2013 22nd Australian Conference on Software Engineering

Support Formal Component-based Development with UML Profile

Dan Li[1,4], Xiaoshan Li[1], Zhiming Liu[2] and Volker Stolz[2,3]
[1]Faculty of Science and Technology, University of Macau, Macau, China
[2]UNU-IIST, Macau, China
[3]University of Oslo, Norway
[4]Guizhou Academy of Sciences, Guiyang, China

***Abstract*—Integrating formal methods into UML opens up a way to complement UML-based software development with precise semantics, development methodologies, as well as rigorous verification and refinement techniques. In this paper, we present an approach to integrate a formal method to practical component-based model driven development through defining a UML profile that maps the concepts of the formal method as UML stereotypes, and implementing the profile into a CASE tool. Unlike most of the previous works in this vein, which concentrate on verifying the correctness of the models built in the development process, we focus on how the full development process can be driven by applying the refinement rules of the formal method in an incremental and interactive manner. The formal method we adopt in this work is the *refinement for Component and Object Systems* (rCOS). We demonstrate the development activities in the CASE tool using an example.**

***Keywords*—Model-driven development, component-based modeling, UML profile, formal methods, rCOS**

I. Introduction

Nowadays, the demand for building more complex and higher quality software systems is increasing, that has led to an urgent need for new languages, methods, and tools. *Component-based development* (CBD) and *model driven development* (MDD) are two of the most important and promising paradigms to deal with the demand. MDD promotes the use of models and model transformations for developing software systems. It supports the principle of *separation of concerns* that allows us to include in each view only the information which is relevant to the immediate purpose. On the other hand, CBD emphasizes on decoupling a system into partially independent and reusable modules - components. A system can be built by gathering together various components, each may have separate concerns and offer different functionalities. Combining the advantages of both techniques, *component-based model driven development* (CB-MDD) is regarded as an effective way to develop complex systems [1].

As a widely accepted standard modeling language, UML [2] provides a certain degree of support for both MDD and CBD. In order to specify a model from different views, UML defines thirteen types of diagrams which are divided into three categories: representing static structure, behavior and interaction. With the help of *component diagrams* and a number of modeling constructs, such as *connector*, *port* and *required/provided interface*, UML provides a semi-formal means for specifying components, their composition and deployment. But from a practical point of view, the lack of a precise semantics, numerous different modeling concepts, and lack of a development methodology make applying the "pure" UML for MDD and CBD difficult in software engineering [3].

The best known MDD initiative, OMG's *Model-Driven Architecture* (MDA) [4], proposed a set of standards for model-based development. MDA suggests a development strategy that focuses on separately defining models for the application domains (PIMs, Platform Independent Models) and models about platform specific techniques (PSMs, Platform Specific Models), and combining the two kinds of models together to build platform specific models or implementations using model transformations. However, MDA also lacks a systematic methodology about how to develop a complete and consistent PIM. In addition, other development methods, such as the *Rational Unified Process* (RUP), provide guidance for the developers to follow in software design, but the development activities and steps are in general not concrete and fine-grained enough. This brings difficulty to the automation the development process.

MDD and CBD need sound theoretical foundations and strong methodology and tool supports. This challenge lies in that the semantic theory must support separation of concerns to allow us to factor the system model into models of different views and to consistently integrate models together under an execution semantics of the whole system. The models are required to be maintainable and verifiable [5]. We also need a systematic development methodology which covers the full evolution process of models, manipulating at, and within, all levels, from requirements through architecture and design, to executable programs. More importantly, the theory and the methodology should be practical enough to be automated into tool supports. In this regard, the use of formal methods, with their rigorous mathematical foundations for system specification, verification and refinement, could be much help to address these challenges.

Integrating formal methods into UML opens up a powerful approach to support MDD and CBD. A formal method can supply UML with a precise semantics, and formal consistency verification and refinement techniques. For the formal method's side, the advantage of combining with UML is the possibility to specify a system with a unified, standardized graphical notation, and to exploit many available UML tools for saving development effort and time. Thus some inherent shortcomings of formal methods, such as the complexity of the notations, the mathematical background required for comprehending the formal background, and the lack of easy-

978-0-7695-4995-8/13 $26.00 © 2013 IEEE
DOI 10.1109/ASWEC.2013.31

to-use tools, can be efficiently overcome. Formal methods could therefore potentially reach a wider user-base.

Many works have been done on the integration of formal methods and UML to support MDD [6]–[8]. Most of them focus on translating the multiple views of a model into a formal specification, thus the formal semantics of the model is defined, and existing analysis tools of the formal language, such as type-checkers, theorem provers, and model checkers, can be used to prove the correctness of the model. However, how to systematically support the development process of MDD using the development methodologies and refinement rules of the formal methods, and how to automate the development process, are not the focus of these works.

In this paper, we present an approach to integrate a formal method, the *refinement for Component and Object Systems* (rCOS) [5], [9], [10], into UML to support MDD and CDB. We focus on how the full development process can be supported in an incremental and interactive manner though applying the rCOS refinement rules, and how the object-oriented and component-based techniques can be seamlessly combined and used in the development process. To integrate rCOS and UML, we adopt the standard strategy of customizing UML using its built-in extension mechanism by defining a UML profile, which describes how UML model elements are extended to support usage in a particular modeling method. We have implemented the profile in a CASE tool, through which the rCOS techniques and methodology can be applied to component-based software development in a standard, model-driven way.

The remainder of the paper is organized as follows. We briefly introduce rCOS major concepts using a metamodel in Section II. Based on this, Section III presents a UML profile for rCOS and its implementation in a CASE tool. Section IV discusses how the development process is supported by the tool using an example. We mention some related work and outline the conclusions in Section V.

II. Basic Concepts of rCOS

rCOS provides a notion and an integrated semantic theory to support separation of concerns, and allows us to factor a system into models of different viewpoints, including static structure, interactions and functionalities, and ensure the consistency among them [9]. The formal semantics is founded on an extension of UTP [11], with a modeling notation with a verification and refinement calculus. For modeling component-based architectures, rCOS defines operators for connecting components, and constructs for defining glue processes.

In addition, rCOS follows a use-case driven, incremental and iterative development methodology that combines object-oriented and component-based design and analysis techniques. The development process goes through the stages of requirements elicitation, object-oriented design, logical component-based design, and through to coding. 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. The main advantage of the rCOS methodology is that we can ensure consistency of the multi-view specifications at different abstract levels [9].

The theoretic background of rCOS has been well-studied in literature such as [5], [9], [10], [12], [13]. Here we discuss only the major concepts and features of the rCOS language using a *Meta Object Facility* (MOF) compliant metamodel, so as to facilitate the definition of the rCOS profile. The description mainly focuses on the syntactic aspects.

A metamodel is a mechanism for defining model concepts, their relationship and constraints, in a comprehensive way. The rCOS metamodel consists of four packages: *Type, Class, Component* and *RCOSP*. The latter three ones have dependencies on the *Type* package. The *RCOSP* package defines the abstract syntax of the specification language, which can be used to specify the rCOS models and method bodies textually. Some concepts, such as *Design*, *Expression*, *Command* and *Predicate*, are defined in the package. Because of space limitations, we will not discuss the *RCOSP* package in this paper.

A. Types and classes of rCOS

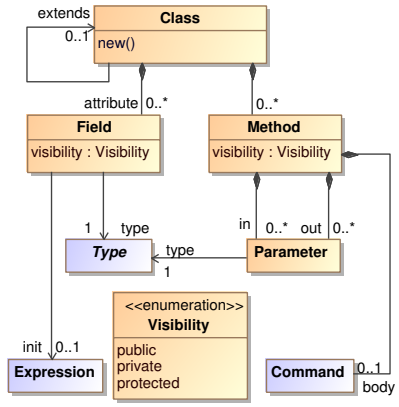

Fig. 1. Class package of rCOS

The *Type* package comprises data types used in rCOS. A *type* may be a *class* (derived from UML), which is the type of an object in a system, or may be a *primitive type*, such as *boolean*, *integer*, or *string*. A *collective type*, such as *Set*, is a collection of elements whose type is already defined in the package. For example, *Set(T)* is the type of sets of type *T*. We define a set of operations, such as *add*, *contains* and *remove* for the *collective type* with their standard semantics. Moreover, like most OO program languages, rCOS has its class model with the notions of classes, associations or attributes, and single inheritance, to represent the application domain concepts. Fig. 1 depicts the rCOS class metamodel.

B. Components in rCOS

The *component* package, as shown in Fig. 2, defines the component-based modeling concepts of rCOS, again borrowing from UML. An rCOS *interface*, which is a syntactic notion providing an interaction point for a component, is a declaration of a set of *fields* and *methods*. In addition, an interaction *protocol* of the interface is a set of finite *traces* over the *events* of method calls. In principle, regular expressions, finite state machines, or CSP processes can be used to describe the traces. The protocol defines the permissible sequences of method invocations for the environment to follow when interacting

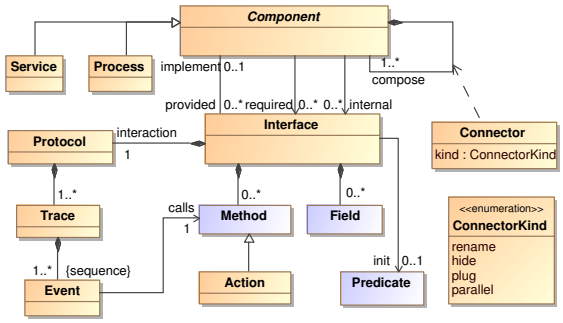

Fig. 2. Component package of rCOS

with the interface. Moreover, the *initial* predicate specifies the allowable initial states of the interface.

A component may provide the clients with services that are defined by the *provided interfaces*. The component is responsible for implementing its provided interfaces, either by the component itself or through some (interface) classes. On the other side, a component may need to use the services provided by other components. These services are called required services and are represented as *required interfaces*. Obviously, the required protocol is determined by the protocol of the provided interfaces.

In rCOS, there are two kinds of components. A *service component*, *component* for short, has provided interfaces, and optionally required interfaces. It is externally passive and only interacts with the outside when a provided service is requested. On the other hand, a *process component*, *process* for short, is active and has its own control thread. A process only has required interfaces and it actively invokes services of other components.

The notion of *component composition* is essential for component-based design. In rCOS, we define the basic composition operators for *renaming* interface methods, *hiding* interface methods, *plugging* a provided interface of one component to a required interface of another component, and *parallel* composition of components. For the semantics of these operators, we refer the reader to [9], [13].

C. The rCOS refinement calculus

The development process of rCOS methodology is driven by applying rules of a well-studied refinement calculus, which includes a set of algebraic laws expressed as equations of OO programs under the UTP semantics [12], and a set of refactoring rules of graph transformations for structure refinement [14]. The soundness and relative completeness of these rules have been proven [9]. In addition, the rules proposed in [15] provide an elegant approach to abstract OO design models into component-based models.

The use of *design patterns* [16], [17] makes rCOS refinement more systematic and thus has much higher possibility for automation. Particularly, the *expert pattern* [17] is repeatedly applied to the OO design models, and to decompose the data functionality of a method into interactions of the related objects, called *information experts*, which maintain or know the information for carrying out parts of the method's responsibilities [5]. Other patterns, such as *high cohesion*, *low coupling*, and the refactoring rules, are used to optimize the structure of the design models through introducing new classes, decomposing existing classes, adding or moving attributes of classes.

III. A UML Profile for rCOS

A. UML profiles

UML profiles provide a standard mechanism to customize and extend UML. The main concept of extension is the stereotype. A stereotype defines a new model element based on an existing UML model element, and provides it with additional attributes and their values (tagged values), additional constraints, and optionally a new graphical representation. The semantics defined for a stereotype must not contradict the semantics of the base model element. As we know UML contains too many model concepts and elements. A profile cannot remove existing model elements and their constraints, but it can tailor UML to allow only the required model elements for the application domain to be visible. Actually, a UML profile is a UML *package* dedicated to group the extensions: stereotypes, tagged values, constraints defining well-formedness rules, icons representing the stereotypes, etc.

Currently, there is no standard approach to define a UML profile for a specific domain. Several authors [18], [19] have proposed a set of steps for the creation of a profile. Based on their approaches, we adopt the following procedure to define our profile: 1) define the domain metamodel; 2) design the structure of the profile; 3) define stereotypes; 4) define well-formedness constraints, and 5) provide tool support.

B. Mapping rCOS concepts to UML

As rCOS is a language for object-oriented and component-based systems, by design it shares a lot of features with UML. To define the rCOS profile, we first go through the full set of rCOS concepts, and identify the most suitable UML model elements as their base metaclasses. In fact, many rCOS concepts can be represented by UML model elements without changing their original semantics, at most imposing some additional constraints. For these rCOS concepts, we directly use UML model elements without introducing stereotypes. Fig. 3 illustrates the mapping from rCOS concepts (below the dashed line) to UML metaclasses. The mapping is intuitive and straightforward, and we are going to discuss the corresponding stereotypes later in this section. For many of the UML concepts mentioned here, analogous description may be extracted from the OMG UML specification [2].

C. The rCOS modeling architecture

Fig. 4 depicts the overall structure of an rCOS model. We introduce the stereotype «*RCOSModel*» extending from UML model to represent the root of an rCOS model, and we define the stereotypes «*ClassModel*», «*UsecaseModel*» and «*ComponentModel*», all as extensions of UML package. Thus an rCOS model consists of three sub-models to specify different aspects of a system:

- **Use case model**, contains all actors, use cases and their relationships, along with a use case diagram.

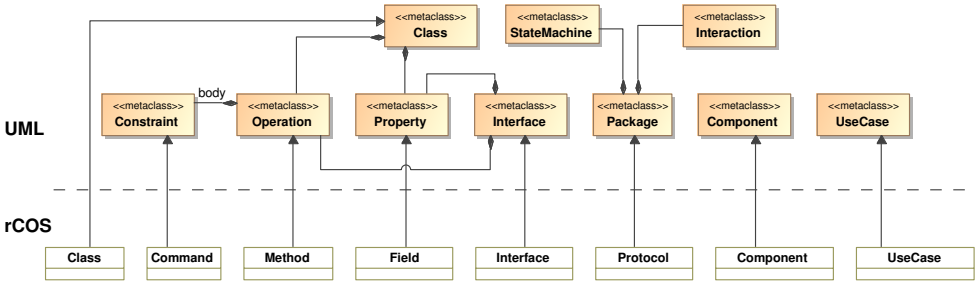

Fig. 3. Mapping from rCOS concepts to UML

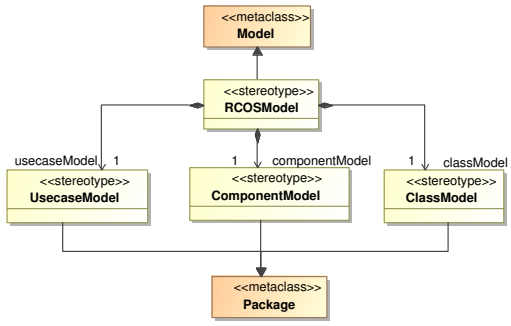

Fig. 4. Overall structure of rCOS model

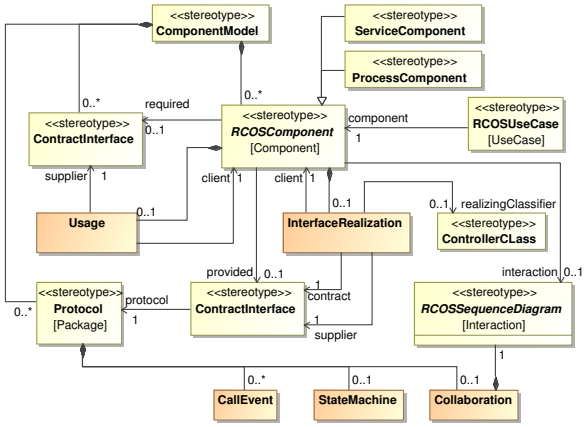

Fig. 5. Modeling rCOS component

Here we define the stereotype «*RCOSUseCase*», and each of the use case is implemented by an rCOS component.

- **Class model**, contains all class definitions and their relationships, represented as a set of class diagrams. Here we define stereotype «*DesignOperation*» as an extension of UML *operation* to represent an rCOS *method*. A textual constraint is introduced with the name *design* to specify the data functionality of the method body.
- **Component model**, contains a collection of components and their interfaces. It also includes component diagrams to describe the relationships between components, and sequence diagrams and state diagrams to specify the behaviors of the components.

D. Component modeling

In rCOS, several components can be composed to a larger component, and a component can be decomposed into a composition of a set of internal components within it. In order to encode rCOS component and composition in a compact way while supporting the semantics of rCOS, we define the rCOS component model, as shown in the UML profile diagram of Fig. 5, which includes:

- *ContractInterface*: Mapped from rCOS *interface*, a contract interface provides an interaction point for a component, and defines the static portion such as fields and methods. A contract interface has a *protocol* that specifies the traces of invocations to the methods of the interface.
- *Protocol*: A protocol is a UML package that contains a *state machine*, a *collaboration* and a set of *call events*. A call event is an invocation to an operation of the contract interface, resulting in the execution of the called operation. As UML behavior state machine is too complex and not suitable here, we adopt the UML *protocol state machine* to represent the reactive behavior of the interface, and each transition of the PSM is trigged by an event in the set of call events. Especially, here a UML *collaboration* owns a UML *interaction* defined as an RCOSSequenceDiagram. However, as both a state machine and a sequence diagram are used to describe the dynamic behaviors of the same entity, we must ensure the consistency of them.
- *RCOSComponent*: We map an rCOS *component* to the abstract stereotype «*RCOSComponent*» as an extension of UML meta-class component, and we define the stereotypes «*ServiceComponent*» and «*ProcessComponent*» to represent the two kinds of component in rCOS. A component itself may also have an rCOS sequence diagram to specify the interactions between its internal sub-components.

In the component model, we realize the connection between a component and its provided interface using a UML *interface realization*. A UML *usage*, a specialized *dependency* relationship, is used to link a required interface to its owner component.

1) Component composition: As shown in Fig. 6, extended from UML *dependency*, we define two stereotypes «*Composition*» and «*Delegation*» to support the following two kinds of component compositions:

- *Horizontal assembly*: A provided interface from one component is plugged to a required interface of another component using a «*Composition*». The requests from the required interface are passed to the provided interface. The two components are composed at the same level of the hierarchical structure.
- *Hierarchical assembly*: If a component has internal sub-components, we use UML *port* as the interaction points to the internal parts. Using the stereotype «*Delegation*», an interface of a sub-component can be associated to a port of the container component. A request to the provided interface of the owning component reaches the port first, and then forwards to a provided interface of a sub-component. Requests originating from a sub-component pass through its required interface to a port of the owning component, and then pass to the required interface of the owning component. The composition process can reach arbitrary depth.

In both cases, the composability between the components must be checked to ensure the *static* (types) and *dynamic* (behavior) correctness of the composition according to the semantics defined in [9].

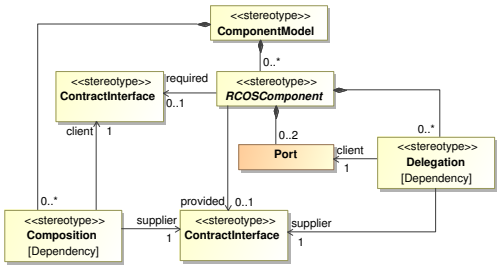

Fig. 6. Realizing component composition

2) Component realization: Realization of a component means the realization of the services provided by the component through its provided interfaces. For a component that does not own any sub-component (called *atomic component*), its provided interface is realized by the *controller class* (use case controller, defined as stereotype «*ControllerClass*» in Fig. 5) of the interface. A controller class is a special class that initializes a system, creates permanent objects, and maintains the main flow of control of the system. On the other side, for a composed component, all requests will be forwarded and realized by the provided interfaces of the internal sub-components. Each method call is processed according to the *run-to-completion* semantics. Only one method call is processed at a time.

E. The rCOS sequence diagram

In order to combine both OO design and component-based design, rCOS has defined two kinds of sequence diagrams, *object sequence diagrams* and *component sequence diagrams*, which are shown in Fig. 7 as two subtypes of abstract stereotype «*RCOSSequenceDiagram*». An object sequence diagram has only one lifeline representing an *actor*, and all other lifelines represent objects or (interfaces of) components, and its messages are *synchronous calls* to an operation provided by the type of the target lifeline, or constructor/create messages. Here the actor could be the use case actor, or other external role that interacts with the system described by the sequence diagram. For a component sequence diagram, all lifelines, except the one representing the actor, represent (interfaces of) components, and each message is a method call to an operation defined in the provided interface of the component represented by the target lifeline. For more discussion about the rCOS sequence diagrams, we refer to [15].

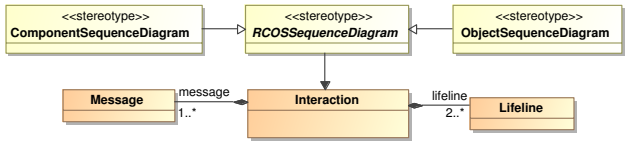

Fig. 7. rCOS sequence diagram

F. Specify profile constraints

An important part of a profile is static semantics that defines the well-formedness constraints. These rules are used to automatically check the well-formedness and the structural consistency of a model. The constraints are typically specified by OCL. This allows validating the models against these constraints automatically.

All these constraints can be divided into two categories: profile conformance constraints, that require a model conforming to the profile, and model consistency constraints, where different viewpoints of a model must be mutually consistent. One simple example of a model consistency constraint is to require that operations called in sequence- or state diagrams must be defined in a structure view, either a class or a component interface. On the other hand, a conformance constraint is evaluated between an rCOS model and the stereotypes of the profile. For example, an rCOS sequence diagram can only have synchronous method calls.

G. Profile implementation

We have developed a CASE tool, called *rCOS modeler* [9], which provides an end-to-end integrated development environment for rCOS, and we implemented the rCOS profile in the tool. The tool is based on *TOPCASED*, "The Open-Source Toolkit for Critical Systems" [20], and developed in Java as a plugin for the Eclipse IDE. We extend and customize UML diagram editors of TOPCASED to limit and/or add menu items in editing palettes and popup menus of the various views. The customized user interface clarifies the usability of the rCOS modeler, enabling the user to drag and drop the correct elements to the appropriate views, and offers a convenient way to guide the user through the full development process. It also contributes to the OCL well-formedness checking by indicating the elements failing the validation and pointing out the possible causes.

IV. Support CB-MDD Development

The rCOS methodology supports the development activities from requirements modeling through object-oriented design to component-based models. This process leads to a sequence of models, evolving in their levels of detail. Within the process, object-oriented analysis and design techniques [17] and component-based techniques can be applied in harmony. Models in later developing phases are constructed from those in earlier phases by correctness preserving model transformations, which implement the rCOS refinement calculus and have been integrated into the rCOS modeler. In this section, we discuss how the rCOS methodology can be supported by the rCOS modeler using an example.

A. Requirements modeling

The rCOS modeling process starts from the creation of an empty rCOS model, and then we capture the business domain concepts with a *conceptual class diagram*. The business processes are described as a set of *use cases*. For each use case, a component is created, along with an empty sequence diagram that has only two lifelines representing the actor and the provided interface of the component. Through adding a message to the sequence diagram, a corresponding method for the interface is automatically produced. Then, we can specify the data functionality of the method as an rCOS design using *RCOSP*. After having completed the sequence diagram, we can generate a *UML protocol state machine* to represent the reactive behavior of the component interface.

We take as example the CoCoME ("Common Component Modelling Example") case study [10] that describes a supermarket trading system. The example originated from Larman's book [17]. In the resulted requirements model, the domain concepts are captured in Fig. 8. One of the use cases, *process sale*, 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. Depicted in the left part of Fig. 9, the use case is modeled by the interface *CInter* of a component *ProcessSale*, and the right part displays the method signatures and fields of the interface. Its scenario is described by a system sequence diagram shown in Fig. 10. In addition, each method has a functionality specification as its body, e.g. the following RCOSP code is for method *enterItem*:

```
public enterItem (Barcode code, int qty) {
  [ qty > 0 ⊢ true] ;
  [ ∃ Item i: store . catalog . contains (i) ∧ i.barcode = code ⊢ true] ;
  [ ⊢ line' = LineItem.new(code, qty)] ;
  [ ⊢ ∃Item j: store.catalog.contains (j) ∧ j.barcode=code
        ∧ line.subtotal '=j . price ∗ qty] ;
  [ ⊢ sale.lines . add(line )] }
```

Here the method body is defined as a sequential composition (using ";") of rCOS *designs*, each in the form of $[p \vdash R]$, meaning that if the program is started at precondition p, the execution will terminate in a state where the postcondition R holds. A variable or field in the post-state is indicated by its primed version, and the omission of p implies the precondition is *true*. As a design is defined in the context of the interface and the class diagram, the navigation of the class diagram and invocation of methods are allowed. For example,

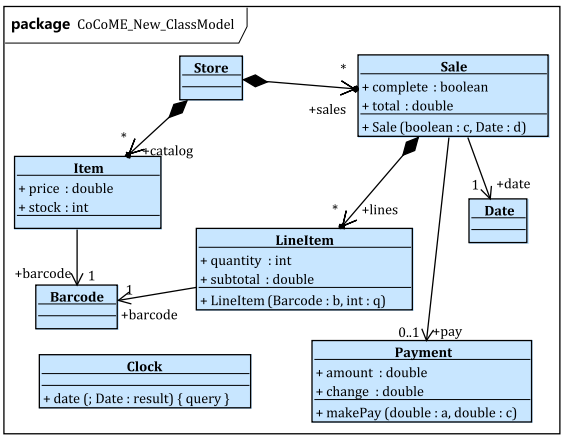

Fig. 8. Domain class diagram

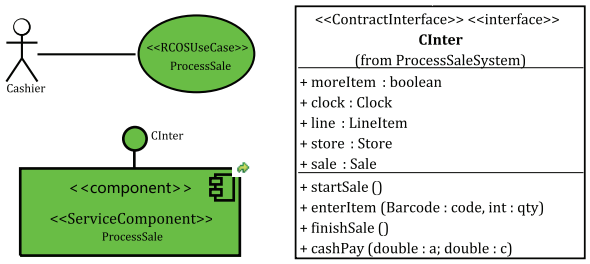

Fig. 9. Use case-, component-, and interface diagrams

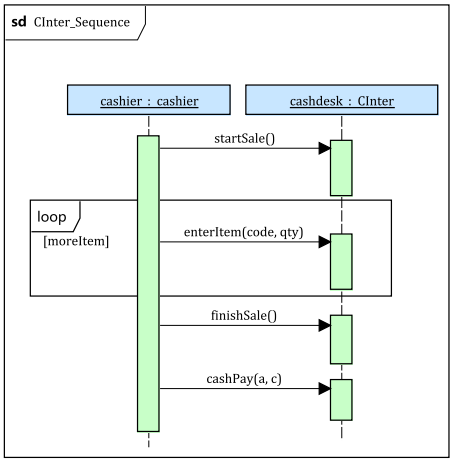

Fig. 10. Sequence diagram for *CInter*

the expression $store.catalog.contains(i)$ means that starting from the field $store$ of type *Store* declared in the interface, we navigate along the association $catalog$ of Fig. 8 to get a collection of *Items*, and check whether *Item* i is contained in the collection.

Through checking the OCL constraints defined in the profile and type checking the method bodies, the static consistency of the requirements model can be ensured. Moreover, we can

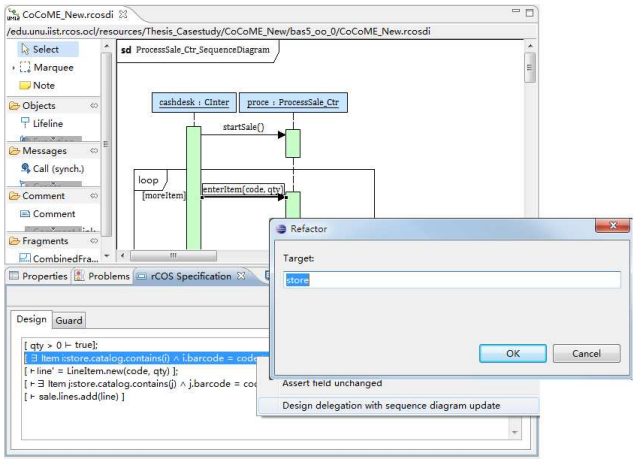

Fig. 11. Interactively applying expert pattern

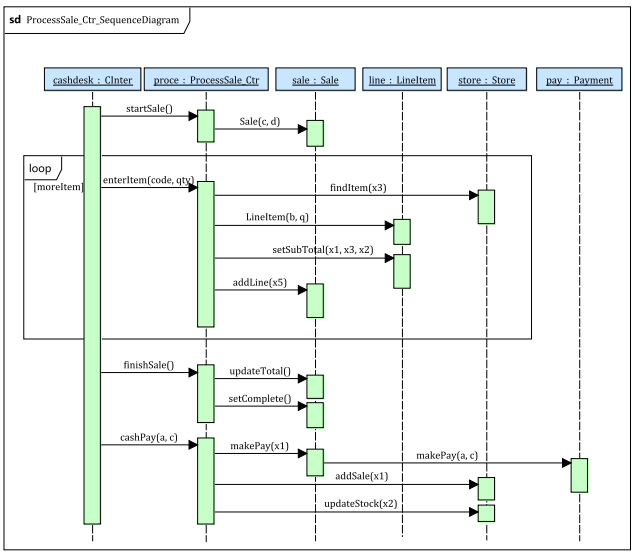

Fig. 12. Refined object sequence diagram

generate CSP specifications for the sequence diagrams, state diagrams and the rCOS designs, and then check the dynamic consistency of the model in the FDR2 model checker [21].

B. Object-oriented design

From a consistent requirements model, we start the OO design in which the rCOS refinement rules can be applied to optimize the model's data structures and functionality specifications. We first generate a controller class for an interface, and we create an initial object sequence diagram by replacing the interface lifeline in Fig. 10 with a lifeline representing the object of the controller class.

Then the expert pattern of responsibility assignments is used to refine the method functionalities of the model. From a message of the object sequence diagram, we open the design of its corresponding method, and select a fragment to delegate to an object of a domain class. For example, as shown in the screenshot of Fig. 11, we select the part "$[\exists Item\ i : store.catalog.contains(i) \wedge i.barcode = code \vdash true]$;" from the specification of message *enterItem*. The tool analyzes the selection, and provides a candidate target object to assign the responsibility, e.g. $store$ in here. After we input the name for the method that will be created to perform the delegated responsibility, e.g. *findItem*, the execution of the transformation replaces the selection part with "$store.findItem(code)$;", and creates a method *findItem* in class *Store* as follows:

```
public findItem(Barcode x3) {
  [ ∃ Item i:catalog . contains (i) ∧ i.barcode = x3 ⊢ true] }
```

Accordingly, the object sequence diagram is updated by adding a new lifeline *store* of type *Store* and a new message *findItem*.

Through repeated applying the expert pattern, the object sequence diagram is finally refined to Fig. 12, and the domain class diagram is therefore refined to a design class diagram by adding methods to the domain classes. Moreover, for class diagrams, we can apply the set of rCOS structural refinement rules [12]. At the end of the OO design, a requirements model becomes an OO design model which includes a set of *object sequence diagrams* and a *design class diagram*.

C. Component-based model design

OO model to component model: From the object sequence diagram of a component's controller class, a number of object lifelines can be selected to convert to a new component using the *object sequence diagram to component sequence diagram transformation* [15]. If the validity checking of the lifeline selection passes, the new component is created as a sub-component of the original component, and the selected object lifelines are collapsed into a lifeline representing the new component. Meanwhile, the component diagram is accordingly updated, and a new sequence diagram and a state machine are generated as the protocol for the new component. By repeating this process, a component is decomposed into a composition of a number of sub-components within the component, and the connection and interaction of these sub-components are illustrated by a component diagram and a component sequence diagram, respectively [15].

For example, by applying the transformation three times on Fig. 12 (selecting lifelines *proce*, *sale* and *line* first, *store* second, and *pay* third), we translate the object sequence diagram to a component sequence diagram shown in Fig. 13, and we accordingly obtain a component diagram depicted in Fig. 15, in where the provided interfaces of new components *COM_store* and *COM_pay* are plugged to the required interface of another new component *COM_proce*, and the last component realizes the functionality of component *ProcessSale*. Fig. 14 shows the protocol generated for component *COM_store*, which consists of a sequence diagram and a protocol state machine.

Component composition: After that, rCOS component composition operators, such as *renaming*, *plugging*, and *parallel composition*, can be continuously applied to optimize the model, or build new components from existing ones. However, even

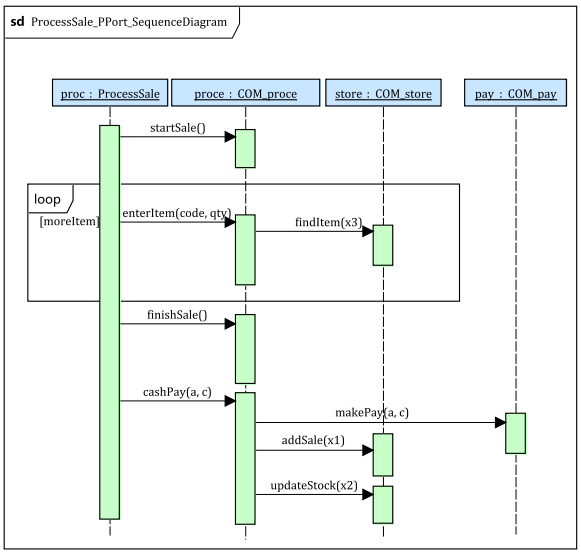

Fig. 13. Component sequence diagram for *ProcessSale*

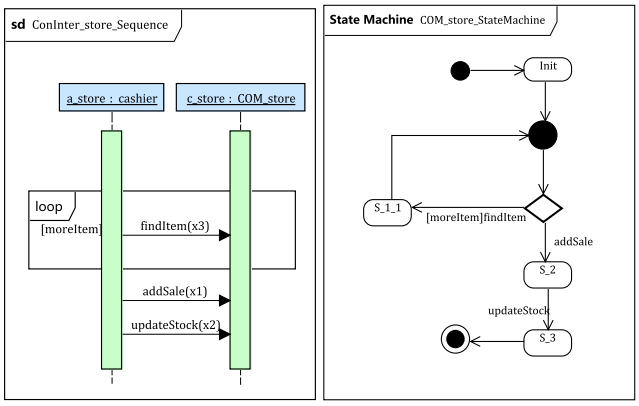

Fig. 14. Provided protocol of component *COM_Store*

the components representing the use cases can be composed to form a single big component to meet the architectural requirement.

For the component diagram of Fig. 15, we consider to combine the components *COM_store* and *COM_pay* into a single component for the purpose of better deployment. First we remove the «Composition» connections from the diagram, and then compose the two components together, that generates a new component *COM_pay_SH*. As the interfaces of *COM_store* and *COM_pay* are disjoint from each other, the new component is resulted from the *disjoint union* of that two components. Finally, we plug the provided interface of *COM_pay_SH* into the required interface of *COM_proce*. The tool will check the composability of these two interfaces:

- *Method signatures*: Both interfaces have at least one identical method signature. It is obvious that all the method signatures of the two interfaces are identical.
- *Behavior protocols*: We generate CSP processes from the PSMs of the two interfaces, and check the generated assertions in the FDR2 tool that the parallel composition of these two processes is deadlock free. Thus, the method invocation traces specified in the protocol of the required interface of *COM_proce* are accepted by the provided protocol of *COM_pay_SH*, and the interactions of the two components will not be blocked.

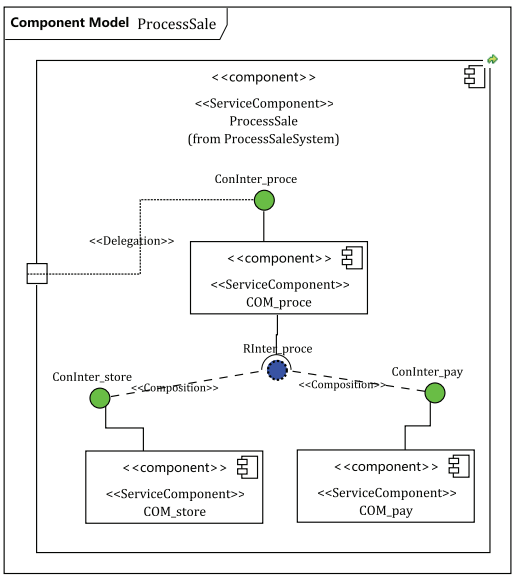

Fig. 15. Component diagram for *ProcessSale*

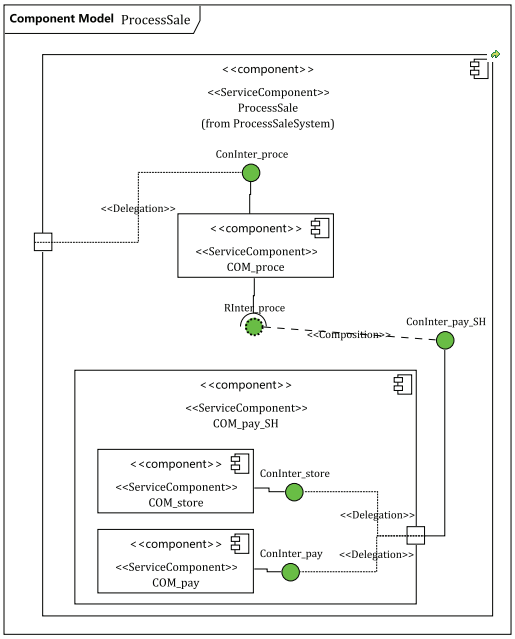

Fig. 16. Final component diagram for *ProcessSale*

Fig. 16 shows the resulted component diagram. The *RCOSP* counterpart of the component diagram is shown for comparison in Listing 1.

Through two stages of interactive, stepwise refinements, we get a component architecture model which refines (im-

plements) the original requirements model. The component model includes component sequence diagrams and component diagrams to define the interactions and relationships of the 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. Based on the component model, we can further generate program code for the components and accordingly deploy them to form a software system.

```
component ProcessSale {
  provided interface CInter
      by COM_proce.ConInter_proce {...}

  composition : COM_proce|COM_pay_SH
    [(COM_proce.RInter_proce
          << COM_pay_SH.ConInter_pay_SH)]

  component COM_proce {
    ...
    provided interface ConInter_proce {...}
    required interface RInter_proce {...}
  }
  component COM_pay_SH {
    provided interface CInter_pay_SH
      by COM_store.ConInter_store,
            COM_pay.ConInter_pay {...}
    composition : COM_proce|COM_pay
            [(COM_store || COM_pay)]

    component COM_store {
      provided interface ConInter_store {...}
    }
    component COM_pay {
      provided interface ConInter_pay {...}
    }
  }
}
```

Listing 1. Final component model in RCOSP

V. Related Work and Conclusion

UML profiles provide a standard extension mechanism for defining modeling languages. A certain number of UML profiles have already been defined, either for generic purposes, or to deal with specific technologies. Among them, the OMG has proposed a set of standard UML profiles, such as SysML [22], which is a UML profile for specifying, analyzing and designing complex systems; and MARTE [23], which is a UML profile supporting specification of real-time and embedded systems. In the field of requirements engineering, a profile [24] is designed to allows the KAOS model to be represented in UML. In [25], a UML profile is proposed to describe distributed and asynchronous software components using UML 2 diagrams for both architectural and behavioral specifications.

Integration of formal methods into MDD has been broadly explored [6]–[8]. Modeling languages must have formally defined semantics, and the multiple views of a complex model need to ensure their consistency. In this perspective, most of the works focus on translating modeling views into a formal specification, thus the formal semantics of the model is defined, and existing analysis tools of the formal language can be used in the proof of desired properties. Among the others, *Z*, *B*, *CSP*, *Petri nets* are the most often chosen target languages. We cite the work of Idani et al. [26] that proposed a metamodel-based transformation of UML models to B. Similarly, there is an MDE based transformation approach for generating *Alloy* specifications from UML class diagrams and backwards [27]. Different from those, our work is mainly concerned with the integrated tool support for the stepwise, incremental development of models by applying the formal refinement calculus of rCOS. In other words, we are more interested in supporting the development process of MDD and CBD with formal methods, not the statically checking of the models at various stages.

In [9], a previous version of rCOS profile was presented as the center part of the rCOS modeler, and the object sequence diagram to component sequence diagram transformation was proposed in [15]. In [10], we have studied the refinement driven development process, carried out by hand, for the CoCoME case study. In the current rCOS profile presented in the paper, we clearly separate the rCOS concepts from UML metamodel concepts, and define rCOS components composition in both horizontally and vertically. We also explicitly define the UML sequence diagrams, protocol state machines and other UML artifacts used in rCOS modeling. In addition, we describe in the paper the concrete steps of how rCOS development process is driven by applying a set of model transformations, and automate the transformations into the rCOS CASE tool.

Conclusion

This paper reports part of our efforts to develop rCOS from a formal theory to a practical tool in the field of component-based model driven software development. We propose a UML profile with necessary stereotypes, tagged values and constraints in order to represent the main concepts of rCOS in UML and support the development methodology of rCOS. We implement the rCOS profile in the rCOS modeler, a CASE tool for rCOS. The tool automates the rCOS refinement rules, such as expert pattern and structural refinements, as correctness preserving model transformations. Thus the full development process, from requirements elicitation through object-oriented design to component-based architectural modeling, can be supported by the tool with a user-friendly interface in an incremental and interactive manner. A model can be analyzed and verified in the tool to ensure its correctness. The rCOS modeler with examples is available from http://rcos.iist.unu.edu.

However, the generation of program code, the last step of the software development process, is not addressed by the paper. Currently, the rCOS tool can only generate monolithic, non-distributed Java programs from the object-originated models. To support component-based, distributed applications, we still need further transforming the component-based models into particular platform, such as the *Model-View-Controller* (MVC) architecture. The MVC design pattern solves the problems arising when applications contain a mixture of domain data (*Model*), GUI presentation (*View*), and business logic (*Controller*). The rCOS model structure provides a basis for implementing an rCOS model as a MVC style application. For example, we can develop rCOS *class model*, *use case model* and *component model* into the *Model*, the *View*, and the *Controller* of MVC, respectively. And finally we could build web applications through generating corresponding SQL,

Java/JML, and JSP codes from different parts of the MVC using template techniques.

Model-driven development is still in its infancy compared to its ambitious goals of having a (semi-)automatic, tool-supported stepwise refinement process from vague requirements specifications to a fully-fledged running program [28]. Integrating formal methods into UML opens up a good way to structure the development activities from object-oriented design and to component-based systems, and we hope the experience and techniques learned from the work of the paper could be helpful in supporting model-driven and component-based software development through formal methods.

ACKNOWLEDGMENT

Supported by grants PEARL, GAVES (Macau Science and Technology Development Fund), Guizhou Intl. Scientific Cooperation Project G[2011]7023, and the National Nature Science Foundation of China (No. 61073022, No. 61103013 and No. 91118007).

REFERENCES

[1] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002.

[2] Object Management Group, "Unified Modeling Language: Superstructure, version 2.4.1," August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4/Superstructure

[3] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, "Model-driven development using UML 2.0: promises and pitfalls," *Computer*, vol. 39, no. 2, pp. 59–66, 2006.

[4] J. Poole, "Model-driven architecture: Vision, standards and emerging technologies," 2001. [Online]. Available: http://www.omg.org/mda/mda_files/Model-Driven_Architecture.pdf

[5] Z. Liu, C. Morisset, and V. Stolz, "rCOS: Theory and Tool for Component-Based Model Driven Development," in *3rd Intl. Symp. on Fundamentals of Software Engineering (FSEN 2009)*, ser. LNCS, vol. 5961. Springer, 2010, pp. 62–80.

[6] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim, "Integrating a formal method into a software engineering process with UML and Java," *Formal Aspects of Computing*, vol. 20, no. 2, pp. 161–204, Mar. 2008.

[7] A. Gargantini, E. Riccobene, and P. Scandurra, "Combining Formal Methods and MDE Techniques for Model-Driven System Design and Analysis," *International Journal On Advances in Software*, vol. 3, no. 1 and 2, pp. 1–18, 2010.

[8] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud, "An Overview of RoZ: A Tool for Integrating UML and Z Specifications," in *Proceedings in Advanced Information Systems Engineering, 12th International Conference (CAiSE 2000)*, ser. LNCS, vol. 1789. Springer, 2000, pp. 417–430.

[9] W. Ke, X. Li, Z. Liu, and V. Stolz, "rCOS: a formal model-driven engineering method for component-based software," *Frontiers of Computer Science in China*, vol. 6, no. 1, pp. 17–39, 2012.

[10] Z. Chen, Z. Liu, A. P. Ravn, V. Stolz, and N. Zhan, "Refinement and verification in component-based model-driven design," *Sci. Comput. Program.*, vol. 74, no. 4, pp. 168–196, 2009.

[11] C. Hoare and J. He., *Unifying theories of programming*. Prentice-Hall International, 1998.

[12] J. He, Z. Liu, and X. Li, "rCOS: A refinement calculus of object systems," *Theor. Comput. Sci.*, vol. 365, no. 1-2, pp. 109–142, 2006. [Online]. Available: http://rcos.iist.unu.edu/publications/TCSpreprint.pdf

[13] X. Chen, J. He, Z. Liu, and N. Zhan, "A Model of Component-Based Programming," in *Proc. Fundamentals of Software Engineering (FSEN 2007)*, ser. LNCS, vol. 4767. Springer, 2007, pp. 191–206.

[14] L. Zhao, X. Liu, Z. Liu, and Z. Qiu, "Graph transformations for object-oriented refinement," *Formal Aspects of Computing*, vol. 21, no. 1–2, pp. 103–131, Feb. 2009.

[15] D. Li, X. Li, Z. Liu, and V. Stolz, "Interactive Transformations from Object-Oriented Models to Component-Based Models," in *Proc. of Formal Aspects of Component Software (FACS'11)*, ser. LNCS, vol. 7253. Springer, 2012, pp. 97–112. [Online]. Available: http://www.iist.unu.edu/www/docs/techreports/reports/report451.pdf

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 3rd ed. Prentice-Hall, 2005.

[18] B. Selic, "A Systematic Approach to Domain-Specific Language Design Using UML," in *Proc. Tenth IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC 2007)*. IEEE, 2007, pp. 2–9.

[19] F. Lagarde, H. Espinoza, F. Terrier, and S. Gérard, "Improving UML Profile Design Practices by Leveraging Conceptual Domain Models," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, 2007, pp. 445–448.

[20] F. Vernadat, C. Percebois, P. Farail, R. Vingerhoeds, A. Rossignol, J. Talpin, and D. Chemouil, "The Topcased project-a toolkit in open-source for critical application and system development," in *International Space System Engineering Conference-Data Systems in Aerospace. Eurospace*, 2006.

[21] Z. Chen, C. Morisset, and V. Stolz, "Specification and Validation of Behavioural Protocols in the rCOS Modeler," in *3rd Intl. Symp. on Fundamentals of Software Engineering (FSEN 2009)*, ser. LNCS, vol. 5961. Springer, 2010, pp. 387–401. [Online]. Available: http://rcos.iist.unu.edu/publications/fsen09.pdf

[22] Object Management Group, "Systems Modeling Language SysML V1.0."

[23] ——, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2."

[24] W. Heaven and A. Finkelstein, "UML profile to support requirements engineering with KAOS," *IEE Proceedings-Software*, vol. 151, no. 1, p. 10, 2004.

[25] S. Ahumada, L. Apvrille, T. Barros, A. Cansado, E. Madelaine, and E. Salageanu, "Specifying Fractal and GCM Components with UML," in *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*. IEEE, 2007, pp. 53–62.

[26] A. Idani, J.-L. Boulanger, and L. Philippe, "A generic process and its tool support towards combining UML and B for safety critical systems," in *Proceedings of the ISCA 20th International Conference on Computer Applications in Industry and Engineering (CAINE 2007)*. ISCA, 2007, pp. 185–192.

[27] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A challenging model transformation," in *MoDELS*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., vol. 4735. Springer, 2007, pp. 436–450.

[28] G. Engels, M. Lohmann, S. Sauer, and R. Heckel, "Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract," in *Proc. Third Intl. Conf. on Graph Transformations*, ser. LNCS, vol. 4178. Springer, 2006, pp. 336–350.