Automated Prototype Generation from Formal Requirements Model

Yilong Yang, Xiaoshan Li, Wei Ke and Zhiming Liu

Abstract—Prototyping is an effective and efficient way of requirement validation to avoid introducing errors in the early stage of software development. However, manually developing a prototype of a software system requires additional efforts, which would increase the overall cost of software development. In this paper, we present an approach with a developed tool RM2PT to automated prototype generation from formal requirements models for requirements validation. A requirements model consists of a use case diagram, a conceptual class diagram, use case definitions specified by system sequence diagrams and the contracts of their system operations. A system operation contract is formally specified by a pair of pre- and post-conditions in OCL. We propose a method with a set of transformation rules to decompose a contract into executable parts and non-executable parts. An executable part can be automatically transformed into a sequence of primitive operations by applying their corresponding rules, and a non-executable part is not transformable with the rules. The tool RM2PT provides a mechanism for developers to develop a piece of program for each non-executable part manually, which can be plugged into the generated prototype source code automatically. We have conducted four case studies with over 50 use cases. The experimental result shows that the 93.65% system operations are executable, and only 6.35% are non-executable, which can be implemented by developers manually or invoking the third-party APIs. Overall, the result is satisfactory. Each 1 second generated prototype of four case studies requires approximate 1 day' manual implementation by a skilled programmer. The proposed approach with the developed CASE tool can be applied to the software industry for requirements engineering.

Index Terms—Formal Requirements Model, Requirements Model, Requirements Validation, Code Generation, Prototype, UML, OCL

1 Introduction

Requirements errors are one of the causes leading failings in software projects [1]. Careful requirements modeling along with systematic validation helps to reduce the uncertainty about target systems [2] [3]. The goal of requirements validation is to construct the consistent requirements for the needs of target users [4]. However, this process is complicated, and it can be hard to produce a correct and complete requirements specification. The complexity is due to the following interrelated attributes [5] [6] [7]: 1) the complexity of application domains and business processes; 2) the uncertainty of clients and domain experts about their needs; 3) the lack of the understanding of system developers about application domains; 4) the difficulties of the understanding between system developers and clients.

Rapid prototyping is an effective approach to requirements validation and evolution via an executable model of a software system to demonstrate concepts, discover requirements errors and find possible fixing solutions, and discover missing requirements [8]. Besides the implementation of main system functionalities, a prototype has a User Interface (UI) [9] that allows the client to validate their requirements visually, make it easy to find out faults in the requirements, and then fix them [10]. In practice, it is very desirable to have a tool that generates prototypes directly from requirements automatically. However, state-of-the-art research and Computer-Aided Software Engineering (CASE) tools still have long distances to reach the goal [11]. The Unified Modeling Language (UML) is a de facto standard for requirements modeling and system design. The current UML modeling tools, such as Rational Rose, SmartDraw, MagicDraw, Papyrus UML, can only generate skeleton code, where classes only contain attributes and signatures of operations, not their implementations [12].

In this paper, we present an approach for generating prototypes automatically from formal requirements models in UML diagrams complemented by the contracts of system operations in Object Constraint Language[1] (OCL). Formal requirements specification has been widely used and proven useful in the software industry, especially in the safety-critical system development [13]. Formal methods can improve the clarity and precision of requirements specifications, and are helpful to find errors in requirements validation and software testing [14]. However, it is difficult for software engineers to write formal specifications in formal specification languages, such as B and Z [15]. In practice, the lightweight formal approach [16] provides a simple yet expressive and flexible formal language for users to specify requirements specification easily and precisely, which offers a cost-effective way of improving the quality of software [14]. Lightweight formal approaches have been applied successfully to many system developments of NASA projects [17]. OCL is a lightweight formal specification language based on the first-order logic, which is mainly used for specifying the constraints of UML models such as invariants, pre- and post-conditions as well as Eclipse foundation projects and IBM product lines [18]. Therefore, we directly use OCL to formally specify the contracts of

- *Yilong Yang (Email: yylonly@gmail.com) and Xiaoshan Li are with the Faculty of Science and Technology, University of Macau, Macau.*
- *Zhiming Liu is with School of Computer and Information Science, Southwest University, Chongqing, China.*
- *Wei Ke is with Macau Polytechnic Institute, Macau.*
- *Xiaoshan Li and Zhiming Liu are the corresponding authors of this paper. Email: xsl@umac.mo and zhimingliu88@swu.edu.cn.*

1. http://www.omg.org/spec/OCL/

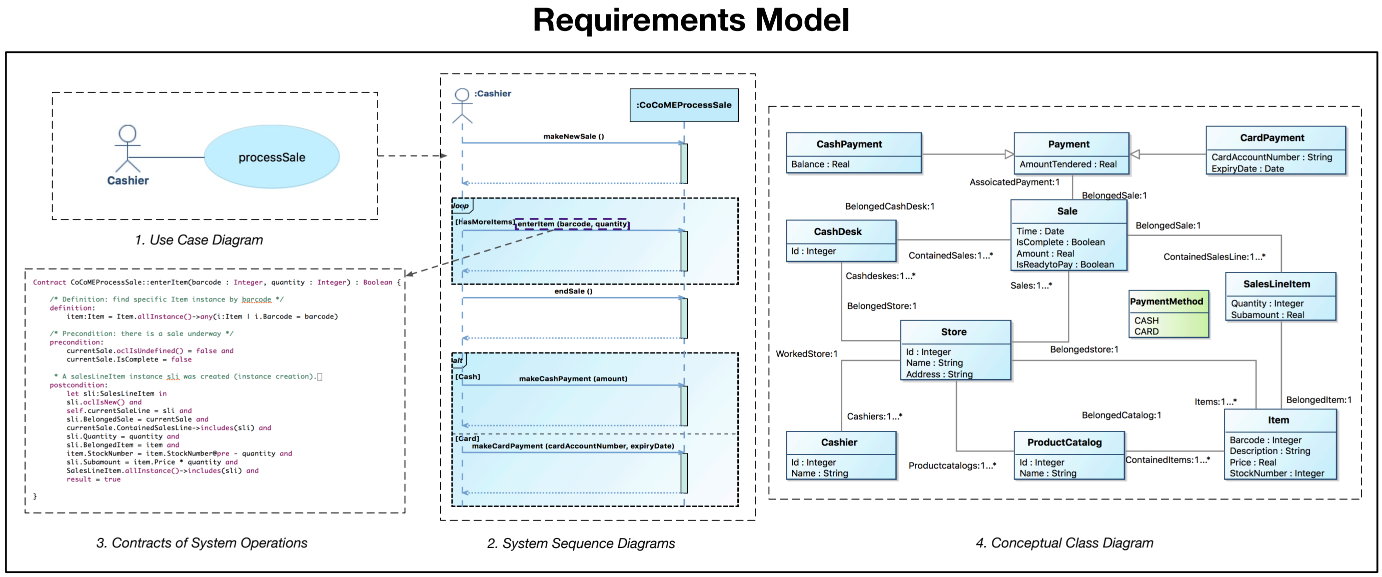


Figure 1. Requirements Model

system operations in requirements models.

Compared with other related work, our approach does not require design models but rely on a lightweight formal requirements model [19] [20] [21] in Figure 1, which contains:

• *A use case diagram:* A use case diagram captures domain processes as use cases in terms of interactions between the system and its users. It contains a set of use cases for a system, actors represented a type of users of the system or external systems that the system interacts with, the relations between the actors and these use cases, and relations among use cases. It helps customers and domain experts specify functional requirements of the target system, and it assists in generating the operation list to hold system operation in prototypes, which is shown in Figure 4.

• *System sequence diagrams.* A system sequence diagram describes a particular domain process of a use case. It contains the actors that interact with the system, the system and the system events that the actors generate, their order, and inter-system events. Compared with the sequence diagram in design models, a system sequence diagram treats all systems as a black box and contains system events across the system boundary between actors and systems without object lifelines and internal interactions between objects. It helps customers to find system operations and provides the sequences to interact with the prototype for requirements validation.

• *Contracts of system operations:* The contract of a system operation [22] [23] specifies the conditions that the state of the system is assumed to satisfy before the execution of the system operation, called the *pre-condition* and the conditions that the system state is required to satisfy after the execution (if it terminated), called the *post-condition* of the system operation. Typically, the pre-condition specifies the properties of the system state that need to be checked when system operation is to be executed, and the post-condition defines the possible changes that the execution of the system operation is to realize. A state of an object-oriented system is about the existing objects together with their properties/states and relations/links. The state is an object diagram defined by a conceptual class diagram plus the input and returns parameters of the operations. The changes of system states are classified into i) new objects created (together with initial values of attributes and links of associations), ii) attributes of existing objects (in the current state) modified, iii) new links among existing objects formed, and iv) existing objects and/or links are removed. The basic operations of changing the state are defined as the *primitive operations* in our approach. We will see, the decomposition from system operation into primitive operations is the theoretical foundation of our approach to automatic generation of an abstract and yet executable model from a requirements model.

• *A conceptual class diagram.* A conceptual class diagram is a concept-relation model, which illustrates abstract and meaningful concepts and their relations in the problem domain, in which the concepts are specified as classes, the relations of the concepts are specified as the associations between the classes, and the properties of the concepts are specified as the attributes of the classes. Compared with the class diagram in design models, a conceptual class diagram does not contain operations in the classes. It focuses on specifying the domain concepts but not for how to encapsulate operations. The proposed approach can directly generate Java classes that represent domain concepts and encapsulates the primitive operations such as getting and setting the values of attributes, adding and remove links, and finding an object through links in the prototypes.

The preliminary idea of automated prototype generation from a requirements model is presented in our earlier work [24] [25]. In this paper, we extend the original work by fully considering the theoretical algorithms of model transformation, and designing and implementing an applicable prototype tool with full functionalities of requirements modeling, automated prototype generation as well as requirements validation, and the tool has been applied to four extensive case studies. Note that the more details of comparisons with our previous work are presented in the section of the related work.

The remainder of this paper is organized as follows: Section 2 presents the preliminary of our approach. Section 3 introduces how to decompose a system operation into primitive operations. Section 4 presents how to generate the prototype. Section 5 provides how to use the generated prototype for requirement validation. Section 6 presents the experiments result of our approach on the four case studies of a Library Management System, CoCoME, ATM and a Loan Processing System. Section 7 and 8 discuss the related work, conclude this paper, and outline the future work.

2 Preliminary

In this section, we introduce the terminology used in the requirements model and prototypes.

2.1 Terminology

The terminology related to the proposed approach except the terminology introduced in the previous section are listed as follows:

• *Association and link.* An association is a relationship between two classes in a conceptual class diagram that specifies how instances of the classes can be linked to work together. A link is an instance of an association, which is a relationship between two objects in an object diagram.

• *Entity class and fabricated class.* To indicate the classes are from domain concepts or fabrications in the prototype, we divide classes into two type: entity classes are Java classes generated in prototypes from conceptual class diagrams, the others are fabricated classes. For example, Java class *Item* is entity class generated from the conceptual class diagram in CoCoME, and Java class *EntityManager* is a fabricated class that helps to find, create, and release the objects in the system.

• *Object reference and reference list.* In object-oriented programming such as Java, an object reference is a variable, the value of which is a type of a class. Object reference provides a way to access an object in the heap of a system. A reference list is a list of object references, which contains the same type of a class. We will see, reference lists are used to record and access the objects of the prototype.

• *System operation.* System operation is an operation that the system executes in response to a system input event in system sequence diagrams.

• *Primitive operation.* Primitive operations are the operations introduced in our approach to covers all actions to manipulate objects, the attributes of objects, and the links of objects in Table 1. The details are shown in Section 3.1.

2.2 Object Constraint Language

Object Constraint Language is a lightweight formal specification language, which is used for specifying constraints of UML diagrams, such as pre- and post-conditions of operations and invariants. In this paper, we adopt OCL in the latest version *v2.4* to specify the contracts of system operations. It not only can specify the pre-condition and post-condition of system operation but also allows to specify shared specifications from pre-condition and post-condition in the *definition* section. The following example shows how OCL specify the contract of the system operation *enterItem* (a cashier scans products in a sale process of a supermarket) of the use case *processSale*.

```
//Signature
Contract CoCoMEProcessSale::enterItem
  (barcode : String, quantity : Real) : Boolean {

//Definition Section
definition:
    //Find Object
    item:Item = Item.allInstance()->any(i:Item |
        i.Barcode = barcode)

//Pre-condition Section
precondition:
    currentSale.oclIsUndefined() = false and
    currentSale.IsComplete = false and
    item.oclIsUndefined() = false and
    item.StockNumber > 0

//Post-condition Section
postcondition:
    //Create an Object
    let sli:SalesLineItem in
    sli.oclIsNew() and
    //Add Links
    self.currentSaleLine = sli and
    sli.BelongedSale = currentSale and
    currentSale.ContainedSalesLine->includes(sli) and
    sli.BelongedItem = item and
    //Modify Attributes
    sli.Quantity = quantity and
    sli.Subamount = item.Price * quantity and
    item.StockNumber = item.StockNumber@pre -
        quantity and
    //Add an Object
    SalesLineItem.allInstance()->includes(sli) and
    result = true
}
```

Signature: The contract first specifies the signature of system operation *enterItem()* of use case *processSale*. The signature declares a *String* variable *barcode* and a *Real* variable *quantity* as input, and output variable typed *Boolean*.

Definition Section: In the *definition* section, we find the object *item* of the class *Item* with the attribute *Barcode* equal to the input variable *barcode*. We will see that the object *item* is used in both the pre- and post-conditions.

Pre-condition Section: The pre-condition of *enterItem* specifies that the current sale object *currentSale* is *existed*, and the value of attribute *IsComplete* of *currentSale* is equal to *false*, the object *item* with the scanned *barcode* is *existed* in the system, and the stock number is greater than zero.

Post-condition Section: The post-condition of *enterItem* specifies that 1) a new object *sli* of class *SalesLineItem* was created, 2) the *currentSaleLine* was linked to the new created object *sli*, 3) the links among *currentSale*, *sli* and *item* were formed, 4) the attributes *Quantity* and *Subamount* of *sli* were set to the value of input variable *quantity* and *item.Price*quantity*, 5) the attribute *StockNumber* of *item* was reduced by the number of *quantity*, 6) the new created object *sli* was added in the object list *SaleLineItem*, 7) the output of system operation *enterItem()* was *true*.

Note that system operations may manipulate the same objects in a system sequence diagram of a use case. OCL allows to access the new created object in the same use case. For example, the object *currentSale* of the class *Sale* is created by system operation *makeNewSale()* of the use case *processSale*. It can be reused in the contract of the system operation *enterItem()*.

3 System Operation Decomposition

In this section, we first present a collection of primitive object-oriented operations and then introduce transformation rules and algorithms that automatically decompose a system operation to primitive operations. Finally, we present an example to show how the transformation rules and algorithms work.

3.1 Primitive Operations

Referring to atomic actions for manipulation tables in relational databases, we introduce a collection of primitive object-oriented operations of the object-oriented system for system operation decomposition in Table 1, which covers all manipulations to a) objects, b) the attributes of objects, and c) the links of objects.

Table 1
Primitive Operations

	Primitive Operation	Return Type
Object	***findObject***(ClassName:String, condition:String)	Object
	findObjects(ClassName:String, condition:String)	Set(Object)
	createObject(ClassName:String)	Object
	addObject(ClassName:String, ob:Class)	Boolean
	releaseObject(ClassName:String, ob:Class)	Boolean
Attribute	***getAttribute***(ob:Class, attriName:String)	PrimeType
	setAttribute(ob:Class, attriName:String, mathExp:String)	Boolean
Link	***findLinkedObject***(o:Class, assoName:String, condition:String)	Object
	findLinkedObjects(o:Class, assoName:String, condition:String)	Set(Object)
	addLinkOnetoMany(ob:Class, assoName:String, addOb:Class)	Boolean
	addLinkOnetoOne(ob:Class, assoName:String, addOb:Class)	Boolean
	removeLinkOnetoMany(ob:Class, assoName:String, removeOb:Class)	Boolean
	removeLinkOnetoOne(ob:Class, assoName:String)	Boolean

Objects: The following primitive operations are used to manipulate objects. An object or objects can be retrieved through primitive operation *findObject()* or *findObjects()* with a class name and a query condition. An object can be created by primitive operation *createObject()* with a class name, and then the created object can be added to the system through primitive operation *addObject()* with providing a class name and an object reference *ob*. Primitive operation *releaseObject()* can be used to delete an object from the system by providing a class name and an object reference *ob*.

Attributes of Objects: The next two primitive operations are used for getting and setting the value of an attribute. Primitive operation *getAttribute()* can retrieve the value of an attribute of an object by providing an object reference *ob* and the name of attribute. The value of an attribute of an object can be changed by the primitive operation *setAttribute()* with an object reference *ob*, the name *attriName* of an attribute, and a math expression *mathExp*.

Links of Objects: The links can be used to find the linked objects. Note that we use different primitive operations to manipulate links corresponding with two different types of the association of classes (one-to-one and one-to-many relations). By providing an object reference *ob*, an association name *assoName*, and an condition, primitive operations *findLinkedObject()* and *findLinkedObjects()* can retrieve the linked object or objects. An link from object *ob* to object *addOb* can be formed by invoking primitive operation *addLinkOnetoMany()* or *addLinkOnetoOne()* with the name of the association, and the object reference *ob* and *addOb*. Primitive operation *removeLinkOnetoMany()* or *removeLinkOnetoOne()* can be used to break the link by providing an object reference *ob*, the name of the association *assoName*. Primitive operation *removeLinkOnetoMany()* requires providing a reference to the target object *removeOb*, and the reference indicates which link will be removed from the object.

3.2 Transformation Rules

We have introduced the contract of system operation and primitive operations. In this subsection, we present how to transform an OCL contract to primitive operations. Transformation rules will be presented in this form:

$$Rule : \frac{\textit{OCL Expression}}{\textit{Primitive Operation in Java Code}}$$

The transformation rule contains two parts: the above section is an OCL Expression, and the bottom part is a primitive operation in Java code. The transformation rules form OCL expressions and primitive operations as pairs, and those pairs are the foundation of the transformation algorithm for automatic system operation decomposition. In short, we refine the original 10 transformation rules in the previous work [24] [25] to 26 transformation rules that cover all the primitive operations corresponding in the contract sections of definition, pre-condition, and post-condition. The details of comparisons are shown in section 7.3. Follow the convention of OCL contract, we present those transformation rules into three parts: a) definition transformation, b) pre-condition transformation, and c) post-condition transformation.

3.2.1 Definition Section Transformation

The definition section of the contract specifies that the objects are further used in pre-condition and post-condition. In object-oriented systems, objects can be reached through the links of objects, which are defined by the associations of the classes. In our approach, we build pure fabricated class *EntityManager* (EM) to record all the references of objects in the system. Therefore, objects can be found through *EntityManager* with a query condition, and then other related objects can be reached through the links of the founded objects. In definition section, seven transformation rules are presented, which involve the primitive operations *findObject()*, *findObjects()*, *findAssociationObject()*, and *findAssociationObjects()* of Table 1.

$$\mathbf{R_1} : \frac{\textit{obs}:\textbf{\textit{Set}}(\textit{ClassName})=\textit{ClassName}.\textbf{\textit{allInstances}}()}{\textit{List<ClassName> obs = EM}.\textbf{\textit{findObjects}}(\textit{ClassName}:\textit{String})}$$

The rule R_1 shows finding all the objects *obs* of the class named *ClassName* in the system. This OCL expression is mapped to the primitive operation *findObjects()*, and the found objects are assigned to a list reference *obs* of the class *ClassName*.

$$\mathbf{R_2}: \frac{obs{:}\textbf{Set}(ClassName){=}ClassName.\textbf{allInstances}()\rightarrow\textbf{select}(o \mid \textbf{conditions}(o))}{List\langle ClassName\rangle\ obs = EM.\textbf{findObjects}(ClassName{:}String,\ conditions(o){:}String)}$$

$$\mathbf{R_3}: \frac{ob{:}ClassName = ClassName.\textbf{allInstances}()\rightarrow\textbf{any}(o \mid \textbf{conditions}(o))}{ClassName\ ob = EM.\textbf{findObject}(ClassName{:}String,\ conditions(o){:}String)}$$

Based on rule R_1, the rules R_2 and R_3 are introduced to find objects *obs* or an object *ob* from all the instances of the class named *ClassName* with the constraints *condition(o)* by using OCL keyword *select* or *any*. The constraint *condition(o)* is a logic formula about object *o*, which composites atomic formulae of pre-condition with the logical operators *and* and *or*. The OCL expressions of R_2 and R_3 are mapped to the primitive operation *findObjects()* or *findObject()*, and then assign the found objects or object to a reference list *obs* or a reference *ob* of class *ClassName*.

For example, when a cashier scans the product by invoking the system operation *enterItem()*, the system will find the object *item* with the specific *barcode*, which has the same value retrieved from the scanner. This part functional semantics are specified in the definition section of contract *enterItem()*. It will map to the primitive operation *findObject()* by R_3 as follows:

$$\frac{item{:}Item = Item.\textbf{allInstances}()\rightarrow\textbf{any}(i \mid i.Barcode = barcode)}{Item\ ob = EM.\textbf{findObject}(\text{"Item"},\ \text{"i.Barcode = barcode"})}$$

Note that the condition *i.Barcode = barcode* will be further mapped by R_{12}. Once we find the target objects, the related objects can be found through the links of the target objects. Those transformations present in the next four rules:

$$\mathbf{R_4}: \frac{o{:}ClassName = ob.assoName}{ClassName\ o = EM.\textbf{findLinkedObject}(ob{:}Class,\ assoName{:}String)}$$

$$\mathbf{R_5}: \frac{obs{:}Set(ClassName) = ob.assoName}{List\langle ClassName\rangle\ obs = EM.\textbf{findLinkedObjects}(ob{:}Class,\ assoName{:}String)}$$

The rules R_4 and R_5 show finding the linked object through *ob.assoName*, where the multiplicity of the association may be one-to-one or one-to-many relationship. If *assoName* is one-to-one association, *ob.assoName* will return a object reference *o* to the object linked with object *ob*; otherwise *ob.assoName* returns a reference list *obs*. Therefore, the OCL expressions of R_4 and R_5 are mapped to primitive operations *findLinkedObject()* and *findLinkedObjects()* with the input variables: an object reference *ob* and an association name *assoName* of object *ob*, and then assign the found object or object list to the reference *o* or reference list *obs* correspondingly.

$$\mathbf{R_6}: \frac{obs{:}Set(ClassName) = ob.assoName\rightarrow\textbf{select}(o \mid \textbf{conditions}(o))}{List\langle ClassName\rangle\ obs = EM.\textbf{findLinkedObjects}(ob{:}Class,\ assoName{:}String,\ preconditions(o){:}String)}$$

$$\mathbf{R_7}: \frac{o{:}ClassName = ob.assoName\rightarrow\textbf{any}(o \mid \textbf{conditions}(o))}{ClassName\ ob = EM.\textbf{findLinkedObject}(ob{:}Class,\ assoName{:}String,\ conditions(o){:}String)}$$

Based on rule R_5, the rules R_6 and R_7 apply OCL expression *select* and *any* with a query condition *condition(o)* to *ob.assoName* with navigation $\rightarrow$ to filter the specific objects from the associated objects.

For example, we find an object *SaleLineItem* under the current sale *s* with the quantity number great than 2. The corresponding OCL contract and R_7 transformation are:

$$\frac{sli{:}SaleLineItem = s.ContainedSaleLine\rightarrow\textbf{any}(sl \mid sl.Quantity > 2)}{SaleLineItem\ sli = EM.\textbf{findLinkedObject}(s,\ \text{"ContainedSaleLine"},\ \text{"sl.Quantity > 2"})}$$

The same as the previous example, the condition *sl.Quantity > 2* will make a further transformation by rule R_{12}.

3.2.2 Pre-condition Transformation

The pre-condition section of the contract specifies the states of the objects before execution of system operation. The related objects have been specified in the definition section of the contract. The pre-condition section specifies the constraints on those objects before execution of system operation. The pre-condition transformation maps those constraints to the primitive operations, which involves *getAttribute()* and a set of basic checking operations under the fabricated class *StandardOPs*. In short, eight transformation rules from R_8 to R_{15} are presented in the pre-condition section.

$$\mathbf{R_8}: \frac{ob.oclIsUndefined() = bool}{\textbf{StandardOPs.oclIsUndefined}(ob{:}Class,\ bool{:}Boolean)}$$

$$\mathbf{R_9}: \frac{var.oclIsTypeOf(type)}{\textbf{StandardOPs.oclIsTypeOf}(«var»,\ type{:}String)}$$

$$\mathbf{R_{10}}: \frac{obs.isEmpty() = bool}{\textbf{StandardOPs.isEmpty}(obs{:}Set(Class),\ bool{:}Boolean)}$$

The rules R_8, R_9, and R_{10} map the basic OCL checking expression about object and object list to the primitive operations. The contract of R_8 checks that the reference *ob* does not refer to an object. The OCL expression of R_9 is used to check that the variable *var* conforms the specific *type*, in which the *var* is a variable of primitive type, an object reference, or a reference list. The contract of R_{10} is to check the object list *obs* is empty.

$$\mathbf{R_{11}}: \frac{obs.size()\ op\ \ mathExp}{\textbf{StandardOPs.size}(obs{:}Set(Class))\ «op»\ «mathExp»}$$

$$\mathbf{R_{12}}: \frac{ob.AttriName\ op\ varPM}{\textbf{getAttribute}(ob{:}Class,\ attriName{:}String)\ «op»\ «varPM»}$$

The rules R_{11} and R_{12} map the logic expression from OCL to Java. The *op* is an infix comparison operator in OCL. «op» represents the corresponding Java code of *op* after compilation. Most operators of OCL and Java are the same, except equal operator = and non-equal <> of OCL. Those operators will be compiled to Java operators == and *!=*. Furthermore, when we compare between two variables *s1* and *s2* of *String* type, the above operators will be compiled to Java code *s1.equals(s2)* and *!s1.equals(s2)* respectively. The rule R_{11} gets the size of list, and then checks the constraint of the list size against a math expression *mathExp*. The *mathExp* may contain numbers, variables, operators, functions and brackets. The rule R_{12} firstly retrieves the value of the attribute *attriName* of the object *ob* by transforming *ob.attribute* to the primitive operation *getAttribute*, and then checks the constraint on the value of the attribute through the expression *op varPM*. The *varPM* is a variable of primitive type or a math expression.

For example, the pre-condition *item.StockNunmber > 0* of contract *enterItem*. ">" is the *op* expression. "0" is the *varPM* expression. This pre-condition will be compiled to Java code *item.getStockNumber() > 0* by rule R_{12}:

$$\frac{sl.Quantity > 2}{\textbf{getAttribute}(sl,\ \text{"Quantity"}) > 2}$$

The rules R_{13} and R_{14} use the OCL expression *ClassName.allInstances()* of the definition section to find all objects of class *ClassName*, and to check whether the specific object *ob* is included in or excluded from the object list. Therefore, *ClassName.allInstances()* is mapped to

EM.findObjects(ClassName), and then the founded object will be passed to the operations *includes()* or *excludes()* of the class *StandardOPs*.

$$\mathbf{R_{13}} : \frac{\textit{ClassName.}\textbf{\textit{allInstances}}\textit{()}\rightarrow\textbf{\textit{includes}}\textit{(ob)}}{\textbf{\textit{StandardOPs.includes}}\textit{(EM.findObjects(ClassName), ob:Class)}}$$

$$\mathbf{R_{14}} : \frac{\textit{ClassName.}\textbf{\textit{allInstances}}\textit{()}\rightarrow\textbf{\textit{excludes}}\textit{(ob)}}{\textbf{\textit{StandardOPs.excludes}}\textit{(EM.findObjects(ClassName), ob:Class)}}$$

$$\mathbf{R_{15}} : \frac{\textit{ClassName.}\textbf{\textit{allInstance}}\textit{()->}\textbf{\textit{isUnique}}\textit{(o:ClassName | o.AttriName)}}{\textbf{\textit{StandardOPs.isUnique}}\textit{(ClassName:String, AttriName:String)}}$$

The last rule R_{15} of precondition section maps the unique detection expression to the operation *isUnique()* of class *StandardOPs*. This opeartion will get all the objects of class *ClassName*, and check the specific attribute *AttriName* has the unique value or not. Note that 1) OCL expressions of pre-condition section can also be used to specify the invariants of the system. Therefore, the transformation rules related pre-condition can also apply to the invariants transformation. 2) The implementation of class *StandardOPs* is not presented in this paper, the details can be found on the GitHub repository[2].

3.2.3 Post-condition Transformation

The post-condition section of the contract specifies the states of objects after execution of the operation. Concretely, the post-condition specifies that the object was created, added, and released, the association was formed and broken, the state of attribute was updated. The related primitive operations are *createObject()*, *addObject()*, *releaseObject()*, *addOnetoManyAssociation()*, *addOnetoOneAssociation()*, *removeOnetoManyAssociation()*, *removeOnetoOneAssociation()*, and *setAttribute()*. Eleven transformation rules of the post-condition are presented in post-condition section. The first rule R_{16} is

$$\mathbf{R_{16}} : \frac{\textbf{\textit{let}}\ \textit{ob:ClassName}\ \textbf{\textit{in}}\ \textit{ob.}\textbf{\textit{oclIsNew}}\textit{()}}{\textit{ClassName ob = EM.}\textbf{\textit{createObject}}\textit{(ClassName:String)}}$$

The OCL expression *let..in* of the rule R_{16} describes the scope of object *ob*. The *ob.oclIsNew()* specifies that the object *ob* was created after the execution of system operation. In order to make system into this state, the rule R_{16} maps *let..in* and *oclIsNew()* to the primitive operation *createObject()*, and then assigns the created object to the reference *ob* of the class named as *ClassName*. For example, the post-condition of system operation *enterItem()* specifies that object *sli* of class *SaleLineItem* was created. By applying the rule R_{16}, this OCL expressions are mapped to:

$$\frac{\textbf{\textit{let}}\ \textit{sli:SaleLineItem}\ \textbf{\textit{in}}\ \textit{sli.}\textbf{\textit{oclIsNew}}\textit{()}}{\textit{SaleLineItem sli = EM.}\textbf{\textit{createObject}}\textit{("SaleLineItem")}}$$

The OCL expression of the rules R_{17} and R_{18} have already been used in the pre-condition to check whether the instance list of class *ClassName* includes or excludes the object *ob* or not. Those OCL expressions in the post-condition indicate that the object list of class *ClassName* includes and excludes the object *ob* after the execution of the system operation.

$$\mathbf{R_{17}} : \frac{\textit{ClassName.}\textbf{\textit{allInstances}}\textit{()}\rightarrow\textbf{\textit{includes}}\textit{(ob)}}{\textit{EM.}\textbf{\textit{addObject}}\textit{(ClassName:String, ob:Class)}}$$

$$\mathbf{R_{18}} : \frac{\textit{ClassName.}\textbf{\textit{allInstances}}\textit{()}\rightarrow\textbf{\textit{excludes}}\textit{(ob)}}{\textit{EM.}\textbf{\textit{releaseObject}}\textit{(ClassName:String, ob:Class)}}$$

It is necessary to add or delete the object *ob* to or from the object list to make the system state conforming the post-condition. Therefore, the *includes()* and *excludes()* with *allInstances()* expression in the post-condition are mapped to the primitive operations *addOjbect()* and *releaseObject()*.

$$\frac{\textit{SalesLineItem.}\textbf{\textit{allInstances}}\textit{()}\rightarrow\textbf{\textit{includes}}\textit{(sli)}}{\textit{EM.}\textbf{\textit{addObject}}\textit{("SalesLineItem", "sli")}}$$

For example, if we apply the rule R_{17} to post-condition of system operation *enterItem()*, the post-condition specifies that the created object *sli* was included in object list of class *SalesLineItem* will be mapped to primitive operation *addObject()* with parameter class name *SalesLineItem* and object name *sli*.

$$\mathbf{R_{19}} : \frac{\textit{ob.assoName}\rightarrow\textbf{\textit{includes}}\textit{(addOb)}}{\textbf{\textit{addLinkOnetoMany}}\textit{(ob:Class, assoName:String, addOb:Class)}}$$

$$\mathbf{R_{20}} : \frac{\textit{ob.assoName}\rightarrow\textbf{\textit{excludes}}\textit{(removeOb)}}{\textbf{\textit{removeLinkOnetoMany}}\textit{(ob:Class, assoName:String, removeOb:Class)}}$$

OCL expression *includes* and *excludes* can also be applied for the association *ob.assoName*. That means the link has been formed or broken after executing the system operation. Therefore, the above rules R_{19} and R_{20} map those expressions to the primitive operations *addLinkOnetoOne()* and *removeLinkOnetoMany()*.

$$\mathbf{R_{21}} : \frac{\textit{ob.assoName = addOb}}{\textbf{\textit{addLinkOnetoOne}}\textit{(ob:Class, assoName:String, addOb:Class)}}$$

The one-to-one association is implemented as an attribute of the class. The rule R_{21} describes if the post-condition specifies that the one-to-one association *ob.assoName* has a linked object *addOb*. The primitive operation *addLinkOnetoOne()* will be executed to make system to satisfy this post-condition. For examples, the links between current sale and the created object *SalesLineItem* object were formed in following transformations:

$$\frac{\textit{currentSale.ContainedSalesLine}\rightarrow\textbf{\textit{includes}}\textit{(sli)}}{\textbf{\textit{addLinkOnetoMany}}\textit{(currentSale, "ContainedSalesLine", sli)}}$$

$$\frac{\textit{sli.BelongedSale = currentSale}}{\textbf{\textit{addLinkOnetoOne}}\textit{(sli, "BelongedSale", currentSale)}}$$

Note that the association from class *Sale* to *SalesLineItem* is multiple, and the reversed association is a one-to-one association. Therefore, the rule R_{19} will be triggered for transformating OCL expression to primitive operation *addLinkOneToMany()* with an object *currentSale*, the association name *ContainedSalesLine*, and the created object *sli* of class *SalesLineItem*. The rule R_{21} is trigger for transformation OCL expression to primitive operation *addLinkOneToOne()* with the object reference *sli* of class *SalesLineItem*, the association name *BelongedSale*, and the reference *currentSale* to current sale object.

$$\mathbf{R_{22}} : \frac{\textit{ob.assoName = }\textbf{\textit{null}}}{\textbf{\textit{removeLinkOnetoOne}}\textit{(ob:Class, assoName:String)}}$$

The rule R_{22} presents if the post-condition specifies that one-to-one association *ob.assoName* has no associated object, the primitive operation *removeLinkOnetoOne()* will be executed to broke the link that makes the system state conforming this post-condition.

$$\mathbf{R_{23}} : \frac{\textit{ob.attriName = mathExp}}{\textbf{\textit{setAttribute}}\textit{(ob:Class, attriName:String, «mathExp»:PrimeType)}}$$

The rule R_{23} indicates that the attribute of the object *ob.attriName* is equal to the math expression *mathExp* in the post-condition. The primitive operation *setAttribute* should be executed to conform this condition. For example, post-condition of *enterItem()* specifies the sub-amount of

2. https://github.com/RM2PT/CaseStudies/wiki/StandardOP

SalesLineItem is computed from the price of the item and the quantity of the item.

$$\frac{\textit{sli.Subamount = item.Price * quantity}}{\textbf{\textit{setAttribute}}\textit{(sli, "Subamount", }\textbf{\textit{getAttribute}}\textit{(item, "Price")*quantity)}}$$

This OCL expression is mapped to primitive operations *setAttribute()* and *getAttribute()*. Primitive operation *getAttribute()* is used to get the price of the item, then the price is multiplied with quantity. Finally, the evaluated result is set to the attribute *Subamount* of object *SalesLineItem* named *sli*.

$$\mathbf{R_{24}} : \frac{\textit{obs->}\textbf{\textit{forAll}}\textit{(o:ClassName | o.AttriName = mathExp)}}{\textbf{\textit{for}}\textit{ (ClassName o:obs) \{}\textbf{\textit{setAttribute}}\textit{(o:Class, AttriName:String, «mathExp»:PrimeType);\}}}$$

We can use *forAll* to literately specify the state of the objects in the post-condition. The rule R_{24} extends the rule R_{23} from specifying the state of a single object to a list of objects, that maps the OCL expression *forAll* to Java *for* loop expression.

$$\mathbf{R_{25}} : \frac{\textbf{\textit{return}}\textit{ = var}}{\textbf{\textit{return}}\textit{ «var»;}}$$

The rule R_{25} specifies the state of return variables after system operation execution. The return variable *var* can be a variable of prime type, a reference to an instance of class, or to an instance list of class. It will be mapped to the Java code correspondingly.

$$\mathbf{R_{26}} : \frac{\textbf{\textit{ThirdPartyServices}}\textit{.opName(vars)}}{\textbf{\textit{service}}\textit{.opName(«vars»)}}$$

The rule R_{26} is a special transformation rule that specifies the third-party APIs such as *cardPayment()* and *sorting()* used in the post-condition. This rule makes our approach extensible to invoking the third-party API library. It will transform the operation *opName* and parameters *vars* into the operation invoking in Java code.

3.3 Transformation Algorithm

We have already presented all transformation rules for system operation decomposition. It is time to introduce the transformation algorithm that transforms the contract of system operation into primitive operations. Transformation algorithm is presented in Algorithm 1. It requires an OCL expression and a tag as input parameters, and return the mapped primitive operations. Note that the tag marks which section of the contract OCL expression comes from. The details of transformation algorithm are that initializing the result set *rs* as empty set and index *i* to zero, and then parsing input *OCLExpression* as a set of sub OCL expressions *sub-formulas* and a set of logic connectors *connectors*, and using *lastn* to record the last element of *sub-formulas*, and iterating each formula *s* of the *sub-formulas* set. In each iteration, algorithm initializes *num* to zero, which indicates whether the appropriate rule is found. According to the value of the tag, different rules are used to check whether the formula *s* is matched, then assign the match rule number to *num* if any rule matched the formula *s*. The formula *s* will be mapped to primitive operation according to the rule. The transformation result is saved in *r*. Note that the OCL expression of pre-condition is mapped into primitive operations connected with Java logic expressions. Therefore, the algorithm appends the transformation result *r* with the connector *connectors[i]* if the tag is *pre-condition* and the index *i* does not point to the last element. Otherwise, the transformation result *r* will be appended to the final result *rs* with a broken line.

```
Input  : OCLExpression, Tag
Output: Primitive Operations
begin
    rs ← ∅;
    i ← 0;
    sub-formulas ← parse(OCLExpression);
    connectors ← parseConnector(OCLExpression);
    lastn ← len(sub-formulas) - 1;
    for s ∈ sub-formulas do
        num ← 0;
        switch Tag do
            case definition do
                num ← matchRule1to7(s);
            end
            case pre-condition do
                num ← matchRule8to15(s);
            end
            case post-condition do
                num ← matchRule16to26(s);
            end
        end
        if num != 0 then
            r ← transform(s, num, Tag);
            if tag == "pre-condition" and i != lastn then
                rs.append(r, connectors[i]);
            else
                rs.append(r, "linebreaks");
            end
        else
            rs.append("transformation error for
              sub-formula:", s);
        end
        i++;
    end
    return rs;
end
```

Algorithm 1: Transformation Algorithm

If no rule is matched to the formula *s*, an error message will be appended to the final result *rs* with the formula *s*. The nature of requirements model implies that, in general, a requirements model may contain non-executable elements, because it focuses on what the system should do rather than how it does it. This error message will help to validate the requirements when generating the prototype. In the end of the iteration, *i* is increased to point to the next sub-formula. After the iteration, the final result *rs* will be returned.

3.4 Transformation Example

We have already introduced the rules for transformation the contracts of system opeartions to the primitive operations. This subsection will present an example of *enterItem()* of use case *processSale* to explain how it work. Note that if a transformation rule is used more than once, we only explain the transformation at the first time. The signature of contract *enterItem* is compiled to the operation signature in Java.

In the definition section, the rule R_3 maps the OCL expression to the primitive operation *findObject()* to find

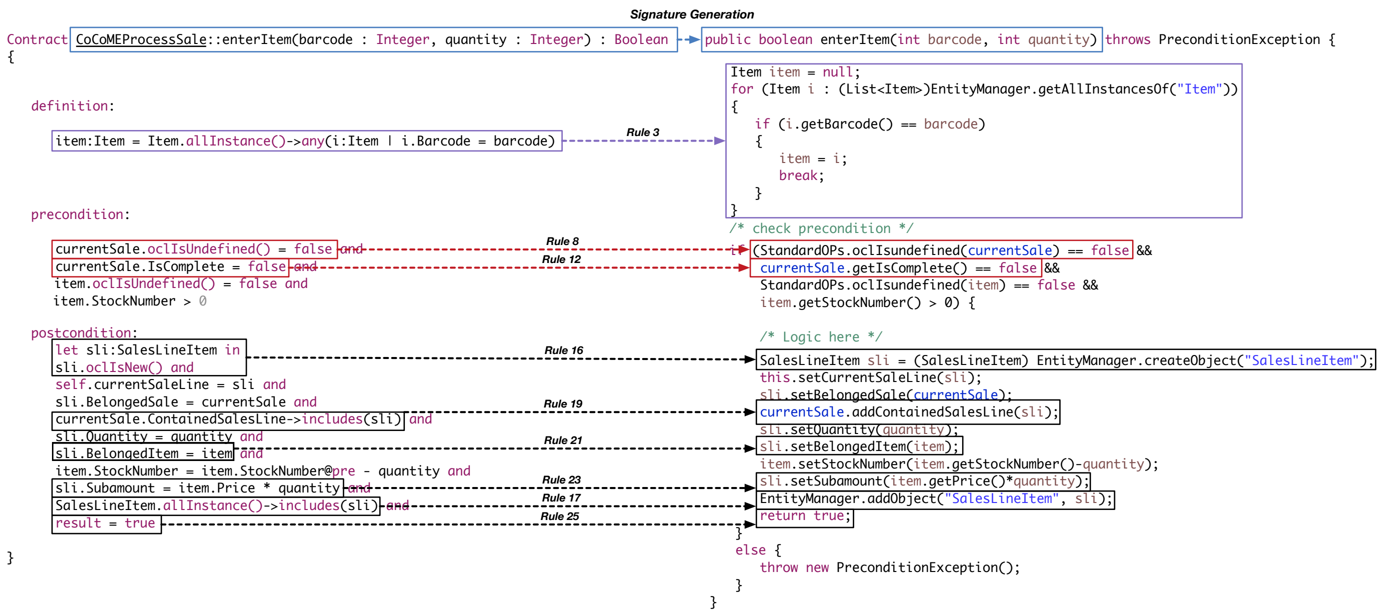


Figure 2. A Transformation Example for the Contract of *enterItem()*

the object *item* with the input barcode. In the pre-condition section, the rules R_8 and R_{12} map the OCL contract to check whether the objects *currentSale* and *item* are undefined, the value of the attribute *IsComplete* is false or not, and the attribute *StockNumber* of object *item* is great than zero.

In the post-condition section, 1) the rule R_{16} maps *let..in* and *oclIsnew()* to the primitive operation *createObject()* to create object *sli* of the class *SaleLineItem* for the scanned product. 2) The rule R_{19} maps *includes* OCL contract to the primitive operation *addOnetoManyAssociation()* by adding the link of the new created object *sli* to the current sale object *currentSale*. 3) The rule R_{21} maps *sli.BelongedItem* = *item* to the primitive operation *addOnetoOneAssociation()* by linking the association *BelongedItem* of object *sli* to the object *item*. 4) The rule R_{23} maps *sli.Subamount* = *item.Price*quantity* to the primitive operations *setAtrribute()* and *getAtrribute()* by setting the attribute *Subamount* of object *sli* to the sub amount value of *item.getPrice()* * *quantity*. 5) The rule R_{17} maps the *includes* expression to the primitive operation *addObject()* by adding the reference *sli* to the records of class *SalesLineItem*. 6) The rule R_{25} maps the result expression to the Java return expression. All rules will be repeatedly applied for transforming until all the OCL contract are mapped to the primitive operations with Java code. This section presented the transformation rules and algorithms for system operation decomposition. Next section will introduce how to generate prototype from the requirements model.

4 Prototype

Graphical User Interface (GUI) is one of the important parts of a prototype for customers and domain experts to validate their requirements. Model-View-Controller (MVC) [26] is one of the most widely used GUI architecture patterns in decades. In this paper, we apply the MVC architecture pattern to prototype, which decouples views and models to make a clear division between models and GUI elements. The prototype contains three modules: view, controller, and model as shown in Figure 3.

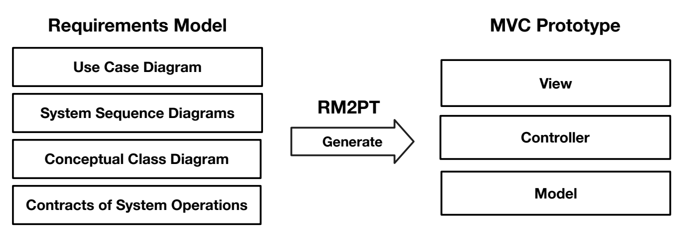


Figure 3. MVC Prototype Generation from Requirements Model

View: The view module of the prototype contains UI widgets for end-users to validate system operations and observe corresponding changes in the system state. It contains two UI panels: system functionality and system state. As an example, the system functionality panel of CoCoME is shown in Figure 4. It includes the widgets of *system operation lists* in the left side, and the *operation widgets* for passing the parameters to system operations and returning the result in the right side. *System operation lists* are generated from the use case diagram and the system sequence diagram. Moreover, the *operation widgets* for inputting the parameters of system operations are generated from the signatures of the contracts of system operations.

The system state panel is used to observe the state of objects of a system before and after executions of system operations. Figure 5 shows the state panel of CoCoME. The system state panel is generated from the conceptual class diagram. The left top side widget of *objects statistics* presents the name and the number of the objects for each class in the current state. The left bottom side widget of *the link of objects* shows the state of associations, which includes the source object, the target object, the name of the association, the number of the linked objects, and whether

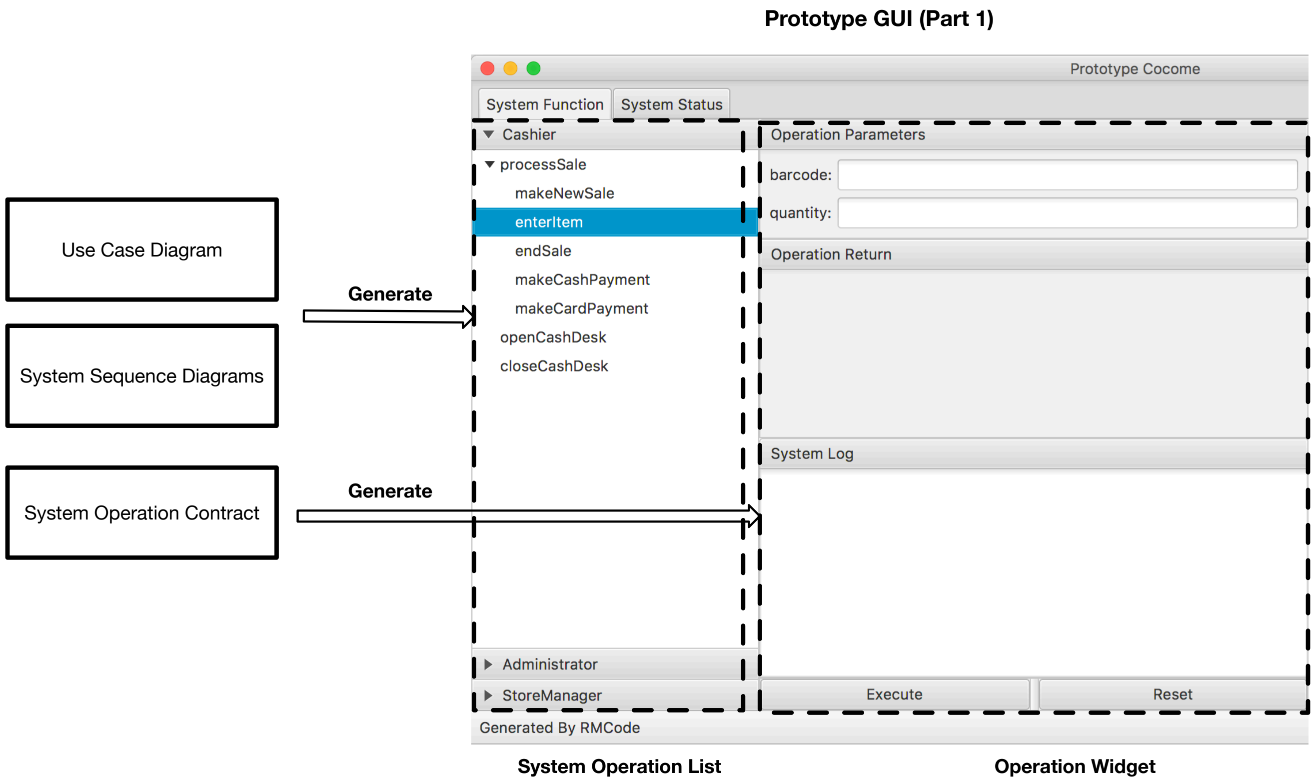


Figure 4. Prototype GUI for System Function

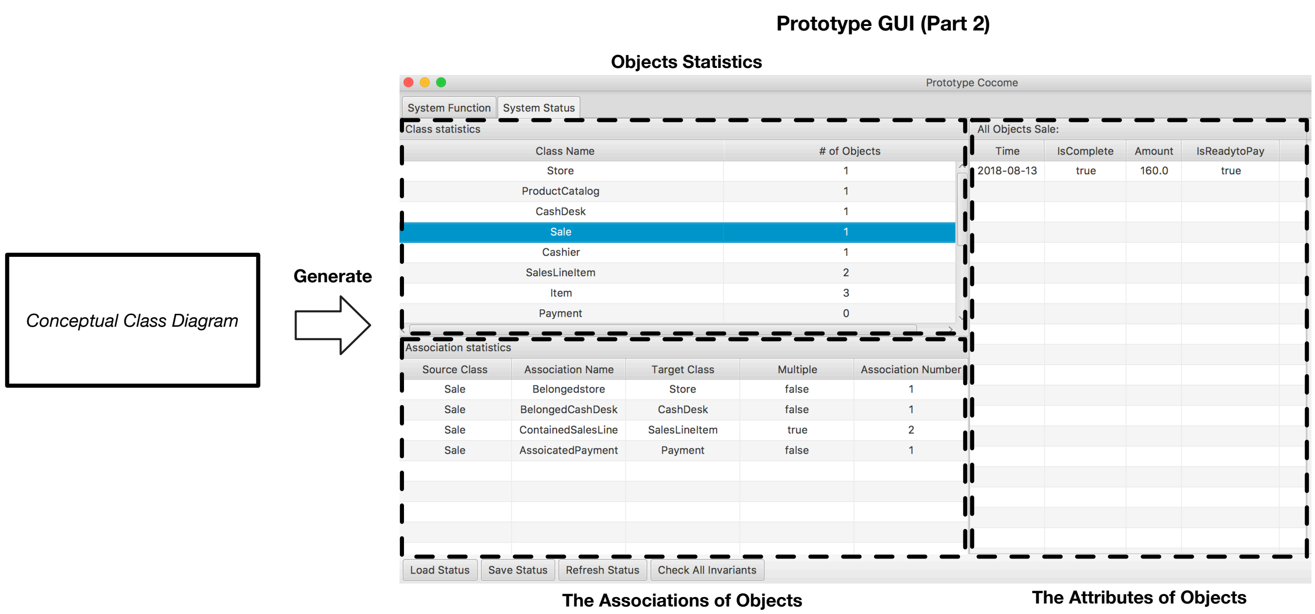


Figure 5. Prototype GUI for System State

this association is a one-to-one or one-to-many relationship. When users click a class entry on the left side, the state of the corresponding attribute will display on the right panel widget of *the attributes of objects*.

Controller: The controller module links the view and model modules, which makes UI events to trigger system operations. The controller listens to the events from UI widgets. When a specific event is captured, the controller retrieves the input parameters from the UI widget, and then deliver the parameters to the corresponding system operation in the model module of the prototype. Finally, the controller will update the UI widgets with the return result. For example, when a cashier clicks the *execute* button on the operation widget of Figure 4, the controller will capture a button clicked event generated by the UI widget, and then the controller will get the input parameters and then invoke the *enterItem()* operation. Finally, the controller will update the

UI widgets with the return result, and then the cashier will see the result from *operation return* panel of Figure 4.

Model: The model module is the core of the MVC architecture pattern. This module contains 1) the classes encapsulating system operations, which can be invoked by the controller with the parameters from UI widgets, and return the result to the controller for displaying on the view widgets, and 2) the classes generated from the conceptual class diagram. Note that we name the classes generated from the conceptual class diagram as *entity classes*, and the others as *fabricated classes*. We will show how to generate fabricated and entity classes in the remaining of this section.

4.1 Fabricated Class Generation

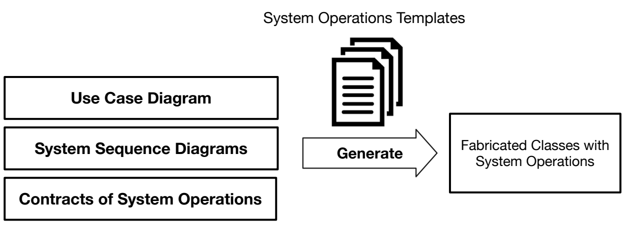


Figure 6. Code Generation for System Operation

Fabricated classes encapsulating system operations are generated from a use case diagram, system sequence diagrams, the contracts of system operations through system operation templates in Figure 6. We already discussed the system operation decomposition in the previous section. To implement the decomposed system operation, we not only need to transform the contracts of system operations into primitive operations through the transformation algorithm and rules but also need to orchestrate them to be a validated system operation like the decomposition example of *enterItem()* in Figure 2. Moreover, the implemented system operations must be encapsulated into classes. In this subsection, we introduce system operation templates to generate the implementations of the system operations from contracts, and how to encapsulate them into classes.

4.1.1 System Operation Template

Xtend [27] is a flexible and expressive template language. In this paper, we adopt Xtend to define the template. The template used to generate the implementation of system operations from the contract *c* is shown as follows:

```
/* operation signature */
public «c.opSign.returnType» «c.opSign.name»(
«FOR para : c.opSign.parameters SEPARATOR ','»
    «para.type» «para.name»
«ENDFOR») throws PreconditionException {

/* contract definition */
«IF c.definition != null»
    «c.definition.mapping»
«ENDIF»

/* check precondition */
if («c.precondition.mapping») {

    /* contract post-condition*/
    «c.postcondition.mapping»

/* result return */
«IF c.opSign.returnType != null»
    return «returnName»;
«ENDIF»
else {
    throw new PreconditionException();
}}
```

Note that the template will be directly as the output result from the prototype generation, except interpolated expressions inside guillemets «». The expressions in «» will be dynamically interpreted in terms of requirements model. Accordingly, the expressions between *«IF»* and *«ENDIF»* will be interpolated when the condition is evaluated as *true*. The expressions between *«FOR»* and *«ENDFOR»* will be repeatedly interpolated through all the elements of the list. For example, if *c* represents the contract of *enterItem()*, *«c.opSign.name»* will be interpreted as the name of the system operation *enterItem()*.

The above template helps to implement the contract signature through generating the Java operation with the keyword *public*, the name of the system operation *«c.opSign.name»*, input variables *«para.name»* with the type *«para.type»*, and return type *«c.opSign.returnType»*. Then the template helps to generate the code *«c.definition.mapping»* for the definition section of the contract, if any. Then, the template generates the Java *if-else* control flow to check the pre-condition of system operation *«c.precondition.mapping»*, and executes the logic code *«c.postcondition.mapping»* to make the system satisfying the post-condition. In detail, 1) if the evaluation result of pre-condition is true, the code of the system operation will be executed, and then return the result by Java code *return «returnName»*, if any. 2) If the evaluation result of pre-condition is not true, an exception *PreconditionIsNotSatified* will be directly emitted without executing any code of the system operation. However, this template only helps to generate the skeleton code of system operation. The *«c.definition.mapping»*, *«c.precondition.mapping»* and *«c.postcondition.mapping»* are interpreted by transformation rules and algorithm in section 3.

4.1.2 System Operation Encapsulation

System operations are captured in system sequence diagrams of use cases. If we follow the suggestion of the expert pattern in General Responsibility Assignment Software Principles (GRASP) [22], system operations should be encapsulated to the classes (experts) in the conceptual class diagram. For example, the class *Item* contains attributes of *barcode* and *price*, *enterItem(barcode, quantity)* should be assigned to the class *Item*. However, entity classes have already held primitive operations for manipulating the attributes and associations. We expect to separate concerns into a different abstract level of class to achieve high cohesion and low coupling.

The mediator pattern of Gang of Four (GoF) [28] suggests an object that encapsulates a set of operations to promote loose coupling. That describes the situation the same as a use case, which includes high cohesion system operations related to one scenario of interactions between the actor and the system. Therefore, we generate a pure fabrication class for each use case and then encapsulate the

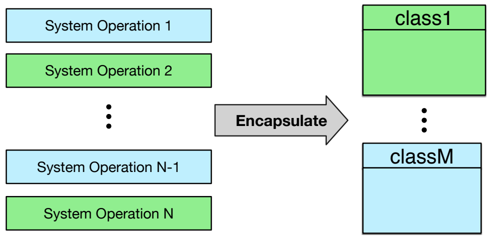


Figure 7. System Operation Encapsulation

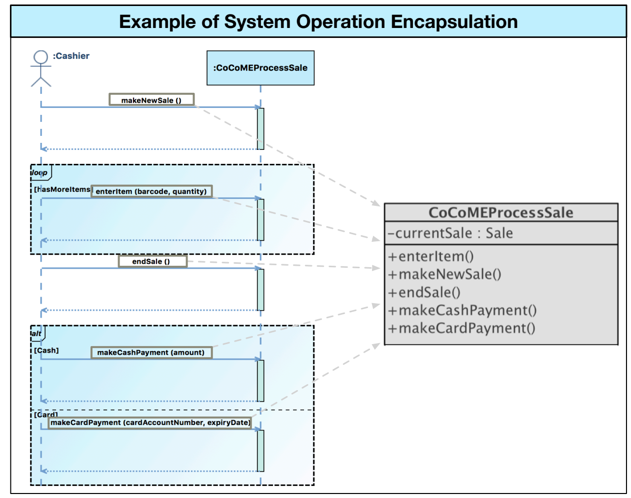


Figure 8. System Operation Encapsulation of Use Case ProcessSale

system operations of the use case to the fabrication class to achieve high cohesion.

Input : *ucd* - Use Case Diagram,
ssds - System Sequence Diagrams,
contracts - Contracts,
ssds - System Sequence Diagrams,
t_{so} - System Operation Template
Output: Fabrication Classes

```
/* Initialize ucClasses as empty set                */
for uc ∈ ucd do
    /* generate fabrication uc class                */
    ucClass ← generateClassSkeleton(uc);
    /* find uc related system sequence diagram      */
    ssd ← findSSD(uc, ssds);
    for op ∈ ssd do
        /* find system operation contract           */
        opCtr ← findContract(op, contracts);
        /* generate system operation sysOp()        */
        generate systemOperation() by template t_so and
          contract opCtr;
        encapsulate systemOperation() to class ucClass;
    end
end
```

Algorithm 2: Generation Algorithm for Fabricated Class and System Operations Encapsulation

Algorithm 2 generates the fabrication classes and encapsulates system operations into them. For each use case, a fabricated class *ucClass* is generated, which encapsulates the system operations from the corresponding system sequence diagram of the use case. Moreover, the implementation of system operation is generated by the contract of system operation and the system operation template t_{so}.

However, if a target problem contains many use cases, and most use cases only include one system operation, Algorithm 2 will generate many fabrication classes with just one operation. That breaks the design principle of high cohesion. The appropriate way is to define one fabrication class charging for several responsibilities to promote high cohesion. In the extreme cases, you can even specify only one pure fabrication class for all use cases. To deal with those situations, our CASE tool provides a mechanism that allows product managers and domain experts to decide how to define those fabrication classes for the use cases, and then automatically generates the implementation of those pure fabricated classes.

For example, the *currentSale* object of class *Sale* created by the system operation *makeNewSale()* is reused in the system operations *enterItem()* and *endSale()* of the use case *processSale*. Algorithm 2 generates a pure fabricated class *CoCoMEProcessSale* to encapsulate those system operations and the shared object *currentSale*. In detail, *CoCoMEProcessSale* is the pure fabrication class for encapsulating system operations *enterItem()*, *makeNewSale()*, *endSale()*, *makeCashPayment()*, *makeCardPayment()* and the object reference to *currentSale* in Figure 8.

This subsection illustrates how to generate fabricated classes and encapsulate the system operations. The next two subsections introduce how to generate entity classes from the conceptual class diagram as well as primitive operations.

4.2 Entity Class Generation

Entity classes generation is a necessary procedure to achieve the auto-prototyping from a requirements model, which is shown in Figure 9. It not only needs to generate the attributes and associations of the entity class but also requires to generate the implementation of primitive operations for setting and getting the attributes, finding linked objects, and adding and removing the links. We will show the details in the subsection.

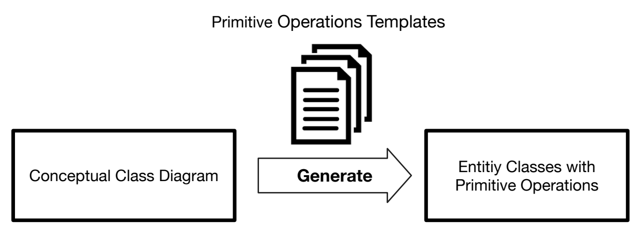


Figure 9. Code Generation for Primitive Operations

4.2.1 Generation Algorithm for Entity Classes

The information expert pattern of GRASP suggests assigning responsibilities to the information expert, and the expert class knows the information necessary to fulfill the responsibilities. Therefore, entity classes encapsulate primitive operations about getting and setting the values of the their attributes, forming and breaking the their links. For example, entity class *Item* has the *Price* attribute, therefore, primitive operation *getPrice()* is encapsulated in class *Item*. Entity class *SaleLineItem* has the *Quantity* attribute and the association *BelongedSale*, therefore, it encapsulates primitive operation *updateAttributeQuantity()* and *addAssociationTosale()*. The algorithm that generates the entity classes from a conceptual class diagram is shown in Algorithm 3.

Input : *ccd* - Conceptual Class Diagram
t_{ec} - Entity Class Template
t_{po} - Primitive Operation Templates
Output: Entity Classes
for *entity* $\in$ *ccd* **do**
  generate entity class skeleton by t_{ec};
  for *attribute* $\in$ *entity* **do**
    generate *getAttribute()* by t_{po};
    generate *setAttribute()* by t_{po};
  end
  for *association* $\in$ *entity* **do**
    if *Is-Multiple(association) == true* **then**
      generate *findLinkedObjects() by* t_{po};
      generate *addLinkOnetoMany() by* t_{po};
      generate *removeLinkOnetoMany() by* t_{po};
    else
      generate *findLinkedObject() by* t_{po};
      generate *addLinkOnetoOne() by* t_{po};
      generate *removeLinkOnetoOne() by* t_{po};
    end
  end
end

Algorithm 3: Generation Algorithm for Entity Classes

This algorithm takes a conceptual class diagram as input and generates entity classes with primitive operations. For each class in the conceptual class diagram, entity class skeleton is generated through the entity class template. Then, primitive operations about getting and setting each attribute of the entity class are generated. For each association of the class, a) if the association is a one-to-many relationship, primitive operations about finding linked objects, adding and removing link of one-to-many association are generated, b) if the association is a one-to-one relationship, primitive operations about finding the linked object, adding and removing link of one-to-one association are generated. Note that above entity skeleton and primitive operations are generated through the code templates, and we will introduce those templates in the remaining of this subsection.

4.2.2 Entity Class Template

The generation skeleton of entity classes is straightforward. Many UML tools support this feature. The template to generate an entity class *c* is:

```
/* Class Skeleton */
public class «c.name»

/* Class Inheritance */
«IF c.superClass != null»
   extends «c.superClass.Name»
«ENDIF» {

/* Attributes */
«FOR attribute : c.attributes»
   private «attribute.type» «attribute.name»;
«ENDFOR»

/* Associations */
«FOR assoc : c.associations»
   private
   «IF assoc.isIsmultiple»
      List<«assoc.class»> «assoc.name» =
         new LinkedList<«assoc.class»>();
   «ELSE»
      «assoc.class.name» «assoc.name»;
   «ENDIF»
«ENDFOR»

/* primitive operations templates */
}
```

The template defines a public class with the keyword *public* and the name *c.name* inside of «». If the class has a super-class, the text of *c.superClass.name* will be replaced by the name of the super-class along with the keyword *extends*. The attribute is declared by the keyword *private* with the type *attribute.type* and the name *attribute.name*. The association is also declared by the keyword *private* with a) a typed list *List<assoc.class>* and association name *assoc.name*, if the association is a one-to-many relation, b) an attribute with the name of associated class *assoc.class.name* and the type of associated class *assoc.class*, if the association is a one-to-one relation.

4.2.3 Primitive Operation Templates

The templates for primitive operations *getAttribute()* and *setAttribute()* are:

```
//Getting Attribute
public «attribute.type» get«attribute.name»() {
   return «attribute.name»;
}

//Setting Attribute
public void set«attribute.name»
         («attribute.type» «attribute.name») {
   this.«attribute.name» = «attribute.name»;
}
```

Getting an attribute can be defined as the public operation with the name *get«attribute.name»* and the type *«attribute.type»* of attribute, which is fulfilled by directly returning the attribute by using the keyword *return*. The setting attribute can be fulfilled by setting the value of attribute *this.«attribute.name»* with an input of typed variables.

The primitive operation templates for finding linked objects, linking a new object, and removing a linked object are:

```
//findLinkedObjects()
public List<«assoc.class»> get«assoc.name»() {
   return «assoc.name»;
}

//addLinkOnetoMany()
public void add«assoc.name»(«assoc.class» ob) {
   this.«assoc.name».add(ob);
}

//removeLinkOnetoMany()
public void remove«assoc.name»(«assoc.class» ob) {
   this.«assoc.name».remove(ob);
```

```
}
```

We use a reference list to store object references for the one-to-many association. Therefore, finding objects through link can be directly implemented by returning the reference list of the association. Forming a link can be fulfilled by invoking the adding operation of the list as well as the deleting operation for breaking a link.

```
//findLinkedObject
public «assoc.class» get«assoc.name»() {
    return «assoc.name»;
}

//addLinkOnetoOne() removeLinkOnetoOne()
public void set«assoc.name»(«assoc.class» ob) {
    this.«assoc.name» = ob;
}
```

The one-to-one association is implemented as an attribute with the type of associated class. Thus, the implementation of finding the object through a link of one-to-one association is the same as getting the value of an attribute. The implementation of adding and removing a link of an one-to-one association is the same as setting primitive operation of an attribute. For example, when removing a link of a one-to-one association, we can just set *null* as the value of the link.

4.3 EntityManager Generation

In object-oriented systems, the object can be eventually found through the links of the objects. However, we need to locate the entrance object then to start the finding procedures. The primitive operation *findObject()* is used to find the first entrance object, then other objects can be reached through the primitive operation *findLinkedObject()* of the links of objects. To implement *findObject()*, a pure fabrication class named as *EntityManager* is required, which contains all the references to the objects of entity classes as well as the primitive operations *findObject()* and *findLinkedObject()*. Moreover, the creator pattern of GRASP suggests assigning the creating responsibility to the class recording the instances of the created objects. *EntityManager* records all instances of entity classes. Therefore, the primitive operations *createObject()* and *releaseObject()* are assigned to *EntityManager*. In addition, *EntityManager* is required a global accessing point. According to the singleton pattern of GoF, we build *EntityManager* as a singleton class, which has the only one instance, and provide a global accessing point. In short, the generation algorithm of *EntityManger* is shown in Algorithm 4.

This algorithm takes a conceptual class diagram, *EntityManager* template, the templates of object-related primitive operation as input, generates *EntityManager* classes. We generate *EntityManager* skeleton through *EntityManager* template, and the implementation of primitive operations *findObject()*, *findObjects()*, *createObject()*, *addObject()* and *releaseObject()* inside of *EntityManager*. Note that *EntityManager* records all the object references of entity classes, and *addObject()* is used for adding the object into the records. We will show those templates in the remaining of this subsection.

Input : ccd - Conceptual Class Diagram
t_{em} - EntityManager Template
t_o - Primitive Operation Templates for Object
Output: EntityManager Class
begin
  `/* Generate EntityManager Skeleton */`
  generate *EntityManager* skeleton by ccd, t_{em};
  `/* Generation Primitive Operations */`
  generate *findObject() by* t_o;
  generate *findObjects()* by t_o;
  generate *createObject()* by t_o;
  generate *addObject()* by t_o;
  generate *releaseObject()* by t_o;
end

Algorithm 4: Generation Algorithm for EntityManager

4.3.1 EntityManager Template

The template for generating *EntityManager* skeleton is shown as follows:

```
/* EntityManager Template */
public class EntityManager {

    /* HashMap Object Records*/
    private static Map<String, List> AllInstance =
        new HashMap<String, List>();

    /* create object reference list */
    «FOR c : classes»
    private static List<«c.name»> «c.name»Instances =
       new LinkedList<«c.name»>();
    «ENDFOR»

    /* Put object reference list into Map */
    static {
    «FOR c : classes»
       AllInstance.put("«c.name»", «c.name»Instances);
    «ENDFOR»
    }

    /* Get all objects of the class */
    public static List getAllInstancesOf
       (String ClassName) {
       return AllInstance.get(ClassName);
    }
}
```

We use Java *HashMap* named *AllInstances* to record all the object reference lists. Each object reference list is implemented as a *LinkedList* of Java with name *«c.name»Instances*. *«c.name»* will be replaced by the name of classes in the conceptual class diagram. All references list of object are added to this *HashMap AllInstance*. After that, we can find the specific object reference list directly by the name of entity class.

4.3.2 Primitive Operation Templates for Finding Objects

By using *EntityManager*, we can do fine-grained search *findObject()* and *findObjects()* with a query condition *precondition(o)* on the reference list. The template of finding object is:

```
/* find object template */
«cName» target = null; //initialize target object
for («cName» o:
```

```
    EntityManager.getAllInstancesOf(«cName»)) {
    //finding the object satisfies the condition
    if («precondtion(o)») {
        target = o;
        return target;
    }
}
```

The template initializes the object reference *target* as *null*, and then iterates the object list by *getAllInstancesOf* to find the object *o* satisfying the search condition *precondition(o)* (refer the details of precondition transformation to section 3.2.2), Finally, the template assigns the finding object *o* to the object *target*, and returns the found object. The template *findObjects* is similar to *findObject*, which is:

```
/* find objects template */
List<«c.name»> targets = = new LinkedList<>();
    //initialize target object lists
for («c.name» o:
    EntityManager.getAllInstancesOf(«c.name»)) {
    //finding the object satisfies the condition
    if («precondtion(o)») {
        targets.add(o);
    }
}
return targets;
```

The differences is the *findObjects()* template initializes the *targets* as a linked list, adds the object *o* to the list *targets* when the object *o* satisfies the condition, and finally, returns the target list *targets*.

4.3.3 Templates for Creating, Adding and Releasing Object

The factory method pattern of GoF suggests defining an interface for object creation, but let sub-classes decide which class to instantiate. Therefore, we impact all the create object responsibilities in the *EntityManager*, and define a factory method that creates all objects of classes. This factory method can invoke a concrete creator that returns an instance of an entity class through the Java reflect mechanism. The primitive operation template for creating objects is:

```
/* create object template */
public static Object createObject(String cName) {
    Class c = Class.forName("EntityManager");
    Method m = c.getDeclaredMethod("create" + cName +
        "Object");
    return m.invoke(c);
}

«FOR c : classes»
public static «c.name» create«c.name»Object() {
    «c.name» o = new «c.name»();
    return o;
}
«ENDFOR»
```

The factory method *createObject* provides a single-point to create object, it will invoke the concrete *create«c.name»Object* method to create and return the object of class *«c.name»*. We also use this pattern to build the *addObject* and *deleteObject* template. The template for generating primitive operation *addObject()* is

```
/* add object template */
public static Object addObject
    (String cName, Object ob) {

    Class c = Class.forName("EntityManager");
    Method m = c.getDeclaredMethod("add" + cName +
        "Object", Class.forName(cName));
    return (boolean) m.invoke(c, ob);

}

«FOR c : classes»
public static boolean add«c.name»Object(«c.name» o) {
    return «c.name»Instances.add(o);
}
«ENDFOR»
```

Object *ob* and the name of class are passed to the factory method *addObject*. Then *addObject* uses Java reflect to invoke the concrete method *add«c.name»Object* to add the object *ob* to the object list *«c.name»Instances*. We use the same pattern to build the template *deleteObject*:

```
/* release object template */
public static boolean deleteObject
    (String cName, Object ob) {

    Class c = Class.forName("EntityManager");
    Method m = c.getDeclaredMethod("delete" + cName +
        "Object", Class.forName(cName));
    return (boolean) m.invoke(c, ob);
}

«FOR c : classes»
public static boolean delete«c.name»Object
    («c.name» o) {
    return «c.name»Instances.remove(o);
}
«ENDFOR»
```

5 REQUIREMENTS VALIDATION

Requirements validation is the process of checking whether requirements meet the customer's real needs. It is critically important because requirements errors will lead to extensive rework when those problems are discovered during the software developments. It mainly concerns about finding problems of requirements. There are different checking types in the requirements validation process. This paper focuses on validity and consistency checking. Validity checking focuses on whether the requirements reflect the real needs of stakeholders. Consistency checking focuses on whether a requirements model contains conflict and contradictory, especially for the contradictory constraints in the same contract of the system function. The remaining parts of this section will introduce how the proposed approach help to check validity and consistency.

5.1 Validity Checking

Validity checking focuses on checking whether the requirements reflect the real needs of stakeholders. To validate the requirements, the stakeholders of the target system must be identified correctly.

- **Stakeholders Identification** The stakeholder [29] is a person or organization who influences a system's requirements or impacted by that system. Typical stakeholders

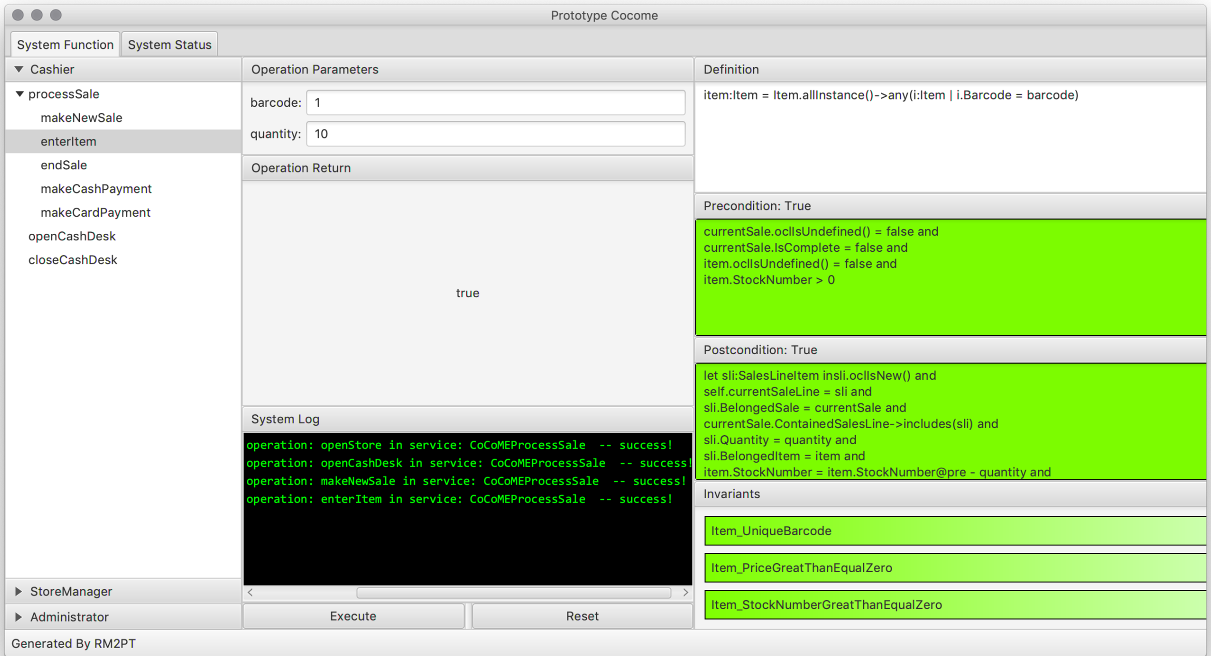


Figure 10. System Operation Execution with Contract Checking

are the end user, project sponsor, developer, tester, quality engineer, project manager, product manager, operator, and maintainer in a software project. In general, stakeholders are the persons or organizations, who 1) directly use the system, 2) involve in developing the system, 3) maintain the system, 4) have a financial interest, 5) constrain the system as regulators, and 6) have negatively affect on the system. In requirements modeling, the use case diagram has already identified end users, which directly interact with the target system. Those stakeholders are included in the generated prototype as well. Therefore, our approach is focusing on validating the stakeholders' requirements, who have already specified as the actors on the use case diagram. For example, the prototype in Figure 10 contains the cashier and store manager on the system function panel. The cashier and store manager can directly execute the use cases on the prototype to check whether requirements meet their real needs.

• **Start-up Objects** When the prototype is opened for the first time, it does not contain any object. That means system operations except adding objects cannot be executed because the pre-conditions of functions are not satisfied. The start-up objects must be added into the prototype before requirements validation. The prototype provides two options to load start-up objects. 1) Load objects from the checkpoint file. The users can click the *Load State* button in the system state panel to load the checkpoint files into the system to restore the status of system saved before. 2) Manually create start-up object by using the Create, Read, Update, and Delete (CRUD) operations provided by prototypes. RM2PT provides mechanisms to automatically generate the contracts and implementations of CRUD operations by marking conceptual classes. The administrator can use this mechanism to add the start-up objects into the prototype.

• **Use Case Execution** Prototype provides a function panel for use case executions. The stakeholders can pick up a system operation of a use case, type the input parameters, and click the execution button to check whether the system returns the expected results. For example, CoCoMe prototype in Figure 10 includes the widgets of system operation list on the left side, the system operation panel in the middle and the contracts panel on the right side. The cashier chooses the system operation *enterItem()*, type input parameters of the bar code 1 and quantity 10, then click the execution button, the prototype returns the value of true as expected. Besides, the prototype involves pre-conditions, post-conditions and invariants checking. When executing a system operation, the background color of pre-condition or post-condition panel will be red if a pre-condition or post-condition is not satisfied (or green when contract passed checking). Moreover, if the execution of the system function makes system state break the related invariants, the color of the invariant bar will become red from green. That indicates the errors in the requirements. Further inspections are required to locate the errors.

5.2 Consistency Checking

Consistency checking focuses on examining whether the requirements model contains conflict and contradictory, especially for the contradictory constraints in the same contract of the system function. As introduced above, the generated prototype can automatically indicate the contradictory in pre- and post-conditions and invariants of the contract when

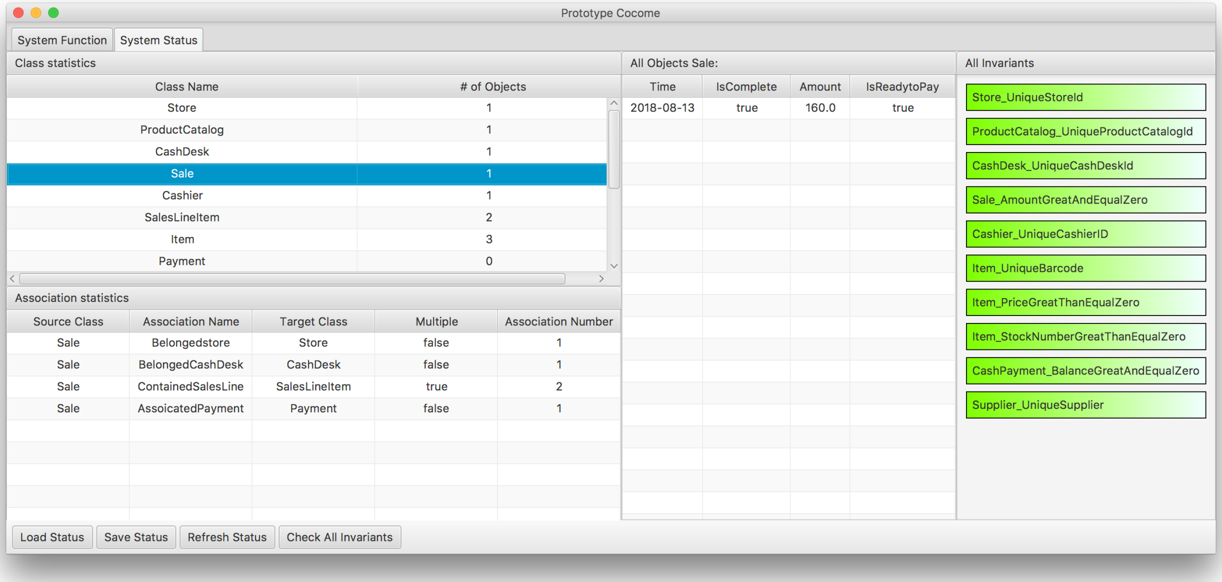


Figure 11. System State Observation

investigating use case executions. But the locating and fixing errors require more efforts. We provide the mechanism about state observations of the objects in the prototype to help developers to locate the contradiction.

• **State Observation** The location of the errors may be in the pre-condition, post-condition of the contract. Invariant checking can only indicate the errors. The state observation of the objects can help to analyze and locate the faults of the requirements. Figure 11 shows the state observation panel of CoCoME prototype. The left top side presents the name and the number of the objects for each class in the current system. For example, the number of the objects of the classes *Store*, *CashDesk*, *Sale* and *Item* in the CoCoME system. The left bottom side shows the state of association, which includes the source object, the target object, the name of the association, the number of the associated objects, and whether this association is a one-to-one or one-to-many relationship. For example, the Figure 11 shows the state of association *BelongedCashDesk* of the object *Sale*. The middle side of the panel shows the state of the attributes of the objects. E.g, the state of the attributes *Time*, *IsComplete*, *Amount* of the *Sale* objects. When clicking a class entry on the left side, the state of the corresponding attributes and associations will display on the middle and left bottom side of the panel. Furthermore, all the invariants of the system are listed on the right side of the panel. When any invariant does not hold, it will become as red like in the system function panel. The remaining parts of this subsection will show how to use the generated prototype to validate the requirements.

• **Pre-condition Errors** The execution of system operation containing pre-condition errors may lead system to an un-expected state that violates system invariants. It contains two cases: parameter constraint missing and constraint errors. Parameter constraint missing means that the constraints of parameters are missing in the pre-condition, the execution of the system operation may lead system to an un-expected state that violates system invariants. For example, the customer makes a cash payment $20 for their $40 products in CoCoMe. If the pre-condition does not check the tendered money is greater than or equal to the total price of products, the invariant of *CashPayment* that the balance must be greater than or equal to zero may be violated. When the invariant bar of the prototype become red, users can add the missed constraints into the pre-condition of *cashPayment* to fix this error.

Constraint errors mean that the constraints are not correct in the pre-condition, the execution of the system operation may lead system to un-expected state that violates system invariants. For example, ATM has the invariant about the one-day maximum withdraw limitation (E.g., $2000 pre-day). When a customer intends to withdraw $500 from her account if she has already withdrawn $1800 on the same day. This request must be rejected by the pre-condition of *withdraw* operation. However, this request can be executed if the pre-condition is that the total withdrew money of one day is lesser than or equal to $2000. In this case, the execution of *withdraw* will make the system state violate the maximum limitation invariant due to the withdrew money $2300 is higher than the maximum limitation $2000. We need to fix this error by replacing the pre-condition to check the value of withdrawing must be lesser than or equal to the value of maximum limitation minus the total value of withdrew today ($withdrawing \leq maxValue - withdrew$).

• **Post-condition Errors** Post-condition errors can also lead the system to an un-expected states that violate system invariants and the pre-condition of the next system operations under the same use case. For examples, 1) one invariant of CoCoME system shows the final price of the current sale process must be greater than zero. The operation *endSale()*

adds all the sub-amount of products together as the final price for the payment. If there is a typo when the plus sign '+' is typed as minus sign '-' in the post-condition, the final price will be less than zero. That will violate the price invariant after execution. 2) The post-condition of *endSale()* should specify the flag *IsReadytoPay* to *true*. If missing this specification in the post-condition, the pre-condition of the system operation *makePayment()* cannot be satisfied, and a warning message will prompt. The product managers can validate and fix those post-condition errors without too many efforts by checking the system state before and after the execution of system operations in the prototypes.

• **Invariants Errors** The invariants may also contain inappropriate constraints. For example, the precondition of *enterItem()* specifies the stock number of the scanned item must be greater than zero. Suppose there is a invariant about the stock number of the best seller product must be higher than 100 for preventing the out of stock problem. If the preconditions and post-conditions of the operations are correct, this invariant will not satisfy when stock number is between 0 and 100.

In short, the prototype can check contract and indicate violations to help requirements validation. Although locating and fixing the errors require carefully observing and analyzing the state of objects and the contracts, the generated prototype provides an intuitive way to help product managers and customer to validate their requirements. Moreover, the requirements cannot be elicited entirely at one time. After validating the requirements, the customers and product managers would fix the faults of the requirements and find new interesting functional requirements of the target system. This fixing and eliciting requirements process will make the previous requirements version from R_{n-1} to next version R_n. Then the evolved requirement R_n will be used to generate a new prototype P_n. Compared with the methods to manually implementing the prototype, customers and project managers can use RM2PT to generate new prototypes and validate the requirements without waiting for the software engineers to implement the prototype. Therefore, RM2PT shortens the timeline and boosts the software development process. Moreover, RM2PT reduces the inconsistency between the requirements and the prototype by automatically transformation rules; otherwise, errors may be involved by the green hands or careless of the human nature. In the next section, we will demonstrate the validity of the RM2PT through four classical case studies using in our daily life.

6 Case Studies

In this section, we use four case studies to demonstrate the validity and capacity of the proposed approach for requirements validation. Those case studies are widely used systems in our daily life: supermarket system (CoCoME), Library Management System (LibMS), Automated Teller Machine (ATM), and Loan Processing System (LoanPS). CoCoME mainly contains the scenarios of the cashier to process sales in supermarkets, and supermarket managers to manage the storage of products. LibMS primarily involves the student to borrow and return books. ATM concerns the customer to withdraw and credit money. LoanPS touches the use cases of submitting and evaluating loan requests, booking a new loan and making the payment. Those four case studies demonstrate the different aspects of requirements modeling and prototyping of RM2PT. ATM provides a quick-start and simple demonstration of RM2PT about requirements modeling and prototyping. CoCoME demonstrates the capacity of generation and validation complex use case such as *processSale*. LibMS demonstrates the capacity of generation and validation of the complex system operations, that makes the case study of LibMS contains the highest number of primitive operations. The last one LoanPS demonstrates the ability to invoking third-party system services by including the remote operations calling from the account management system of banks and the credit management system of the governments. We will show the requirements modeling and validation results in the following parts. More details of the requirements models can be found at GitHub[3].

6.1 Complexity and Cost of Requirements Model

Requirements complexity is measured by the dimensions of the number of the actors, use cases, system operations, entity classes, and associations of the entity classes in the requirements model. We present the requirements complexity of four case studies in Table 2. The requirements model of

Table 2
The Complexity of Requirements Models

Case Study	Actor	Use Case	SO	AO	Entity Class	Association	INV
ATM	2	6	15	103	3	4	5
CoCoME	3	16	43	273	13	20	10
LibMS	7	19	45	433	11	17	25
LoanPS	5	10	34	171	12	8	12
Sum	17	51	137	980	39	49	52

* Above table shows the number of elements in the requirements model. SO and AO are the abbreviations of system and primitive operations respectively. INV is the abbreviation of invariant.

CoCoME contains three actors, sixteen use cases, forty-three system operations, two hundred-seven-three primitive operations, thirteen entity classes, twenty associations between those objects, and ten invariants. The requirements model of LibMS includes seven actors, nineteen use cases, forty-five system operations, four hundred-thirty-three primitive operations, eleven entity classes, seventeen associations, and twenty-five invariants. The requirements model of ATM includes two actors, six use cases, fifteen system operations, one hundred and three primitive operations, three entity classes, four associations, and five invariants. The requirements model of LoanPS includes five actors, ten use cases, thirty-four system operations, one hundred and seventy-one three primitive operations, twelve entity classes, eight associations, and twelve invariants. In short, we provide **17** actors, **51** use cases, **137** system operations, **980** primitive operation, **39** entity classes, **49** associations of entity classes, and **52** invariants.

The cost of requirements modeling has an inevitable impact on the applicability of our proposed approach. Therefore, we investigate the costs of specifying and modeling the

3. https://github.com/RM2PT/CaseStudies

above requirements models. We invite 50 software engineers from industry and academia into our experiments. 70% of them have a bachelor's degree in computer science, 20% of them have a master's degree, and the remaining 10% have a Ph.D. degree. Moreover, we divide them into four category levels based on their programming experience. The elementary level contains 13 software engineers with 1-3 years programming experience. The intermediate level includes 18 software engineers with 4-6 years programming experience. The advanced level consists of 14 software engineers with 7-9 years programming experience, and the expert level contains 7 software engineers, who have at least 10 years' experience. We have a brief training about how to do the requirements modeling in our CASE tools. For the interested readers, all experiment materials and tutorials are available on the website[4].

The time cost of developing the requirements models are shown in Table 3. We separately investigate the UML diagrams and OCL contracts of the requirements models. Most developers are familiar with case study ATM, CoCoME, and LibMS but not LoanPS. That makes modeling LoanPS requires more time to communicate with domain experts. Note that UML diagram modeling includes the time of communication with stakeholders, and OCL contracts modeling includes the time of correctness checking. On average, 50 software engineers require 2.33 hours for modeling ATM, which contains 1.01 hours for UML diagram and 1.32 hours for OCL contract. CoCoME requirements modeling requires 9.46 hours, which contains 4.55 hours for UML diagram and 4.91 hours for OCL contracts. LibMS requires 11.01 hours, which contains 4.64 hours for UML diagram modeling and 6.37 hours for OCL contracts modeling. LoanPS requires 12.45 hours, which contains 5.51 hours for UML modeling and 6.94 hours for OCL contract modeling. In short, four case studies require **8.81** hours for requirements modeling, which requires **3.92** hours for UML diagrams and **4.88** hours for OCL contracts. The results show that contract modeling requires more time than the UML diagram. Because OCL contracts require more time to specify the signature of system operations as well as the system conditions before and after system operation execution. Moreover, the software engineers have more programming experience and familiarity of the target domain, the less time will spend on requirements modeling. Especially for the software engineers has a higher education background or more experience of using OCL in their projects, they require less time for specifying the contracts of system operations.

Table 3
Cost of Requirements Modeling

Case Study	UML Diagram	OCL Contracts	Total (hours)
ATM	1.01	1.32	2.33
CoCoME	4.55	4.91	9.46
LibMS	4.64	6.37	11.01
LoanPS	5.51	6.94	12.45
Average	3.92	4.88	8.81

* UML diagram contains a use case diagram, system sequence diagrams, and a conceptual class diagram.

4. http://rm2pt.yilong.io

6.2 Results of Prototype Generation

The generation results of four case studies are shown in this subsection. The failures of requirements modeling and generation can be divided into two situations: 1) the contracts of system operations cannot be correctly specified in OCL expression without invoking third-party APIs such as sorting algorithm, sending email, and etc. 2) The contracts of system operations can be correctly specified in OCL expression, but no transformation rule is matched to generate the implementation for the system operations correctly. We count both failures of those case studies by the measurements *MSuccess* and *GenSuceess* in Table 4.

Table 4
The Generation Result of System Operations

Case Study	NumSO	MSuccess	GenSuccess	SuccessRate (%)
ATM	15	15	15	100
CoCoME	43	41	40	93.02
LibMS	45	43	42	93.33
LoanPS	34	30	30	88.23
Average	34.25	32.25	31.75	93.65

* MSuccess is the number of SO which is modeled correctly without external event-call, GenSuccess is the number of SO which is successfully generated, SuccessRate = GenSuccess / NumSO.

ATM is the most straightforward case study of four. It does not contain a complex workflow. All of the fifteen system operations can be successfully modeled and generated. The successful generation rate reached 100%.

CoCoME contains forty-three system operations. Forty-one can be successfully generated, in which two system operations cannot be specified correctly. The two failures of requirements modeling are 1) listing top 10 out of stock products, 2) sending email notification to the store manager when the ordered products are delivered to the store. Preconditions of those two system operations can be specified correctly, but the sub-expressions of post-condition, *listing top 10* and *sending email*, can not be specified correctly in OCL expression. Therefore, the corresponding transformation and implementation cannot be generated correctly as well.

Another case is that listing almost out of stock the products with storage less than 10 in ascending order. That post-condition can be specified correctly, but the sub-expression of *ascending order* cannot be generated because no transformation rule can be used to map that sub-expression to the primitive operations. Counting the failures of system operations modeling and generation, the successful rate of prototype *CoCoME* is 93.02%.

LibMS contains forty-five system operations, forty-three can be correctly modeled, and forty can be correctly generated. The two modeling failures are 1) listing the top 10 the student holding the overdue book over a week, 2) sending an email notification to the student, who holds the book copy will be due in the next two days. Like CoCoMe, the failures of modeling are also due to the expression *top 10* and *sending email*. The failure of generation is listing the holding book records by the due day in *ascending order*. Counting in those failures, the successful generation rate is 93.33%.

LoanPS contains thirty-four system operations, and thirty system operations can be correctly specified and generated. Four failures of modeling include 1) listing the top 10 loan requests with loan assistant, 2) sending a notification to the applicant when her loan request is approved, 3) printing the loan agreement, 4) sending the notification to the customer when their loan is due soon. The successful generation rate of *LoanPS* is 88.23%.

In short, RM2PT can successfully generate average **93.65%** system operations of those four case studies without any extension. There still has **6.35% errors** of requirements modeling and prototype generation. As we mentioned, the errors mainly caused by the failures of specifying the post-conditions of the contract such as sending email, printing document, sorting and retrieving top N elements.

To help users easier finding the failures of modeling and prototype generation, 1) our case tool RM2PT can automatically indicate the sub-expression error with red underlines in the post-conditions when generating the prototype from the requirements model and the code errors in the generated prototype. 2) RM2PT provides an extension mechanism to allow specifying that the third-party APIs were invoking in OCL expression. Once errors were identified by RM2PT, we can replace the error expression in post-condition with the calling expression of the third-party service. This calling expression will be a transformation to the operation calling in the prototype. The developer can manually implement the operation, or invoking the operation system APIs (for sending email and printing document) and reusing the collection algorithm library (for sorting and top N elements retrieving). After specified the calling expression of third-services into the requirements model, all the system operations of four case studies can be correctly modeled and generated.

This subsection exposes the results of requirements modeling, prototype generation, and how to fix the failures of modeling and generation. Based on the techniques of requirements validation in the section 5, we will show the result of validation in the next subsection.

6.3 Results of Requirement Validation

The result of requirement validation of four case studies are shown in Table 5 and 6, which contain the statistic of requirements errors and missing. During the three-round requirements modeling, prototype generation, requirements validation, we found **99** requirements errors, which includes 31 errors in the pre-condition and 68 errors in the post-condition. In details, 5 pre-condition and 12 post-condition errors are founded in ATM. 8 pre-condition and 23 post-condition are founded in CoCoME case study. 12 pre-condition and 26 post-condition errors are founded in LibMS case study. 6 pre-condition and 21 post-condition are founded in LoanPS case study.

Table 5
Requirements Errors

	Requirements Errors	
Name	Pre-condition	Post-condition
ATM	5	12
CoCoME	8	23
LibMS	12	26
LoanPS	6	21
Total	31	68

Table 6
Requirements Missing

	Requirements Missing					
Name	Actor	UseCase	SO	Entity Class	Association	INV
ATM	1	3	9	1	2	3
CoCoME	1	11	22	5	10	5
LibMS	4	12	14	11	15	12
LoanPS	2	3	15	4	2	8
Total	8	29	60	21	29	28

Requirement validation can help to find errors in requirements, but also can help to elicit missing requirements. During the requirement validations and refinements from first version to third version, ATM adds 1 actor, 3 use cases, 9 system operations, 1 entity class, 2 associations of entity classes, and 2 invariants. CoCoME adds 1 actor, 11 use cases, 22 system operations, 5 entity classes, 10 associations of entity classes, and 5 invariants. LibMS adds 4 actors, 12 use cases, 14 system operations, 11 entity classes, 10 associations of entity classes, and 12 invariants. LoanPS adds 2 actors, 3 use cases, 15 system operations, 4 entity classes, 2 associations of entity classes, and 8 invariants. Totally, we find **175** missing requirements during the requirements validation about **8** actors, **29** use cases, **60** system operations, **21** entity classes, **29** the associations of entity classes, and **28** invariants.

6.4 Automated Prototyping vs Manual Prototyping

We demonstrate the advantages of the proposed approach to automated prototype generation in this subsection. The invited software engineers are requested to implement the four requirements models by their favorite object-oriented programming language. Note that the prototype is mainly used for requirements validation, that means the correctness of the prototype program is more important than the quality. The manual prototypes must be correct before used to validate the requirements. Therefore, we count manual prototyping from the aspects of implementation, testing, and debugging. The manual prototyping results are shown in Table 7.

The results show that manual prototyping requires more efforts in testing and debugging stages to achieve consistent implementation with the requirements model. On average, ATM prototype requires 14.62 hours in total, which include 6.09 hours for implementation, 4.63 hours for testing, and 3.90 hours for debugging. CoCoME prototype requires 32.19 hours in total, which include 15.09 hours for implementation, 8.80 hours for testing, and 8.31 hours for debugging. LibMS prototype requires 34.74 hours in total, which include 18.28 hours for implementation, 9.18 hours for testing, and

Table 7
Manual Prototyping

Case Study	Implementation	Testing	Debugging	Total
ATM	6.09	4.63	3.90	14.62
CoCoME	15.08	8.80	8.31	32.19
LibMS	18.28	9.18	7.29	34.74
LoanPS	13.23	8.96	8.79	30.98
Average	13.17	7.89	7.07	28.13

8.79 hours for debugging. In short, manual prototyping of four case studies requires **28.13** hours, which includes **13.16** hours for implementation, **7.89** hours for testing. The results show that manual prototyping requires efforts for implementation. After implementing, the testing and debugging require more costs than implementation. In practice, developers can not guarantee there is no consistency between the requirements model and prototype by one-time coding. The manual prototype is inevitable to introduce costs of testing and debugging in software engineering.

In contrast, we evaluate the RM2PT prototyping performance. The performance is measured on the dimensions: a) line of code, which is calculus by the tool *cloc*[5], b) prototype time of RM2PT, c) generation time for the system operations. All the measurement are computed on a PC with 3.5 GHz Intel Core i5, 16 GB 1600 MHz DDR3, and 500 GB Flash Storage.

Table 8
Automated Prototyping

Name	Line of Code	Automated Prototype (ms)	System Operation (ms)
ATM	3897	309.74	2.26
CoCoME	9572	788.99	9.78
LibMS	12017	1443.39	18.22
LoanPS	7814	832.78	5.52
Average	8325	843.73	8.95

The automated prototyping result is presented in Table 8. ATM contains 3897 lines of code, the generation of prototype spends 309.74 ms, in which system operations only spend 2.26 ms. CoCoME contains 9572 lines of code, generation prototype spend 788.99 ms. LibMS includes 12017 lines of code, the generation time of prototype is 1443.39 ms. LoanPS contains 7814 lines of code, the generation time of prototype is 832.78 ms. On average, the case studies contains 8325 lines of code, generation prototype spend **843.73 ms** less than 1 second, the system operations generations only require **8.95 ms**.

RM2PT auto-prototyping is much more efficient than manual prototyping (**~1 second vs ~28 hours**). Although the experienced sofware engineers take less prototyping time than others, that is still more in-efficient than auto-prototyping approach. Moreover, the manual prototyping usually introduces inconsistency between prototype and requirements because of careless and not understanding of requirements model. That makes more than half of the whole prototyping time to debugging. Furthermore, if we only count the implementation time of system operations, the spending time **~9 ms)** is much less than the prototyping (**~850 ms**). In short, RM2PT is an efficient and effective approach to generate prototype without introducing inconsistency between the requirements model and prototype.

5. https://github.com/AlDanial/cloc

6.5 Scope and Limitation

Our approach has the scopes of application for practical problems. The requirements model of our approach requires a subset of UML diagrams - use case diagram, system sequence diagrams, conceptual class diagram, and OCL contracts. That makes our approach suitable for object-oriented information systems, enterprise systems, and interactive systems. Especially when the target system has many external interactions with environments. The batching systems have heavy internal workloads are not suited for. Moreover, our approach focuses on functional requirements but not non-functional requirements such as time, dependability, security, and space. That means the real-time systems, embedding systems, and cyber-physical systems are not suitable for our approach.

In addition, the proposed approach has the following limitations. 1) The first one is that 6.91% system operations cannot be correctly specified or successfully generated without introducing third-party services, but this limitation has been solved by specified the thrid-part service is invoking in OCL expression. That is a significant extension of our previous work. 2) The second limitation that is although the formal specification OCL has short learning cure than other formal specification, it still needs time for learning to specify the correct contract. We will try to find the better solution to alleviate this problem. 3) The third limitation is the performance of generation, and it can be further optimized in the future.

7 Related Work

The related work contains three parts: 1) automated prototype approach, 2) requirements modelers, 3) requirements validation, and 3) the comparison with our previous work.

7.1 Automated Prototype Generation

The following approaches are the most closely related to our work. Umple [30] can generate a prototype from a class model (conceptual class diagram) and state machine models. However, the state machine only contains abstract states and descriptions of actions (system operations). To generate fully functional prototypes, the actions must be implemented by programming languages. Moreover, invariant checking and requirements validation are not considered.

ActionGUI [31] can generate a multi-tier application from a design model, which includes a data model (specified by ComponentUML [32]), a security model (specified by SecurityUML [32]) and a GUI model (specified by GUI Modelling Language). Compared with our approach, there are three main differences. ActionGUI a) requires to provide system operation design by using data actions (primitive operations) to specify statements for events (system operations) in GUI model. A statement (the workflow of primitive operations) is either an action, an iteration or a sequence of statements. Our approach only requires providing contracts (pre and post conditions in OCL) to system operations, then the implementation of system operation can be automated generated. b) ActionGUI requires GUI engineer to construct a GUI model through their specific tool manually. Our approach can be automated prototype generation from a requirements model without providing any GUI design. c) The generated GUI of our approach is implemented by state-of-arts GUI architecture design, i.e., separated concerns into GUI content (XML) and GUI style (CSS). That makes prototype high reusable by only applying user-friendly styles to the same content.

The paper [33] proposed an intermediate approach to generate UI prototypes from UML models. It generates state chart diagrams from a design model specified by a class diagram and collaborations diagrams for each use case, and then generate the UI prototype from a use case diagram and the intermediate state chart diagrams. Compared with our work, we only require a requirements model, which includes a use case diagram, conceptual class diagram (without system operations), system sequence diagrams and the contracts of system operations (only including the interaction between actors and system interface without internal interactions among objects, that required in collaborations diagrams).

JEM [34] can generate an n-tiered prototype from a conceptual model and an execution model. The conceptual model contains a) an object model specified in a class diagram, b) functional models for the attributes of the class and c) dynamic model for each class defined in state chart diagram. Then the prototype can be generated by the derived formal specification OASIS from the conceptual model and the implementation related execution model, which includes the generation templates and details mapping about OASIS to the implementation. Compared with our approach, pre- and post-conditions only contain simple attribute checking and updating (only one attribute involved) that is hard for working on the practical requirements model and prototype generation.

SCORES [35] proposed a semi-automatically approach to generating prototypes from an enhancement of the requirements specification with user interface model in FLUID [36]. Requirements specification contains a use case diagram, activities diagrams to each use case, and a class diagram (including operations). User interface model includes a specification for view widgets, their navigation, and selection or manipulations (primitive operations) of the domain objects. It does not include the specification or contract for system operations other than simple manipulations of domain objects. Therefore, the sophisticated system operations cannot be generated such as *borrowBook*, which includes collaborations of primitive operations such as find an object, form a link, update the attribute. Moreover, the class diagram in SCORES already contains the system operations in the activities diagram, stickily speaking, that is a design model.

In short, the related works 1) require providing a design model, which contains a class diagram encapsulating system operations, the design or implementation of system operations specified in collaboration diagrams or a programming language. Moreover, SCORES, ActionGUI and JEM require a GUI design for generating prototype user interface. 2) They lack the mechanism to deal with the non-executable elements in the requirements model. 3) The generated prototype from their tools does not provide the automatic mechanisms to pre-condition, post-condition, invariants checking and object state observations, which really helps the process of requirements validation.

7.2 Requirements Modelers

Most UML modeling tools support OCL-based contracts and can generate skeleton code for entity classes in the conceptual class diagram. Moreover, Visual Paradigm (VP) [37] not only supports entity class generation but also supports generating code for primitive operations of entity class such as creating, deleting, modifying, finding the entity object. The generated entity classes can be automated mapped into tables of relational database thought ORM and the corresponding RESTFul-based web service wrappers could be automatically generated. Eclipse Foundation provides many open source CASE tools and frameworks. For example, Papyrus UML [38] is well developed and widely used open source tool. EMF Forms[6] cannot only generate primitive operations (like in VP) but also can generate GUI for validating those operations. CDO[7] provides open source solutions to support ORM for EMF[8] model. Commercial CASE tool Enterprise Architect[9] supports generating system operations of objects from the presented design model. The study [39] proposed sequence integration graph (SIG), which acts as an intermediate to help automatically fulfill system operations from sequence diagram. Morever, if providing a design model, MasterCraft [40] could generate information system prototype. AndroMDA[10] could generate Java EE and .NET system prototype. However, all the current CASE tools can not generate a prototype without providing an explicit design model. USE (UML-based Specification Environment) [41] supports analysts, designers, and developers in executing UML models and checking OCL constraints and thus enables them to employ model-driven techniques for software production. The project of Eclipse OCL[11] provides an OCL parser and evaluator on UML models. Although USE and Eclipse OCL can evaluate the invariants and precondition of operation contracts, they can not generate the implementation of system operation to conform the post-condition.

7.3 Requirements Validation

Requirements views [42] are the approach of validating the requirements in the early stage. The paper [43] focuses on examining requirements in the different viewpoints. It proposes a viewpoint language to describe the different viewpoint requirements in a unified form. After analyzing the viewpoints, missing and conflicting information can be found. This approach is a cooperative method, where different people can check their understanding of a subject and discuss the issues. However, this approach has some defects:

6. http://www.eclipse.org/ecp/emfforms
7. http://www.eclipse.org/cdo/
8. https://eclipse.org/modeling/emf/
9. http://www.sparxsystems.com
10. http://andromda.sourceforge.net
11. https://projects.eclipse.org/projects/modeling.mdt.ocl

a) the extra-costs for learning the viewpoint language, b) no UI support, c) lack of the automation, the consistency check fully depends on stakeholders.

Test-case generation is another way for requirements validation. CosTest [44] can automatically generate test cases and test reports for requirements validation from a Foundations UML (fUML) model compensated with Action Language for Foundational UML (Alf). fUML is a subset of standard UML, which contains a class diagram and activity diagrams, the details of implementation are specified by Alf. That means CosTest not only need requirements specifications but also a design which at least contains how to encapsulate the system operations into the classes, and the implementations of the system operations. In addition, test-case based validation is friendly for developers but not customers and clients, that makes it hard for them to validate their requirements through CosTest in practice.

EuRailCheck [45] allows the users to specify the requirements in natural language and formalize them into a subset of UML with temporal constraints and properties, which are defined in Controlled Nature language (CNL). The satisfaction of constraints and properties can be further verified by NuSMV or CEGAR model checkers. Stickily speaking, the subsets of UML used in EuRailCheck are not just a requirements model. It uses the class diagram to capture the design information about how to encapsulate the system operations into classes, state machines for specifying state transition for each class, as well as sequence diagrams for describing the interaction among a set of objects. Compared with our work, RM2PT only requires a requirements model, and the requirements can be validated visually by investigating the process of use case executions along with system state observations, pre- and post-condition checking in prototypes.

Animation [46] is another important technique for requirements validation. The paper [47] presents a tool for interactively validating requirements through animation. It takes BPMN as input, and generates a mock-up user interface prototype. The generated prototype contains UI pages and navigation to animate workflow execution. Moreover, all interactions of the stakeholders are recorded and incorporated into the prototype models. The papers [48] [49] present a web animation tool for requirements validation through exploring goals and scenarios, in which it uses linear temporal logic to express goals, and XML to define state transitions and UI components. TestMEReq [50] can automatically generate mock-up user interface as well as abstract test cases from requirements description. It allows multiple stakeholders to collaboratively validate the same set of requirements.

Compared with our work, the prototypes generated from their tools are only mock-up, which do not contains the details implementation of system operations. That means only the coarse-grained requirements validation can be done through the basic animation, other functionalities of prototypes such as validity and consistency checks through the investigating execution of use cases along with system state observation, invariant, pre- and post-conditions checking are not included. Moreover, the mock-up prototype will be throw-way after validation. The prototypes generated from our tools are evolutionary prototypes. They embedded architecture patterns and design patterns in Java EE and .NET enterprise system. That makes the generated prototypes from our tools easier evolving to be the practical software systems without too much cost.

There are some other works about requirement validation. The paper [51] presents an approach to describe and validate high-level timed requirements by using the Timed Use Case Maps language. The paper [52] presents a systematic approach for early-stage validation of security requirements. The paper [53] proposes a method for Global Software Development (GSD). It specifies requirements documents to understand the details of different GSD sites and then circulated them between different sites for reviewing and validation. The paper [54] introduces a new Hazard Relation Diagrams to increase validation objectivity of requirements based hazard mitigations. The paper [55] presents an approach to feature-oriented requirements validation for automotive systems in both functional behaviors and nonfunctional properties.

7.4 Compared with Our Previous Work

Compared with our previous work about Automated Prototype Generation and Analysis (AutoPA) [24] [25], we extend the original work by fully accomplishing the theoretical work of automated prototype generation from a requirements model in UML with formal specifications of contracts in OCL as well as developing the corresponding CASE tool: RM2PT, which is applicable in practical software development. Moreover, the generated prototypes from RM2PT are evolutionary prototypes rather than throwaway prototypes in the previous work. The extension details are presented as follows:

1) *Extensions of transformation rules.* To generate a more practical prototype, the current work directly applies object reference concept of Java to generate prototypes rather than introduce an explicit unique object ID for each object in the previous work. Based on the previous work about finding an object by object ID, creating an object, adding a link, getting the value of attribute, setting an attribute, removing a link, and deleting an object, we refine the previous rules and present the new rules for supporting larger set of OCL contract transformation, which include a) finding an object with a condition through OCL keywords *any()* and *select()*, b) OCL standard operations such as *size()* and *isEmpty()*, c) the iteration expression *forAll()* for iterating object from a list, d) third-party operation invoking, and e) the *return* expression.

In detail, we introduce the rules from R_1 to R_3 for finding objects with a condition through *EntityManager*, which stores all references of objects. The rules from R_4 to R_7 support finding objects through links. The rules from R_8 to R_{11} support OCL standard operations. The rule R_{15} can check the uniqueness of an object. Since the current work applies reference implementing prototype generation, two rules R_{16} and R_{17} are for creating an object. The rule R_{16} is used for creating a new object first, and then the rule R_{17} is used for adding its corresponding reference into *EntityManager*. Adding a link between objects is carried out by the rule R_{19} for the one-to-one association and the rule R_{21} for the one-to-many association, respectively. Similarly, the rules R_{20} and R_{22} are for removing a link. The rule R_{24} is for setting the same value to all attributes of a set of objects. For example, when the time reaches 00:00 a.m. in the midnight, the attribute *withdrawal* of every bank account object should be set to zero. The rule R_{25} is for *return* expression, and the rule R_{26} is for invoking third-party APIs.

2) *Transformation failure handling.* In the previous work, when a sub-expression of OCL contracts cannot be matched with one of the given transformation rules, the tool will be failed to generate a prototype. In this paper, a mechanism is designed to handle these transformation failures of the non-transformable OCL sub-expressions of contracts by wrapping them as an interface, which can be further implemented by developers manually or invoking third-party APIs.

3) *Enhancing prototyping functionality for requirements validation.* The generated prototype in previous work only contains the function of investigating the process of use case execution and invariant checking. In this paper, we enhance the prototyping functionality for requirements validation by adding system state observation, pre- and post-condition checking, which makes developers and customers find requirement errors easily.

4) *Evolutionary prototype rather than throwaway prototype.* The previous work used the way of introducing an explicit unique ID for each object in order to identify and store the object. Therefore, the objects only can be found through object IDs, which is different from the real way of object-oriented programming. The generated prototype would be throwaway after requirements validation. In this paper, we adopt object references to store and identify objects. In addition, with embedding architecture patterns and design patterns inside the generated prototypes, the generated prototypes could be further evolved to be the practical software systems such as Java EE and .NET enterprise system after requirements validation. This contribution is significant.

5) *Fully designing and implementing a CASE tool: RM2PT.* To make a proposed approach applicable, it is necessary and useful to develop its corresponding support tool. We fully design and implement a CASE tool: RM2PT by applying architectural and design patterns, version control, and automated testing. It provides a one-stop environment for software requirements modeling and analysis, prototype generation, and requirements validation.

6) *Evaluation through four case studies over 50 use cases rather than a small example.* The feasibility of our preliminary idea is demonstrated with a small example in our previous work. In this paper, we apply our developed tool: RM2PT to four case studies to evaluate our approach, and obtaining a satisfactory result with successfully generating correct source code of 93.65% system operations in 1 second without invoking third-party APIs. Besides, the remains 6.35% failures can be identified and further implemented by developers manually or invoking third-party APIs.

8 Conclusion and Future Work

This paper presents an approach to automated prototype generation from a formal requirements model, and the requirements model can be validated by the generated prototype. It includes executable analysis of formal specification and designs a set of transformation rules for translating the executable parts of the contract into Java source code. The non-executable parts of a contract can be identified and wrapped by an interface, which can be fulfilled by third-party APIs. Four cases studies, which are library management system, ATM, CoCoME and loan processing system, have been investigated, and the experiment result is satisfactory that the **93%** of system operations of use cases can be generated successfully. The CASE tool: RM2PT and its tutorials are available for the public at GitHub[12].

In the future, we will improve the current transformation algorithm to cover the more substantial subset of the executable specification. Meanwhile, we will integrate current prototyping tool with our another work on automated translating use case definitions in natural language into their corresponding formal contract in OCL. That will make this work more applicable to software industrial developers. Generally, they can read formal specification, but they have difficulties in writing a formal specification. With the tool support, their task is to confirm whether the translated formal specification is conformance with the natural language requirement description. Furthermore, after a system requirements model is validated by prototyping, we can generate the prototype into its corresponding real system with another developed transformation software tool. Besides, we will investigate how to generate test cases from the OCL specification, so that we can enhance our tool for automated prototyping and testing. Finally, the tool can be used and checked with more case studies, and hopefully, it can benefit the software industry during requirements engineering.

Acknowledgments

This work was supported by Macau Science and Technology Development Fund (FDCT) (No. 103/2015/A3) and University of Macau (No. MYRG 2017-00141-FST), Southwest University Grant (No. SWU116007), and National Natural Science Foundation of China (NSFC) (No. 61472779, 61562011, 61672435 and 61732019). Thanks to the proofreading of this paper by Quan Zu (Tongji University) and Xiaohong Chen (University of Illinois at Urbana-Champaign).

References

[1] A. G. Sutcliffe, A. Economou, and P. Markis, "Tracing requirements errors to problems in the requirements engineering process," *Requirements Engineering*, vol. 4, no. 3, pp. 134–151, 1999.

[2] H. F. Hofmann and F. Lehner, "Requirements engineering as a success factor in software projects," *IEEE Software*, vol. 18, no. 4, pp. 58–66, Jul. 2001.

[3] I. Sommerville, *Software Engineering*. Addison Wesley, 2015.

[4] G. Atladottir, E. T. Hvannberg, and S. Gunnarsdottir, "Comparing task practicing and prototype fidelities when applying scenario acting to elicit requirements," *Requirements Engineering*, vol. 17, no. 3, pp. 157–170, Sep. 2012.

[5] H. Lichter, M. Schneider-Hufschmidt, and H. Zullighoven, "Prototyping in industrial software projects-bridging the gap between theory and practice," *IEEE Transactions on Software Engineering*, vol. 20, no. 11, pp. 825–832, Nov. 1994.

12. http://rm2pt.yilong.io

[6] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)*, 2000, pp. 35–46.

[7] I. Erfurth and W. Rossak, "CUTA4UML: bridging the gap between informal and formal requirements for dynamic system aspects," in *Proceedings of ACM/IEEE 32th International Conference on Software Engineering (ICSE'10)*, vol. 2, May 2010, pp. 171–174.

[8] F. Kordon and Luqi, "An introduction to rapid system prototyping," *IEEE Transactions on Software Engineering*, vol. 28, no. 9, pp. 817–821, Sep. 2002.

[9] D. Baumer, W. Bischofberger, H. Lichter, and H. Zullighoven, "User interface prototyping-concepts, tools, and experience," in *Proceedings of IEEE 18th International Conference on Software Engineering (ICSE'96)*, Mar. 1996, pp. 532–541.

[10] M. Kamalrudin and J. Grundy, "Generating essential user interface prototypes to validate requirements," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, Nov. 2011, pp. 564–567.

[11] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of UML models: a systematic review of research and practice," *Software and Systems Modeling*, Apr. 2018.

[12] D. Regep and F. Kordon, "Using metascribe to prototype a UML to C++/Ada95 code generator," in *Proceedings of the 11th International Workshop on Rapid System Prototyping (RSP'00)*, 2000, pp. 128–133.

[13] P. D. Bois, E. Dubois, and J.-M. Zeippen, "On the use of a formal requirements engineering language: The generalized railroad crossing problem," *Requirements Engineering*, vol. 2, no. 4, pp. 171–183, Dec. 1997.

[14] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 4–14, Jan 1998.

[15] H. Nakagawa, K. Taguchi, and S. Honiden, "Formal specification generator for KAOS: Model transformation approach to generate formal specifications from KAOS requirements models," in *Proceedings of the 22th IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, Nov 2007, pp. 531–532.

[16] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE'16)*, Nov 2016, pp. 373–383.

[17] F. A. C. Polack, "A case study using lightweight formalism to review an information system specification," *Software: Practice and Experience*, vol. 31, pp. 757–780, 2001.

[18] G. Vanwormhoudt, O. Caron, and B. Carré, "Aspectual templates in UML - enhancing the semantics of UML templates in OCL," *Software and System Modeling*, vol. 16, no. 2, pp. 469–497, 2017.

[19] X. Li, Z. Liu, and J. He, "Formal and use-case driven requirement analysis in UML," in *Proceedings of the 25th International Computer Software and Applications Conference (COMPSAC'01)*, Chicago IL, USA, 2001, pp. 215–224.

[20] Z. Liu, H. Jifeng, X. Li, and Y. Chen, "A relational model for formal object-oriented requirement analysis in UML," in *Proceedings of the 5th International Conference on Formal Engineering Methods (ICFEM'2003)*, Berlin, Heidelberg, 2003, pp. 641–664.

[21] Z. Chen, Z. Liu, A. P. Ravn, V. Stolz, and N. Zhan, "Refinement and verification in component-based model-driven design," *Science of Computer Programming*, vol. 74, no. 4, pp. 168–196, 2009.

[22] C. Larman, *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Pearson Education India, 2012.

[23] B. Meyer, *Design by contract*. Prentice Hall, 2002.

[24] D. Li, X. Li, J. Liu, and Z. Liu, "Validation of requirement models by automatic prototyping," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 241–248, Oct 2008.

[25] X. Li, Z. Liu, M. Schäf, and L. Yin, "AutoPA: Automatic prototyping from requirements," in *Proceeding of the 4th International International Symposium On Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA'10)*, Oct. 2010, pp. 609–624.

[26] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26–49, Aug. 1988.

[27] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[28] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[29] M. Glinz and R. J. Wieringa, "Stakeholders in requirements engineering," *IEEE Software*, vol. 24, no. 2, pp. 18–20, March 2007.

[30] A. Forward, O. Badreddin, T. C. Lethbridge, and J. Solano, "Model-driven rapid prototyping with umple," *Software: Practice and Experience*, vol. 42, no. 7, pp. 781–797, Jul. 2012.

[31] D. Basin, M. Clavel, M. Egea, M. A. G. de Dios, and C. Dania, "A model-driven methodology for developing secure data-management applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 324–337, Apr. 2014.

[32] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security: From UML models to access control infrastructures," *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, pp. 39–91, Jan. 2006.

[33] M. Elkoutbi, I. Khriss, and R. K. Keller, "Automated prototyping of user interfaces based on UML scenarios," *Automated Software Engineering*, vol. 13, no. 1, pp. 5–40, Jan. 2006.

[34] O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano, "The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming," *Information Systems*, vol. 26, no. 7, pp. 507 – 534, 2001.

[35] A. Homrighausen, H.-W. Six, and M. Winter, "Round-trip prototyping based on integrated functional and user interface requirements specifications," *Requirements Engineering*, vol. 7, no. 1, pp. 34–45, Apr. 2002.

[36] G. Kosters, H. W. Six, and J. Voss, "Combined analysis of user interface and domain requirements," in *Proceedings of the 2th International Conference on Requirements Engineering (RE'96)*, Apr. 1996, pp. 199–207.

[37] V. Paradigm, "Visual paradigm for UML," *Visual Paradigm for UML - UML tool for software application development*, 2013.

[38] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier, "Papyrus UML: an open source toolset for mda," in *Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09)*. Citeseer, 2009, pp. 1–4.

[39] D. Kundu, D. Samanta, and R. Mall, "Automatic code generation from unified modelling language sequence diagrams," *IET Software*, vol. 7, no. 1, pp. 12–28, 2013.

[40] V. Kulkarni, R. Venkatesh, and S. Reddy, "Generating enterprise applications from models," in *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*. Springer, 2002, pp. 270–279.

[41] M. Gogolla, F. Büttner, and M. Richters, "USE: a UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1, pp. 27–34, 2007.

[42] M. Felderer and A. Beer, "Using defect taxonomies for requirements validation in industrial projects," in *Proceedings of the 21th IEEE International Requirements Engineering Conference (RE'13)*, July 2013, pp. 296–301.

[43] J. C. S. P. Leite and P. A. Freeman, "Requirements validation through viewpoint resolution," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1253–1269, Dec 1991.

[44] M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor, "CoSTest: A tool for validation of requirements at model level," in *2017 IEEE 25th International Requirements Engineering Conference (RE'17)*, Sep. 2017, pp. 464–467.

[45] R. Cavada, A. Cimatti, A. Mariotti, C. Mattarei, A. Micheli, S. Mover, M. Pensallorto, M. Roveri, A. Susi, and S. Tonetta, "Supporting requirements validation: The EuRailCheck tool," in *2009 IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, Nov 2009, pp. 665–667.

[46] A. Gemino, "Empirical comparisons of animation and narration in requirements validation," *Requirements Engineering*, vol. 9, no. 3, pp. 153–168, Aug 2004.

[47] G. Gabrysiak, H. Giese, and A. Seibel, "Deriving behavior of multi-user processes from interactive requirements validation," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. New York, NY, USA: ACM, 2010, pp. 355–356.

[48] S. Uchitel, R. Chatley, J. Kramer, and J. Magee, "Fluent-based animation: exploiting the relation between goals and scenarios for requirements validation," in *Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04)*, Sep. 2004, pp. 208–217.

[49] R. Chatley, S. Uchitel, J. Kramer, and J. Magee, "Fluent-based web animation: exploring goals for requirements validation," in *Pro-*

ceedings of the 27th International Conference on Software Engineering (ICSE'05)*, May 2005, pp. 674–675.

[50] N. A. Moketar, M. Kamalrudin, S. Sidek, M. Robinson, and J. Grundy, "An automated collaborative requirements engineering tool for better validation of requirements," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, Sep. 2016, pp. 864–869.

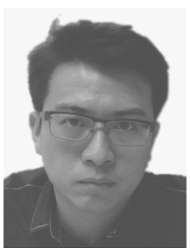
[51] J. Hassine, "Early modeling and validation of timed system requirements using timed use case maps," *Requirements Engineering*, vol. 20, no. 2, pp. 181–211, Jun 2015.

[52] M. El-Attar and H. A. Abdul-Ghani, "Using security robustness analysis for early-stage validation of functional security requirements," *Requirements Engineering*, vol. 21, no. 1, pp. 1–27, Mar 2016.

[53] N. Ali and R. Lai, "A method of software requirements specification and validation for global software development," *Requir. Eng.*, vol. 22, no. 2, pp. 191–214, Jun. 2017.

[54] B. Tenbergen, T. Weyer, and K. Pohl, "Hazard relation diagrams: a diagrammatic representation to increase validation objectivity of requirements-based hazard mitigations," *Requirements Engineering*, vol. 23, no. 2, pp. 291–329, Jun 2018.

[55] J. Zhou, Y. Lu, K. Lundqvist, H. Lönn, D. Karlsson, and B. Liwång, "Towards feature-oriented requirements validation for automotive systems," in *Proceedings of 22th International Requirements Engineering Conference (RE'14)*, Aug 2014, pp. 428–436.



Yilong Yang received his B.S. degree in Computer Science from China University of Mining and Technology, China in 2010. His M.S. degree is from Guizhou University, China in 2013. He has been a fellow at United Nations University - International Institute for Software Technology, Macau. Currently, he is a Ph.D. candidate in Software Engineering at University of Macau. His research interests include Automated Software Engineering and Machine Learning.

Xiaoshan Li received his Ph.D. degree from the Institute of Software, the Chinese Academy of Sciences, Beijing in 1994. Currently, he is an Associate Professor in the Department of Computer and Information Science at University of Macau. His research interests include Automated Software Engineering, Formal Specification and Verification, Formal Semantics of UML, and AI on Chinese Calligraphy and Painting.

Wei Ke received his Ph.D. degree in Computer Applied Technology from Beihang University in 2012. He is currently an associate professor in School of Public Administration of Macau Polytechnic Institute. He had successfully completed in a couple of research projects funded by Macau FDCT. His research interests include Programming Languages, Formal Methods, and Software Engineering.

Zhiming Liu holds Ph.D. degree in Computer Science from University of Warwick, UK. He worked as senior research fellow at the United National University, International Institute for Software Technology, and Chair Professor of Software Engineering at Birmingham City University. He currently leads the Centre for Research and Innovation in Software Engineering (RISE) at Southwest University. His research interests include Formal Methods, Real-time Systems, Fault-tolerant Systems, Health Information Systems, and Object-oriented and Component-based Systems.