

# Separation of Concerns and Consistent Integration in Requirements Modelling

Xin Chen<sup>1,2</sup>, Zhiming Liu<sup>1 \*</sup>, and Vladimir Mencl<sup>1,3</sup>

<sup>1</sup> International Institute for Software Technology  
United Nations University, Macao SAR, China  
{chenxin, z.liu, mencl}@iist.unu.edu

<sup>2</sup> Department of Computer Science and Technology, Nanjing University, China

<sup>3</sup> Department of Software Engineering, Charles University, Czech Republic  
mencl@nenya.ms.mff.cuni.cz

**Abstract.** Due to their increasing complexity, design of software systems is not becoming easier. Furthermore, modern applications ranging from enterprise to embedded systems require very high levels of correctness and dependability assurance. The most effective means to handle complexity is *separation of concerns* and *incremental development*, and assurance of correctness requires *formal modelling* and *formal analysis*. When separation of concerns splits the model into several parts, an important issue is to ensure consistency among these parts. We propose an approach supporting separation of concerns and consistent and incremental modelling of requirements.

## 1 Introduction

The growing complexity of software applications requires a UML-based (or similar) software development approach [11, 12] that supports separation of concerns and incremental development. In such a development framework, artifacts are modelled and analysed using different modelling notations: class diagrams representing the *static structural view*, *state machines* specifying and validating *dynamic behaviour*, *sequence and collaboration diagrams* describing the interactions between objects, while a textual notation, such as *OCL*, is used to specify the functionalities and state constraints. However, several challenging issues arise from such a multi-view and multi-notational approach:

- *Consistency*: the models of various views need to be syntactically and semantically compatible with each other (i.e. horizontal consistency) [2, 4],
- *Transformation and evolution*: a model must be semantically consistent with its refinements (i.e. vertical consistency) [4, 2].
- *Integration*: models of different views need to be seamlessly integrated for prototyping and system construction.

To deal with these issues in applications that require high levels of correctness and dependability assurance, we need *precise semantic definitions* of the models and their

---

\* partially supported by the project HighQSoftD: Integrating Methods and Tools for High Quality Software Development funded by Macao Science and Technology Development Fund

integration. In this paper, we study the consistent use of different UML models for describing different views and their integration for the overall system requirements modelling and analysis. We propose an approach to incremental requirements capture and analysis so that new requirements are added without disturbing the consistency of the requirements that are already modelled.

Consistency checking and formal analysis of UML models have been widely studied in recent years [1, 3, 5, 7, 18]. The majority of them focus on the formalisation of individual diagrams and only treat the consistency of the models of one or two views. Another phenomenon in research on formal use of UML is that different communities intend to emphasise different notations and use the full or even extended power of individual diagrams such as sequence or state machines. This would lose the advantages of the multi-view modelling, increase the complexity of a certain kind of models, and reduce the role that the other kinds of UML models can play.

We begin in Sect. 2, where we define the model of requirements. Sect. 3 defines our formal notation, which is used in Sect. 4 to formalise the model views and their integration. Sect. 5 defines the consistency of a model. We discuss in Sect. 6 how requirements can be modelled incrementally and conclude our paper in Sect. 7. We use the Point of Sale Transactions (POST) system [13] as a case study to illustrate the models and ideas.

## 2 Models of Requirements Analysis

Software development is about constructing a sequence of models that allows us to verify that the final model, i.e. the code, implements the application requirements correctly.

### 2.1 Model of requirements

With a UML-based object-oriented approach, to build the application requirements model, one needs to

- identify *domain (business) processes* and represent them as *use cases*,
- identify *domain concepts* and their *relations* and model them by a *conceptual class diagram*,
- identify *business rules* and *constraints* and describe them as *system invariants*.

Although UML is widely used to describe the above notions, different people use different UML models to describe elements of the requirements model. We define an *application requirements model*  $\mathcal{M} = \langle \Gamma, \Delta, \Omega, \Phi, \Theta \rangle$  consisting of the following parts:

1.  $\Gamma$  is a *conceptual class diagram*, in which all classes are *conceptual* and have no methods assigned to them and all associations have no directions. We also include a *use case controller class* for each use case.
2.  $\Delta$  is a family of *sets of system sequence diagrams*, where each set of sequence diagrams represent the *different courses of interaction events* between the *actors* and the *use case controller* in a use case, and each event is called a *system operation* (or *use case operation*). The operations are declared as methods of the use case controller class.

3.  $\Omega$  is a set of *state diagrams* (or *statecharts*), one for each use case controller class. The (symbolic) states of a state diagram form the value space of the attribute *state* which belongs to the corresponding use case controller class and represents the current control state.
4.  $\Phi$  is a *specification mapping* that equips each system operation  $m(T_1 \text{ in}; T_2 \text{ out})$  with a pair of pre- and post-conditions written in the form  $m(T_1 \text{ in}; T_2 \text{ out})\{pre_m \vdash Post_m\}$ .
5.  $\Theta$  is a system invariant that has to be preserved by each system operation. Liveness properties can also be specified, but they are not considered in this paper.

Rather than simple treatment of use-case diagrams, we define an application requirements model in terms of multi-views. The reason is that use-case diagrams alone can't contain all the concerns with respect to desired applications. For example, a business process can be clearly expressed by a use-case, but business rules tightly coupled with it can't be derived from that diagram. In practice, the combination of use-case diagrams, class diagrams, segments of sequence diagrams and state diagrams is widely adopted to provide detailed behaviour descriptions. Such formal treatment as proposed in this paper can be effectively applied to obtain a formal model from it.

It is assumed that a class in the requirements model has no methods, as what a class can and should do must be designed later, according to the needs of the use case operations. Similarly, navigability (directions) of associations can't be decided until decisions are made on how objects interact with each other to realise the use cases. Attributes declared in the use-case controller class serves as the variables for defining functionality specifications for the operations in the same class. These assumptions could be regarded as instructions on how to carry out analysis of class methods and association navigability with method refinements [8] along the vertical dimension.

## 2.2 A model of requirements for POST

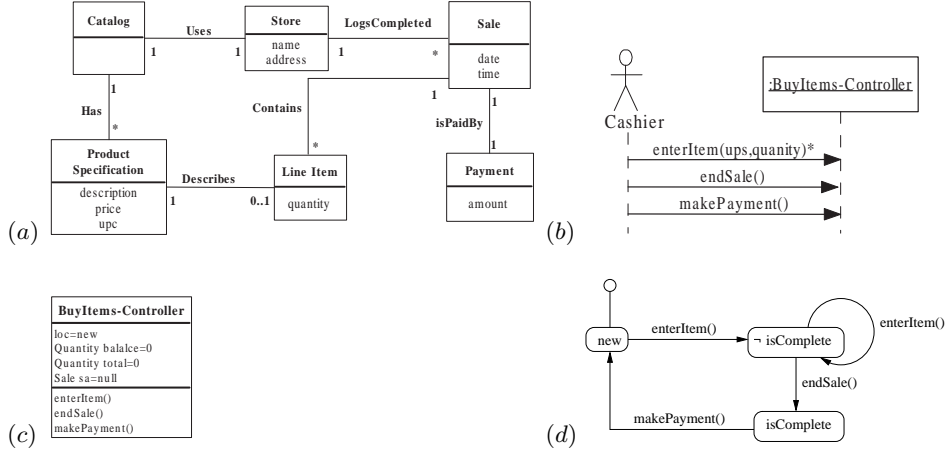
We show the different views of the POST [13] system example in Fig. 1: its *conceptual class diagram* is shown in Fig. 1(a), a sequence diagram for the use case *BuyItems* in Fig. 1(b), the controller class for this use case in Fig. 1(c), and its state diagram in Fig. 1(d). The use case *BuyItems* identifies three use case operations *enterItem()*, *endSale()*, and *makePayment()*; these will be specified later in Sect. 4.

Please note that the adoption of a use-case controller class is to make the modelling approach programmatic, and to set up a starting point which formal analysis and refinement start from. For further details about the Controller Pattern, readers can refer to Larman's book. [13].

## 3 Formal Notation: Designs

Before we proceed to formalise the specifications, we introduce our notation for writing specifications. The most fundamental notion in our framework is a *design* [10]. A design  $D$  over an input alphabet  $in\alpha$  and an output alphabet  $out\alpha$  is a predicate of the form:  $p(in\alpha) \vdash R(in\alpha, out\alpha)$  and its meaning is defined by

$$(p \vdash R) \stackrel{def}{=} (ok \wedge p) \Rightarrow (ok' \wedge R).$$



**Fig. 1.** (a) class diagram of POST, (b) sequence diagram of use case *BuyItems*, (c) use case controller of *BuyItems*, and (d) state diagram of *BuyItems*

which asserts that if execution of the design  $D$  starts successfully, i.e.  $ok$  is *true*, from a state in which the precondition  $p$  holds, it will terminate successfully, i.e.  $ok'$  is *true*, in a state satisfying the postcondition  $R$ . Notice that in a design, an input variable is an unprimed identifier and an output variable is a primed one. We will simply write  $R$  for a design  $true \vdash R$ .

With the use of the auxiliary boolean variables  $ok$  and  $ok'$ , designs are closed under the conventional programming operators, such as assignment “ $x := e$ ”, sequencing “;”, and conditional choice “ $D_1 \triangleleft b \triangleright D_2$ ” (**if**  $b$  **then**  $D_1$  **else**  $D_2$ ). We also define the special designs  $\perp$  as the “weakest design” *true* and  $\top$  as “strongest design” *false*. An *assertion* (or *guard*)  $g$  is a predicate over the input alphabet and can also be defined as a design, denoted by  $g_\top$

$$g_\top \stackrel{def}{=} skip \triangleleft g \triangleright false$$

where  $skip$  is the design  $true \vdash \bigwedge_{x \in in_\alpha} x' = x$  that does not change any variable. Thus, a guarded design  $g_\top; D$  behaves like  $D$  when  $g$  holds, and deadlocks otherwise.

We also define parallel composition and nondeterministic choice

$$(p_1 \vdash R_1) \parallel (p_2 \vdash R_2) \stackrel{def}{=} (p_1 \wedge p_2 \vdash R_1 \wedge R_2)$$

$$(p_1 \vdash R_1) \sqcap (p_2 \vdash R_2) \stackrel{def}{=} (p_1 \vee p_2) \vdash ((p_1 \wedge R_1) \vee (p_2 \wedge R_2))$$

Refinement between designs is defined as logical implication, and data refinement is handled by introducing a refinement mapping between the state spaces of the two designs. All the above operations on designs are monotonic with regard to the refinement relation. For details, please see [10].

The *weakest precondition* ( $wp$ ) has been widely used to analyse programs. The link from the design calculus to the theory of predicate transformers is given by the following definition

$$wp(p \vdash R, q) \stackrel{def}{=} p \wedge \neg(R; \neg q)$$

## 4 Formalisation, Consistency and Integration

We now discuss how formal techniques can be used for the treatment of the sub-models.

### 4.1 Class diagrams

The class diagram  $\Gamma$  of a model of requirements defines the types (value space) and static structure of the program:

- *cname*: the set of all class names declared in  $\Gamma$ .
- *aname*: the set of association names that can be treated as classes, each having two attributes representing the roles of the associations.
- *superclass*: the function defines that  $N$  is a direct superclass of  $M$  in  $\Gamma$ . We define the general superclass class relation  $\succ$  to be the transitive closure of *superclass*, and  $N \succeq M$  if  $N \succ M$  or  $N = M$ .
- *attr*: for each class name  $C \in \text{cname} \cup \text{aname}$ ,  $\text{attr}(C)$  is the set of attributes declared in  $C$  itself. Each attributes  $a$  is declared with a type and we use  $\text{dtype}(a)$  to denote this type. Let *Attr* be the function that extends  $\text{attr}(C)$  for each  $C$  to include the protected and public attributes inherited from its superclasses.

*Example 4.1.* For the class diagram in Fig. 1(a) have

$$\begin{aligned} \text{cname} &= \{\text{Store, Sale, Catalog, ProductSpecification, LineItem, Payment}\} \\ \text{aname} &= \{\text{Has, Uses, Describes, Contains, Logscompleted, IsPaidBy}\} \end{aligned}$$

The relation *superclass* is empty, and the function *attr* can be easily defined.

### 4.2 Object diagrams and OO commands as designs

Assume a set  $\mathcal{T}$  of *built-in data types*, and an infinite set  $\mathcal{R}$  of *object identities* (or *references*), and  $\text{null} \in \mathcal{R}$ . A *value* is either a member of a primitive type in  $\mathcal{T}$  or an object identity in  $\mathcal{R}$ . An *object*  $o$  is either the special object *null*, or a structure  $\langle r, C, \sigma \rangle$ , where reference  $r$ , denoted by  $\text{ref}(o)$ , is in  $\mathcal{R}$ ;  $C$ , denoted by  $\text{type}(o)$ , is a class name;  $\sigma$  is called the *state* of  $o$ , denoted by  $\text{state}(o)$ , and is a mapping that assigns to each  $a \in \text{Attr}(C)$  a value in  $\text{dtype}(a)$  if  $\text{dtype}(a) \in \mathcal{T}$  and the *null* object or a value in  $\mathcal{R}$  otherwise. We use  $o.a$  to denote  $\sigma(a)$ .

In rCOS [8], the semantics of a command in an OO language is defined as a design. Here we only give the meaning of attribute modification and object creation, where we use a state variable  $\Pi$  to denote the current state (i.e. the *heap*) of the system, and  $\Pi(C)$  for the set of all objects of class  $C$  that currently exist in  $\Pi$ :

- Let  $le$  be either a simple variable  $x$  of type  $C \in \text{cname} \cup \text{aname}$ , or  $le.b$ . The attribute assignment  $le.a := e$  changes the attribute  $a$  of the object  $o$  (attached to  $le$  in the heap) to the value of  $e$ , provided the assignment is *well-defined*:

$$le.a := e \stackrel{\text{def}}{=} \mathcal{D}(le.a := e) \wedge a \in \text{Attr}(\text{type}(le)) \vdash \left( \begin{array}{l} \Pi(\text{dtype}(le))' = \Pi(\text{dtype}(le)) \uplus \\ \{o \oplus \{a \mapsto \text{value}(e)\} \mid o \in \Pi \wedge \text{ref}(o) = le\} \end{array} \right)$$

where  $le.a := e$  is *well-defined* (denoted  $\mathcal{D}(le.a := e)$ ) if  $le$  and  $e$  are well defined expressions and they have consistent types [8].

- The command  $C.new(le)$  is well-defined if  $C \in cname \wedge \mathcal{D}(le) \wedge dtype(le) \succeq C$ . The command creates a new object, attaches the object to  $le$  and set the attributes of  $le$  to their initial values.

$$\llbracket C.new(le) \rrbracket \stackrel{def}{=} D(C.new(le)) \vdash \exists r \notin ref(\Pi) \cdot (AddNew(C, r) \wedge Modify(le))$$

where

$$AddNew(C, r) \stackrel{def}{=} \Pi(C)' = \Pi(C) \cup \{ \langle r, C, \{a_i \mapsto init(C.a_i) \mid a_i \in Attr(C)\} \rangle \}$$

$$Modify(le) \stackrel{def}{=} le' = r$$

Here we assume if  $dtype(C.a_i) = M$  and  $M$  is a class name, the initialisation assignment  $a_i \mapsto init(C.a_i)$  is  $a_i \mapsto M.new(C.a_i)$ .

It is easy to understand that the class diagram  $\Gamma$  establishes the type system of the system, and can be formalised as a class declaration section [8, 14].

### 4.3 System sequence diagrams

In general, a system sequence diagram of a use case accepts invocations of operations of the use case and calls operations of other use cases, in a temporal order required by the business process. We define the semantics of a system sequence diagram as the *prefix closed* set of finite traces of received method invocations  $?m(v; u)$  and calls to methods of other use case controllers  $!c.n(v; y)$ . Each trace is a prefix of an execution of the use case. We use the prefix closed set of traces, as an execution of a use case may be infinite. We also call this set of traces the *interaction protocol* of the use case controller.

*Example 4.2.* The set of traces of the sequence diagram in Fig. 1(b) can be represented by the following regular expression.

$$Tr(BuyItems) \stackrel{def}{=} (\{?enterItem(u, q; ) \mid u \in UPC, q \in Quantity\})^* \\ + (\{?enterItem(u, q; ) \mid u \in UPC, q \in Quantity\})^+ \cdot ?endSale() \\ + (\{?enterItem(u, q; ) \mid u \in UPC, q \in Quantity\})^+ \cdot ?endSale() \cdot \\ \{?makePayment(a; ) \mid a \in AMOUNT\}$$

Later, we will omit the parameters when all correctly typed inputs are allowable and an invoked method will return. The above set of traces can thus be described as

$$Tr(BuyItems) \stackrel{def}{=} ?enterItem()^* + ?enterItem()^+ \cdot ?endSale() \\ + ?enterItem()^+ \cdot ?endSale() \cdot ?makePayment()$$

When there is more than one sequence diagram for a use case, the set of traces of the use case is defined to be the union of all the traces of its sequence diagrams. For example, if it is allowed to *cancel* the sale at any point before the payment is made, we need two additional sequence diagrams: *cancelSale()* after a number of *enterItem()*, and *cancelSale()* after *endSale()*. Then the set of traces of *BuyItems* is

$$Tr(BuyItems) \stackrel{def}{=} ?enterItem()^* + ?enterItem()^+ \cdot ?cancelSale() \\ + ?enterItem()^+ \cdot ?endSale() \\ + ?enterItem()^+ \cdot ?endSale() \cdot ?cancelSale() \\ + ?enterItem()^+ \cdot ?endSale() \cdot ?makePayment()$$

In theory, the traces of a use case may not be a regular language [9]. However, regular languages are easier to be automatically checked for consistency between sequence diagrams and state diagrams.

If a use case  $U_1$  includes another use case  $U_2$ , the actors of  $U_1$  still directly interact with the use case controller of  $U_1$ , and this use case controller *delegates* the calls to the use case controller  $U_2$ . Only after the delegated call returns, use case controller of  $U_1$  can respond to the call from the actor and become ready to accept the next incoming call. As in the example, we use the traces of method invocations of the use case by actors, called the *provided protocol* of the use case, as the semantics of sequence diagrams. We thus omit the question mark prefixes in events.

#### 4.4 State diagrams

A *state diagram*  $S_C = (S, s_0, \zeta)$  for a use case controller  $C$  consists of a set  $S$  of *control states*, an initial state  $s_0 \in S$ , and a transition relation  $\zeta \subseteq S \times Label \times S$ . A transition  $t \in \zeta$  from a location  $s$  to  $s'$  is labelled with a *triple*  $\ell = \langle m(in; out), g, a \rangle \in Label$ , where  $m()$ , called a *triggering event*, is an operation of the use case,  $a$  is an *action* specified as an OO command, composed from assignments, object creation, method calls to other use case controllers, and designs in the formal notation defined in Sect. 3 (the semantics of such a command is a design too), and  $g$  is a predicate about the attributes of the class and input parameters of  $m()$ . We denote such a transition  $t$  by  $s \xrightarrow{\ell} s'$ .

As a syntactical restriction, we require that all transitions are triggered by external invocations. That means the execution of a method after invocation cannot be interrupted by external events. Neither do we permit the action of a transition to call other methods of the state diagrams, as the effect of the recursive invocation can be calculated into the design functionality. The state machine in Fig. 1(d) for the use case *BuyItems* of POST only shows the triggering events and the changes of control states.

Let *state* be a state variable. The change of control state by a transition  $t = s \xrightarrow{\ell} s'$  is specified as a *guarded design*:

$$cs(t) \stackrel{def}{=} (state = s)_{\top}; (state' = s')$$

For each triggering event  $m()$ , we can identify a set  $\mathcal{E}(m())$  of transitions that has  $m()$  as their triggering event. The behaviour of the state machine, in response to the invocation of  $m()$ , can be specified by the method definition:  $m() \{ \sqcap_{t \in \mathcal{E}(m())} cs(t) \}$ .

*Example 4.3.* The state diagram shown in Fig. 1(d) for the use case controller class *BuyItems-Controller* (Fig. 1(c)) corresponds to the following definitions. We use the shorthands  $New$  and  $New'$  to denote the expressions  $state = new$  and  $state' = new$  for a location.

```
BuyItems-Controller ::
enterItem() { ((New)⊤; (¬IsComplete' ∧ ¬New')) □ (¬New ∧ ¬IsComplete)⊤ }
endSale() { (¬New ∧ ¬IsComplete)⊤; IsComplete' }
makePayment() { (IsComplete)⊤; New' }
```

## 4.5 Functionality specification

For a transition  $t = s \xrightarrow{\ell} s'$  with  $\ell = \langle m(), g, a \rangle$ , the action  $a$  changes the data state while control moves from location  $s$  to  $s'$ . The specification of action  $a$ , denoted as  $m()@s \xrightarrow{\ell} s'$  (or just  $m()@s$  when not ambiguous), is defined as the *functionality specification* of transition  $t$  and is specified in terms of a design, as introduced in Sect. 3.

*Example 4.4.* The use case controller class *BuyItems-Controller* has three methods: *enterItem* (*UPC upc, Quantity qty;*), *endSale*(), and *makePayment*(*AMOUNT amount;*). Their functionality specifications are as follows:

$$\begin{aligned}
\text{known}(upc) &\stackrel{\text{def}}{=} \exists sp \in \Pi(\text{ProductSpecification}) \cdot (sp.upc = upc) \\
\text{newLine}(li) &\stackrel{\text{def}}{=} \exists li \in \Pi(\text{LineItem}) \cdot \text{LineItem.New}(li); (li.quantity' = qty) \\
\text{addLineToSale}(x, li) &\stackrel{\text{def}}{=} \exists y \in \Pi(\text{Contains}) \cdot \text{Contains.New}(y); (y.sale' = x) \\
&\quad \wedge (y.line' = li) \\
\text{addTotal} &\stackrel{\text{def}}{=} total' = total + qty \times sp.price \\
\text{enterItem}()@new &\stackrel{\text{def}}{=} \text{known}(upc) \Rightarrow (\text{Sale.New}(sa); \\
&\quad \bigvee_{sp.upc=upc} (\text{NewLine}(li); \text{addLineToSale}(sa, li)) \wedge \text{addTotal}) \\
\text{enterItem}()@\neg complete &\stackrel{\text{def}}{=} \text{known}(upc) \Rightarrow \\
&\quad \bigvee_{sp.upc=upc} (\text{NewLine}(li); \text{addLineToSale}(sa, li)) \wedge \text{addTotal}) \\
\text{endSale}()@\neg complete &\stackrel{\text{def}}{=} \text{skip} \\
\text{makePayment}()@complete &\stackrel{\text{def}}{=} \exists p \in \Pi(\text{Payment}) \cdot \text{Payment.New}(p); p.amount' = amount; \\
&\quad amount \geq total \vdash balance' = amount - total
\end{aligned}$$

The functionality specification function  $\Phi$  thus connects via disjunction the specifications of each method at all its possible locations.

*Example 4.5.*

$$\Phi(\text{BuyItems-Controller}::\text{enterItem}()) \stackrel{\text{def}}{=} \text{enterItem}()@new \vee \text{enterItem}()@\neg complete$$

Static behaviour analysis, including reasoning about class and system invariants, can be only done with the functionality specifications.

## 4.6 Integrated specification

The *integrated specification*  $\text{Spec}(t)$  of a transition  $t = s \xrightarrow{\ell} s'$  with  $\ell = \langle m(), g, a \rangle$ , considering both control state change and functionality specifications, is defined as:

$$\text{Spec}(t) \stackrel{\text{def}}{=} (g \wedge \text{state} = s)_{\top}; m()@s \xrightarrow{\ell} s' \wedge (\text{state}' = s')$$

It means that the transition can take place only when guard  $g$  holds and the control is in the state  $s$  and when it takes place, it changes the control to the state  $s'$  and changes the data state according to the functionality specification  $m()@s \xrightarrow{\ell} s'$ .

The *integrated specification* of method  $m()$  of class  $C$  is defined as disjunction of the integrated specifications of all transitions that may react to invocation of  $m()$ . Let

$t_i = s_i \xrightarrow{\ell_i} s'_i$ , with  $t_i = \langle m(), g_i, a_i \rangle$ ,  $i \in 1..k$  be the transitions in the state diagram of class  $C$  triggered by event  $m()$ . The *integrated specification* of method  $m()$  of class  $C$  is defined as

$$Spec(C :: m()) \stackrel{def}{=} \{Spec(t_1) \sqcap \dots \sqcap Spec(t_k)\}$$

We use  $body(C :: m())$  to represent the design  $Spec(t_1) \sqcap \dots \sqcap Spec(t_k)$ , and  $\mathcal{G}(C :: m())$  to denote  $\exists i \cdot (1 \leq i \leq k \wedge g_i \wedge (state = \ell_i))$ .

The integrated specification which gives complete behaviour definition for each method is formalised in our model in a compositional way. Such a kind of formalisation supports the separation of concerns and facilitates the verification of models.

## 5 Consistency

We focus on the inter-consistency among the different views of a system model, as the well-formedness conditions of individual diagrams have been widely studied in the past and defined in OCL.

### 5.1 Static consistency

The consistency between the class diagram  $\Gamma$  and the other models is simply that the class diagram defines the type system for the functional specifications defined by  $\Phi$ . That is, all types, classes and attributes that are used in the definitions of  $\Phi(m())$  are defined in  $\Gamma$ , and all the specification statements are *well-defined* in the context of  $\Gamma$ . This consistency relation is similar to the consistency of a program statement and the type definitions and variable declarations in an imperative program. With this regard, adding classes, associations and attributes preserves consistency. Furthermore, the system invariant  $\Theta$  has to be satisfied by the functionality specifications of all methods. Due to its simplicity and the space limits, we will not go further into the formalisation of this consistency problem.

### 5.2 Dynamic consistency

We first consider traces with only triggering events, i.e. the provided protocols of use cases. In this case, method calls made by a use case to another use case or actor are not included, based on the assumption that their effect is included in the guarded designs of the actions triggered by triggering events.

Assuming  $v_0$  are the initial values of the attributes  $v$  of the controller class  $C$ , the integrated specifications  $\{C :: m() \{Spec(t_1) \sqcap \dots \sqcap Spec(t_k)\} \mid m() \text{ is a method of } C\}$  define the full semantics of the state diagram of  $C$  as a set of sequences of transitions, each of the form

$$s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \dots$$

such that the weakest precondition  $\mathbf{wp}(v' = v_0; a_1; \dots; a_i, g_{i+1} [in_{i+1}]) = true$ , for  $i \geq 0$ . This means that every execution prefix establishes the guard condition of the following transition, where  $g_{i+1}$  is the guard condition of the transition  $t_{i+1}$ .

The sequence diagrams  $\Delta$ , the state diagrams  $\Omega$  and the specification function  $\Phi$  are *consistent* if all traces of each use case defined by its sequence diagrams are completely

*realisable* by the state diagram and the functional specification of the use case controller class. That is, for any trace  $tr = m_0() \dots m_k() \in Tr(U)$  of a use case  $U$ , and any prefix of the trace  $tr$  of the form  $tr = m_0() \dots m_j(), j < k$ , the weakest precondition

$$\mathbf{wp}(v' = v_0 \wedge state' = s_0; body(m_0()); \dots; body(m_j()), \mathcal{G}(m_{j+1}())) = true$$

Therefore, the consistency ensures that every sequences of method invocations in the provided protocol of the use case is a prefix of the triggering events of an allowable sequence of transitions of the state diagram. In other words, deadlock will not happen if the actors follow the protocol defined by the sequence diagrams. This also implies the initial values  $v_0$  are *consistent* with the functionality and guards of the transitions so that preconditions and guards are satisfied for all the transitions: *preconditions ensure livelock freedom and guards ensure deadlock freedom*. We can check that the model of the POST system is consistent, i.e., its sequence diagram can be realised by the state machine.

When a use case  $U_1$  *includes* another use case  $U_2$ , UML allows a sequence diagram to directly “include” (with the “ref” keyword) another sequence diagram. In this case, the traces of the sequence diagrams of  $U_1$  include those of  $U_2$ , and the state diagrams of use case  $U_2$  should be included in the state diagram of  $U_1$  properly. This is of course done in the notation of statecharts so that the statechart of  $U_2$  is contained as a submachine in that of  $U_1$ . A statechart can always be flattened into a state diagram.

One can also directly create the sequence diagrams of the two use cases to make sure the correct inclusion of the traces is ensured. Accordingly, the state diagrams for each of the use cases are created.

In yet another approach, actors directly interact with the use case controller  $UC_1$  of  $U_1$ , and  $UC_1$  *delegates* the requests of the actors to the use case controller  $UC_2$  of  $U_2$ . Then the provided protocol of  $U_1$  obviously includes that of the  $U_2$ . However, the analysis requires to compare the entire interaction protocols of the two use cases. Then for checking the consistency of the sequence diagrams of  $U_1$  and the state diagrams, we need the *parallel composition* of the state diagrams. In such a composed state machine, there are internal triggering events resulting from the method calls by actions of the state diagram of  $U_1$  to the triggering events of the state diagram of  $U_2$ . However, it is fair to argue that such a model is already a design model and not a requirements model.

## 6 Incremental Requirements Analysis

To deal with the complexity, a software development process is carried out in two dimensions of refinement. In *horizontal refinement*, new features are added into the model, without changing any of its existing elements. Adding new use cases (system sequence diagrams) corresponds to *incremental requirements elicitation* and leads to adding new classes, new associations and attributes, as well as new sequence diagrams, state diagrams and functionalities of them. For reusability, we can also carry out *use case decomposition*, restructuring a use case by existing ones. For the newly introduced elements, we need to be sure that the originally established consistency is preserved and no inconsistency is introduced. The method we have established in this paper applies these checks.

*Example 6.1.* In the POST system, we first identified in the requirements model the use case *BuyItems*. During the requirements elaboration, *makePayment*, originally only a system operation in *BuyItems*, has been identified as a standalone use case. Subsequently, the use case *BuyItems* is remodelled to *include* (use) the use case *MakePayment*. This preserves the original functionality, but at the same times allows to use from *BuyItems* new features introduced into *MakePayment* (such as a credit card payment). We can also refine *BuyItems* to deal with inventory control, discount vouchers, and tax.

In *vertical refinement*, an analysed requirements model is transformed into a (sequence of) design models, and eventually to an implementation model. In a single step of vertical refinement, classes can be decomposed; new classes, attributes and associations can be introduced; methods of classes can be decomposed and functionalities can be delegated. These will introduce internal object interactions that have to be modelled by sequence diagrams and state diagrams. Consistent vertical refinement is not trivial and will be a major part of our future work.

## 7 Conclusion

We have defined a model of requirements in terms of five views, each of which represents a separate concern of the system. Different views can be analysed and refined separately. A simple integrated semantics is given for consistency analysis. The model can be represented by UML diagrams. Even though restricted, the application of this approach to modelling [15] the Point of Sale System (POST) [13] shows that these diagrams are expressive enough for quite realistic systems. This is due to the feature of incremental analysis.

Our future work mainly concerns with consistent refinement of the different views to design models. We are interested in the problem of how to derive a consistent refinement of the whole model from a refinement of a sequence diagram or a state diagram. This will require a lot of enrichment of the vertical refinements of different models consistently, and the treatment of composition and decomposition of state diagrams and sequence diagrams. When models at different levels in the vertical dimension are described in different modelling languages, the relation between these models have to be precisely defined to obtain a uniformed semantic framework. Ideas and techniques in [6, 18, 16, 17] can be useful. We are going to investigate techniques and algorithms for checking the consistency among different models.

## References

1. P. Andre, A. Romanczuk, J.-C. Royer, and A. Vasconcelos. Checking the consistency of UML class diagrams using Larch Prover. In *Proc. ROOM'2000, York, UK, 2000*.
2. E. Astesiano and G. Reggio. An attempt at analysing the consistency problems in the UML from a classical algebraic viewpoint. In *Proc. WADT 2002, LNCS 2755*. Springer Verlag, 2003.
3. A. Egyed. Scalable consistency checking between diagrams: The Viewintegra approach. In *Proc. 16th IEEE ASE, San Diego, USA, 2001*.

4. G. Engels, J.M. Kuester, and L. Groenewegen. Consistent interaction of software components. In *Proc. of IDPT2002*, 2002.
5. G. Engels, *et al.* A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *Proc. FSE-10*, Austria, 2001.
6. Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programmin. *Journal of the ACM*, 39(1):95–146, 1992.
7. D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Israel, September 2000.
8. J. He, X. Li, and Z. Liu. rCOS: A refinement calculus for object systems. *Theoretical Computer Science*, 2006. accepted, also available as Technical Report 322, UNU-IIST, P.O. Box 3058, Macao SAR China. <http://www.iist.unu.edu/>.
9. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
10. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
11. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
12. P. Kruchten. *The Rational Unified Process – An Introduction*. Addison-Wesly, 2000.
13. C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.
14. Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, ICFEM03, LNCS 2885*, pages 641–664. Springer, 2003.
15. Quan Long, Zongyan Qiu, Zhiming Liu, Lingshuang Shao, and Jifeng He. POST: A case study for an incremental development in rCOS. In Dang Van Hung and Martin Wirsing, editors, *Proc. ICTAC 2005, LNCS 3722*. Springer, 2005.
16. Iman Poernomo. The meta-object facility typed. In *Proc. SAC '06*, pages 1845–1849. ACM Press, 2006.
17. Iman Poernomo. A type theoretic framework for formal metamodelling. In Ralf Reussner, Judith Stafford, and Clemens Szyperski, editors, *Architecting Systems with Trustworthy Components, LNCS 3938*, pages 262–298. Springer, 2006.
18. Gianna Reggio, Maura Cerioli, and Egidio Astesiano. Towards a rigorous semantics of UML supporting its multiview approach. In Heinrich Hußmann, editor, *Proc. FASE2001, LNCS 2029*, pages 171–186. Springer, 2001.