. This is also an indicator of the advantage of the tool for scaling up.

3.2 Use case model

The use case model includes a use case diagram representing the static relations among the use case cases, an activity diagrams for the dynamic behavior of the use cases, and the specification of the functionality of the use case operations.

Use case diagram In Fig. 2 we present the use case diagram of the library system. The use case diagram models, for each actor, the use cases that can be executed. This figure shows one of the main advantages of AutoPA. The translation of the requirements model into an executable prototype gives the user the opportunity to introduce use cases that can test the functionality of other use cases: we introduce a use case setCurrentDate with which the Librarian actor can do dailyCheck. With the setCurrentTime operation, a librarian can set the system time of the prototype as the date when a loan becomes overdue and run use case checkOverdueLoan. In a real program this use case would usually implemented by an event listener and thus, it would not appear in the specification. By adding a semantic meaning to use cases, a user can actively test her use case diagram by adding other use cases. This leads to a broader and more reliable specification.
In AutoPA, the use case diagram is used as the vehicle for creating and integrating (in terms of contexts) the UML models. For the development of AutoPA, we need a formally defined syntax and semantics of use cases. To this end, we adopt the idea of rCOS model of components to use a use case controller class to declare a use case. This idea is inspired by the Controller Pattern in [10].

A use case defines a number of operations as methods that are to be called by its actor to perform the use case. Use cases can be related in a way that one use is included (or used in the UML terminology) in another.

**Definition 2 (Use Case Classes)** A use case class is of the form

\[
\text{Class UseCase Name includes U1,..., Uk } \{
\begin{array}{c}
 f: T, \ldots, f: T; \\
 \text{m(in: T;out: R)(pre: P; post: Q);} \\
 \ldots \ldots \\
 \text{m(in: T;out: R)(pre: P; post: Q);} \\
\end{array}
\}
\]

where the keyword “includes” has the same semantics as “extends” in rCOS (or Java), and the types fields \( f \) can be data types or classes defined in the conceptual class model, and the pre- and post-conditions are written in OCL.

We can also add use case invariants in the use case class when needed.

---

1. In rCOS, a use case is model as a component with a provided interface and an interface class is used to “implement” the interface.
Actors and activity diagrams  Actors interact with the system following a protocol in performing a use case. Each interaction event is initiated by the actor by sending an invoking message to the system. This message causes a state change of the system by execution of some a program statement and a return message (possibly with outputs) to the actor.

The actor class defines the association between an actor and a use case. For simplicity in the implementation each use case has its own actor named UseCaseNameUser. For example, the actor of borrowCopies is named as borrowCopiesUser.

An actor is similar to a Java thread and its activity() is the run() of the thread. The activity method defines the order in which the operations of the use are called and executed to realize the business policy. In rCOS this order is called the interaction protocol and defines the dynamic behavior of the use case. The syntax of the body $S$ of activity() is given by

$$S ::= \text{m() | A.activity() | S; S | if } b \text{ then } S \text{ else } S | \text{while } b \text{ do } S | S \| S$$

where $\text{m()}$ must be a method of the use case class, and $A$ a name of the actor of a use case class included in the use case class and its meaning can be understood as $A$.activity() in Java.

Note that a method in a use case class or the activity method in an actor do not call methods of the conceptual classes, they are not designed anyway. We use an activity diagram to represent the behavior of the activity() method of an actor. Fig. 3 shows the activity diagrams of two use cases, borrowCopies on the left and dailyCheck on the right. The diagram on the left shows that use case borrowCopies is carried outs in the following way

1. Input the cardID of the user and locate the corresponding object user in current system state.
2. If the user can be identified, call use case borrowCopy to lend the copy, otherwise, end this use case directly.
3. If user borrows more than one copy, repeat process from Step 2 until no further copy to borrow and then finish this run of the use case.

Definition 3 (Use Case Model)  The use case model of an application is the list of classes for all the use case and actors defined for the application.
Design of AutoPA

The AutoPA GUI is designed for creating UML models is designed, but AutoPA also allows its users to use a UML CASE tool, such as MagicDraw, to create requirements models of their applications. These models are saved into XMI files. However, the design of the AutoPA GUI and models created by MagicDraw must conform to those conditions defined in Section 3. For this, AutoPA defines a meta model, called UML+OCL meta model. An instance of this meta model generated from the XMI file is stored in AutoPA and used for syntactic checking.

AutoPA stores the requirements model (e.g., actor and control classes) and the OCL specification separately to allow further editing of the OCL constraints. During the prototype generation, the OCL specifications are translated into a sequence of atomic actions defined below. These actions are needed to avoid potential side effects that may arise when translating pre- and post-conditions into sequential code. Finally an executable prototype is created. The prototype implements the provided requirements model according to our semantics and further provides a graphical user interface allowing the user to trigger different use cases by simply pressing a button.

An architectural overview of AutoPA is shown in Fig. 4. In the following, we describe the used components in more detail.
XMI Parser and OCL Parser  AutoPA XMI file generates an internal representation of the model (IRM) from a model of requirements provided by the user (e.g. as XMI file). The IRM includes a model file, that records the model elements of the requirements model (classes, associations, actors/control classes, etc.), and a OCL file that specifies constraints on the model elements, and the pre- and post-conditions of the use case operations.

The OCL Parser takes generates an Abstract Syntax Trees (ASTs) of the OCL constraint expressions in the IRM. The OCL parser is a reuse of Octopus parser of the OCL validation tool [16].

Model Transformer  takes the IRM file and the ASTs as its input and generates an instance of UML+OCL model. It then transforms each pair of pre- and post-conditions of each use case operation into a sequence of atomic actions to implement the state check and change specified by the pre- and post-conditions. For this, AutoPA defines the atomic actions FindObject, FindObjects, FindLink, FindLinks, CheckAttribute, CreateObject, RemoveObject, CreateLink, RemoveLink, and Update. Each of them is specified as a pair pre- and post-conditions in OCL and implemented in Java. The first five actions do not change the system state while the other five do. The semantics of these atomic actions are simple and clear from their names. These operations are sufficient to checking conditions and updating a state (the object heap).

In the use case model, the user does not have to write the code for the activity methods. Instead, Model Transformer parses each activity diagram and generates a tree of ActivityNodes, from which the Java code of of the activity methods are generated.

Code Generator and Prototype Template  Before generating code, AutoPA first check the well-formedness conditions of models. For this, we define and implement a meta model for the syntax of UML elements with OCL constraints for the models defined in Section 2. After the model passes the well-formedness checking, Code Generator takes the output of Model Transformer and generates the Java code of the prototype. The code include the implementation the conceptual classes, associations relations between these classes, inheritance relations, the use classes, actor classes, and the statements of the use case operations and the activity methods of actor classes.
Prototype Template is the framework for the prototype GUI and static class declaration. Based on the framework, the Code Generator can generate the prototype java classes from the conceptual class diagram and use case diagram. Finally, the Code Generator generates the prototype GUI that extends the GUI classes of the Prototype Template.

4.1 Implementation of OCL expressions

OCL is a description language, i.e. pre- and post-conditions are side-effect free. They specify a relation between states (i.e. a next state relation), there is no indication of the orders and combinations of atomic actions for the implementation of the next state relation. We have identified and implemented the patterns between OCL expressions and combinations of atomic actions. For example, Fig. 5 shows the AST of

\begin{align*}
\text{Publication} & \rightarrow \text{includes}(\text{pub}) \text{ and Copy } \rightarrow \text{excludes}(\text{copy})
\end{align*}

the following OCL expression.

By traversing this tree, the Model Transformer decomposes the OCL expression into a number of sub-expressions according to the logic structure of expression. Each such sub-expression is an AST fragment that can match one of the atomic action (also specified in OCL). AutoPA defines the rules for matching AST fragments with atomic actions, using AST fragment templates. For example, the atomic action \texttt{FindObject (ObjId,Classifier)} checks whether a given instance object with identifier \texttt{ObjId} of class \texttt{Classifier} exists in the current system state. If it does not exist, the corresponding precondition is false, and the prototype should go to the exception handling. The template \textbf{T1} in Fig. 6 and rule \textbf{R1} shows a case for generating a \texttt{FindObject} action.

\begin{align*}
T(fg) &= T1, \forall \text{ OP1 } \in \{ \text{asSet, asBag, asSequence} \}, \\
& \quad \forall \text{ OP2 } \in \{ \text{includes, notEmpty, one} \} \\
\textbf{R1} & \triangleq \frac{\text{FindObject}(V,C)}{}
\end{align*}

If \( T(fg) = T1 \) and the contents of the fragment parameters satisfy the premises of rule \textbf{R1}, the atomic action corresponding to the fragment \texttt{fg} is \texttt{FindObject(ObjId,Classifier)}. 
Execution order of the atomic actions  The OCL expression of a post-condition specifies the change of a state. AutoPA generates the corresponding atomic actions of a post-condition in a correct order. For example, if an expression specifies the removal of an object and the removal of a link to the object, the action for the removal of an object cannot be executed before the actions for removing the link to the object.

AutoPA implements a sorting algorithm when generating the sequence of atomic actions for a pair of pre- and post-conditions. We assign the highest priority to atomic actions FindObject and FindObjects, and then in a decreasing order for FindLink, FindLinks, CheckAttribute, CreateObject, CreateLink, UpdateAttribute, RemoveLink and RemoveObject. For the use case borrowCopy of the library system, AutoPA generates the following sequence of atomic actions from its specification (that we omit due to page limit).

1. FindObject(u, User)  
2. FindObject(c, Copy)  
3. CheckAttr(u, User, "status", "=", "normal")  
4. CheckAttr(c, Copy, "status", "=", "available")  
5. if FindObject(u, Staff)  
   then CheckAttr(u, User, "borrowNO", "<", 30)  
   else CheckAttr(u, User, "borrowNO", "<", 10)  
6. CreateObject(ln)  
7. Publication pub=FindLink("copyOf", c).getOtherRole()  
8. CreateObject(lr: LoanRecord)  
9. UpdateAttr(c, Copy, "status", "=" , "loan")  
   then UpdateAttr(ln, Loan, "dueDate",CalendarNow+10)  
   else if FindObject(u, Staff)  
   then UpdateAttr(ln, Loan, "dueDate",CalendarNow+60)  
   else UpdateAttr(ln, Loan, "dueDate",CalendarNow+30)  

AutoPA can generate checks for system invariants, which are specified as OCL expressions. System invariants should hold on all system stable states, i.e. In any state before or after a use case operation. Multiplicities in a class diagram are special kinds of invariants. Code for an invariant is generated in the
same as for a precondition, it is in general an iteration statement that checks all objects and links of the current system state.

When generating code from post-conditions, the functionality of AutoPA is limited to post-conditions that are formulated from the primitive assignable expressions of the form $e_1 = e_2$, where $e_1$ is a navigation path $x.a_1 \ldots a_k$ and $e_2$ can be evaluated on the current state, i.e. the pre-state in term of OCL. To generate code from expressions like $x + y = x@pre + 10$ and $x + 3y = y@pre - 5$ an equation solver or SAT solver is needed. For the same reason, AutoPA does not handle nondeterministic post-conditions such as $x > y@pre + x@pre$.

5 Prototype of the Library System

We build the requirements model of the library system by drawing diagrams and OCL specifications. The XMI files of this model is then uploaded to AutoPA to generate its executable prototype in Java. We validate the requirements of library system by running the Java prototype, This section demonstrates the results of the generated library prototype system.

We use UML CASE tool MagicDraw to build the requirements model, which includes conceptual class diagram, use case diagram (Fig. 1 and 2) and the activity diagrams for complex use case (the left of Fig. 7), and the associated OCL specifications which are attached to use case operations and classes. Finally, the requirements model can be saved the model into an XMI file shown the right of Fig. 7.
5.1 Generating a prototype

We run AutoPA to generate an executable prototype from the XMI file produced from the UML model of requirements in the following steps

1. Run AutoPA and create a new project, see the left of Fig. 8.

2. Import the XMI file produced above, see the right Fig. 8. AutoPA parses it and checks its syntax, and it then reports the result of the syntactic checking as shown in Fig. 9. If the syntactic checking of the XMI file succeeds, AutoPA generates an IRM which can be edited in AutoPA, as shown in Fig. 10.

3. Then we generate the prototype from the IRM file imported, as shown Fig. 11.

5.2 Execution of generated prototype

For this case study, the prototype generated by AutoPA contains 156 .java files with a total of 34,522 lines of code. We execute the generated prototype to validate the requirements of the Library system. The interface of the prototype is shown in Fig. 12. Once we select an actor on the left area of the window, the use cases associated with it appears on the right side. For example, in Fig. 12 the actor "Librarian" is
selected, and the use cases, "registerUser", "addFaculty" etc., are shown on the right area of the window.

![Prototype generation](image1)

**Figure 11: Prototype generation**

We can click a use case on the right side and execute it step by step. We run *borrowCopy* use case to illustrate the execution process. Use case *borrowCopy* is specified by a pair of pre- and post-conditions. First, we input the parameters it requires, as shown in Fig. 13, and then execute it by clicking the sequence of atomic actions shown in Fig. 14. A "Green" button means the execution of the atomic action has been finished, a "Yellow" button means the atomic action is to be executed next, a "Red" button shows the atomic action has not been executed yet. The results of the execution of each atomic action are shown in the *OutPut* area.

![Interface of use case *borrowCopy*](image2)

**Figure 12: Interface of generated Library prototype**

**Figure 13: Interface of use case *borrowCopy***
Checking invariants  We can also check multiplicities and invariants on current system state. A screen short of the checking is shown in Fig. 15.

An invariant related of a use case is checked after the execution of each atomic action, as shown on the left side of Fig. 14. The methods of checking an invariant are generated as shown in Fig. 16. In the example, the method INV_Copy_On_Available is generated for checking the invariant Copy_On_Available specifying the assertion that a copy is available only if the publication of the copy is not reserved or each reservation of it has a copy held for it but this copy not held for a reservation. This is a complex invariant on class Copy, Reservation, Publication and the associations between them. It is specified as follows and
implementation of the checking is shown in Fig. [16]

context Copy
inv Copy_On_Available: self.status = CopyStatus::available
implies (self.borrowed -> isEmpty() and
self.copyOf.reserves -> isEmpty() or
self.copyOf.reserves -> forAll(r | r.heldOn -> notEmpty()))

Figure 16: Methods for checking invariant INV_Copy_On_Available

6 Conclusion and Discussion

We have presented the formal definition of models of requirements, and the design and implementation of the prototyping tool AutoPA. With the graphical user interface or an existing UML CASE tool, the structural aspects of the models of different views can be prepared, and the use case activities separately by drawing. The model constraints and the pre- and post-conditions of use case operations specified in OCL are prepared in text. The key algorithm of the tool is to translate use case operation in OCL specification into a sequence of atomic actions. The prototype is used for validation functional properties, as well as for syntactic consistency checking.

Comparing with other prototype tools, a distinct feature of AutoPA is prototyping from a model of requirements directly, rather than from a design model, such as design sequence diagrams, a design state diagram, or a model of live sequence charts [15, 6]. The tool USE [2] mainly focuses on validating UML models with OCL constraints by testing the given system states (object diagrams) [5]. As discussed in Section 3 and 4, this function is also supported by AutoPA, as constraints on states can be checked in addition to automatic generate of an executable prototype.
Further development of AutoPA includes the OCL MessageExp expressions for prototyping real-time interactive systems. We also plan to develop visual animation functionality for demonstrating the dynamic behavior of state changes using the graph-based operational semantics in [8]. Another application could be to use AutoPA for test case generation. This can be realized either by deriving input values from the OCL specification or by introducing special use cases that describe how other use case are tested.

Our experiments with the library system show that AutoPA is a useful tool to support software design by contracts [13] [14] [4]. AutoPA contributes to the landscape of CASE tools in two ways. 1) By checking the semantic consistency of the use case model we can improve the quality, especially for complex diagrams. 2) The generated prototype allows to validate the functional properties of the system in an interactive way, i.e. the user can experiment with the system.

Acknowledgements We would like to thank our colleagues Cristiano Bertolini and Volker Stolz for their comments.

References
