A Refinement Driven Component-Based Design*

Zhenbang Chen, Zhiming Liu, Volker Stolz and Lu Yang
International Institute for Software Technology
United Nations University
{zbchen,lzm,vs,yl}@iist.unu.edu

Anders P. Ravn
Department of Computer Science
University of Aalborg, Denmark
apr@cs.aau.dk

Abstract

Modern software applications ranging from enterprise to embedded systems are becoming increasingly complex, and require very high levels of dependability assurance. The most effective means to handle complexity is separation of concerns and incremental development, and assurance of dependability requires formal methods*. We report here our experience on these issues in an application of a formal calculus, rCOS, to a component-based design of the point of sale system (POS). We demonstrate the possibility in scaling-up correctness by design and discuss how rCOS may be integrated with current and emerging software engineering tools.*

Keywords: *Software development process, object-orientation, component-based modeling, refinement.*

1. Introduction

Modern software applications ranging from enterprise to embedded systems are complex, and in order to produce high quality systems efficiently, component-based technology is used. This entails that the focus moves from programming in the small to composition of modules. Components are not the small standard library units that we know from current OO-languages. They are richer and thus it is not sufficient to specify them by their syntactical interfaces and rely on informal semantics. A more precise contract is needed to specify the properties of the functionality provided or required by a given component. Contracts are specifications, and the components are implementations. Within the "formal methods" community this is a well known setting, and theories and techniques, developed since at least the 1970'ies, provide answers to questions which now become of very practical concern:

1. is an implementation consistent with the contract?
2. are the two contracts compatible when a required and a provided interface are connected?
3. when is one component replaceable by another?
4. what are the interfaces and contracts of a composite component, or assembly of components?

Traditionally, the question of implementation correctness has attracted most attention, because it is closely related to programming language semantics, program synthesis and program verification. Our work with rCOS [9, 10] takes this question further and looks at the issues of contract composition and compatibility. rCOS has some facets of a coordination language.

Since rCOS focuses on all of requirements, architecture and design modeling, it must incorporate different views of a system: the *static structure*, *dynamic behavior*, *interaction between components*, and the *static data functionalities*. UML [8], as a de facto standard, provides syntax for modeling these views: the static structural view is modeled by *class diagrams* or *component diagrams*, dynamic behavior by *state diagrams*, and interactions by *sequence diagrams*. rCOS complements such descriptions with precise semantics and rules for reasoning about relations among the constructs.

We report on our experience with rCOS in a model driven development of a point of sale system (POS) for a retail store. We give examples that illustrate:

- how different aspects, including static structure, interactions, and data functionality, of the system are specified and analyzed in rCOS, and
- how different aspects are designed by effectively using refinement rules provided in rCOS.

Finally we consider how different tools for verification and analysis may strengthen the rCOS framework.

Related formalisms. Eiffel [16] first introduced the idea of design by contacts for object-oriented programming. The notion of designs for methods in object-oriented rCOS is similar to the use of assertions in Eiffel, and thus also supports similar techniques for static analysis and testing. JML [2] has recently become

*This work is partially supported by the project HighQSoftD funded by Macao Science and Technology Development Fund, and the NSF project 60573085 and 863 of China2006AA01Z165.

a popular language for modeling and analysis object-oriented designs. It shares similar ideas of using assertions and refinement as behavioral subtypes in Eiffel. The strong point of JML is that it is well integrated with Java and comes with parsers and tools for runtime checking and testing.

In Fractal [18], 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. 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 modeling 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 [17] in that a multi-notational modeling language is used to encompass 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.

Overview. Section 2 introduces rCOS which is used in Section 3 to specify the requirements of POS. In Section 4 we apply the refinement rules to construct an OO design of the system. In Section 5, we transform the model of the OO design to a component-based architecture. We will discuss there how different interaction mechanisms can be used for implementing interfaces of components. In Section 6, we report on results of applying tools for verification and analysis. Finally, Section7 concludes and discusses the findings.

2. A Summary of rCOS

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

The essential idea of UTP is that any program is a binary relation on the *observables* of the program. For a sequential imperative program, we observe the initial values of the variables before execution and their final values after execution, and furthermore we should know whether the execution terminates. The observables of such a program include the program variables, say x representing the values of the program variables before the execution, and the variables x' representing the values after the execution. These observables that reflect the values of program variables are supplemented by Boolean model variables ok and ok' that denote the termination status of the program.

The execution of the program is then specified by a predicate $pre(x) \vdash post(x,x')$, called *a design*. The meaning of the design $pre(x) \vdash post(x,x')$ is defined by the predicate $pre(x) \wedge ok \Rightarrow ok' \wedge post(x,x')$. It asserts that if the program is activated from a well-defined, terminated state, that satisfies the *precondition* $pre(x)$ then the program terminates, i.e. $ok' = true$, in a state satisfying the postcondition $post$. For example, an assignment $x := x+y$ is defined as $true \vdash x' = x+y$.

In UTP, the healthiness conditions of designs are studied and it is proven that all designs are closed under all standard programming constructs like sequential composition, choice, and iteration. Sequential composition $D_1;D_2$ is simply relational composition.

For concurrency with communicating processes, additional model observables are used to record communication traces. Communication readiness is expressed by additional *guard* predicates. The healthiness conditions of these *guarded designs* give the semantic basis in rCOS for the integration of the static functionality specified by simple designs with interaction protocols specified by traces. Their dynamic behavior is linked by guarded designs that define the semantics of a state machine [10, 5].

The *refinement* relation between designs is defined in UTP as well as in rCOS as logical implication.

rCOS extends UTP to formalize the concepts of object oriented programming: classes, object references, method invocation, subtyping and polymorphism [9]. Like Java, an OO program in rCOS is specified as $Classes \bullet main$ consisting of a number of *class declarations* and a *main program*. A class can be public or private and declares its attributes and methods; these can be public, private or protected. The main program is given as a *main class*. Its attributes are the global variables and it has a main method $main()$. We assume that there are no hidden side effects: the main method only accesses and modifies objects of the classes by invoking methods of these classes.

2.1. Specification of classes

With UML, some information of the list of class declarations can be represented in a class diagram, including class names, their attributes and some constraints such as multiplicities of associations. The definition of

method bodies has to be given in sequence diagram and state diagrams, which usually involve low level design details. In rCOS, we write a class specification in the following format:

class C [**extends** D]{
attr $T\,x = d, \ldots, T_k\,x = d$
meth $m(T\ in; V\ return)$ {
pre: $c \vee \ldots \vee c$
post: $\wedge$ $(R;\ldots;R) \vee \ldots \vee (R;\ldots;R)$
$\wedge$ ……
$\wedge$ $(R;\ldots;R) \vee \ldots \vee (R;\ldots;R)$ }
……
meth $m(T\ in; V\ return)$ {…… }
……
invariant Inv }

The initial value of an attribute is optional, and an attribute is assumed to be public unless it is tagged with reserved words *private* and *protected*. If no initial value is declared it will default to *null*.

Each c in the precondition represents a condition to be checked; it is a conjunction of primitive predicates.

A *design* $p \vdash R$ for a method is written as **Pre** p and **Post** R An R in the postcondition is of the form $c \wedge (le' = e)$, where c is a condition, le an *assignable expression* and e an expression. An assignable le is either a primitive variable x, or an attribute name a or $le.a$. An expression e can be 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 reader can see the influence of TLA$^+$ [12], UNITY [4] and Java on the above format.

2.2. Datatypes

In rCOS, we distinguish data from objects and thus a datum, such as an integer or a boolean value does not have a reference. For this paper, we assume the data types of $V ::= long \mid double \mid char \mid string \mid bool$. Let C be a class name in CN the set of *class names*, then types are generated by:

$$T ::= V \mid C \mid set(T) \mid bag(T)$$

We use the operations $add(T\,a)$, $contains(T\,a)$, $delete(T\,a)$ on a set or a bag with their usual meaning. For a variable s of type $set(T)$, the specification statement $s.add(T\,a)$ equals $s' = s \cup \{a\}$, and $s.contains(T\,a)$ equals $a \in s$, and $s.sumAll()$ is the sum of all elements of s, which is assumed to be a set of numbers. We use curly brackets $\{e_1, \ldots, e_n\}$ and square brackets $[[e_1, \ldots, e_m]]$ to define a set respectively a bag. For a set s such that each element has a key, $s.find(ID\ id)$ denotes the function that returns the element whose key equals id if there is one, or *null* otherwise. Notice that Java implements these types via the *Collection* interface. Therefore, these operations in specification statements can be easily coded in Java.

In a specification, we use $C\ o$ to denote that object o is of type C, and $o \neq null$ to denote 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. We use the short hand $o \in C$ to denote $o \neq null$ and its type is C.

In rCOS, we require side effect free evaluation of expressions, thus the Java expression $newC()$ is not a rCOS expression. Instead, we have $C.New(x)$ as a command that creates an object of C and assigns it to variable x.

However, in the specification of POS, we use the shorthand $x' = C.New()$ to denote $C.New(x)$, and $x' = C.New[v_1/a_1, \ldots, v_k/a_k]$ to denote the predicate $C.New[v_1/a_1, \ldots, v_k/a_k](x)$ that a new object of class C is created with the attributes a_i initialized with v_i for $i = 1, \ldots, k$, and this object is assigned to variable x. Similarly, we use $x' = m()$ for a method with a return parameter *return* of the same type of x.

2.3. Refinement

OO design is to design object interactions so that objects interacts with each other to realize the functionality specified in the class declarations. 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 behavior of the main method with respect to the global variables is preserved. It is an extension to the notion of data refinement in imperative programming, with a semantic model dealing with object references, method invocation, and polymorphism. In such a refinement, all non-public attributes of the objects are treated as local (internal) variables [9].
2. Refinement of the class declaration section: $Classes_1$ is a refinement of $Classes$ if $Classes_1 \bullet main$ refines $Classes \bullet main$ for all $main$. This means that $Classes_1$ supports at least as many services as $Classes$.
3. Refinement of a method of a class in $Classes$. Obviously, $Classes_1$ refines $Classes$ if the public class names in $Classes$ are all in $Classes_1$ and for each public method of each public class in $Classes$ there is a refined method in the corresponding class of $Classes_1$.

Very interesting results on completeness of the refinement calculus are available in [14].

In an OO design there are mainly three kinds of refinement: *Delegation of functionality* or *responsibilities*, *attribute encapsulation*, and *class decomposition*.

Delegation of functionality. Assume that C and C_1 are classes in *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 the method c are accessible in C_1, we have $Classes \sqsubseteq Classes_1$, where $Classes_1$ is obtained from *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*.

This rule and other refinement rules can prove big-step refinement rules, such as the following **expert pattern**, that will be repeatedly used in the design of POS.

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

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

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

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

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 $\{a_{11}.....a_{1k_1}.x_1, ..., a_{\ell 1}.....a_{\ell k_\ell}.x_\ell\}$ *have a common prefix, say 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* $a_{11}...., a_{ij}$ *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 *knows* the data needed for the responsibilities assigned to them.

Encapsulation. 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 two $D_1 :: T_1\ x; m()\{...\}$ 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 in [9].

An important point 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 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 transformation can be derived for the change in the specification of the functionality and object interactions. For details, please see our work in [14].

2.4. Component-based modeling in rCOS

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

A closed component has a provided interface and a code that implements the *contract* of the interface. The *contract* of the interface of a component describes what is needed for the component to be *used* in building and maintaining software systems. It specifies *functionality*, *behavior*, *protocols*, *safety*, and through additional model variables it may extend to *real-time*, *power*, *bandwidth*, *memory consumption* and *communication mechanisms*, that are needed for composing the component in the given architecture for the application of the system. For the POS example, we only consider *functionality*, *behavior* and *protocols*.

Note that an interface can be the union of a number of interfaces. Therefore, in a specification one can write the interfaces separately.

An *open component* requires methods from other components. It implements a contract *IC* of its provided interface *under an assumed contract OC of its required interface* if each provided method with the assumed specification of the method of *OC* refines the specifi-

cation of the corresponding method in *IC*.

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.

Compositions for *disjoint union* of components and for *plugging* components together, for *gluing components* by processes are defined, and their closure properties and the algebraic properties of these compositions are studied [5].

The contracts together with the interfaces of a component provide a black-box specification of the component. A contract in rCOS defines the unified semantic model of implementations of interfaces in different programming languages, and thus clearly supports interoperability of components and analysis of the correctness of a component with respect to its interface contract. The theory of *refinement* of contracts and components in rCOS characterizes component substitutivity, and supports independent development of components.

3. Requirements Elicitation

Requirements capture in general presupposes a problem domain analysis [15] which is also called *domain engineering* [1]. It defines the context or environment for the system under development as a collection of interrelated objects (sometimes called actors) and the relevant operations on them. It thus defines the vocabulary for specifying requirements, and later on for design: the necessary interfaces for interaction, and the state information that must be recorded internally by the system in order to implement the functionality. The concepts are recorded in a *structure view*. It can be depicted by a class diagram or packages of class diagrams.

The *functional* and *behavioral* requirements are facilitated through an analysis of the *business processes* described as *use cases*. A use case specification includes three views:

1. *Interaction* with the environment. Interactions are described by a protocol over the operations of the actors. In rCOS, this protocol is a set of *traces* of method invocations, and here we define it by a UML sequence diagram, called a *use case sequence diagram*.

2. The *dynamic behavior* is modeled by a *guarded state transition system*, that also can be defined by a UML state diagram. The sequence diagram and the state diagram must be trace-equivalent. In addition to its operational semantics, a state diagram also has a denotational *failure-divergence* semantics [9, 19]. This view may be used for verifying deadlock and livelock freedom by model checking state reachability.

3. The static *functionality view* that specifies what operations should do to the internal representation of its external object when invoked. For the purpose of *compositional* and *incremental* specification, we introduce a designated class for each use case, called the *use case controller class* of that use case, and specify each operation of the use case as a method of this class. The signatures of the methods must be consistent with those used in the interaction and dynamic views.

The different views are closely related and they all together form a consistent whole view of the system. There is no sequential order for producing the models. An initial structural view is needed for describing the interactions. The detailed specification of the interactions and functionality provide the full information needed to construct the final model of the structure view. The *consistency* and *integrated semantics* of the different views are studied in [6]. In the following, requirements for the POS example are used to illustrate this.

3.1. The Point of Sale System

The point of sale system (POS) was originally used as running example in Larman's book [13] to study concepts OO systems design. An extended version is now used as the case study in the Common Component Modeling Contest [7] and identified as a pilot project at the Asian Working Conference on Verified Software [20].

3.1.1. Problem domain. POS is a computerized system typically used in a retail store. It records sales, handles both cash payments and card payments as well as inventory management. Furthermore, the full system deals with ordering goods and generates various reports for management purposes. The system can be a small system, containing only one terminal for checking out customers and one terminal for management, or a large system that has a number of terminals for checking out customers in parallel, or even a network to support an enterprise of a chain of supermarkets. We consider the development of a POS that is used in one store, but it has a number of checkout points. The whole system includes hardware components such as computers, bar

code scanners, card readers, printers, and a software to run the system. To handle credit card payments, orders and delivery of products, we assume a *Bank* and a *Supplier* that interact with POS.

3.1.2. Requirements – specification of interactions. The problem description provides the initial context and vocabulary for further requirements elicitation through studying the business processes.

A *use case* specifies how the system interact with actors and its environment in a business process. An actor, such as a *Cashier* who checks out customers, interacts with the system by calling a *system operation* to request a service of the system.

There are many use cases for a full POS, depending on what business processes the client of the system want the system to support. One of the main ones is *processing sales*, that is denoted by the use case **UC1**: *Process sales*. It can informally described as follows.

The main courses of interactions between the actors and the system is described as follows.

1. When a *customer* comes to the *checkout point* with their *items*, the cashier initiates a new *sale*.
2. The cashier enters each item, either by typing or scanning in the *bar code*, if there is more than one of the same item, the cashier can also enter the *quantity*. The system records each item and its quantity and calculates the subtotal.
3. When there are no more items, the *total* of the sale is calculated. The cashier tells the customer the total and asks her to pay.
4. The customer can pay by cash or a credit card. If by cash, the amount received is entered. The system records the *cash payment* amount and calculates the change. If by credit card, the card information is entered. The system sends the credit payment to the *bank* for *validation*. The payment only succeeds if a positive validation reply is received.

   After payment, the inventory of the store is updated and the completed sale is logged.

There are exceptional courses of interactions, e.g., the entered bar code is not known in the system, the customer does not have enough money for a cash payment, or the authorization reply is negative. A system needs to provide means of handling these exception cases, such as canceling the sale or change to another way of paying for the sale. At the requirements level, we capture these exceptional conditions as preconditions.

We model the interaction protocol that the system offers the actor, i.e. Cashier. This is given as the *use case sequence diagram* in Fig. 1; the set of traces of the

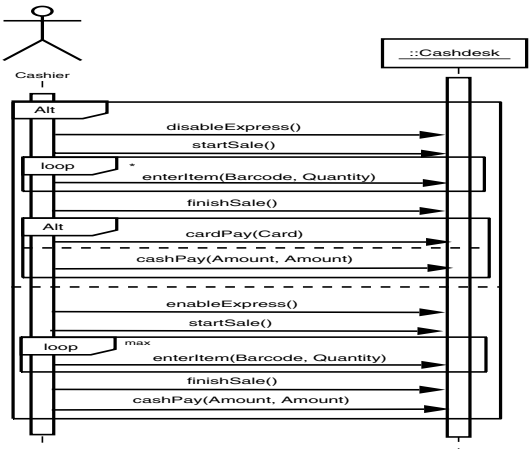

Figure 1. Sequence diagram for UC1

diagram is given by the regular expression:

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

The flow of control in the use case could also be given by a state diagram that we omit here.

3.1.3. Specification of the functionality. The functionality specification of use case **UC1** consists of the class declarations represented by the class diagram in Fig. 2, and the following specification of the methods of the use case controller class *Cashdesk*.

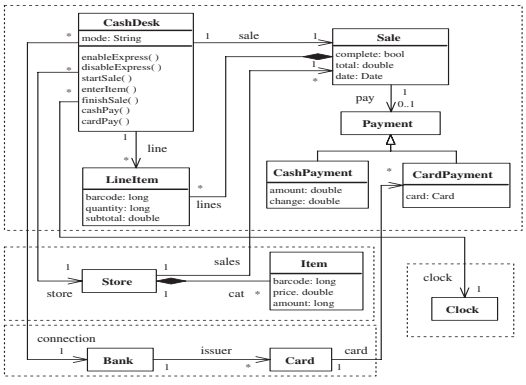

Figure 2. Class Diagram for POS

Use Case UC1: Process Sales

class *Cashdesk*{
meth *enableExpress*() {
pre: *true*
post: $\wedge$ *light.display*$'$ = *green* }
meth *disableExpress*() {
pre: *true*
post: $\wedge$ *light.display*$'$ = *yellow* }
meth *startSale*() {
pre: *true*
post: $\wedge$ *sale*$'$ = *Sale.New*(*clock.date*()/*date*)}
meth *enterItem*(*long c*,*double q*) {
pre: /** the input barcode *c* is valid **/
store.cat.find(*c*) $\neq$ *null*
post:
$\wedge$ *line*$'$=*LineItem.New*(*c*/*barcode*,*q*/*quantity*)
$\wedge$ *line*$'$*.subtotal*=*store.cat.find*(*c*)*.price* $\times$ *q*
$\wedge$ *sale.lines.add*(*line*$'$) }
meth *finishSale*() {
pre: *true*
post:
$\wedge$ *sale.complete*$'$ = *true*
$\wedge$ *sale.total*$'$=*addAll*[[*l.subtotal*|*l* $\in$ *sale.lines*]]}
meth *cashPay*(*double a*;*double c*) {
pre: /** amount no less than total **/
a$\geq$*sale.total*
post:
$\wedge$ *sale.pay*$'$ = *CashPayment.New*(
a/*amount*,*a* $-$ *sale.total*/*change*)
$\wedge$ (*c*$'$ = *a* $-$ *sale.total* **;** *store.sales.add*(*sale*))
$\wedge$ $\forall$*l* $\in$ *sale.lines*,$\forall$*p* $\in$ *store.cat* : (
if *p.barcode* = *l.barcode* **then**
p.amount$'$ = *p.amount* $-$ *l.quantity*) }
meth *cardPay*(*Card c*) {
pre: /** the card is valid **/
valid(*c*,*sale.total*) /** authorized by bank **/
post:
$\wedge$ (*sale.pay*$'$ = *CardPayment.New*(*c*/*card*)
; *store.sales.add*(*sale*))
$\wedge$ $\forall$*l* $\in$ *sale.lines*,$\forall$*p* $\in$ *store.cat* : (
if *p.barcode* = *l.barcode* **then**
p.amount$'$ = *p.amount* $-$ *l.quantity*) }
}

UC2: Order products. This use case is similar to use case **UC1**. It starts with *startOrder* and then a number of times *orderItem* followed by *makeOrder*. We only specify the functionality of the use case operations, and omit the declaration of the related classes which are obvious. We use *OrderDesk* to denote the name of the use case handler class.

Use Case UC2: Order products

class *OrderDesk*{
attr *Store store*, *Order order*, *Supplier supplier*, *LineItem line*
meth *startOrder*() {
pre: *true*
post: /** a new *order* is created and its *lines*
initialized to empty **/
$\wedge$ *order*$'$ = *Order.New*() }
meth *orderItem*(*long c*,*double q*) {
pre: /**the input barcode exists in the catalog**/
supplier.cat.find(*c*) $\neq$ *null*
post:
$\wedge$ *line*$'$ = *OrderLine.New*(*c*/*identifier*,*q*/*quantity*)
$\wedge$ *order.lines.add*(*line*$'$) }
meth *makeOrder*(*order*) {
pre: *true* /** validity of order is checked
by supplier **/
post:
$\wedge$ *store.orders.add*(*order*)
$\wedge$ *store.supplier.receiveOrder*(*order*) }
}

UC3: Manage inventory. This use case carries out changes to the inventory items. Here we only specify the operations for changing the price of an item and adding a new item. Also, the protocol of this use case allows any sequence of invocations of these operations and is thus omitted.

Use Case UC3: Managing inventory items

class *InventoryDesk*{
attr *Store store*
meth *changePrice*(*long code*,*double newPrice*) {
pre: *store.cat.find*(*code*) $\neq$ *null*
post:
$\wedge$ $\forall$*p* $\in$ *cat* : (**if** *p.barcode* = *code* **then**
p.price$'$ = *newPrice*) }
meth *addItem*(*long code*,*amt*,*double prc*) {
pre: *valid*(*code*) /** not defined here **/
post:
$\wedge$ *store.cat.add*(*Item.New*(*code,amt,prc*/
barcode,amount,price) }
}

In a fully developed POS there are many other use cases: order products from the supplier, managing inventory, including changing the amount of an item, changing the price of a product, and adding a new product, and deleting an old product; not to say producing many different reports. However, the selection above illustrates the approach.

3.2. Discussion

With an informal description, we are not able to describe the functionality of a use case completely and clearly. Yet, it would be too complex to describe both interactions and functionalities in a single notation. rCOS gives a clear separation of these two aspects, and we can change one aspect without the need to change the other.

A complete (as complete as possible) and precise domain specification is crucial to identify the classes and their attributes and associations needed for the use cases. The specification of the functionality also determines later in the design how objects should interact with each other to realize the specified use cases. Therefore, without a full specification of the use case interaction protocol, the functionality, the classes and their attributes and associations, it would be difficult to enter the design phase.

4. Design by Refinement

This section illustrates how refinement rules in rCOS are effectively used to work out a correct design for POS. It is effective because rCOS has big-step refinement rules (design patterns), and they map directly to high level programming language structures, such as those implemented in Java. We only refine a few operations of use case **UC1** in order to show the approach.

Operation *startSale*(). The specification says to get the date, *clock.date* and create a new *sale*, and the expert pattern allows to delegate responsibility for getting the date to the clock and responsibility for creating the new *sale* to the class *Sale*:

$$\begin{array}{ll} \mathit{CashDesk}:: & \mathit{startSale}()\ \{\mathit{Date}\ d := \mathit{clock.date}();\\ & \qquad \mathit{sale} := \mathit{Sale.New}(d)\}\\ \mathit{Clock}:: & \mathit{date}()\{\mathit{return} := \mathit{date}\} \end{array}$$

In Java, sets are implemented by classes that implement the interface *Collection*. The constructor of the set class initializes the instance as an empty set. An object is always created by the constructor of the class and this is a special case of the expert pattern. According to the specification, the constructor *Sale*() of *Sale* is:

$$\begin{array}{ll} \mathit{Sale}:: & \mathit{Sale}(\mathit{Date}\ d)\{\mathit{complete} := \mathit{false};\\ & \mathit{lines} := \mathit{set}(\mathit{LineItems}).\mathit{New}(d);\\ & \mathit{total} := 0; \mathit{date} := d; \mathit{complete} := \mathit{false}\} \end{array}$$

Operation *enterItem*(). We first introduce a method *know*(*long code*), for checking the precondition $\mathit{store.catalog.find}(\mathit{code}) \neq \mathit{null}$. Following the expert pattern, we introduce methods in the classes in the navigation path.

$$\begin{array}{ll} \mathit{CashDesk}:: & \mathit{bool\ know}(\mathit{long\ code})\ \{\\ & \mathit{return} := \mathit{store.know}(\mathit{code})\}\\ \mathit{Store}:: & \mathit{Set}(\mathit{Item})\ \mathit{cat};\\ & \mathit{bool\ know}(\mathit{long\ code})\{\mathit{cat.know}(\mathit{code})\}\\ \mathit{Set}(\mathit{Item}):: & \mathit{bool\ know}(\mathit{long\ code})\{\\ & \mathit{return} := \mathit{find}(\mathit{code}) \neq \mathit{null}\} \end{array}$$

We omit further refinement and coding of the method *find*() in the set class.

For the postcondition, we directly refine each conjunct in the specification following the expert pattern:

$$\begin{array}{ll} \mathit{CashDesk}:: & \mathit{enterItem}(\mathit{long\ code}, \mathit{long\ qty})\{\\ & \quad /^{**}\ \text{combine pre and postconditions}\ ^{**}/\\ & \quad \textbf{if}\ \mathit{know}(\mathit{code})\ \textbf{then}\ \mathit{makeLine}(\mathit{code}, \mathit{qty})\\ & \quad \textbf{else throw}\ \mathit{Unknown(code)}\\ & \}\\ & \mathit{makeLine}(\mathit{long\ code}, \mathit{long\ qty})\ \{\\ & \quad \mathit{line} := \mathit{LineItem.New}(\mathit{code}, \mathit{qty});\\ & \quad \mathit{line.subtotal}(\mathit{store.getPrice}(\mathit{code}), \mathit{qty});\\ & \quad \mathit{sale.addLine}(\mathit{LineItem\ line})\\ & \}\\ \mathit{Store}:: & \mathit{double\ getPrice}(\mathit{long\ code})\\ & \quad \{\mathit{cat.getPrice}(\mathit{code})\}\\ \mathit{set}(\mathit{Item}):: & \mathit{double\ getPrice}(\mathit{long\ code})\\ & \quad \{\mathit{return} := \mathit{find}(\mathit{code}).\mathit{price}\}\\ \mathit{Sale}:: & \mathit{set}(\mathit{LineItem})\ \mathit{lines};\\ & \quad \mathit{addLine}(\mathit{LineItem}\ l)\{\mathit{lines.add}(l)\}\\ \mathit{LineItem}:: & \mathit{LineItem}(\mathit{long\ code}, \mathit{long\ qty})\{\mathit{barcode} := \mathit{code};\\ & \quad \mathit{quantity} := \mathit{qty}\}\\ & \mathit{subtotal}(\mathit{double\ price}, \mathit{long\ qty})\\ & \quad \{\mathit{subtotal} := \mathit{qty} \times \mathit{price}\} \end{array}$$

We leave the design for exception handling unspecified. Any decision is formally a refinement of the original specification.

The *CashDesk* can also get the price first and then pass it as a parameter to the constructor of *LineItem*, but then the constructor has to set the subtotal of the line as well. Further refactoring can introduce more methods to a class so that method calls does not occur in method parameters. For example, we introduce in *makeLine*() the command $\mathit{double}\ p := \mathit{store.getPrice}(\mathit{code})$ and then pass p as a parameter to the constructor of *LineItem*. Correct refactoring is also formalized as refinement in rCOS.

Operation *cashPay*(). We consider the last part in the postcondition that updates the inventory:

$$\forall l \in \mathit{sale.lines}, \forall p \in \mathit{store.cat} : (\textbf{if}\ p.\mathit{barcode} = l.\mathit{barcode}\ \textbf{then}\ p.\mathit{amount}' = p.\mathit{amount} - l.\mathit{quantity})$$

We give a name *updateInventory*() for the method that the above formula specifies, and introduce the following methods to realize this functionality:

$$\begin{array}{ll} \mathit{CashDesk}:: & \mathit{updateInventory}()\ \{\forall l \in \mathit{sale.line} :\\ & \mathit{store.updateInventory}(l.\mathit{barcode}, l.\mathit{quantity})\}\\ \mathit{Store}:: & \mathit{updateInventory}(\mathit{long\ code}, \mathit{double\ qty})\{\\ & \mathit{cat.updateInventory}(\mathit{code}, \mathit{qty})\}\\ \mathit{set}(\mathit{Item})\ s:: & \mathit{updateInventory}(\mathit{long\ code}, \mathit{double\ qty})\{\\ & \forall p \in s : (\textbf{if}\ p.\mathit{barcode} = l.\mathit{barcode}\\ & \textbf{then}\ p.\mathit{amount}' = p.\mathit{amount} - l.\mathit{quantity})\} \end{array}$$

A Java implementation of $\forall o \in s$: *statement* for s being a set of type $\mathit{set}(T)$ is the pattern:

$$\text{Iterator}\ i := s.\mathit{iterator}(); \textbf{while}\ \ i.\mathit{hasNext}()\{\mathit{statement}\}$$

Applying this pattern to the above refinement specification of *updateInventory*(), we obtain an implementation. This shows the advantage of the combination of rCOS refinement rules and advanced features and libraries implemented in modern languages.

Discussion. The expert pattern is a work horse for localizing design of functionalities to the classes involved. After the design of the functionalities of classes and interactions among them, encapsulation on attributes can be syntactically applied.

The expert pattern is a promising candidate for automated transformations that generate a detailed design, only leaving the specification of operations on datatypes that do not require much inter-object communication to be coded by a programmer.

5. A Component-Based Architecture

We first refine the OO design into a logical model of a component-based architecture. We then discuss the design of different interaction mechanisms that can implement the interfaces between the components in a distributed setting. Finally, we discuss the design of the GUI and hardware components. The aim is to show the separation of these different concerns.

5.1. Application components

The considerations on what should be designed into one component include: 1) the provided operations in a use case should be handled in the same component, 2) use cases for realizing business processes that may be carried out in different physical places are organized in different elementary components, 3) a decomposition of the OO model to a component-based model should lower the coupling among different objects so that less related functionalities are performed by different components, and 4) permanent objects of the OO model are allocated to the components whose functionality depends intimately on these objects.

The component decomposition requires specification of a *system invariant* property of the permanent objects and their relations. The system invariant has to be established when the system is set up. For the POS system, we assume that all cash desks in a retail store share the same *store* object that contains the *catalog*, the *logged sales*, and records of *orders* made and their *deliveries* received. Let *UC* be the names of the use case handler classes that are associated with the *Store* class. We need the following invariant

$$\forall H_1, H_2 \in UC, \forall H_1\ d_1, H_2\ d_2 : (d_1 \neq null \wedge d_2 \neq inull \Rightarrow d_1.store = d_2.store \neq null)$$

According to the above considerations, we organize the OO model in Section 3 into two components.

SalesHandler This component does not contain any permanent object, except for the cash desk instances that are specified as

component *SalesHandler*{
interface *CheckOutIf*{
/** method signatures of UC1**/};
protocol {
/**trace expression of UC1**/};
class *CashDesk* **implements** *CheckOut*;
required interface *CashDeskIf* {*store.know*(),
store.getPrice(),*store.updateInventory*()}
required interface *ClockIf* {*Clock.date*()}
required interface *BankIf*{*Bank.valid*()}
}

Notice that we omit the parameters and the protocol of the required interface that is needed when the component is wrapped as a black box to be used by a third party. A protocol for the required interface can be derived from the provided interface protocol and the implementation.

Inventory This component consists of the use case handler objects for use cases **UC2** and **UC3**, the store object and the data contained in it: the catalog items, the logged sales with their payments, the orders and deliveries. Its interface is the union of the interfaces *OrderIf* providing the methods of **UC2** and implemented by class *OrderDesk*, *ManageIf* providing the methods of use case **UC3**, and *CashDeskIf* that provides to component *SalesHandler* the methods *store.know*(), *store.getPrice*(), and *store.updateInventory*(). The last two interfaces can be implemented by a class that extends the class *InventoryDesk* with *store.know*() and *store.getPrice*().

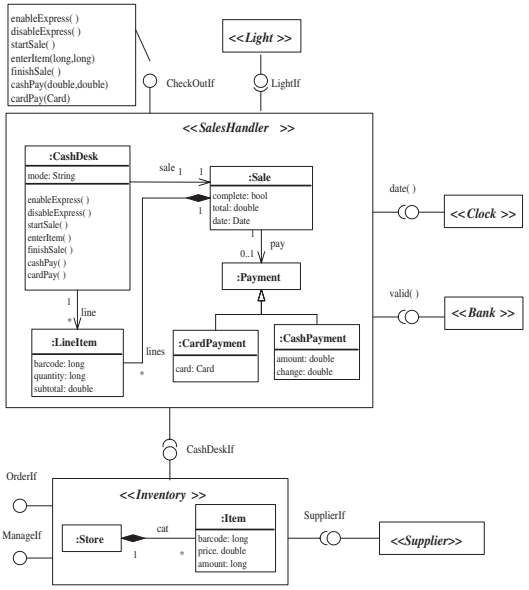

Figure 3. The two components of POS

This component requires the method *order*() from the remote product supplier component *supplier*.

We do not have the space to write the full specification of these components, but they are shown in Fig. 3.

5.2. A further decomposition

We can further decompose the inventory component into two components, *Application* and *Data*. *Application* consists of the use case handlers and provides the interfaces that the overall inventory component provides, but it requires interface of *Data*. *Data* consists of the *store* object and the data contained in it and provides an interface, we call it *StoreIf*, to the application to receive the invocations that are made to the *store*. This decomposition is shown in Fig. 4.

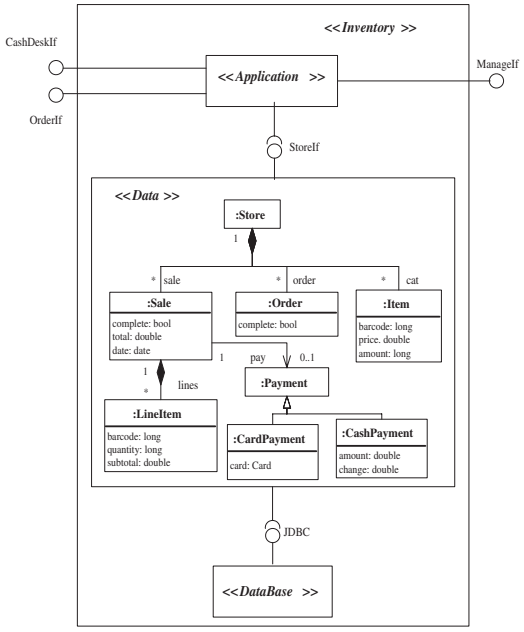

Figure 4. Inventory decomposition

The component *Data* can be further decomposed into components for different kinds of data, such as *SaleData* and *OrderData*. Also, *Data* can be further decomposed into a component *DataRepresentation* and a *Database* component, to make the *Inventory* component a classical three-layer-architecture. All the data are stored in the Database and the component *DataRepresentation* is the data representation, that provides the interface *StoreIf* to the application layer, interactions with the database can be made through JDBC.

5.3. Inter-component interaction mechanisms

So far the semantics of the interfaces between the components is still OO interfaces and thus interactions are supposed to be made via local object method invocation. This works if there is only one *SalesHandler* instance and all the components share a central memory. Now we assume there are more than one *SalesHandler* instance each having its own clock (all clocks are assumed to be synchronized) and only one *Inventory* instance. We must change some of the OO interfaces by introducing *connectors*, for example:

- We keep the interface *StoreIf* between the application layer and the data representation layer in *Inventory* as an OO interface. The interactions between *DataRepresentation* and the *Database* interact through JDBC.
- As all *SalesHandler* instances share the same inventory, we can introduce a connector by which the instances get product information or request the inventory to update a product by passing a product code. This can be implemented asynchronously using an event channel.
- The interaction between the *SalesHandler* instances and the bank or the product supplier can be made via RMI or CORBA.

5.4. GUI components

In our approach, we keep the application design independent of the GUI. The GUI design is only concerned with linking methods of GUI objects to the interface methods of the application components. It delegates a requested operation and gets the information to display on the GUI. So in general, the application components do not call methods of the GUI objects. However, if there is a need to pass a signal to the GUI it may be implemented by an observer pattern - essentially a call-back functionality. This requires that all information that are to be displayed on the GUI reside in the application components and corresponding interface operations are provided by the application components in the GUI components. Then existing GUI builders can be used.

5.5. Controllers of hardware devices

Each *SalesHandler* instance is connected to a bar code scanner, a card reader, a light, a cash box, and a printer. The hardware controllers also communicate with both application interfaces and the GUI objects. For example, when the cashier press the *startSale* button at his cash desk, the corresponding *SalesHandler* instance should react create a sale and the printer controller should also react to start to print the header of the receipt. The main communication can be done by using events which are sent through event channels. An obvious solution is that each *SalesHandler* has its own event channel, called *checkOutChannel*.

This channel is used by the *CheckOut* instance to enable communication between all device controllers, such as a *LightDisplayController*, a *CardReaderController* and the GUIs. The component, the device controllers and the GUI components have to register at their *checkOutChannel*, event handlers have to be implemented and a message middleware, such as JMS, must call the event handlers. All the channels can be organized as a component called an *EventBus*.

Note that a controller is active and thus is a *process component* in rCOS. Its interface protocol is a set of traces of incoming and outgoing events. On the other hand a *service component*, such as *Inventory*, has its provided and required interfaces with their own protocols.

After all the components discussed in the previous subsections are designed and coded, the system is ready for deployment, that we leave out of this paper.

5.6. Discussion

The component-based design in this section shows that an OO design model is very useful for identification of components and the interfaces between them. We can also see the clear separation of the concerns of functionality, interaction mechanisms, GUI and hardware controllers. There are very mature techniques for designing interaction mechanisms and GUIs. There are existing implementations of interaction mechanisms and middlewares, such as RMI, CORBA and JMS, that are ready for use. The design of the controllers and their communications with the GUI and the application components are carried out in a pure event based model. Automated tools, such as FDR, for verification and analysis are likely to be feasible for this design.

6. Verification and Analysis

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 (not shown in this paper) 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.

In the whole CoCoME POS case study, we use 20 pages to carry out the full design of seven use cases by using rCOS refinement rules. 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 [2] and carried out runtime checking and testing. In what follows, we show the taste of checking of JML models.

Runtime checking and testing in JML. 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*.

```
/*@ public normal_behaviour
  @   requires (\exists Object o; theStore.theProductList.contains(o);
  @                ((Product)o).theBarcode.equals(code));
  @   assignable theLine, theSale;
  @   ensures    theLine != \old(theLine) &&
  @                theLine.theBarcode.equals(code) &&
  @                theLine.theQuantity == quantity &&
  @                (\exists Object o; theStore.theProductList.contains(o);
  @                ((Product)o).theBarcode.equals(code) ==>
  @                theLine.theTotal == ((Product)o).thePrice * quantity) &&
  @                theSale.theLines.size() == (\old(theSale.theLines.size()) + 1) &&
  @                theSale.theLines.contains(theLine);
  @ 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;
```

Figure 5. Refined specification of *enterItem*.

For example, the design of *enterItem*() given in Section 4 is translated to the *.jml* file shown in Fig. 5. Notice that the text in the dotted rectangle gives the specification of the exception that was left unspecified in Section 4. If the same testing was taken against a *.jml* file containing the functionality specification of *enterItem*() given in Section 3, there would be a *NormalPostconditionError* reported if the input *code* does not exist. This indicates that the implementation does not handle the input that falsifies the precondition. We now modify the implementation to the code given in the left side of Fig. 6. An exception will be thrown if the variable t is *false*, which represents the nonexistence of the bar

code in the catalog. However, with this implementation, an *InvariantError* will be reported. The unsatisfied *invariant* is given in Fig. 6, asserting that each bar code of a sale's line item must have a product with that code in the catalog. Debugging is required to make sure that the code must be checked before a line is created and added to the sale. The corrected code is shown on the right of Fig. 6.

Testing is not sufficient for correctness. Therefore, it is also desirable to carry out static analysis, for instance with ESC/Java [3].

```
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();
}
```

```
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();
}
```

```
/*@ public instance invariant ((theSale != null && theSale.theLines != null) ==>
  @          (\forall Object o; theSale.theLines.contains(o);
  @            (\exists Object p; theStore.theProductList.contains(p);
  @               ((Product)p).theBarcode.equals(((LineItem)o).theBarcode))));
  @*/
```

Figure 6. Improved code of *enterItem*.

Checking interactions among components. The design of the interactions among the controllers, the GUI and the application components is in fact the design of a concurrent system. In the POS case study, we have given a CSP model of the interactions and used FDR to check deadlock freedom, and the refinement of the use case protocols by the protocols that the system offers to the clients.

7. Conclusions

We have presented different kinds of models that are used in a component-based development process and the relations among them. We emphasize on the separation of concerns and views of the system in each development stage. The whole system, including the application software, the GUI and controllers of hardware devices, is estimated to have about 130 classes of total of 30k lines of code. Without the separation of the concerns, the development would not be quite feasible.

Another point that we would like to make is that correctness preserving design by refinement is important as it would not be possible to verify the code at the end of the development. Our experience is that the use case protocols are usually simple and does not need much time to construct a model for them. However, making the functional specification of the use cases are both critical and time consuming. Also, this task is infeasible for a programmer without experience in formal specification. One experienced formal rCOS expert spent one day working out the specification of UC1, and other six use cases took a 4-man weeks of people who are PhD students, supervised by the experienced person. The OO design of UC1 took the rCOS expert less than one hour with writing, but the design of the other use cases took another 4-student weeks. One can see that a transformation tool can automated the expert pattern and data encapsulation. After the design, coding does not take much time, and in fact most of the code can be automatically generated.

The component architecture design was not time consuming, and can be aided by an automated transformation tool. Existing interaction mechanisms, such as RMI, CORBA and JMS can be used for implementing the interactions among components. However, integrating the interfaces provided to clients with the design of the GUI and controllers of the devices is not an easy task. A lot of coding is needed to glues the interfaces of GUI and controllers with those of the application components, and this can be difficult to formalize. The CSP model of the interactions among these components took a week of a postdoctoral research fellow. A special point here is that until we are clear about the provided interfaces of component-based architecture of the application components, it would be difficult to work out the GUI and the controllers of the devices.

Our conclusion is that rCOS provides a beneficial integration of a number of formal theories. It is effective in specification the functionality, OO refinement and component-based architecture design. It supports event-based methods for the design of embedded controllers, but is not better or worse than the existing ones.

Out future work includes development of automated transformations for OO refinement and component-based architecture design from an OO design, and a concrete syntax for rCOS to have too support.

References

[1] D. Bjørner. *Software Engineering, vol. 3: Domains, Requirements and Software Design*. Texts in Theoretical Computer Science. Springer, 2006.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.M. Leino, and E. Poll. An overview of JML tools and applications. *Intl. Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[3] P. Chalin, J.R. Kiniry, G.T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *LNCS*, pages 342–363. Springer, 2006.

[4] K.M. Chandy and J. Misra. *Parallel Program Design: a Foundation*. Addison-Wesley, 1988.

[5] X. Chen, J. He, Z. Liu, and N. Zhan. A model of component-based programming. Technical Report 350, UNU-IIST, P.O. Box 3058, Macao SAR, China, 2006. http://www.iist.unu.edu, Accepted by FSEN'07.

[6] X. Chen, Z. Liu, and V. Mencl. Separation of concerns and consistent integration in requirements modelling. In J. van Leeuwen et al, editor, *SOFSEM'2007: Proc. Current Trends in Theory and Practice of Computer Science*, volume 4362 of *LNCS*. Springer, 2007.

[7] Common component modelling example (CoCoME). http://agrausch.informatik.uni-kl.de/CoCoME, 2007.

[8] Object Management Group. Unified Modeling Language: Superstructure, version 2.0, final adopted specification. http://www.omg.org/uml/, 2005.

[9] J. He, X. Li, and Z. Liu. rCOS: A refinement calculus for object systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.

[10] J. He, Z. Liu, and X. Li. A theory of reactive components. *Electronic Notes on Theoretical Computer Science*, 160:173–195, 2006.

[11] C.A.R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[12] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall Intl., 2nd edition, 2001.

[14] X. Liu, Z. Liu, and L. Zhao. Object-oriented structure refinement - a graph transformational approach. In *Intl. Workshop on Refinement (REFINE'06), ENTCS*, 2006. Extended version submitted for journal publication.

[15] L. Mathiassen, A. Munk-Madsen, P.A. Nielsen, and J. Stage. *Object-oriented Analysis and Design*. MARKO Publishing, Aalborg, 2000.

[16] B. Meyer, I. Ciupa, A. Leitner, and L. Liu. Automatic testing of object-oriented software. In *Proc. Current Trends in Theory and Practice of Computer Science, LNCS*. Springer, 2007.

[17] E-R. Olderog and H. Wehrheim. Specification and (property) inheritance in CSP-OZ. *Science of Computer Programming*, 55:227–257, 2005.

[18] F. Plasil and S. Visnosky. Behavior protocols for software components. *IEEE Trans. Software Eng.*, 28(11):1056–1070, 2002.

[19] A.W. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[20] UNU-IIST. 1st Asian Working Conference on Verified Software. http://www.iist.unu.edu/www/workshop/AWCVS2006/.