6 Modelling with Relational Calculus of Object and Component Systems - rCOS

Zhenbang Chen[1,5], Abdel Hakim Hannousse[1], Dang Van Hung[1], Istvan Knoll[3], Xiaoshan Li[2], Zhiming Liu[1,⋆], Yang Liu[1], Qu Nan[4], Joseph C. Okika[1,3], Anders P. Ravn[3], Volker Stolz[1], Lu Yang[1], and Naijun Zhan[1,4]

[1] International Institute for Software Technology – United Nations University, Macao
Z.Liu@iist.unu.edu
[2] Faculty of Science and Technology, The University of Macau
[3] Department of Computer Science, Aalborg University, Denmark
[4] Lab. of Computer Science, Institute of Software, CAS, China
[5] National Laboratory for Parallel and Distributed Processing, China

Abstract. This chapter presents a formalization of functional and behavioural requirements, and a refinement of requirements to a design for CoCoME using the *Relational Calculus of Object and Component Systems* (rCOS). We give a model of requirements based on an abstraction of the use cases described in Chapter 3.2. Then the refinement calculus of rCOS is used to derive design models corresponding to the top level designs of Chapter 3.4. We demonstrate how rCOS supports modelling different views and their relationships of the system and the separation of concerns in the development.

Keywords: Requirements Modelling, Design, Refinement, Transformation.

6.1 Introduction

The complexity of modern software applications ranging from enterprise to embedded systems is growing. In the development of a system like the CoCoME system, in addition to the design of the application functionality, design of the interactions among the GUI, the controllers of the hardware devices and the application software components is a demanding task. A most effective means to handle complexity is *separation of concerns*, and assurance of correctness is enhanced by *formal modelling* and *formal analysis*.

Separation of concerns. Separation of concerns is to divide and conquer. At any stage of the development of a system, the system is divided into a number

⋆ Team Leader and corresponding author, UNU-IIST, P.O.Box 3058, Macao. This work is supported by the projects HighQSoftD and HTTS funded by Macao Science and Technology Development Fund; Nordunet3; the 973 Project of China 2005CB321802; the 863 Project of China 2006AA01Z165; and the NSFC Projects 60573081, 60573085, and 90612009.

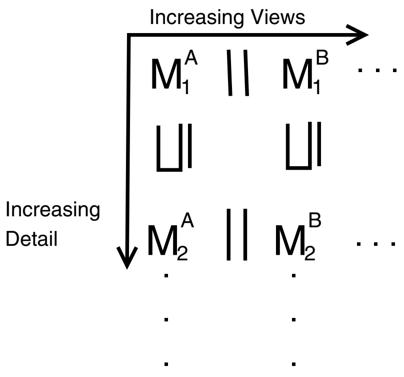
Increasing Views
Increasing Detail
$M_1^A \parallel M_1^B \cdots$
$\sqsubseteq \quad \sqsubseteq$
$M_2^A \parallel M_2^B \cdots$

Fig. 1. rCOS approach with views and refinements

of views or aspects: the static structural view, the interaction view, the dynamic view and their timing aspects. These views can be modeled separately and their integration forms a model of the whole system. Different concerns require different models and different techniques; state-based static specifications of functionality and their refinement is good for specification and design of the functionality, while event-based techniques are the simplest for designing and analyzing interactions among different components including application software components, GUI components and controllers of hardware devices.

Formalization. In recent UML-based development, the static structural view is modeled by packages of *class diagrams* and/or *component diagrams*, dynamic behavior by state diagrams, and *interactions* by sequence diagrams. However, UML has a formal syntax only, and its semantics is not formally definable without imposing strong restrictions.

To assure correctness, we need to incorporate semantic reasoning through specification, refinement, verification and analysis into the development process.

To provide formal support to multi-view and multi-level modelling and analysis in a model-driven development process, a desirable method should

1. Allow to model different views of the system at different levels of abstraction,
2. Provide analysis and verification techniques and tools that assist in showing that the models have the desired properties,
3. Give precise definitions of correctness preserving model transformations, and provide effective rules and tool support for these transformations.

Based on these considerations, we have recently developed a refinement calculus, named rCOS, for design of object and component oriented software systems [9,14,10,5]. It provides a two dimensional refinement framework, that is, *consistent increments* to the models for the multiple parallel ($\parallel$) views in the horizontal dimension, and *refinement*, a relation ($\sqsubseteq$) between models at different levels of abstraction in the vertical dimension.

6.1.1 Goals and Scope of the Component Model

rCOS is an extended theory of Hoare and He's Unifying Theories of Programming (UTP) for object-oriented and component-based programming. The key concepts in the rCOS approach are:

– A component is an aggregation of objects and processes with interfaces. Each interface has a contract, that specifies what is needed for the component to be *used* in building and maintaining a software *without the need to know its design and implementation.* A contract only specifies the functional behavior in terms of pre and post conditions and a protocol defining the permissible traces of method invocations.
– Components can be composed hierarchically by plugging, renaming, hiding and feedback. These operators are defined by a relational semantics for object-oriented programs. Procedural programs are special cases of object-oriented programs in rCOS.
– Composition is supported by required and provided interfaces.
– The refinements are verifiable through rCOS laws. The compatibility of the compositions in the concrete example have been checked for the protocols using the FDR tool, and some of the functionality specifications have been checked using JML.
– Application aspects are dealt with in a use-case driven Requirements Modelling phase. Each use case is specified as a contract of the provided interface of a component, in which the interaction protocol is specified by a set of traces and illustrated by a UML sequence diagram. The functionalities of the provided interface methods are given by the their pre and post conditions, the dynamic behavior is given by guarded design and depicted by a UML state diagram. The data and class structure are specified by class declarations and depicted by a class diagram. The consistency conditions of the requirements include the consistency between the interaction protocol and the dynamic behavior for each use case, and system invariants that relate all use cases.
– The use cases are then designed by using object-oriented refinements, including functionality delegation (via the *expert pattern*), class decomposition and data encapsulation. Such a model of object-oriented design describes how the interface methods (i.e. methods of the use cases) are realized via interaction of internal domain objects of the class diagram. This model is then transformed into a component-based architecture, called a *model of logical design*, by wrapping related objects into components and identifying the interfaces among them from interactions between objects in different components.
– The logical design is refined and refactored to a specific component architecture suitable for a given technical platform, and each component can be further refined to a model which is ready for coding (or code generation).
– The approach does not address deployment but can include code to the extent that the implementation language is given an rCOS semantics.

6.1.2 Modelled Cutout of CoCoME

For the example, we have covered the following:

– The aspects of Requirements Modelling, Logical Design and Component Architecture Design are illustrated. Testing can be done using JML, although translation of an rCOS functionality specification of a class to its JML counterpart is carried out manually. Deployment has not been done.
– The rCOS approach is not currently able to handle dynamic composition.
– The strong parts of the treatment are formal component specifications, refinement and multi-view consistency checks.
– The weak parts are extra-functional properties and missing tool support.
– We did not model the exact architecture of the CoCoME because we focused on the systematic refinements step from requirements to a component architecture very similar to the one of CoCoME implementation.
– The protocol and multi-view consistency were verified for the logical design using FDR and JML respectively.

6.1.3 Benefit of the Modelling

In a model driven development process, the rCOS approach offers:

– A formal specification and analysis of different views including static structure, interactions and functionalities.
– High-level refinement rules for adding details in moving from one level of design to another.
– How different tools for verification and analysis may be incorporated.

6.1.4 Effort and Lessons Learned

Getting acquainted with rCOS requires three months with reasonable background in formal methods in general. In the context of the CoCoME experiment, the current treatment took experienced rCOS people about 2 work months (about 5 people over 1.5 calendar month). We estimate that a complete treatment of the example to the level of a component architecture design would require about 12 work months.

In more detail, one experienced formal rCOS expert spent one day working out the specification of **UC 1**, and the other six use cases took a 4-person week of PhD students, supervised by the experienced person. The OO design of **UC 1** took the rCOS expert less than one hour with writing, but the design of the other use cases took another 4-student week. A lot of effort had to be spent on ensuring model consistency manually. Just from the presented design we have derived around 65 Java classes with over 4000 lines including comments and blank lines. The packages of the *classes* for the resulting component-based model are shown in Appendix B.

Among the important lessons from the exercise are that after developing the functional specifications, the other tasks follow almost mechanically. However, there is still much room in both design and implementation for an efficient solution, taking into account for example, knowledge about the uniqueness of keys in lookups resulting from existential quantification.

6.1.5 Overview

In the next section the fundamental concepts and ideas of the rCOS Component Model are introduced. Section 6.3 shows an rCOS formalization of the requirements for the common example including specifications of classes with local protocols and functional specification. Based on the functional specifications, an object-oriented design is generated. This is refined to components that are fitted into the prescribed architecture of the exercise. In Section 6.4, we discuss Extra-functional properties and related Tool Support. Finally, Section 6.5 concludes and discusses the perspectives and limitations of the solution. In particular we comment on how extra-functional properties can be included through observables, and likely approaches to extensive tool support.

6.2 Component Model

We introduce the basic syntax of rCOS that we will use in the case study, with their informal semantics. We will keep the introduction mostly informal and minimal that is enough for the understanding of the rCOS models of CoCoME. For detailed study, we refer the reader to our work in [9,14,10,5].

The *Relational Calculus of Object and Component Systems* (rCOS) has its roots in the *Unified Theory of Programming* (UTP) [11].

UTP - The background of rCOS. UTP is the completion of many years of effort in the formalization of programming language concepts and reasoning techniques for programs by Tony Hoare and He Jifeng. It combines the reasoning power of ordinary predicate calculus with the structuring power of relational algebra. All programs are seen as binary relations on a state space which is a valuation of a set X of program variables or other observables. An example of an observable is a variable ok', which is true exactly when a program terminates. In all generality, the partial relation for a sequential program P is specified by a pair of predicates over the state variables, denoted by $pre(x) \vdash post(x, x')$ and called a *design*, where x represents the values of the variables before the execution of the program and x' denotes the new values for the variables after the execution, $pre(x)$ is the *precondition* defining the domain of the relation, and $post(x, x')$ is the *postcondition* defining the relation itself.

The meaning of the design $pre(x) \vdash post(x, x')$ is defined by the predicate: $pre(x) \wedge ok \Rightarrow ok' \wedge post(x, x')$, asserting that if the program is activated from a well-defined state (i.e. its preceding program terminated) that satisfies the precondition $pre(x)$ then the execution of the program will *terminate* (i.e. $ok' = true$) in a state such that the new values in this state are related with the old values before the execution by $post$. For example, an assignment $x := x + y$ is defined as $true \vdash x' = x + y$.

In UTP, it is known that designs are closed under all the standard programming constructs like sequential composition, choice, and iteration. For example, $D_1; D_2$ is defined to be the relational composition $\exists x_0 : (D_1(x_0/x') \wedge D_2(x_0/x))$.

For concurrency with communicating processes, additional observables record communication traces; communication readiness can be expressed by *guard* predicates. A major virtue of using a relational calculus is that *refinement* between programs is easily defined as relation inclusion or logical implication.

It is clear that UTP needs adaptation to specific programming paradigms, and rCOS has emerged from the work of He Jifeng and Zhiming Liu to formalize the concepts of object oriented programming: classes, object references, method invocation, subtyping and polymorphism [9]. They have continued to include concepts of component-based and model-driven development: interfaces, protocols, components, connectors, and coordination [10,5]. Thus rCOS is a solid semantic foundation for component-based design. Also, its refinement calculus has been further developed such that it offers a systematic approach to deriving component based software from specifications of requirements.

With a programming language like Java, OO and component-based refinement can effectively ensure correctness. Most refinement rules have corresponding design patterns and thus Java implementations. This takes refinement from programming in the small to programming in the large.

6.2.1 Object Modelling in rCOS

Just as in Java, an OO program in rCOS has *class declarations* and a *main program* [9]. A class can be public or private and declares its attributes and methods, they can be public, private or protected. The main program is given as a *main class*. Its attributes are the global variables of program and it has a main method *main*(), that implements the application processes. Unlike Java, the definition of a method allows specification statements which use the notion of design $pre \vdash post$. Notice that the use of the variables ok and ok' implies that rCOS is a total correctness model, that is if the precondition holds the execution terminates correctly.

Types and Notations. Another difference of rCOS from an OO programming language is that we distinguish data from objects and thus a datum, such as an integer or a boolean value does not have a reference. For the CoCoME case study, we assume the following data types:

$$V ::= long \mid double \mid char \mid string \mid bool$$

Assuming an infinite set CN of symbols, called class names, we define the following type system, where C ranges over CN

$$T ::= V \mid C \mid array[1..n](T) \mid set(T) \mid bag(T)$$

where $array[1:n](T)$ the type of arrays of type T, and $set(T)$ is the type of sets of type T. We assume the operations $add(T\,a)$, $contains(T\,a)$, $delete(T\,a)$ and $sum()$ on a set and a bag with their standard semantics. For a variable s of type $set(T)$, the specification statement $s.add(a)$ equals $s' = s \cup \{a\}$, $s.sum()$ is the sum of all elements of s, which is assumed to a set of numbers. We use curly brackets $\{e_1, \ldots, e_n\}$ and the square brackets $[[e_1, \ldots, e_m]]$ to define a set and a bag. For set s such that each element has an identifier, $s.find(id)$ denotes the function that returns the element whose identifier equals id if there is one, it returns *null* otherwise. Java provides the implementations of these types via the *Collection* interface. Thus these operations in specifications can be directly coded in Java.

In specifications, $C\ o$ means that object o has type C, and $o \neq null$ means that o is in the object heap if the type of o is a class, and that o is defined if its type is a data type. The shorthand $o \in C$ denotes that $o \neq null$ and its type is C.

In rCOS, evaluation of expressions does not change the state of the system, and thus the Java expression `new` C`()` is not a rCOS expression. Instead, we take $C.New(C\ x)$ as a command that creates an object of C and assigns it to variable x. The attributes of this object are assigned with the initial values or objects declared in C. If no initial value is declared it will be *null*. However, in the specification of CoCoME, we use $x' = C.New()$ to denote $C.New(x)$, and $x' = C.New[v_1/a_1, \ldots, a_k/v_k]$ to denote the predicate $C.New[v_1/a_1, \ldots, a_k/v_k](x)$ that a new object of class C is created with the attributes a_1 initialized with v_i for $i = 1, \ldots, k$, and this objects is assigned to variable x.

A *design* $pre \vdash post$ for a method in rCOS is here written separately as **Pre** *pre* and **Post** *post*. For the sake of systematic refinement, we write the specification of static functionality of a use case handler class in the following format:

```
class        C [extends D] {
attributes                    T x = d, ..., T_k x = d
methods                       m(T in; V return) {
                               pre:     c ∨ ... ∨ c
                               post:  ∧ (R; ...; R) ∨ ... ∨ (R; ...; R)
                                      ∧ ......
                                      ∧ (R; ...; R) ∨ ... ∨ (R; ...; R) }
                              ......
                              m(T in; V return) {...... }
invariant                     Inv
                              }
```

where

- The list of class declarations can be represented as a UML class diagram.
- The initial value of an attribute is optional.
- Each c in the precondition, is a conjunction of primitive predicates.
- Each relation R in the postcondition is of the form $c \wedge (le' = e)$, where c is a boolean condition and le an *assignable expression* and e is an expression. An assignable le is either a primitive variable x, or an attribute name, a, or $le.a$ for an attribute name a. We use **if** c **then** $le' = e_1$ **else** $le' = e_2$ for $c \wedge (le' = e_1) \vee \neg c \wedge (le' = e_2)$ and **if** c **then** $le' = e$ for $c \wedge (le' = e) \vee \neg c \wedge$ **skip**. Notice here that the expression e does not have to be an executable expression. Instead, e is a logically specified expression, such as the greatest common divisor of two given integers.

We allow the use of *indexed conjunction* $\forall i \in I : R(i)$ and *indexed disjunctions* $\exists i \in I : R(i)$ for a finite set I. These would be the quantifications if the index set is infinite.

The above format has been influenced by TLA$^+$ [12], UNITY [4] and Java. We also need a notation for traces; in this setting, they are given by UML sequence diagrams and the UML state diagrams.

6.2.2 Refinement

In rCOS, we provide three levels of refinement:

1. Refinement of a whole object program. This may involve the change of anything as long as the visible behavior of the main method is preserved. It is an extension to the notion of data refinement in imperative programming, with a semantic model dealing with object references. In such a refinement, all non-public attributes of objects are treated as local (or internal) variables [9].
2. Refinement of the class declaration section $\mathit{Classes}_1$ is a refinement of $\mathit{Classes}$ if $\mathit{Classes}_1 \bullet \mathit{main}$ refines $\mathit{Classes} \bullet \mathit{main}$ for all main. This means that $\mathit{Classes}_1$ supports at least as many functional services as $\mathit{Classes}$.
3. Refinement of a method of a class. This extends the theory of refinement in imperative programming, with a semantic model dealing with object references. Obviously, Class_1 refines Class if the public class names in $\mathit{Classes}$ are all in $\mathit{Classes}_1$ and for each public method of each public class in $\mathit{Classes}$ there is a refined method in the corresponding class of $\mathit{Classes}_1$.

An rCOS design has mainly three kinds of refinement: *Delegation of functionality* (or *responsibility*), *attribute encapsulation*, and *class decomposition*. Interesting results on completeness of the refinement calculus are available in [14].

Delegation of functionality. Assume that C and C_1 are classes in $\mathit{Classes}$, $C_1\ o$ is an attribute of C and $T\ x$ is an attribute of C_1. Let $m()\{c(o.x', o.x)\}$ be a method of C that directly accesses and/or modifies attribute x of C_1. Then, if all other variables in command c are accessible in C_1, we have that $\mathit{Classes}$ is refined by $\mathit{Classes}_1$, where $\mathit{Classes}_1$ is obtained from $\mathit{Classes}$ by changing $m()\{c(o.x', o.x)\}$ to $m()\{o.n()\}$ in class C and adding a fresh method $n()\{c[x'/o.x', x/o.x]\}$. This is also called the *expert pattern of responsibility assignment*.

Encapsulation. When we write the specifications of the methods of a class C before designing the interactions between objects, we often need to directly refer to attributes of the classes that are associated with C. Therefore, those attributes are required to be public. After designing the interactions by application of the expert pattern for functionality assignments, the attributes that were directly referred are now only referred locally in their classes. These attributes can then be encapsulated by changing them to protected or private.

The *encapsulation rule* says that if an attribute of a class C is only referred directly in the specification (or code) of methods in C, this attribute can be made a *private attribute*; and it can be made *protected* if it is only directly referred in specifications of methods of C and its subclasses.

Class decomposition. During an OO design, we often need to decompose a class into a number of classes. For example, consider classes $C_1 ::\ D\ a_1$, $C_2 ::\ D\ a_2$, and $D ::\ T_1\ x, T_2\ y$. If methods of C_1 only call a method $D :: m()\{...\}$ that only involves x, and methods of C_2 only call a method $D ::\ n()\{...\}$ that only involves y, we can decompose D into $D_1 ::\ T_1\ x; m()\{...\}$ and $D_2 ::\ T_2\ y; n()\{...\}$, and change the type of a_1 in C_1 to D_1 and the type of a_2 in C_2 to D_2. There are other rules for class decomposition [9,14].

With these and other refinement rules in rCOS, we can prove a big-step refinement rule, such as the following **expert pattern**, that will be repeatedly used in the design of CoCoME.

Theorem 1 (Expert Pattern)
Given a class declarations section Classes and its navigation paths $r_1.\ldots.r_f.x$, *(denoted by* le *as an assignable expression),* $\{a_{11}.\ldots.a_{1k_1}.x_1, \ldots, a_{\ell 1}.\ldots.a_{\ell k_\ell}.x_\ell\}$, *and* $\{b_{11}.\ldots.b_{1j_1}.y_1, \ldots, b_{t1}.\ldots.a_{tj_t}.y_t\}$ *starting from class C, let* $m()$ *be a method of C specified as*

$$\begin{aligned} C :: m()\{ \quad & c(a_{11}.\ldots.a_{1k_1}.x_1, \ldots, a_{\ell 1}.\ldots.a_{\ell k_\ell}.x_\ell) \\ & \wedge\ le' = e(b_{11}.\ldots.b_{1s_1}.y_1, \ldots, b_{ts1}.\ldots.b_{ts_t}.y_t)\ \} \end{aligned}$$

then Classes can be refined by redefining $m()$ *in C and defining the following fresh methods in the corresponding classes:*

$$\begin{array}{ll} C :: & \mathit{check}()\{\mathit{return}'{=}c(a_{11}.\mathit{get}_{\pi_{a_{11}} x_1}(), \ldots, a_{\ell 1}.\mathit{get}_{\pi_{a_{\ell 1}} x_\ell}())\} \\ & m()\{\mathbf{if}\ \mathit{check}()\ \mathbf{then}\ r_1.\mathit{do}\text{-}m_{\pi_{r_1}}(b_{11}.\mathit{get}_{\pi_{b_{11}} y_1}(), \\ & \qquad\qquad \ldots, b_{s1}.\mathit{get}_{\pi_{b_{s1}} y_s}())\} \\ T(a_{ij}) :: & \mathit{get}_{\pi_{a_{ij}} x_i}()\{\mathit{return}'{=}a_{ij+1}.\mathit{get}_{\pi_{a_{ij+1}} x_i}()\}\ (i : 1..\ell, j : 1..k_i - 1) \\ T(a_{ik_i}) :: & \mathit{get}_{\pi_{a_{ik_i}} x_i}()\{\mathit{return}'{=}x_i\}\ (i : 1..\ell) \\ T(r_i) :: & \mathit{do}\text{-}m_{\pi_{r_i}}(d_{11}, \ldots, d_{s1})\{r_{i+1}.\mathit{do}\text{-}m_{\pi_{r_{i+1}}}(d_{11}, \ldots, d_{s1})\} \\ & \mathit{for}\ i : 1..f - 1 \\ T(r_f) :: & \mathit{do}\text{-}m_{\pi_{r_f}}(d_{11}, \ldots, d_{s1})\{x' = e(d_{11}, \ldots, d_{s1})\} \\ T(b_{ij}) :: & \mathit{get}_{\pi_{b_{ij}} y_i}()\{\mathit{return}'{=}b_{ij+1}.\mathit{get}_{\pi_{b_{ij+1}} y_i}()\}\ (i : 1..t, j : 1..s_i - 1) \\ T(b_{is_i}) :: & \mathit{get}_{\pi_{b_{is_i}} y_i}()\{\mathit{return}'{=}y_i\}\ (i : 1..t) \end{array}$$

where $T(a)$ *is the type name of attribute* a *and* π_{v_i} *denotes the remainder of the corresponding navigation path* v *starting at position* j.

If the paths $\{a_{11}.\ldots.a_{1k_1}.x_1, \ldots, a_{\ell 1}.\ldots.a_{\ell k_\ell}.x_\ell\}$ *have a common prefix up to* a_{1j}, *then class C can directly delegate the responsibility of getting the* x*-attributes and checking the condition to* $T(a_{ij})$ *via the path* $a_{11}.\ldots, a_{ij}$ *and then follow the above rule from* $T(a_{ij})$. *The same rule can be applied to the b-navigation paths.*

The expert pattern is the most often used refinement rule in OO design. One feature of this rule is that it does not introduce more couplings by associations between classes into the class structure. It also ensures that functional responsibilities are allocated to the appropriate objects that *know* the data needed for the responsibilities assigned to them.

An important point to make here is that the expert pattern and the rule of encapsulation can be implemented by automated model transformations. In general, transformations for structure refinement can be aided by transformations in which changes are made on the structure model, such as the class diagram, with a diagram editing tool and then automatic transformations can be derived for the change in the specification of the functionality and object interactions [14].

6.2.3 Component Modelling in rCOS

There are two kinds of components in rCOS, *service components* (simply called *components*) and *process components* (also simply called *processes*).

Like a service component, a *process component* has an interface declaring its own local state variables and methods, and its behavior is specified by a process contract. Unlike a service component that is passively waiting for a client to call its provided services, a process is active and has its own control on when to call out to required services or to wait for a call to its provided services. For such an active process, we cannot have separate contracts for its provided interface and required interface, because we cannot have separate specifications of outgoing calls and incoming calls. So a process only has an interface and its associated contract (or code).

Compositions for *disjoint union* of components and *plugging* components together, for *gluing components* by processes are defined in rCOS, and their closure properties and the algebraic properties of these compositions are studied [5]. Note that an interface can be the union of a number of interfaces. Therefore, in a specification we can write the interfaces separately.

The contracts in rCOS also define the unified semantic model of implementations of interfaces in different programming languages, and thus clearly support interoperability of components and analysis of the correctness of a component with respect to its interface contract. The theory of refinements of contracts and components in rCOS characterizes component substitutivity, as well as it supports independent development of components.

6.2.4 Related Work

The work on rCOS takes place within a large body of work [15] on modelling and analysis techniques for object-oriented and component based software. Some of these works we would like to acknowledge.

Eiffel [17] first introduced the idea of design by contract into object-oriented programming. The notion of designs for methods in the object-oriented rCOS is similar to the use of assertions in Eiffel, and thus also supports similar techniques for static analysis and testing. JML [13] has recently become a popular language for modelling and analysis of object-oriented designs. It shares similar ideas of using assertions and refinement as behavioral subtype in Eiffel. The strong point of JML is that it is well integrated with Java and comes with parsers and tools for UML like modelling.

In Fractal [20], behavior protocols are used to specify interaction behavior of a component. rCOS also uses traces of method invocations and returns to model the interaction protocol of a component with its environment. However, the protocol does not have to be a regular language, although that suffices for the examples in this chapter. Also, for components rCOS separates the protocol of the provided interface methods from that of the required interface methods. This allows better pluggability among components. On the other hand, the behavior protocols of components in Fractal are the same for the protocols of coordinators and glue units that are modeled as processes in rCOS. In addition to interaction protocols, rCOS also supports state-based modelling with guards and pre-post conditions. This allows us to carry out stepwise functionality refinement.

We share many ideas with work done in Oldenburg by the group of Olderog on linking CSP-OZ with UML [18] in that a multi-notational modelling language is used for encompassing different views of a system. However, rCOS has taken UTP as its single point of departure and thus avoids some of the complexities of merging existing notations. Yet, their framework has the virtue of well-developed underlying frameworks and tools.

6.3 The Example

The requirements capture starts with identifying *business processes* described as *use cases*. The use case specification includes four views. One view is the *interactions* between the external environment, modeled as *actors*, and the system. The interaction is described as a protocol in which an actor is allowed to invoke *methods* (also called *use case operations*) provided by the system. In rCOS, we specify such a protocol as a set of *traces* of method invocations, and depict it by a UML *sequence diagram* (cf. Fig. 2), called a *use case sequence diagram*.

In the real use of the system the actors interact with the system via the GUI and hardware devices. However, in the early stage of the design, we abstract from the GUI and the concrete input and output technologies and focus on specifying what the system should produce for output after the input data or signals are received. The design of the GUI and the controllers of the input and output devices is a concern when the application interfaces are clear after the design of the application software. Also, a use case sequence diagram does not show the interactions among the domain objects of the system, as the interactions among those internal objects can only be designed after the specification of what the use case operations do when being invoked. There are many different ways in which the internal objects can interact to realize the same use case.

The interaction trace of a use case is a constraint on the flow of control of the main application program or processes. The flow of control can be modeled by a *guarded state transition system*, that can be depicted by a UML state diagram (cf. Fig. 4). While a sequence diagram focuses on the interactions between the actors and the system, the state diagram is an operational model of the *dynamic behavior* of the use case. They must be trace equivalent. This model may be used for verification of deadlock and livelock freedom by model checking state reachability.

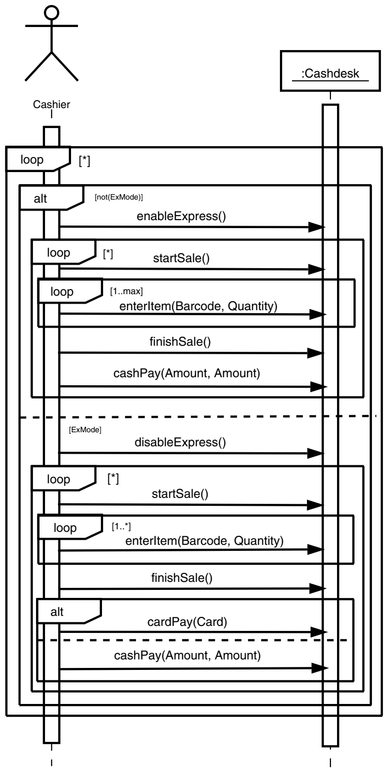

Fig. 2. Use case sequence diagram for **UC 1 & 2**

Another important view is the static *functionality view* of the system. The requirements should precisely specify what each use case operation should do when invoked. That is, what state change it should make in terms of what new objects are to be created, what old objects should be destroyed, what links between which objects are established, and what data attributes of which objects are modified, and the precondition for carrying out these changes. For the purpose of *compositional* and *incremental* specification, we introduce a designated *use case controller class* for each use case, and we specify each method of the use case as a method of this controller class. A method is specified by its signature and its design in the form $pre \vdash post$. The signatures of the methods must be consistent with those used in the interaction and dynamic views. During specification of the static functionality of the use case operations, all types and classes (together with their attributes) required in the specification must be defined.

The type and class definitions in the specification of functionality of the methods of the use case controls form the *structure view* of the system. It can be depicted by a class diagram or packages of class diagrams (cf. Fig. 3). The consistency and integrated semantics of the different views are studied in [6].

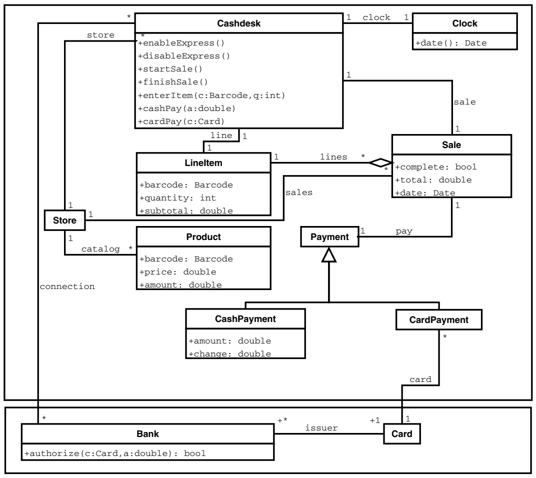

Fig. 3. Use case class diagram for **UC 1 & 2**

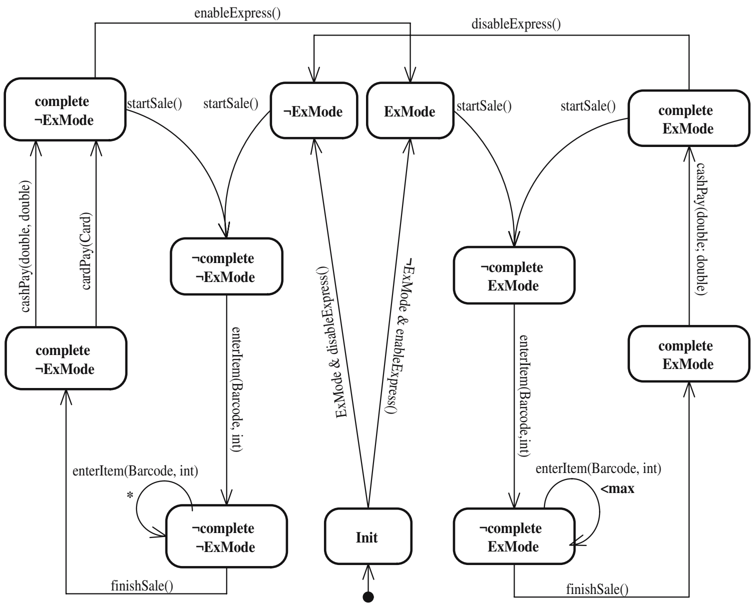

Fig. 4. State diagram for **UC 1 and 2**

UC 1 & UC 2: Process Sale. As both the first and the second use case relate to a single sale, we handle them in a single section. It would be possible to keep the mode change in a separate use case, but the combination saves space.

We first model the interaction protocol that the system offers the actor, i.e. Cashier. This is given in a *use case sequence diagram* in Fig. 2. As a simplification, we assume that the Cashier controls switching between normal and express mode; in the end it makes no difference who does it. In this sequence diagram, *max* denotes the maximum number of items that the current sale can process.

The protocol that the sequence diagram defines is specified by the set of traces represented by the following regular expression:

$$\begin{aligned} &tr(SD_{uc1}) = \\ &(\quad (enableExpress\ (startSale\ enterItem^{(max)}\ finishSale\ cashPay)^*\ ) \\ &\ \ +\ (disableExpress\ (startSale\ enterItem^*\ finishSale\ (cashPay\ +\ cardPay))^*\ )\ )^* \end{aligned}$$

These traces are accepted by the state diagram given in Fig. 4 (note that labels of states only serve as documentation and are not UML compliant). We assume that the *ExMode* guard is initialized non-deterministically in the *Init* state.

Functionality Specification. We now start to analyze the functionality of each of the operations in the use case. An informal understanding of the functionality is to identify the classes, their properties, and to construct an initial class diagram, see Fig. 3. For the specification of the operations we assume:

1. There exists a *Store* object, *store : Store*.
2. The object *store* owns a set of *Product* objects with their barcode, amount, and price, denoted by *store.catalog*. It accesses the attribute *catalog : set(Product)* (we omit properties not relevant to modelling, like product descriptions).
3. The *Cashdesk* object accesses the store via an association *store*.
4. There exists a *Clock* object associated with the *desk* via the association *clock*.
5. There is a *Bank* class with a method *authorize(Card c,double a;bool returns)*, which checks a credit card transaction with amount *a* and returns whether it is valid.

We now specify the functionality of the methods in *Cashdesk*.

Use Case	**UC 1: Process Sale**
Class	*Cashdesk*
Method	*enableExpress()*
	pre: *true*
	post: *ExMode' = true*
Method	*disableExpress()*
	pre: *true*
	post: *ExMode' = false*
Method	*startSale()*
	pre: *true*
	post: /* a new *sale* is created, and its *line items* initialized to empty, and the date correctly recorded */
	$sale' = Sale.New(false/complete,\ empty/lines,\ clock.date()/date)$
Method	*enterItem(Barcode c, int q)*
	pre: /* there exists a product with the input barcode *c* */
	$store.catalog.\mathrm{find}(c) \neq null$
	post: /* a new *line* is created with its *barcode c* and *quantity q* */
	$line' = LineItem.New(c/barcode, q/quantity)$
	; $line.subtotal' = store.catalog.\mathrm{find}(c).price \times q$
	; $sale.lines.\mathrm{add}(line)$

Method *finishSale()*
pre: *true*
post: $sale.complete' = true$
$\wedge\ sale.total' = sum[[l.subtotal \mid l \in sale.lines]]$

Method *cashPay(double a; double c)*
pre: $a \geq sale.total$
post: *sale.pay′=CashPayment.New(a/amount, a-sale.total/change)*
/* the completed *sale* is logged in *store*, and */
; *store.sales*.add(*sale*); /* the inventory is updated */
$\forall l \in sale.lines,\ p \in store.catalog \bullet$ (**if** $p.barcode = l.barcode$ **then**
$p.amount' = p.amount - l.quantity$)
; *store.sales*.add(*sale*);
$\forall l \in sale.lines,\ p \in store.catalog \bullet$ (**if** $p.barcode = l.barcode$ **then**
$p.amount' = p.amount - l.quantity$)

Invariants. A *class invariant* is established on initialization of an instance, i.e. through the constructor. It must hold after each subsequent method call to that class. We specify correct initialization of the cash desk as a *class invariant*:

Class Invariant *Cashdesk* : $store \neq null \wedge store.catalog \neq null$
$\wedge\ clock \neq null \wedge bank \neq null$

Static and Dynamic Consistency. When we have the constituents of the specification document: *class diagram*, *state diagram*, *sequence diagram* and the *functional specification*, we need to make sure that they are consistent [6]. The *static consistency* of the requirement model is ensured by checking that

1. All types used in the specification are given in the class diagram,
2. All data attributes of any class used in the specification are correctly given in the class diagram,
3. All properties are correctly given as attributes or associations in the class diagram, and the multiplicities are determined according to whether the type of the property is *set*(C) or *bag*(C) for some class C,
4. Each method given in the functional specification is used in the other diagrams according to its signature, that is, the arguments (and return values) and their types match,
5. Expressions occurring as guards are (type) consistent with their functional specifications, i.e., of right type, initialised before first use, etc.

Dynamic Consistency. means that the dynamic flow of control and the interaction protocol are consistent:

1. If the actors follow the interaction protocol when interacting with the use case controller, the state diagram should ensure that the interaction is not blocked by guards. Formally speaking, the traces of method calls defined by the sequence diagram should be accepted by the state machine.
2. On the other hand, the traces that are accepted by the state diagram should be allowable interactions in the protocol defined by the sequence diagram.

The above two conditions are formalised and checked as trace equivalence between the sequence diagram and the state diagram in FDR [23,21]. We point the interested reader to [22] and [19] for more detailed applications of CSP to different flavours of state diagrams. However, we note that the reasons for having a sequence diagram and a state diagram are different:

– The denotational trace semantics for the sequence diagram is easy to use as the specification of the protocol in terms of temporal order of the events,
– The state diagram has an operational semantics which is easier to use for verification of both safety and liveness properties.

The event-based sequence diagrams and state diagrams abstract the data functionality away and thus make checking practically feasible—i.e., naive model-checking of an OO program would require considering all possible values for attributes and arguments of methods.

6.3.1 Detailed Design

At this point, we illustrate the refinement rules of rCOS (see Sec. 6.2.2) to the operations that were specified for the use cases in the previous section. We take each operation of each use case and decompose it, assigning functionalities to use cases according to attributes of classes. This happens mainly through application of the refinement rule for functional decomposition, called the *expert pattern*.

Navigation in the functional specification will be translated to setters and getters for attributes, and direct access for associations.

Occasionally, refinement will not directly introduce a concrete implementation, but may also lead to refinement on the functionality specification level. For an example, observe how the handling of sets in the following examples evolves first through further refinement before being eventually modeled in code.

Refinement of UC 1 & 2

We successively handle the previously specified operations. The refinement of the mode handling to code is trivial. We remind the reader that according to the problem description changing the physical light will be handled by a separate component.

```
class Cashdesk:: enableExpress() { exmode := true }
                 disableExpress() { exmode := false }
```

The *startSale()* operation is refined by making the *Cashdesk* instance invoke the constructor of the *Sale* class. As the *Clock* is an entity located in the *Cashdesk*, we have to pass the current *Date* as an argument. This follows the *expert pattern*:

```
class Cashdesk:: startSale() { sale:=Sale.New(clock.date()) }
class Sale::     Sale(Date d)
                  { date := d; complete := false; total := 0; lines := empty }
```

In Java, sets are implemented as a class that *implements* the *interface Collection*. The constructor of the set class initializes the instance as an empty set. The formal treatment of set operations like *find()*, *add()*, and constructors in general is given in the existing rCOS literature. Thus, the constructor *Sale()* can be further refined to the following code:

```
class Sale:: Sale(Date d)
      { date := d; complete := false; total := 0; lines := set(LineItem).New() }
```

However, when design of a significant algorithm is required, such as calculating the greatest common divisor of two integer attributes or finding the shortest path in a directed graph object, the specification of the algorithm instead of code can be first designed in the refinement. For operation *enterItem()*, the precondition is checked by finding the product in the catalog that matches the input bar code. From the refinement rule for the *expert pattern*, the *navigation path* *store.catalog*.find() indicates the need for a method *find()* in the use case handler, that calls a method *find()* which in turn calls the method *find()* of the set *catalog*. Thus, we need to design the following methods in the relevant classes:

```
class Cashdesk::     find(Barcode code; Product returns) { store.find(code; returns) }
class Store::        find(Barcode code; Product returns) {catalog.find(code; returns)}
Class set(Product):: Method find(Barcode code; Product returns)
                     Pre  $\exists p : Product \bullet (p.barcode = code \wedge contains(p))$
                     Post $returns.barcode' = code$
```

Applying the expert pattern to the navigation paths *line.subtotal*, *store.catalog*.find() and *sale.lines*.add(), we can refine the specification of *enterItem()* to:

```
class Cashdesk:: enterItem(Barcode code, int qty) {
                   if find(code) ≠ null then {
                     line:=LineItem.New(code, qty);
                     line.setSubtotal(find(code).price × qty);
                     sale.addLine(line)
                   } else { throw exception e() }   }
class Sale::     addLine(LineItem l) { lines.add(l) }
class LineItem:: setSubtotal(double a) { subtotal:=a }
```

Note that we use exception handling to signal that the precondition is violated. This allows us to introduce more graceful error handling later through refinement. This is different to translating the condition into an `assert` statement which would terminate the application, as that would preclude refinement.

We now refine method *finishSale()* using the expert pattern and define a method *setComplete()* and a method *setTotal()* in class *Sale*. These methods then will be called by the use case handler class.

```
class Cashdesk:: finishSale()   { sale.setComplete(); sale.setTotal(); }
class Sale::     setComplete() { complete:=true }
                 setTotal()     { total:=lines.sum() }
```

For *cashPay()*, we need the total of the sale to check the precondition, accordingly we define *getTotal()* in class *Sale*. To create a payment, we define a method *makeCashPay()* called by the cash desk, and creates an object of type *CashPayment*. For logging the sale, we define a method *addSale()* in class *Store* that is called by the cash desk, that will use the method *add()* of the set of sales.

For updating the inventory, the universal quantification will be implemented by a loop, so we defer the implementation to a helper method:

```
class Cashdesk:: cashPay(double amount; double return) {
                 if (amount ≥ sale.getTotal()) then  {
                    sale.makeCashPay(amount; return);
                    store.addSale(sale);
                    updateInventory() /* defined separately */
                 } else { throw exception e(amount ≥ sale.getTotal())}   }
class Sale::     getTotal(; double returns) { returns := total }
                 makeCashPay(double amount; double returns)
                   { payment:=CashPay.New(amount); returns:=getChange() }
                 getChange(; double returns) { returns := amount - total }
class Store::    addSale(Sale s) { sales.add(s) }
```

Recall the functional specification corresponding to *updateInventory()*:

Class *Cashdesk*::
$\forall l \in sale.lines,\ p \in store.catalog \bullet (\ \mathbf{if}\ p.barcode = l.barcode\ \mathbf{then}$
$p.amount' = p.amount - l.quantity)$

It involves universal quantification over elements of a set. Such a specification is usually covered by some *design pattern*. The solutions always require loop statements, which, in an object-oriented setting, are for example covered by (Java) *iterators*, or they might be implemented in a database.

A design pattern is to first define a method for changing the variables, i.e., to update the amount of the product in the catalog. This implies a method *update(int qty)* in class *Product*, and then a method *update(Barcode code, int qty)* in *catalog* whose type is *set(Product)* and which implements the loop for the quantification on p (we consider the iteration over *sale.lines* in the next step):

```
class Product::      update(int qty) { amount := amount-qty }
class set(Product):: update(Barcode code, int qty) {
                       Iterator i := iterator();
                       while (i.hasNext()) {
                         Product p := i.next();
                         if p.barcode=code then p.update(qty);
                       }   }
class Store::        update(Barcode code, int qty) { catalog.update(code,qty) }
```

The quantification on *sale.lines* is then designed as another loop in the class of the method that contains the formula in its specification:

```
class Cashdesk:: updateInventory() {
                   Iterator j := sale.lines.iterator();
                   while (j.hasNext()) {
                     LineItem l := j.next();
                     store.update(l.barcode,quantity)
                 } }
```

Now we can also give an equivalent, more direct encoding of the two quantifications, where the inner loop is for the objects whose state is being modified by the specification.

```
class Cashdesk:: updateInventory() {
                    Iterator j := sale.lines.iterator();
                    while (j.hasNext()) {
                      LineItem l := (j.next();
                      /* inlined store.update()/catalog.update() call: */
                      Iterator i := store.catalog.iterator();
                      while  (i.hasNext()) {
                        Product p := i.next();
                        if p.barcode=l.barcode then p.update(l.quantity)
                      } } }
```

In *cardPay()*, the precondition invokes the function *authorize(Card, double)* of the *Bank*. We reuse *addSale(sale)* and *updateInventory()* unchanged from the refinement for *cashPay()*. At this stage, where the *Bank* is an external class we do not need to specify the *authorize(Card, double)* method.

```
class Cashdesk:: cardPay(Card c) {
                    if (Bank.authorize(c,sale.total)) then  {
                       payment:=CardPay.New(c);
                       store.addSale(sale);
                       updateInventory()
                    } else { throw exception e(c)}   }
```

The other use cases expand in a similar way. The refinement of specifications involving universal and existential quantifications over a collection of objects/-data to Java implementation of the *Collection interface* show that formal methods should now take the advantages of the libraries of the modern programming languages such as Java. This can significantly reduce the burden on (or the amount of) verification.

6.3.2 Component-Based Architecture

The component architecture is designed from the object-oriented models in the previous sections. In contrast to the component layout in Chapter 3, where already deployment has been taken into account for the component mapping, we will first map the object-oriented model to logical components, and then discuss how they are affected by deployment. Also, we have some *a priori* components, like the hardware devices and the *Bank*.

The adaptation of the object-oriented model to a component-based model *reduces system coupling*, such that less related functionalities are performed by different components. This is done according to use cases and users (i.e. actors).

Logical Model of the Component-Based Architecture. The primary use case **UC 1** is performed by the the *SalesHandler* component, while the composition of the handler with the components for the peripherals yields the *CashDesk*. A *Store* component aggregates several *CashDesk*s and an *Inventory*.

For the other use cases, we obtain a similar structure with a controller and supporting classes (not shown in detail): Ordering stock (**UC 3**), handling deliveries (**UC 4**), stock report (**UC 5**), and changing prices (**UC 7**) are components within a *Store*.

Delivery reports (**UC 6**) are generated inside the *Enterprise* component, while product exchange between stores (**UC 8**) is managed in the *Exchange* component, which resides within *Enterprise*.

The model is called a *logical component-based architecture* because

1. It is the model of the design for the application components,
2. The interfaces are object-oriented interfaces, that is interactions are only through local object method invocations.

However, it is important to note that the object-oriented functional refinement are needed for the identification of the components and their interfaces.

We take some liberties with the design of Chapter 3: we do not model a *CashDeskLine* (see Chapter 3, Fig. 12), but only a single cash desk that accesses the inventory. Also, we omit the *CashBox* as it does not contribute to the presentation. Note that in the following, only the *SalesHandler* is actually derived from the requirements (as would be the *Clock*, the *Bank*, and the *Inventory*).

The *SalesHandler* component will be the "work horse" of our cash desk. It implements the actual *Sale* use case protocol and also provides the necessary API for accessing the ongoing sale from the GUI. As a simplification, we assume that they can happen atomically at anytime, that is, the `pure` keyword indicates that the methods calls can be interleaved with those from the protocol. The provided protocol corresponds to the trace given in the Functional Description of the **UC 1**. The method invocations on the required side are derived (manually) from the refinement of the functional specification. The multiple `update` call-outs stem from the iteration when a sale is logged and the inventory updated.

```
component SalesHandler
required interface ClockIf { date() }
required interface BankIf { authorize(..) }
required interface StoreIf { update(..), find (..), addSale(..) }
provided pure interface CashdeskIf { getItem(..), getSubTotal(..), getTotal(..), getPayment() }
provided interface SaleIf
protocol { ( [ ?enableExpress ( ?startSale date! (?enterItem find!)^(max) ?finishSale
                                ?cardPay authorize! addSale!)*
             | ?disableExpress ( ?startSale date! (?enterItem find!)* ?finishSale
                                [ ?cardPay authorize! addSale! update!*
                                | ?cashPay addSale! update!* ] )* ] )* }
class Cashdesk implements SaleIf, CashdeskIf
```

Design of the Concrete Interaction Mechanisms. After obtaining the model of the logical component-based architecture, we can replace the object-oriented interfaces by different interaction protocols according to the requirement descriptions and the physical locations of the components.

There are more than one *CashDesk* component instance, each having its own clock and sharing one *Inventory* instance per store. The interaction between them can then be implemented asynchronously using an event channel. RMI or CORBA can be used for interactions between a *Store* component and the *Enterprise* component.

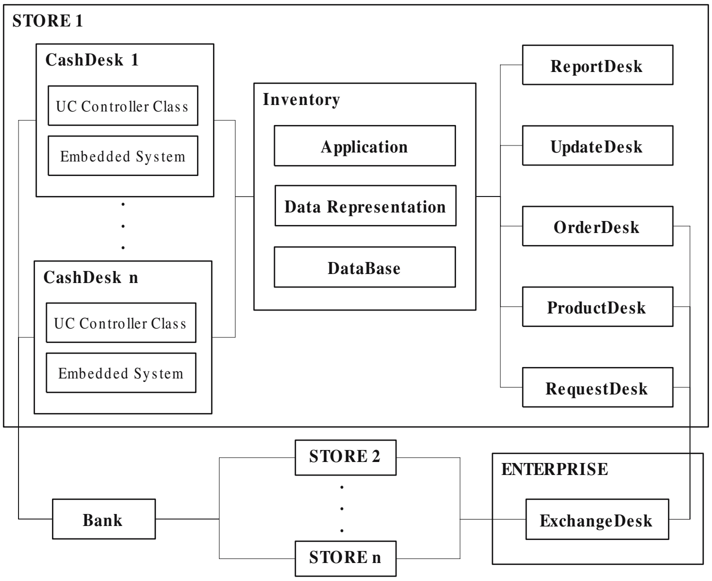

Fig. 5. Overall component view of the system

If we decompose the *Inventory* into sub-components (the three layer architecture): the *application layer*, the *data representation layer* and the *database*, we can

- Keep the OO interface between the application and data representation layer,
- Implement interaction between the data representation and the database in JDBC.

Most of these interactions mechanisms are international standards and the change from an OO interface to any of them has been a routine practice. We believe that there is no need for providing their formal models and analysis, though a formal method like rCOS is able to do this with some effort. Fig. 5 gives the overall view of the system. In the following, we discuss the detailed decomposition of the peripherals of a cash desk.

Hardware Components. The peripheral device components are modelled in rCOS only at the *contract* level, that is, with regard to the protocols. We do not give their functional description or implementation here and assume they implement their behaviour correctly. The required protocols (call-outs with trailing exclamation mark; see [8]) have been derived from the functional specifications/refinement.

The input devices are modelled as (active) rCOS processes that call provided methods of another component on input. For manual input, we model the cash desk terminal as a *black box* (we dispense with the implementing class) with buttons for starting/ending a sale, and manual input of an item and its quantity. Remember from Sec. 6.3 that we designed the controller class to handle both express mode changes. Nonetheless, here we stick to the original problem description and allow the cashier to disable it only. Thus, the protocol is still a subset of the one induced by the use case (we omit method signatures for conciseness of the presentation):

```
// define short−hand for methods
define SaleIf { enableExpress(), disableExpress(), startSale (), enterItem (..),
                finishSale (), cardPay(..), cashPay(..) }

component Terminal
required interface SaleIf
protocol { ([disableExpress!] startSale ! enterItem!* finishSale ! [cardPay! | cashPay!])* }
```

Furthermore, we assume that the bar code scanner has the same interface (although it will in practice only ever invoke the *enterItem()* method). To connect both devices to the cash desk application, we have to introduce a *controller* which merges input from both devices. For later composition, we introduce unique names to the two provided interfaces of the same type and specify the class which handles the call-ins (implementation not shown). Here, we give the *combined* required/provided protocol of call-ins (methods prefixed by `?`)/call-outs (suffixed by `!`). rCOS also permits separate protocols for a component interface, which does not reveal any dependencies on method calls.

```
component InputController
required interface SaleIf
provided interface SaleIf at PortA, PortB // interleaving
protocol { ( [?disableExpress disableExpress!] ?startSale startSale ! // relay messages
             // fan in from both devices:
             (?enterItem enterItem!)*
             ? finishSale finishSale ! [ ?cardPay cardPay! | ?cashPay cashPay!])* }
class Merge implements SaleIf
```

The cash desk display provides a way of updating the display with the current sale. For each event, the display controller queries the cash desk's current sale via getter-methods and updates the screen. The interface will be provided by the *SaleHandler* component. Also, we handle displaying the mode here. Note that the GUI has a more general protocol as we do not need to take mode changes into account for an individual sale.

```
component CashDeskGUI
required interface LightIf { lightExpress(), lightNormal() }
required interface CashdeskIf { getItem(..), getSubTotal(..), getTotal (..), getPayment() }
required interface ClockIf { date }
provided interface GUIIf { enterItem(..), startSale(), finishSale (), cardPay (..),
                           cashPay (..), enableExpress(), disableExpress() }
protocol { ( [?enableExpress lightExpress!| ?disableExpress lightNormal!] ?startSale date!
             (?enterItem getItem! getSubTotal!)* ?endSale getTotal!
             [?cardPay | ?cashPay] getPayment! )* }
class GUI implements GUIIf
```

The *Printer* component shall employ the same design, providing PRINTER. PRINTERIF.

As the system should use a bus architecture, updates have to be done in an event-based fashion, i.e., we need a *BusController* component that proxies between all devices and acts as a fan-out when an event has multiple subscribers. Contrary to the design document, we do not employ a broadcast architecture: for example, the controller makes sure that the business logic processes an `enterItem` event *first*, and only *then* notifies the display. Likewise, it drives the printer.

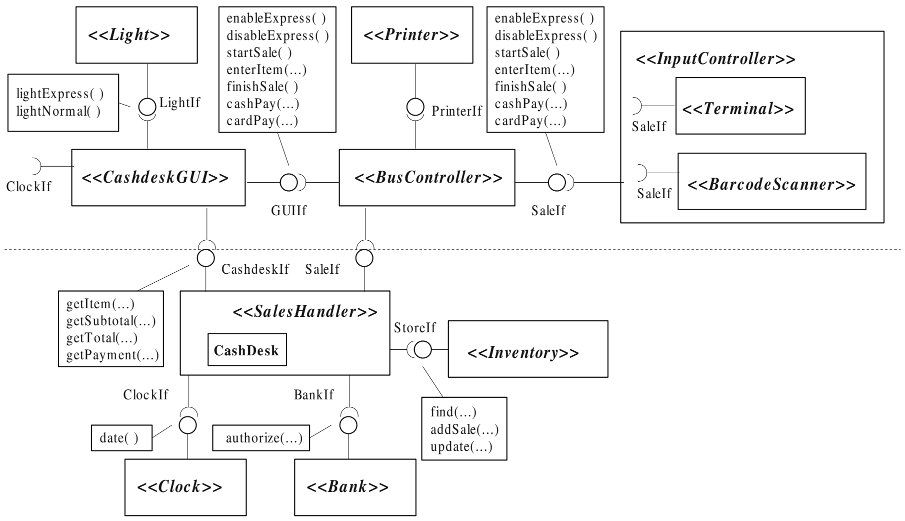

Fig. 6. Deployed components for a single Cashdesk

component BusController
provided interface SaleIf *// to InputController*
required interface CashDeskGUI.GUIIf *// elided, see above*
required interface Printer.PrinterIf *// ditto*
required interface SaleIf *// from business logic*
protocol { /∗ fan−out for each call−in elided ∗/ }
class Bus **implements** SaleIf

We concede that this component design means that the *BusController* must be modified each time a new subscriber is added to the system.

We plumb the *BarcodeScanner* and *Terminal* component into the *InputController*, which we connect to the *BusController*. That is in turn connected to the *SalesHandler*. We omit detailed discussion of the other interfaces; dependent components mentioned in `with`-clauses are deployed automatically as long as there are no ambiguities with regard to interfaces:

component Cashdesk
  deploy CashDeskGUI **with** Clock, Light
  deploy InputController **with** Barcode **at** PortA, Terminal **at** PortB
  deploy BusController **with** InputController, CashDeskGUI, Printer
  deploy SalesHandler **with** Clock,Bank,Inventory,CashDeskGUI,BusController

Assuming availability of the required components of the *SalesHandler*, the resulting component is *closed*, as all required interfaces are provided. For the resulting component diagram, see Fig. 6; the upper half indicates the peripherals, the lower half the components derived from the use case.

With regard to formal rCOS (see e.g. [5,10]), we note that only components whose traces start with a call-in are *rCOS-components*. Those that have a call-out at the start of their trace, are actually *rCOS-processes*. For **UC 1**, only the input devices used by the actor are processes.

Modelling Deployment. For a consistent rCOS model, this means we must modify the existing model to take into account the *deployment boundaries*, *middle-ware* and their effect on communication object references.

We also note that the different failure modes of remote communication must be taken into account and may require to revisit the Design, as for example, suddenly functions (in the mathematical sense) may *fail* when they are invoked on remote hosts. This is a field of ongoing investigation.

6.4 Analyses

This section outlines how to add the specification and analysis of the extra functionalities given in the description document. It then continues with a description of the actual analyses of the functional and behavioural properties that have been carried out with tool support.

Extra-Functional Properties

We specify extra functionality of a method as a property for the time interval for the execution of the method. We use temporal variables whose values depend on the reference time interval for the execution of methods for our specification. Those variables could be ET_m which is the duration of the execution of method m in the worst case, or $N\_Customers$ which is the number of customers in the referenced observation time interval. From the intended meaning, the variable ET_m is rigid, its value does not depend on the reference interval. For a formula f on the rigid and temporal variables, for a probability p, $[f]_p$ is a formula saying that f is satisfied with the probability p. As it is well-known in the interval logic, the formulas $\phi;\psi$, which corresponds to the sequential composition of formulas ϕ and ψ, holds for an interval $[a,b]$ iff there is $m \in a..b$ such that ϕ holds for interval $[a,m]$ and ψ holds for interval $[m,b]$. Let ℓ be a temporal variable denoting the length of the interval it applied to. Intuitively, the formula

$[0 \leq ET\_ScanItem < 5]_{0.9} \wedge [0 \leq ET\_ScanItem < 1.0]_{0.05}$

says that the execution time for the operation $ScanItem$ is within 5 seconds with probability 0.9, and it is less than 1 second with probability 0.05.

Since the arrival and leaving rates are the same: 320/3600 arrivals per second, and constant, with an exponential distribution,we can derive that $[N\_Customers = \frac{2}{45}\ell]_1$ holds for all intervals. As another example, by estimating the average waiting time for customers here we show how to include QoS analysis in our framework. Let $ET\_Service$ stand for the average service time for customers. It is easy to calculate the possibility of $ET\_Service$ from the above specification, i.e. $ET\_Service = 32.075$. Therefore, the rate of service is $\mu = 1/ET\_Service = 1/32.075 = 0.0311$. Also, we have that the rate of customers arriving is $\lambda = N\_Customers/\ell = 4/45$.

6.4.1 Verification, Analysis and Tool Support

Various verifications and analyses are carried out on different models. For the requirement model, the trace equivalence between the sequence diagram and its state diagram has been experimentally checked with FDR. We manually checked the consistency between the class declarations (i.e. the class diagrams) and the functionality specification to ensure that all classes and attributes are declared in the class declarations. This is obviously a syntactic and static semantic check that can be automated in a tool. We can further ensure the consistency by translating the rCOS functionality specification into a JML specification and then carry out runtime checking and testing. Also, some of the development steps involving recurrent patterns can be automated.

Runtime Checking and Testing in JML. We have not checked the correctness of the design against the requirement specification for removing possible mistakes made when manually applying the rules. However, we have translated some of the design into JML [13] and carried out runtime testing of specifications and the validity of an implementation.

We translate each rCOS class *C* into two JML files, one is *C.jml* that contains the specification translated from the rCOS specification, and the other is a Java source file *C.java* containing a code that implements the specification. During the translation, the variables used in the rCOS specification are taken as specification-only variables in *C.jml*, that are mapped to program variables in *C.java*. The translated JML files can be compiled by the JML Runtime Assertion Checker Compiler (*jmlc*). Then, test cases can be executed to check the satisfaction of the specification by the implementation. The automatic unit testing tool of JML (*jmlunit*) can be used to generate unit testing code, and the testing process can be executed with *JUnit*.

For example, a JML code snippet of the *enterItem*() design in Section 6.3 is shown on the left of Fig. 7. Notice that the code in the dotted rectangle gives the specification of the exception that was left unspecified in Section 6.3.

```
/*@ public normal_behaviour
  @   requires (\exists Object o; theStore.theProductList.contains(o);
  @            ((Product)o).theBarcode.equals(code)); ...
  @   ensures   theLine != \old(theLine) &&
  @     theLine.theBarcode.equals(code) &&...
  @ also
  @ public exceptional_behaviour
  @    requires !(\exists Object o; theStore.theProductList.contains(o);
  @            ((Product)o).theBarcode.equals(code));
  @    signals_only Exception;
  @*/
public void enterItem(Barcode code, int quantity) throws Exception;
```

```
public void enterItem(Barcode code, int quantity)
        throws Exception{
  line = new LineItem(code, quantity);
  Iterator it = store.productList.iterator();
  boolean t = false;
  while (it.hasNext()){
    Product p = (Product)it.next();
    if (p.barcode.equals(code)){
      line.total = p.price * quantity;
      t = true;
      sale.lines.add(line);
    }
  }
  if (!t)   throw new Exception();
}
```

Fig. 7. JML Specification and Implementation

The final code implementing the *enterItem*() specification is shown on the right of Fig. 7. Before getting the final code, we encountered two runtime errors reported by the testing process. One error resulted from the implementation which did not handle an input that falsifies the precondition. The reason for the other error is that one invariant is false after method execution. Testing is not sufficient for correctness. Therefore, it is also desirable to carry out static analysis, for instance with ESC/Java [3].

QVT Transformation Support. Our long term goal is to implement correctness preserving transformations that support a full model driven development process. The problems we are concerned with are the consistency among models on the same level, and the correctness relation between models in different levels. The meaning of consistency among models on the same level is that the models of various views need to be syntactically and semantically compatible with each other. The meaning of correctness relation between models on different levels is that a model must be semantically consistent with its refinements [16].

We plan to use QVT [7], a model transformation language standard by OMG, to implement these model transformations. We have already defined the required rCOS metamodels for object diagrams, object sequence diagrams, component diagrams, component interaction diagrams and state machines. Pre- and post-conditions can also be translated into the respective clauses of a QVT program.

The refinement of the use cases on the object level through the expert pattern is done manually now, but it can be implemented using QVT, and automated. The correctness of the expert pattern is proved by rCOS. We have already explored correctness preserving transformations in a object-oriented design in [24].

Then we can apply architectural design to decompose the object model into a component model by allocating use cases, classes, associations and services to components. The component model should be a refinement of the application requirement model. This step can also be implemented as a QVT transformation. The correctness of the transformation from object model to component model should be proved in rCOS.

Verifying Interaction Protocols. Composition of components not only requires that the interfaces and their types match. Also, the interaction protocols must be compatible; if two interfaces are composed, the corresponding traces must match, i.e., the sequences of call-ins/call-outs must align: unexpected call-ins are ignored by the callee and will deadlock the caller.

We automatically check protocol consistency by generating CSP processes for each interface. Interface composition is then modelled through pairwise parallel composition. As a composed interface is uniquely defined in the specification, we can successively check each composition for deadlock freedom and incrementally add successive interfaces. Model checking with FDR would indicate a deadlock if an operation call required by some component is not provided by any of its partner components at some moment in time.

In an application, these can also be implemented as runtime checks using extensions for aspect-oriented programming that capture temporal behaviour like Tracematches [1] or Tracechecks [2].

6.5 Summary

We have presented our modelling of the Common Component Example in rCOS, the Relational Calculus of Object and Component Systems. Based on the problem description, we have developed a set of interrelated models for each use case which separately models the different concerns of control and data. The rigorous approach ensures that we can be of *high confidence* that the resulting *program* implements the desired behaviour *correctly* without having to prove this on the generated code, which usually is very difficult or even impossible. As the problem description is not always amenable to modelling in rCOS, we occasionally had to simplify the model.

For each use case, a state diagram, a sequence diagram, its trace and the functional specification of its operations with pre- and postconditions are provided. These different aspects shall help all participants involved in the development process (designers, programmers) to share the same overall understanding of the system. Consistency of models is checked through processes that can be automated, e.g. by type checking of OO methods and model checking of traces.

The functional specifications in rCOS are then refined to a detailed design very close to Java code through *correct* rules for patterns like the Expert Pattern or translation of quantification. The generated code can be enriched with JML annotations derived from the functional specification and invariants. The annotations can then be used for runtime checking or static analysis.

From the OO model, we then derive a more convenient component model using Class Decomposition and grouping classes into components. The rCOS component model allows us to reason about component interaction, defined by the traces from the specifications, ruling out "bad" behaviour like deadlocks. We discuss issues of (distributed) deployment and necessary middle-ware.

For extra-functional analysis, we applied the Probabilistic Interval Temporal Logic to specify extra-functional properties given in the problem description. Then, we conducted the estimation of the average waiting time for customers.

Apart from concrete tool support, we also point out ongoing work and research on automating the different parts of the development process.

The generated code and additional information is available from the project web page at `http://www.iist.unu.edu/cocome/`.

Acknowledgements. The authors thank Wang Xu and Siraj Shaikh at UNU-IIST for helpful suggestion on CSP and FDR, and the careful reviewers.

References

1. Allan, C., Avgustinov, P., Simon, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittamplan, G., Tibble, J.: Adding Trace Matching with Free Variables to Aspect J. In: OOPSLA 2005 (2005)
2. Bodden, E., Stolz, V.: Tracechecks: Defining semantic interfaces with temporal logic. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, Springer, Heidelberg (2006)
3. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)

4. Chandy, K.M., Misra, J.: Parallel Program Design: a Foundation. Addison-Wesley, Reading (1988)
5. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. Technical Report 350, UNU-IIST, P.O. Box 3058, Macao SAR, China, Accepted by FSEN 2007 (2006), http://www.iist.unu.edu
6. Chen, X., Liu, Z., Mencl, V.: Separation of concerns and consistent integration in requirements modelling. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, Springer, Heidelberg (2007)
7. Object Management Group. MOF QVT final adopted specification, ptc/05-11-01 (2005), http://www.omg.org/docs/ptc/05-11-01.pdf
8. He, J., Li, X., Liu, Z.: Component-based software engineering. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, Springer, Heidelberg (2005)
9. He, J., Li, X., Liu, Z.: rCOS: A refinement calculus for object systems. Theoretical Computer Science 365(1-2), 109–142 (2006)
10. He, J., Li, X., Liu, Z.: A theory of reactive components. In: Liu, Z., Barbosa, L. (eds.) Intl. Workshop on Formal Aspects of Component Software (FACS 2005). ENTCS, vol. 160, pp. 173–195. Elsevier, Amsterdam (2006)
11. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
12. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Reading (2002)
13. Leavens, J.L.: JML's rich, inherited specification for behavioural subtypes. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, Springer, Heidelberg (2006)
14. Liu, X., Liu, Z., Zhao, L.: Object-oriented structure refinement - a graph transformational approach. Technical Report 340, UNU-IIST, P.O. Box 3058, Macao SAR, China (2006), http://www.iist.unu.edu; In: Proc. Intl. Workshop on Refinement, ENTCS (Extended version accepted for journal publication)
15. Liu, Z., He, J. (eds.): Mathematical Frameworks for Component software: Models for Analysis and Synthesis, Series on Component-Based Software Development, vol. 2. World Scientific, Singapore (2006)
16. Liu, Z., Mencl, V., Ravn, A.P., Yang, L.: Harnessing theories for tool support. International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA06), Full version as UNU-IIST Technical Report 343 (2006), http://www.iist.unu.edu
17. Meyer, B.: Object-oriented software construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
18. Möller, M., Olderog, E.-R., Rasch, H., Wehrheim, H.: Linking CSP-OZ with UML and Java: A case study. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, Springer, Heidelberg (2004)
19. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In: 1st Intl. Conf. on Software Engineering and Formal Methods (SEFM 2003), IEEE Computer Society Press, Los Alamitos (2003)
20. Plasil, F., Visnosky, S.: Behavior protocols for software components. IEEE Trans. Software Eng. 28(11), 1056–1070 (2002)
21. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
22. Roscoe, A.W., Wu, Z.: Verifying Statemate statecharts using CSP and FDR. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, Springer, Heidelberg (2006)

23. Schneider, S.: Concurrent and Real-time Systems. Wiley, Chichester (2000)
24. Yang, L., Mencl, V., Stolz, V., Liu, Z.: Automating correctness preserving model-to-model transformation in MDA. In: Proc. of Asian Working Conference on Verified Software, UNU-IIST Technical Report 348 (2006), http://www.iist.unu.edu

A Full CSP/FDR Listing for Consistency

```
-- Define events:
channel enableExpress, disableExpress, startSale, enterItem, finishSale, cashPay, cardPay

-- Define the process corresponding to the regular expression:
Trace = (TraceExMode [] TraceNormalMode) ; Trace
TraceNormalMode = disableExpress -> TraceNormalSale
TraceNormalSale = startSale -> enterItem -> TraceEnterItemLoopStar
                  ; finishSale -> ((TraceCashPay [] TraceCardPay)
                  ; (SKIP [] TraceNormalSale))
TraceEnterItemLoopStar = SKIP [] (enterItem -> TraceEnterItemLoopStar)
TraceCashPay = cashPay -> SKIP
TraceCardPay = cardPay -> SKIP

TraceExMode = enableExpress -> TraceESale
TraceESale = startSale -> enterItem -> TraceEMode(7)
             ; ( finishSale -> (TraceCashPay ; (SKIP [] TraceESale)))
TraceEMode(c) = if c == 0 then SKIP
                              else (SKIP [] (enterItem -> TraceEMode(c-1)))

-- State Diagram:
datatype Mode = on | off
State = Init(on) [] Init(off)
-- Resolve outgoing branches non-deterministically:
Init(mode) = (if mode == on then disableExpress -> StateNormalMode(off)
                                else STOP)
             [] (if mode == off then enableExpress -> StateExpressMode(on)
                                else STOP)
StateNormalMode(mode) = (startSale -> enterItem -> StateEnterItemLoopStar)
                        ; finishSale -> ((StateCashPay [] StateCardPay)
                          ; ((enableExpress -> StateExpressMode(on))
                            [] StateNormalMode(mode)))

StateEnterItemLoopStar = SKIP [] (enterItem -> StateEnterItemLoopStar)
StateCashPay = cashPay -> SKIP
StateCardPay = cardPay -> SKIP

StateEMode(c) = if c == 0 then SKIP
                              else (SKIP [] (enterItem -> StateEMode(c-1)))

StateExpressMode(mode) = startSale -> enterItem -> StateEMode(7)
                         ; finishSale -> (StateCashPay
                           ; ((disableExpress -> StateNormalMode(off))
                             [] StateExpressMode(mode)))

-- Check trace equivalence:
assert State [T= Trace
--^ does not hold as trace abstracts from the guard,
-- permits: enableExpress -> ... -> enableExpress
assert Trace [T= State

-- Make sure both mechanisms can't deadlock:
assert Trace :[deadlock free [F]]
assert State :[deadlock free [F]]
```

B Packages

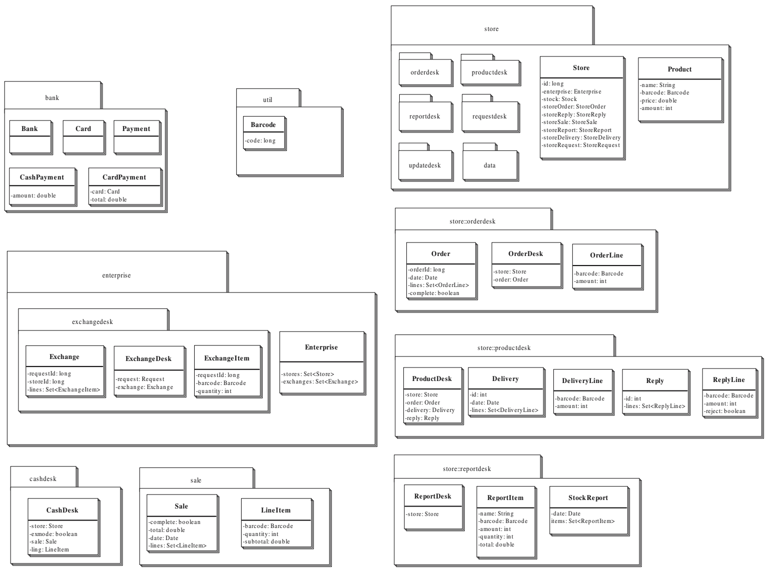

Fig. 8. Packages

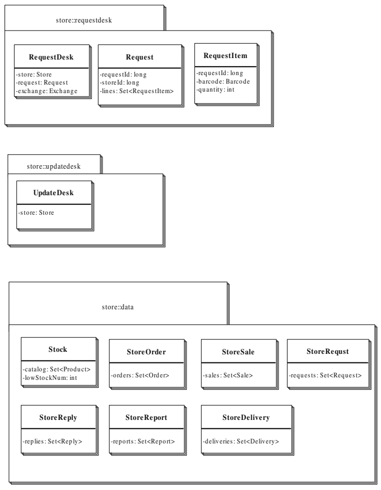

Fig. 9. Packages