Interactive Transformations from Object-Oriented Models to Component-Based Models

Dan Li[1⋆], Xiaoshan Li[1], Zhiming Liu[2], and Volker Stolz[2,3]

[1] Faculty of Science and Technology, University of Macau, China
[2] UNU-IIST, Macau, China
[3] Dept. of Informatics, University of Oslo, Norway

Abstract. Consider an object-oriented model with a class diagram, and a set of object sequence diagrams, each representing the design of object interactions for a use case. This article discusses how such an OO design model can be automatically transformed into a component-based model for the purpose of reusability, maintenance, and more importantly, distributed and independent deployment. We present the design and implementation of a tool that transforms an object-oriented model to a component-based model, which are both formally defined in the rCOS method of model driven design of component-based software, in an interactive, stepwise manner. The transformation is designed using QVT Relations and implemented as part of the rCOS CASE tool.

Keywords: Model-driven development, OO design model, sequence diagram, component model, model transformation, QVT

1 Introduction

In the rCOS [3, 12] model-driven design of component-based software, the model of the requirements is represented in a component-based architecture. Each *use case* is modeled as a component in the requirements model. The *interface* of the component provides methods through which the actors of the use case interact with the component. The functionality of each method $m()$ of the interface is specified by pre- and post-conditions $m()\{pre \vdash post\}$, and the order of the interactions (called the use-case *protocol*) between the actors and the component as a set of traces of method invocations, graphically represented by a UML sequence diagram. One component may have a *required interface* through which it uses the provided methods of other components. The linkages (dependency) between components forms a static component-based structure modeled as a *component diagram*. The types of the variables of the components, i.e. its objects and data, are modeled by a UML class diagram, that has a textual counterpart specification in rCOS. Therefore the model of the *component-based architecture of the requirements* consists of a model of the component-based static structure (graphically represented as a UML component diagram), a class model (graphically a class diagram), an interaction protocol (graphically a sequence diagram for each component), and a specification of the data functionality of the interface methods.

⋆ On leave from Guizhou Academy of Sciences, Guizhou, China.

In the design, the functionality specification of the interface methods of each component is then refined by decomposition and assignment of responsibilities to objects of the component, obtaining an OO model of object interactions represented by an object sequence diagram. This object sequence diagram refines the sequence diagram of the component (use case). For the purpose of reusability, maintenance, and more importantly, distributed and independent deployment (third party composition) [19], the OO model is abstracted to a model of *interactions of components*, that is graphically represented as a *component sequence diagram* defined in the UML profile for rCOS.

This paper presents the design and implementation of a tool for the transformation of a model of object interaction to a model of component interaction. The tool requires user interactions. In each step of interaction, the users decide which objects will be turned into a component, then the tool automatically performs the model transformation. However, we need to define the criteria for the selection of objects to form a component as the validity conditions of the selection. The tool automatically checks the validity, and the transformation of the sequence diagram is carried out if selection passes the check. The transformation also automatically and consistently transform the static structure and reactive behavior (state machine diagram), obtaining a model of component-based design architecture, that correctly refines the component-based architecture of the requirements.

Through a finite number of transformation steps with valid selection on the OO model of each component in the model of requirements, the object sequence diagram is transformed to a component sequence diagram in which the lifelines represent only components. Also, a complete component diagram is generated with the interface protocols as sets of traces and the reactive behavior modeled by state machine diagrams of the components. The transformations of the OO design of all components thus, one by one, obtain a correct refinement of the model of requirements architecture to a component-based design architecture in which each component in the requirements is a composition of a number of components.

The semantic correctness of the transformation and consistency among the different resulting views (diagrams) can be reasoned about within the rCOS framework. The tool is not only applicable in a top-down design process. If object-interaction models can be obtained from packages (modules) of OO programs, the tool can be used to transform OO programs to components, at least on the modeling level. An extension would be required to transform existing source code within a transformation step.

The paper is organized as follows. We start in Section 2 to discuss the concepts of rCOS model to facilitate the definition of the transformation. We present the major principles of the transformation in Section 3, and describe the implementation of the transformation tool. Section 4 shows how the transformation be applied to a case study. Our conclusions and the related work of this paper are discussed in Section 5.

2 UML Profile of rCOS Models

rCOS provides a notation and an integrated semantic theory to support separation of concerns and allows us to factor a system model into models of different viewpoints [3, 12]. The formal semantics and refinement calculus developed based on it are needed for

the development and use of tools for model verification and transformations. The aim of the development of rCOS tools is to support a component-based software development process that is driven by automatic model transformations. The model transformations implement semantic correctness preserving refinement relations between models at different level of abstraction. It is often the case that the models before and/or after a transformation need to be verified or analyzed, and in that case verification and analysis tools are invoked. The rCOS project focuses on tool development for model transformations, and this paper in particular is about the transformation from object-oriented design models to component-based design models.

UML Profile [15] is a mechanism to support extending and customizing standard UML. This mechanism is carried out by defining stereotypes, tagged values and additional constraints. Through such a UML profile, rCOS models can be supported by standard UML infrastructure and CASE tools, minimizing the effort to develop a new tool, and meeting the requirements for standardization and interoperability.

The rCOS development process involves the following models:

1. The requirements model includes a *component diagram*, a *conceptual class diagram* in which classes do not have methods, and a set of *sequence diagrams*. They all have their formal rCOS textual counter parts. Also, each method of the provided interface has a *pre-* and *post-condition specification*. The sequence diagrams are *component sequence diagrams* in which the lifelines are components, and interactions are inter-component interactions.
2. Each component in the requirements model go through an OO design phase and its sequence diagram is refined into an *object sequence diagram* in which each lifeline is an object, and interactions are intra-object interactions within the component. The conceptual class diagram in the requirements model is also refined into a *design class diagram* in which methods for the intra-object interactions of the object sequence diagrams are assigned to the classes.
3. Then each OO sequence diagram of a component in the previous stage is abstracted to a component sequence diagram; thus the component is decomposed into a composition of a number of components. After the abstraction transformation is done for all components of the requirements model, the component-diagram of the requirements is refined to another component diagram with more hierarchical components being introduced.

Note that the transformation described here is not limited to rCOS models—rather, rCOS just prescribes the wellformedness of the input models, and the semantics of the communication model that will be preserved through the transformation. We refer to our publications [3, 12] for detailed discussions. The rCOS class model is rather a UML standard class model. In the rest of this section, we define the metamodels of rCOS components and sequence diagrams.

2.1 The metamodel of rCOS components

The component model is an essential part of rCOS. Its metamodel is defined by a UML profile diagram shown in Fig. 1, in where an element in the light yellow box represents a

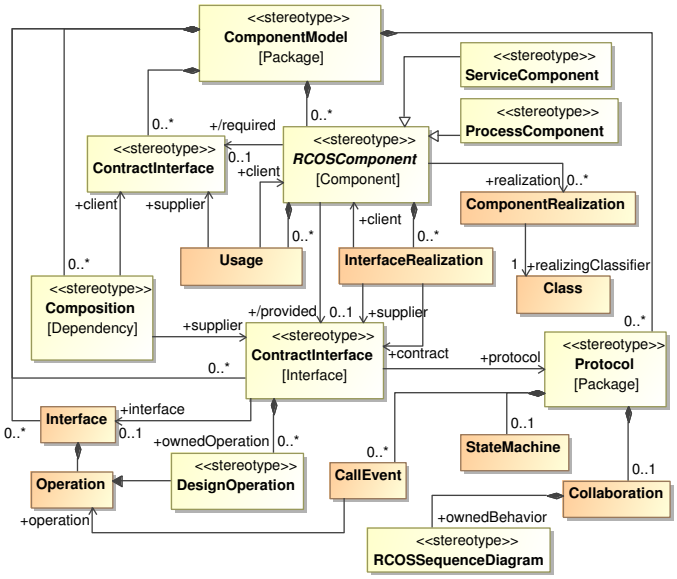

Fig. 1. The metamodel of rCOS component model

stereotype of rCOS, and the ones in the dark yellow boxes are standard UML metamodel elements. In the metamodel, an rCOS component model consists of:

- ContractInterface: Extended from UML *Interface*, a contract interface provides an interaction point for a component, and defines the static portion of a rCOS interface contract. DesignOperations specify the static functionality of an operation. It is defined as an rCOS design in the form of $pre \vdash post$. An rCOS Field, which is not shown in the figure explicitly, is implemented as a UML *Property* of a contract interface. (For ease of layout of the diagram, the same ContractInterface element appears twice in Fig. 1.)
- Protocol: A contract interface has a Protocol that specifies the traces of invocations to the *Operations* of the *Interface* of the contract interface. A protocol contains a *StateMachine*, a *Collaboration* and a set of *CallEvents*. A *call event* is an invocation of an operation of the contract interface, resulting in the execution of the called operation. Especially, here a *Collaboration* owns a UML *Interaction* defined as a RCOSSequenceDiagram, whose metamodel is given in the next subsection.
- RCOSComponent: There are two kinds of components in rCOS, ServiceComponents and ProcessComponents. A service component, for short a component here, provides services to the environments through its *provided* interfaces, and requires services from other components through it required interfaces. rCOS defines separate contracts for the provided interface and required interface of a component. Thus, the metamodel defines one provided contract interface, and optionally a required contract interface.

– We realize the connection between a component and its provided interface using a UML *InterfaceRealization*. A UML *Usage*, a specialized *Dependency* relationship, is used to link a required interface to its owner component. In addition, we define a stereotype Composition, which is also an extension of UML *dependency*, to plug a provided interface of a component to a required interface of another component (here, rCOS component operations do not translate naturally to UML component composition). Furthermore, a component may be realized by a set of classes through *ComponentRealizations*.

2.2 Metamodel of rCOS sequence diagrams

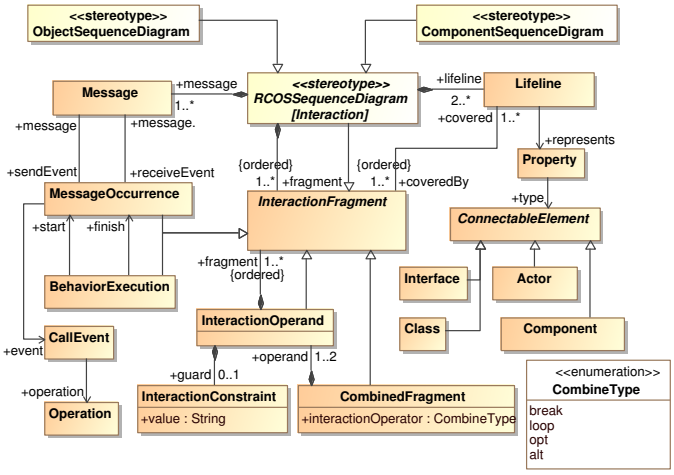

Fig. 2. Metamodel of rCOS sequence diagram

Fig. 2 shows the metamodel of rCOS sequence diagrams. It conforms to the interaction metamodel provided by OMG [15]. In the metamodel, a UML *Interaction* contains a number of *Lifelines*, and a set of *Messages*.

A *message* specifies a communication from a sender lifeline to a receiver lifeline. It has a *sendEvent* and *receiveEvent* which express the *MessageOccurrences* along the lifelines, appearing in pairs. A message occurrence represents the synchronous invocation of an *operation*. The *BehaviorExecution* (green segment of a lifeline in the later diagrams) represents the duration of an operation, and plays no role in our models (yet it is an artefact from the graphical editor).

rCOS has two kinds of sequence diagrams, object sequence diagrams and component sequence diagrams. A lifeline may represent an actor, an object (of a particular class), or a component. When a lifeline represents an object or a component, we call it *object lifeline* or *component lifeline*. A *CombinedFragment* represents a nested block that covers lifelines and their messages to express flow of control, such as an alternative

block (`alt`) or an iteration block (`loop`), with their attached boolean *guard* conditions. Sequence diagrams here do not express recursion.

The two kinds of sequence diagrams are needed to combine both OO design and component-based design in rCOS. The abstract stereotype RCOSSequenceDiagram has subtypes of ObjectSequenceDiagram and ComponentSequenceDiagram, that satisfy the following well-formed conditions, respectively.

1. ObjectSequenceDiagram:
   - There is one lifeline representing an *Actor*, and all other lifelines represent objects or components.
   - Messages are *synchronous calls* to an operation provided by the type of the target lifeline, or a constructor/create messages.
   - A message flow starts with a message from the actor to a single component, from components to components or objects, or from objects to objects, but never from objects to components.

   Therefore, and object-sequence diagram can contain both component and object lifelines, and thus also serves as an intermediate data structure for the transformations, until all objects have been transformed.
2. ComponentSequenceDiagram:
   - One lifeline represents an *Actor*, and all other lifelines represent components.
   - All receive events occur on the lifelines representing components.
   - Each message is a method call to an operation defined in the provided interfaces of the component represented by the target lifeline.
   - There should be a *composition* relation between two component if there is a message between them in a sequence diagram.
   - No create messages exists in the diagram.

The static semantics, i.e. well-formedness of the rCOS sequence diagrams, including the above conditions, is defined by a set of OCL rules in the rCOS CASE tool. These rules are used to automatically check the well-formedness conditions and the structural consistency of the UML model: for example, the object creation event on a lifeline must precede all other events on the lifeline, and a fragment must include both the sender and the receiver of any event occurring in the fragment.

rCOS also has a dynamic model represented by state diagrams. The metamodel of state diagrams is largely the same as the labelled transition systems provided by standard UML state diagrams, where guarded transitions are again linked to interface methods. We leave the metamodel definition out of this paper.

3 Transformation from Object- to Component Sequence Diagrams

We now describe the interactive transformations from an object sequence diagram to a component sequence diagram. The transformations start with an object sequence diagram and a design class diagram. Through a number of steps of interactions between the user and the tool, they generate a component diagram, a component sequence diagram, and the protocols of the provided interface and required interface of each component in the component diagram. In each step, the user selects a set of object lifelines that she

intends to make into a component. The tool will check the validity conditions for this set to form a component. If the selection passes the check, the tool combines the selected object lifelines into a component lifeline, adding a component to the component diagram, and generates the protocols for the component. We describe the principles of the selection and the validity of selection, as well as the generation of a component from the selected lifelines below. As the UML metamodel especially for sequence diagrams is quite verbose as shown in the previous section, we use an alternate, more concise representation here (at the cost of not having established the formal correspondence between the two levels).

3.1 Selection of object lifelines

First, one object lifeline is designated as the *controller object* of the selection by the user. The principles for picking such a control object not only depend on checkable conditions of the object but also on design considerations of reusability, maintainability, and organization of the system being modeled. The major checkable condition is that this object should be a *permanent object* in the sequence diagram. This means it should have existed before the start of the execution of the sequence diagram (specified by the precondition of the first message), and it will not be destroyed during the execution (rCOS does not have destructor methods). This also includes software objects representing the control of physical devices, such as barcode readers, controllers of printers, lights, and operating system objects, such as the system clock.

Then the selection of further objects should be made by the user with consideration of the following conditions and principles:

1. any object lifeline that is a receive end of a creation event from a lifeline that is already included in the selection must be selected,
2. the objects in the selection must be *strongly connected*, i.e. for any lifeline ℓ in the selection there is at least one message path from the controller object to ℓ,
3. consider low coupling and high cohesion principle that the selected lifelines have more intensive interaction with each other than with lifelines outside the selection.
4. lifelines that represent objects which will be deployed on different nodes of a distributed system should not be included in the same selection.

The first two conditions are must condition and can be easily checked, as discussed in the next sub-section. The third condition is a desirable principle, and the fourth is a platform dependent condition. The latter two can never lead to an inconsistent model, but to a model that does not capture the intentions correctly, and a detailed discussion of them is out the scope of the paper.

3.2 Validating the lifeline selection

Given an object sequence diagram D, we define some notations for the describing the validation of a selection. We use *D.lines* to denote the set of all lifelines of D, *D.messages* the set of messages, and a message is represented by $m[\ell_i, \ell_j]$ as an invocation of m of ℓ_j from ℓ_i. *Create*-messages indicate constructor invocations.

Let $D.selection \subseteq D.lines$ be a selection, and ℓ_c the designated controller object, and define $D.rest = D.lines - D.selection$. Further, we define

IntraM	$= \{m[\ell_i, \ell_j] \ : \ \ell_i, \ell_j \in D.selection\}$	Messages among the selected lifelines
InM	$= \{m[\ell_i, \ell_j] \ : \ \ell_i \in D.rest \wedge \ell_j \in D.selection\}$	Incoming messages to selected lifelines
OutM	$= \{m[\ell_i, \ell_j] \ : \ \ell_i \in D.selection \wedge \ell_j \in D.rest\}$	Outgoing messages from selected lifelines
OutsideM	$= \{m[\ell_i, \ell_j] \ : \ \ell_i, \ell_j \in D.rest\}$	Messages outside the selected lifelines

A lifeline ℓ in sequence diagram *D* can be either an object lifeline, denoted by $type(\ell) = Class$, or a component lifeline, denoted by $type(\ell) = Component$. Now we define the conditions below for checking the validity of a selection.

1. All lifelines selected must be object lifelines

   $\forall \ell \in D.selection \cdot type(\ell) = Class$

2. The controller object ℓ_c must be a permanent object. This is done by checking it is not on the receive end of an object creation message.

   $\forall \ell \in D.lines \cdot (create[\ell, \ell_c] \notin D.messages)$

3. The transformation starts with those lifelines that directly interact with the actor, then those directly receiving message from the lifelines that have been made into component lifelines. Therefore any incoming message to the current selection should be from either the actor or a component lifeline

   $\forall m[\ell_i, \ell_j] \in InM \cdot (type(\ell_i) = Actor \vee type(\ell_i) = Component)$

4. Creation messages can only be sent between lifelines inside the selection or between objects outside the selection

   $\forall \ell_i, \ell_j \in D.lines \cdot (create[\ell_i, \ell_j] \in IntraM \vee \text{create}[\ell_i, \ell_j] \in OutsideM)$

5. Any incoming message to the selection is received either by the controller object or by a lifeline which has a direct path of message from the controller object

   $\forall m[\ell_i, \ell_j] \in InM \cdot (\ell_j = \ell_c \vee \exists m[\ell_c, \ell_j] \in IntraM)$

6. The lifelines of the selection must be strongly connected, meaning that for any selected lifeline ℓ, there must be a path of messages from the controller object

   $m[\ell_c, \ell_1], m_1[\ell_1, \ell_2], \ldots, m_i[\ell_i, \ell]$

Notice that Conditions 4&6 are closure properties required of the section, and that the initial object-sequence diagram of a use case in rCOS always has a use case controller object that satisfies Conditions 2,3&5. Using induction on the number of lifelines, these conditions all together ensures existence of a valid selection for any well-formed sequence diagram that contains object lifelines. Every OO sequence diagram can be translated into the trivial component sequence diagram which internalises *all* object lifelines into the controller.

3.3 Generating a component from selected lifelines

If the selection passes the validity checking, the transformation will be executed to generate the target models, otherwise an error message is fed back to the tool user. The transformation is specified in the QVT relational notation (see Section 3.4). For the understandability to the formal specification community, we describe the specification in terms of the relation between the source model and the target model, similar to the pre- and postcondition specification of a program.

Given a source sequence diagram *D*, that is an object sequence diagram, and a valid selection *D.selection*, let *D'* denote the target sequence diagram after one step of the transformation. For a lifeline ℓ in *D* (or *D'*), we use *type*(ℓ, D) to denote the type of the lifeline ℓ in *D* (respectively *type*(ℓ, D') in *D'*). For a component lifeline ℓ in *D* (or *D'*), *pIF* denotes the provided interface of the component that ℓ represents, and *rIF* the required interface. We now describe the relation between *D* and *D'* as the conjunction of the following predicates.

1. The controller object ℓ_c in *D* is changed to a component lifeline in *D'*

$$\begin{aligned} &\ell_c \in \textit{D.selection} \wedge \textit{type}(\ell_c, D) = \textit{Class} \\ \wedge\ &\ell_c \in D'.\textit{lines} \wedge \textit{type}(\ell_c, D') = \textit{Component} \end{aligned}$$

2. An incoming message to the selection in *D* becomes an invocation to the interface methods of ℓ_c in *D'*

$$\forall m[\ell_i, \ell_j] \in \textit{InM} \cdot (m[\ell_i, \ell_c] \in \textit{D'.messages} \wedge m \in \textit{pIF}(\ell_c))$$

   Notice that the order of the messages and fragments are not to be changed.
3. All the intra-object interactions in the selection in D are collapsed, more precisely hidden inside the component ℓ_c

$$\forall m[\ell_i, \ell_j] \in \textit{IntraM} \cdot (\ell_i, \ell_j \notin \textit{D'.lines} \wedge m[\ell_i, \ell_j] \notin \textit{D'.messages})$$

4. All the outgoing messages from the selection become sending messages from the component that ℓ_c represents in D', with the order and fragments preserved, and they become the required methods of the component

$$\forall m[\ell_i, \ell_j] \in \textit{OutM} \cdot (m[\ell_c, \ell_j] \in \textit{D'.messages} \wedge m \in \textit{rIF}(\ell_c))$$

5. No lifelines and messages outside the selection are changed

$$\forall m[\ell_i, \ell_j] \in \textit{OutsideM} \cdot (m[\ell_i, \ell_j] \in \textit{D'.messages})$$

From the definition of the resulting sequence diagram *D'*, its static counterparts, the components can be defined. The change for the component diagram can be specified in a similar way. The protocols of the provided interface *pIF*(ℓ_c) and the required interface *rIF*(ℓ_c) of the newly constructed component ℓ_c in *D'* will be generated.

Next, we give an intuition into how the relations defined above can be directly put to use through QVT-Relations.

3.4 Implementation of the transformation

The object sequence diagram to component sequence diagram transformation is implemented through the QVT Relations language using the QVTR-XSLT tool we recently developed [10]. The MOF 2.0 Query/View/Transformation (QVT) [14] is a model transformation standard proposed by OMG. QVT has a hybrid declarative/imperative nature. In its declarative language, called QVT Relations (QVT-R), a transformation is defined as a set of *relations* between the elements of source metamodels and target metamodels. QVT-R has both textual and graphical notations, and the graphical notation provides a concise, intuitive way to specify transformations.

The QVTR-XSLT tool supports the graphical notation of QVT-R. It provides a graphical editor in which a transformation can be specified using the graphical syntax, and a code generator that automatically generates executable XSLT [21] programs for the transformation. The tool supports *in-place transformations* so we can focus on defining rules only for the parts of a model we want to change. Multiple input and output models are also supported in the tool.

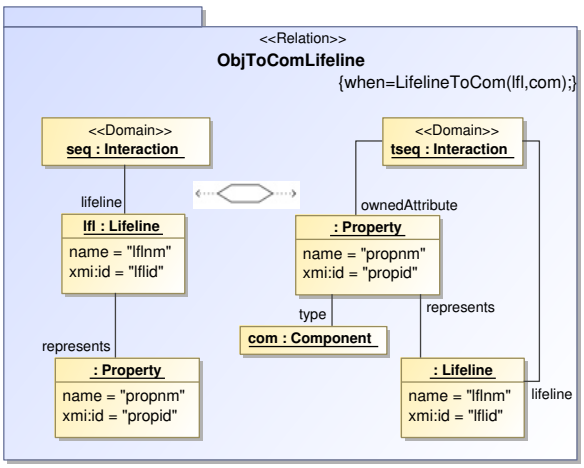

Fig. 3. An example of a QVTR relation

In the graphical notation, a *relation* defines how two object diagrams, called *domain patterns*, relate to each other. Fig. 3 illustrates an example QVT relation in graphical notation which specifies the generation of a component lifeline from an object lifeline. Starting from the root object *seq* tagged with label $\ll Domain \gg$, the source domain pattern (left part) of the relation consists of a *Lifeline lfl* with its representing *Property* under the *seq*. The target domain pattern (right part) has a similar structure. The patterns are used for structural matching in the source- and target model, respectively.

When the relation is executed, the source domain pattern is searched in the source model. If a match is found, the *lifeline* and the *property* are bound to instances of source

model elements. The target domain pattern of the relation acts as a template to create objects and links in the target model. In this example, the target domain pattern creates a *lifeline* object and a *property* object. Both objects own a *name* and an *xmi:id* attributes. These two attributes get values from the corresponding model instances bound by the source domain pattern. Moreover, the *property* object of the target model has now the association *type* set to the component *com*, which is bound (and possible created) by invoking relation *LifelineToCom* in the *when* clause. These clauses specify additional matching conditions and can either refer to other relations, or OCL expressions.

At the implementation level, a complete model consists of a UML model and a DI (diagram interchange) [13] model. The former contains the abstract syntax information that is described in Section 2, and it is stored in Eclipse Modeling Framework (EMF) XMI format, which is supported by many UML CASE tools. The latter contains the layout information in the form of UML 2.0 Diagram Interchange standard [13]. In fact, these two models are technically separate models and saved in different XML files. When the UML model is modified by the transformation, the DI model must be synchronously updated in order to correctly display the corresponding diagrams. The changes to the DI model are also specified using QVT-R, and transformed by the QVTR-XSLT tool. The resulting diagrams for the case study are the result of those transformations after minimal visual cleanup. The transformation is specified by three transformation models. In total, they contain 105 relations, and 45 functions and queries. About 6300 lines of XSLT code are generated for the implementation of the transformation.

To support the rCOS methodology, we have developed a CASE tool [4] with graphical interfaces for designing use cases, classes, component-, sequence- and state diagrams, and the syntactic consistency among these views can be checked. The tool is implemented as an Eclipse-plugin on top of the Eclipse Graphical Modelling Framework and TOPCASED [16]. We have integrated the XSLT programs of the transformation into the user interface of the tool. A user can select a group of lifelines from the interface, and then the XSLT transformation programs are invoked by the tool with these lifelines as parameters. If these lifelines are allowed to become a component, the transformation is executed and the user interface will be automatically refreshed to show the transformation results.

4 Case Study

The Common Component Modelling Example (CoCoME) [3, 17] describes a trading system that is typically used in supermarkets. This case study deals with the various business processes, including processing sales at a cash desk, handling payments, and updating the inventory. The system maintains a catalog of product items, as well as the amount of each item available. It also keeps the historical records of sales; each of them consists of a number of line items, determined by the product item and the amount sold.

At the end of the object-oriented design stage, we get a design model which contains a set of design class diagrams and object sequence diagrams. Fig. 4 shows a simplified version of the design class diagram for the CoCoME example, where the class *CashDesk* is the control class. Fig. 5 depicts the object sequence diagram of use case

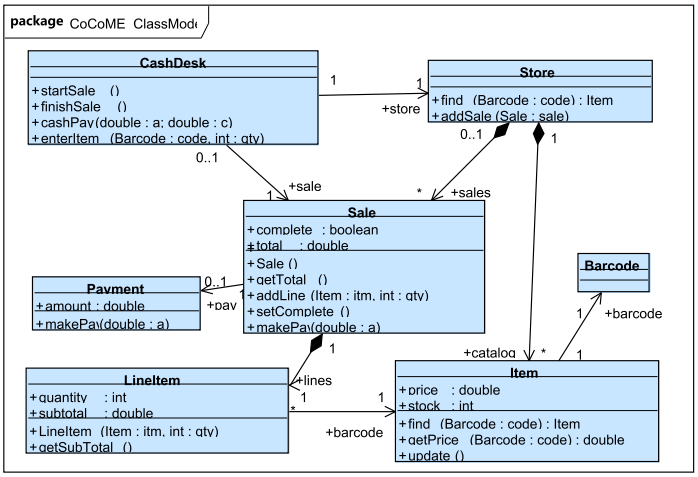

Fig. 4. The design class model of CoCoME

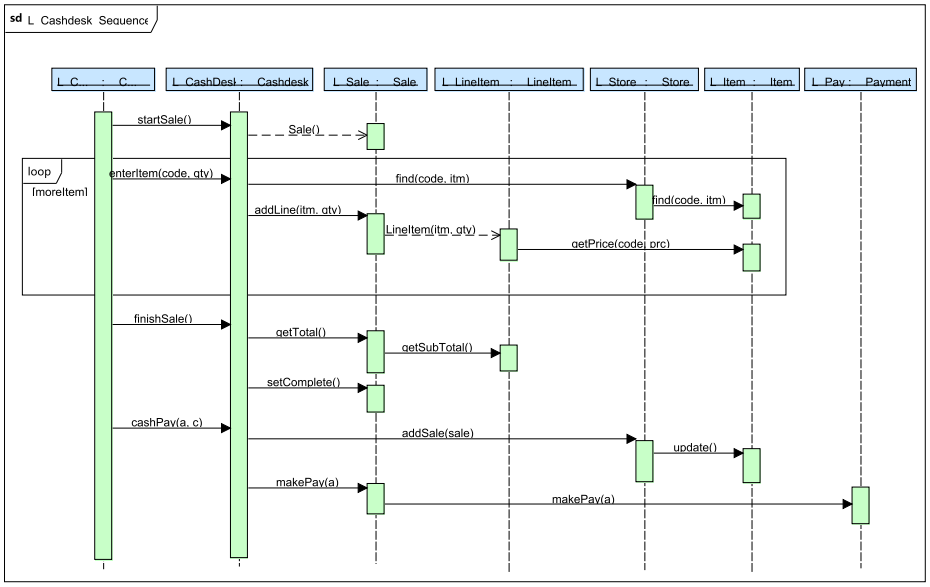

Fig. 5. The object sequence diagram of usecase *process sale*

process sale, which describes the check out process: a customer takes the products she wants to buy to a cash desk, the cashier records each product item, and finally the customer makes the payment. Applying the transformations discussed in the previous sections, we transform the object sequence diagram into an rCOS component sequence diagram in a stepwise, incremental manner. Meanwhile the object model automatically evolves to a component-based model.

The object sequence diagram of Fig. 5 consists of seven lifelines. The leftmost lifeline is the *Actor*, and followed by lifelines *L_CashDesk*, *L_Sale*, *L_LineItem*, *L_Store*, *L_Item* and *L_Pay*, representing objects of class *CashDesk*, *Sale*, *LineItem*, *Store*, *Item* and *Payment*, respectively. Based on our interpretation of the case study, we decide to apply the transformation three times.

The first step deals with the lifeline *L_CashDesk*, which is directly interacting with the actor. Since lifeline *L_Sale* is created by *L_CashDesk*, and *L_LineItem* is created by *L_Sale*, they have to be in the same component. As shown in Fig. 6, we select these three lifelines from the sequence diagram, set *L_CashDesk* as the controller object (main lifeline in the figure), and transform them into a service component *COM_L_CashDesk*. The component has a provided interface *ConInter_L_CashDesk* and a required interface *RInter_L_CashDesk*. The resulting sequence diagram is shown in Fig. 8, in which lifeline *L_CashDesk* now represents the new component, and lifelines *L_Sale* and *L_LineItem*, along with their internal messages, are removed from the diagram.

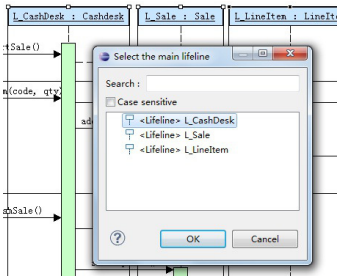

Fig. 6. Select lifelines

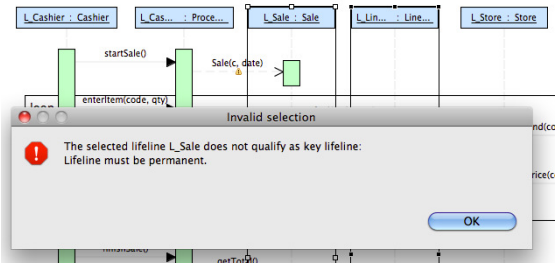

Fig. 7. Validation error message

As we mentioned before, the tool will check whether the selected lifelines can be transformed to a component, and provides an error message if the selection is not valid. For instance, if we choose lifelines *L_Sale*, *L_Store* and *L_Item* to become a component, the tool will display an error message, as shown in Fig. 7.

For the second transformation, we select the lifelines *L_Store* and *L_Item* from the sequence diagram of Fig. 8, and indicate *L_Store* as the controller object. Since class *Store* is composed with class *Item*, the transformation is allowed, and the two lifelines are transformed into a service component *COM_L_Store*.

As the result of the second transformation, the lifeline *L_Store* now represents the component *COM_L_Store*. Accordingly, the component diagram is changed, where the provided interface *ConInter_L_Store* is plugged to the required interface *RInter_L_Cash Desk* (we only show the final resulting component diagram later in Fig. 11).

For each generated component, we also generate an rCOS protocol, which consists of a sequence diagram and a state diagram, for its provided interface. The protocol for component *COM_L_Store* is shown in Figs. 9 & 10. The left part of the sequence diagram in Fig. 9 specifies the interactions of the component with its environment (represented by a fresh actor), and the right part defines the interactions between the com-

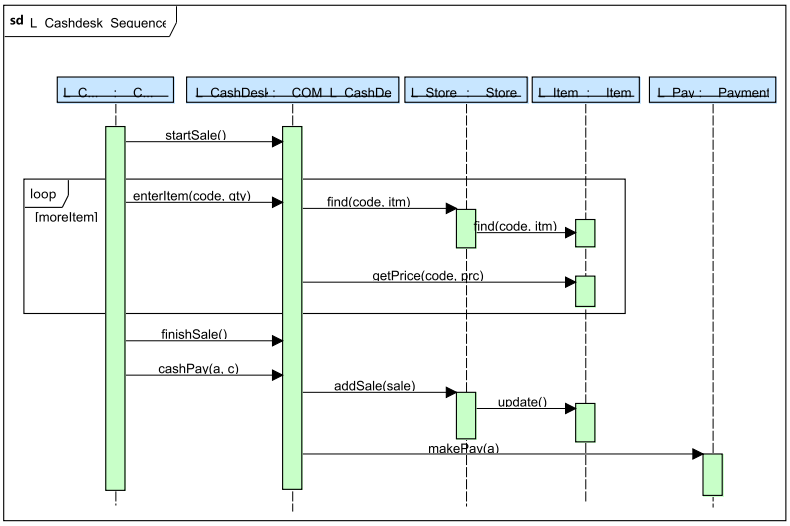

Fig. 8. The sequence diagram after the first transformation

ponent and its internal objects. We notice that a message originally sent from a non-selected lifeline and received by another selected lifeline, such as the *getPrice* message in Fig. 5, now becomes two messages. The first *getPrice* message is received by the component *COM_L_Store*, and then delegated to the original receiving lifeline *L_Item* using the second *getPrice* message.

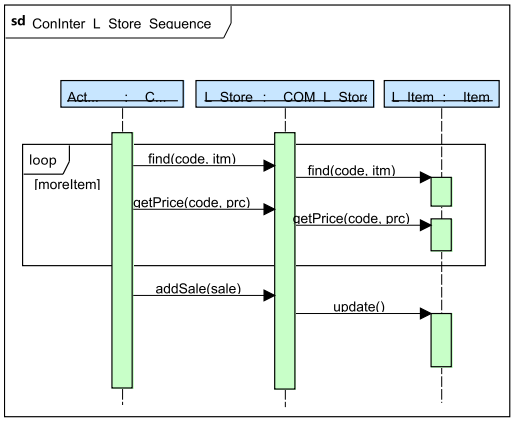

Fig. 9. Sequence diagram of *COM_L_Store*

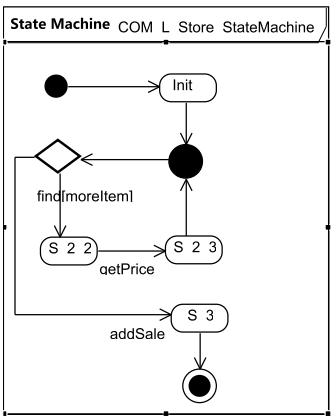

Fig. 10. State diagram of *COM_L_Store*

In the third transformation, we turn *L_Pay*, the only object lifeline left, into component *COM_L_Pay*. Thus we get the final component diagram shown in Fig. 11, which

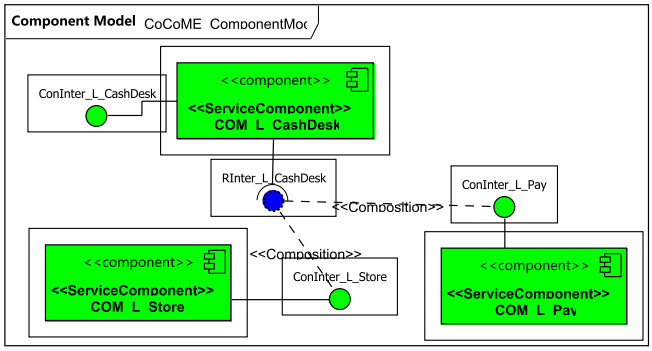

Fig. 11. Final component diagram of the CoCoME example

depicts the relationships among the three components of the model. We obtain the final component sequence diagram, in which all lifelines represent components, except the one representing the actor (see Fig. 12), fulfilling the structural well-formedness rules of component sequence models as discussed in Section 2.

Through applying the object sequence diagram to component sequence diagram transformation three times, we have successfully developed the design model of CoCoME into a component model. The component model includes component sequence diagrams and component diagrams to define the relationship of components. Each component has its provided/required interfaces, as well as a protocol, that consists of a sequence diagram and a state diagram, to define the behaviors of the component.

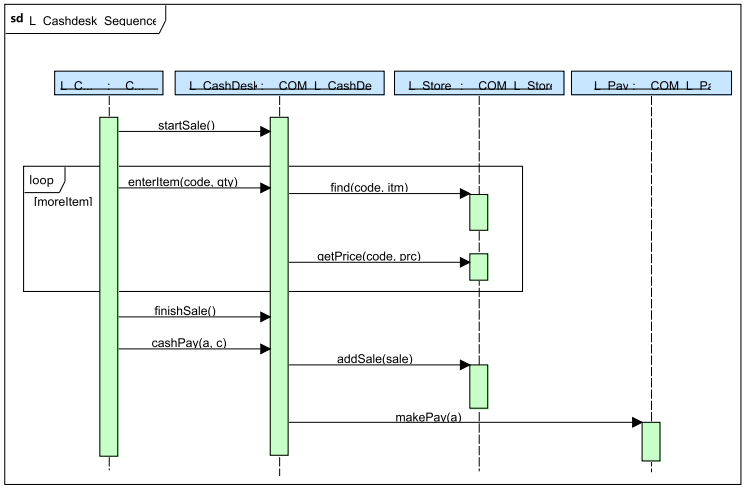

Fig. 12. Final rCOS component sequence diagram for usecase *process sale*

5 Conclusion

A major research objective of the rCOS method is to improve the scalability of semantic correctness preserving refinement between models in model-driven software engineering. The rCOS method promotes the idea that component-based software design is driven by model transformations in the front end, and verification and analysis are integrated through model transformations.

As nearly all existing component-based technologies are realized in object-oriented technologies, most design processes start with an OO development process and then at the end of the process an OO design is directly implemented by using a component-based technology, such .COM or .NET. It is often the case that an OO program is developed first and then it is transformed into component software. Our approach improve this practice by allowing a component-based model of the requirements, and a seamless combination of OO design and component-based design for each components in the requirements. The combination is supported by the interactive transformations from OO design to component-based design presented in this paper, in a stepwise and compositional manner. This allows the object-oriented and component-based design patterns to be used in the OO design and captured in the specification of the transformation.

In the tool implementation, the transformation is specified in a subset of the graphical QVT Relations notation. The correct implementation of the interactive transformation requires the definition of a UML profile of the abstract syntax of the rCOS model that is presented in the paper. The QVT specification of the transformation is automatically transformed to an executable XSLT program, that can be run through an Eclipse-plugin. The presented technique and tool can be combined with reverse engineering techniques for transformation of OO programs into component-based programs.

5.1 Related work

As a natural step of model driven development, object-oriented models are further evolved to component-based models to get the benefits of reusability, maintenance, as well as distributed and independent deployment. Surveys of approaches and techniques for identification reusable components from object-oriented models can be found in [2,20]. Based on the principle of "high cohesion and low coupling", researchers try to cluster classes into components. The basic ideas are: calculate the strength of semantics dependencies between classes and transform them into the form of weighted directional graph, then cluster the graph using graph clustering or matrix analysis techniques [20]. Using clustering analysis, components with high cohesion and low coupling are expected to be obtained in order to reduce composition cost.

Particularly, since use cases are applied to describe the functionality of the system, the work of [18] focuses on applying various clustering methods to cluster use cases into several components. In [6], the static and dynamic relationships between classes are used for clustering related classes in components, where static relationship measures the relationship strength, and dynamic relationship measures the frequency of message exchange at runtime. COMO [9] proposed a method which measures inter-class relationships in terms of *create*, *retrieve*, *update* and *delete* (CRUD) operations of model elements. It uses dynamic coupling metric between objects to measure the

potential number of messages exchanged. All above approaches are based on clustering algorithms, which makes them much different from our approach, where transformations are applied at the *design stage* by a human.

Identifying reusable components from object-oriented models was considered to be one of the most difficult tasks in the software development process [6]. Most existing approaches just provide general guidelines for component identification. They lack more precise criteria and methods [5]. Because of the complexity of source information and the component model itself, it is not advisable for component designers to manually develop component-based models from object-oriented models [20]. Alas, there are almost no (semi)-automatic tools to help designers in the development process [18]. The work of the paper makes a useful attempt to address this problem, and provide a tool supporting.

Sequence diagrams have of course already been used informally in UML-based modeling since their conception. Recently, [7] presents a rigorously defined variant called "Life Sequence Charts" with tool support to use them for system design. The focus there is however not on component modeling, but giving a formal semantics to sequence charts for synthesis.

In [3], we have studied this top-down development process, carried out by hand, for the CoCoME case study. Our process is motivated by an industrial CASE tool, MASTERCRAFT [11]. There, the focus is on the design and refinement of the relational method specifications using the rCOS language [8, 22].

5.2 Future work

There are still many challenges in the automation of model transformations, especially on the level of method specifications, such as applying the expert pattern in the object-oriented design stage. It is not enough to only provide a library of transformations, but more importantly, the tool should provide guiding information on which rule is to be used [12]. Since our methodology (unsurprisingly) coincides with textbook-approaches to design of OO- and component software, we hope that the tool can also become a foundation for education in software engineering. It should guide the user through the different stages with recommendations, e.g. where detail should be added to the model, or where refinement is necessary. Based on metrics, the tool could also propose concrete transformation parameters. It is also difficult to support consistent and correct reuse of existing components when designing a new component. We will continue working in this direction to overcome these challenges.

The rCOS Modeler that implements the transformations discussed here can be downloaded together with examples from `http://rcos.iist.unu.edu`.

Acknowledgements Partially supported by the ARV and GAVES grants of the Macau Science and Technology Development Fund, and the Guizhou International Scientific Cooperation Project G[2011]7023 and GY[2010]3033.

References

1. F. Arbab and M. Sirjani, editors. *Fundamentals of Software Engineering (FSEN 2009)*, volume 5961 of *Lecture Notes in Computer Science*. Springer, 2010.
2. D. Birkmeier and S. Overhage. On Component Identification Approaches - Classification, State of the Art, and Comparison. In *CBSE'09*, pages 1–18, 2009.
3. Z. Chen, Z. Liu, A. P. Ravn, V. Stolz, and N. Zhan. Refinement and verification in component-based model driven design. *Sci. Comput. Program.*, 74(4):168–196, 2009.
4. Z. Chen, C. Morisset, and V. Stolz. Specification and validation of behavioural protocols in the rCOS modeler. In Arbab and Sirjani [1], pages 387–401.
5. M. Choi and E. Cho. Component Identification Methods Applying Method Call Types between Classes. *J. Inf. Sci. Eng*, 22:247–267, 2006.
6. M. Fan-Chao, Z. Den-Chen, and X. Xiao-Fei. Business Component Identification of Enterprise Information System: A Hierarchical Clustering Method. *Proc. of the 2005 IEEE Intl. Conf. on e-Business Engineering*, 2005.
7. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag, 2003.
8. J. He, Z. Liu, and X. Li. rCOS: A refinement calculus of object systems. *Theor. Comput. Sci.*, 365(1-2):109–142, 2006.
9. S. Lee, Y. Yang, F. Cho, S. Kim, and S. Rhew. COMO: A UML-based component development methodology. In *6th Asia Pacific Softw. Eng. Conf.*, pages 54–61. IEEE, 1999.
10. D. Li, X. Li, and V. Stolz. QVT-based model transformation using XSLT. *SIGSOFT Softw. Eng. Notes*, 36:1–8, January 2011.
11. Z. Liu, V. Mencl, A. P. Ravn, and L. Yang. Harnessing theories for tool support. In *Proc. of the Second Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 371–382. IEEE Computer Society, Aug. 2006.
12. Z. Liu, C. Morisset, and V. Stolz. rCOS: theory and tools for component-based model driven development. In Arbab and Sirjani [1], pages 62–80.
13. Object Management Group. UML 2.0 Diagram Interchange Specification. http://www.omg.org/cgi-bin/doc?ptc/2003-09-01, September 2003.
14. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, Dec. 2009.
15. Object Management Group. Unified Modeling Language: Superstructure, version 2.3, May 2010. http://www.omg.org/spec/UML/2.3/Superstructure.
16. N. Pontisso and D. Chemouil. TOPCASED Combining formal methods with model-driven engineering. In *ASE'06: Proc. of the 21st IEEE/ACM Intl. Conf. on Automated Software Engineering*, pages 359–360, Washington, DC, USA, 2006. IEEE Computer Society.
17. A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008.
18. G. Shahmohammadi, S. Jalili, and S. M. H. Hasheminejad. Identification of System Software Components Using Clustering Approach. *Journal of Object Technology*, 9(6):77–98, 2010.
19. C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002.
20. Z. Wang, X. Xu, and D. Zhan. A survey of business component identification methods and related techniques. *International Journal of Information Technology*, 2(4):229–238, 2005.
21. WWW Consortium. XSL Transformations (XSLT) Version 2.0, W3C Recommendation. http://www.w3.org/TR/2007/REC-xslt20-20070123/, January 2007.
22. L. Zhao, X. Liu, Z. Liu, and Z. Qiu. Graph transformations for object-oriented refinement. *Formal Aspects of Computing*, 21(1–2):103–131, Feb. 2009.