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

1. Introduction

The challenge of increasing complexity of software applications requires design methods that support separation of concerns and incremental development. In such a method, artifacts are modeled and analyzed using different modeling notations. For this, OO and component-based techniques are widely used. The different views of a system that are concerned include the *static structure*, *dynamic behavior*, the *interactions between objects* or *components*, *static data functionalities*. In UML, 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 can only be used in an informal or semiformal design. For the assurance of correctness, we need to incorporate formal specification, refinement, verification and analysis into the development process.

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

1. allow us to specify the models of different views of the system in 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, of designs of object and component software systems [7, 13, 8, 4]. It provides a two dimensional refinement framework, that is *consistent increments* to the models for the multi views of different aspects in the horizontal dimension and refinement relation between models at different levels of abstraction in the vertical dimension.

In this paper, we report our experience of using rCOS in the design of the point of sale system (POS), that can be used in a retail store (or a chain of stores). We show in a model-based development process

1. how different aspects, including static structure, interactions, and data functionality, of the system can be formally specified and analyzed in rCOS,
2. how different aspects can be designed by effectively using refinement rules proved in rCOS,
3. how different tools for verification and analysis of properties can be incorporated into the rCOS framework.

Overview. Section 2 gives an introduction rCOS. Section 3 provides an informal description of POS. We give the rCOS specification of the requirements in Section 4. In Section 5 we apply the refinement rules to construct an OO design of the system. In Section 6, 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 7, we report the results of application of tools for verification and analysis. Finally, Section 8 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) [9].

The essential idea of UTP is that any program can be modeled a binary relation on the *observables* of the program. For a sequence imperative program, we are interested to observe the initial values of the variables before the execution and their final values after the execution, and whether the execution terminates. The observables of such a program include the variables, say x representing the values of the program variables before the

*This work is partially supported by the project HighQSoftD funded by Macao Science and Technology Development Fund, and the NSF project 60573085 and and 863 of China 2006AA01Z165.

execution, the variables x' representing the values of the program variables after the execution, and the boolean variables ok and ok' for modeling the activation and termination of of the execution. 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 $pre(x) \wedge ok \Rightarrow ok' \wedge post(x,x')$, asserting that if the program is activated from a well-defined state (i.e. its preceding program terminated in that state) that satisfies the *precondition* $pre(x)$ then the execution of the program will *terminate* (i.e. $ok' = true$) in a state such that the news values are related with the old values by the *postcondition* $post$. For example, an assignment $x := x+y$ can be 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 the standard programming constructs like sequential composition, choice, and iteration. The sequential composition $D_1;D_2$ 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 are used to record communication traces and communication readiness can be expressed by additional *guard* predicates. The healthiness conditions of these *guarded designs* gave the semantics basics in rCOS for the integration of the specification of the static functionality by simple designs, interaction protocols by traces, and the dynamic behavior by guarded designs that defines the semantics of a state machines [8, 4]. The *refinement* relation between programs is defined as logical implication. rCOS extends UTP to formalize the concepts of object oriented programming: class, object references, method invocation, subtyping and polymorphism [7].

Like Java, an OO program in rCOS is specified as $Classes \bullet main$ consisting of a number *class declarations* and a *main program* [7]. A class in the class declarations 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 $main()$. We assume that the main method only access and modified objects of the classes in the class declarations via invoking methods declared in those methods.

2.1. Types and notations

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 POS, we assume the data types of $V ::= long \mid double \mid char \mid string \mid bool$. Let CN be an infinite of identifiers, called *class names*. We define the following type system, where C ranges over CN

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

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 $addAll()$ on a set and a bag with their standard semantics. 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 $s.addAll()$ is the sum of all elements of s, which is assumed to a set of numbers. We allow to 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, $s.find(ID\ id)$ denotes the function that returns the element whose identifier equals id if there is one, it returns $null$ otherwise. Notice that Java provides the implementations of these types via the *Collection* interfaces. 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, evaluation of expressions does not change the state of the system, and thus the Java expression $C.New()$ 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 with the initial values or objects declared in C. If no initial value is declared it will be $null$. However, in the specification of POS, 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. Similarly, we use $x' = m()$ for a method with a return parameter *return* of the same type of x.

2.2. rCOS model of requirements

OO requirements capture in general starts with identification *business processes* described as *use cases*. The use case specification includes the four views. One view is the *interactions* between the external environment, modeled as *actors*, and the system. The interactions are described as a protocol in which the actors 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.1), called a *use case sequence diagram*.

While a sequence diagram focuses on the interactions between the actors and the systems, the *dynamic behavior* of the use case is model by a *guarded state transition system*, that can be depicted 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 [8, 4, 17]. This view is used for verification deadlock and livelock freedom by model checking state reachability.

Another important view is the static *functionality view* of the system. The requirements specifications should precisely specify what each use case operation should do when being invoked. That is what state change it should make in terms of what new objects are to created, what old objects should be destroyed, what links between which objects are established, and what data attributes of which objects are modified. And what is the precondition for carrying out these changes. 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 controller class. Each method is specified by its signature and its design in the form $m()\{pre \vdash post\}$. The signatures of the methods must be consistent with those used in the interaction and dynamic views. During the 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 controllers forms the *structure view* of the system. It can be depicted by a class diagram or packages of class diagrams (cf. Fig. 2). The consistency and integrated semantics of the different views are studied in [5].

In the example of POS, a *design* $p \vdash R$ for a method in rCOS is written separately as **Pre** p and **Post** R. At the requirement level, we write specification the static functionality of their methods in the following format.

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

where,

- The list of class declarations can be represented as a UML class diagram.
- The initial value of an attribute is optional, and an attribute is assumed to be public and can be further tagged with reserved words *private* and *protected*.
- Each c in the precondition, representing a condition to be checked, is of 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** $le' = e$ **else** $le' = e$ for $c \wedge (le' = e) \vee \neg c \wedge (le' = e)$ and **if** c **then** $le' = e$ for $c \wedge (le' = e) \vee \neg c \wedge$ **skip**. 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$^+$ [10], UNITY [3] and Java on the above format.

2.3. Object-oriented refinement in rCOS

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 regarding to the global variables is preserved. It is an extension to the notion of data refinement in imperative programming, with a semantics model dealing with object references. In such a refinement, all non-public attributes of the objects are treated as local (or internal) variables [7].
2. Refinement of the class declaration section *Classes*. $Classes_1$ 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 functional services as *Classes*.
3. Refinement of a method of a class in *Classes*. $Classes_1$ refines *Classes* if the public class names in *Classes* are all in $Classes_1$ and for each public methods of each public class in *Classes* there is a refined method in the corresponding class of $Classes_1$.

The first step in an OO design is to refine methods of use case use case handler classes, without changing the interaction protocols of the use cases with the external actors. There are mainly three kinds refinement: *Delegation of functionality* (or *responsibilities*), *attribute encapsulation*, and *class decomposition*. Very interesting results on completeness of the refinement calculus are available in [13].

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 direct accesses and/or modifies attribute x of C_1. Then, if all other variables in command c are accessible in C_1, we have $Classes \sqsubseteq Classes_1$, where $Classes_1$ is obtained from $Classes_1$ 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 patterns of responsibility assignment*.

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

Theorem 1 (Expert Pattern) *Given a list of class declarations Classes and its navigation paths* $le := r_1.....r_f.x, \quad \{a_{11}.....a_{1k_1}.x_1, ..., a_{\ell 1}.....a_{\ell k_\ell}.x_\ell\}$, *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*

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

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.

If the paths $\{a_{11}.....a_{1k_1}.x_1, ..., a_{\ell 1}.....a_{\ell k_\ell}.x_\ell\}$ *has 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 require to 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. When we write the specifications of the use case operations, we need to directly refer to attributes of the classes that are associated with the use case controllers. Therefore, those attributes are required to be public. After designing the interactions by the 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 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 [7, 13].

One important notice 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 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 [13].

2.4. Component-based modeling in rCOS

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

A closed component has a provide 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 be *used* in building and maintaining software systems. It contains the information about the viewpoints among, for example *functionality*, *behavior*, *protocols*, *safety*, *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 POS, we only consider *functionality*, *behavior*, *protocols*, *safety*, *real-time* and *communication mechanisms*.

In an *open component*, a provided method may call methods provided by another component. An open component *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 specification 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 a process, we cannot have separate contracts for its provided interface and required interface.

Compositions for *disjoint union* of components and *plugging* components together, for *gluing components* by processes are defined, and their closure properties and the algebraic properties of these compositions are studied in [4]. Note that an interface can be the union of a number of separately specified interfaces.

The contracts together with the interfaces of a component provide a black-box specification of the component. The contracts in rCOS also define the unified semantic model of implementation of interfaces in different programming languages, and thus support 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, as well as it supports independent development of components.

2.5. Related formalisms

Eiffel [14] first introduced the idea of design by contract into 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 [12] has recently become a popular language for modeling and analysis of 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 UML like modeling.

In Fractal [16], behavior protocols are used to specify the 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. 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 [15] in that a multi-notational modeling language is used for combining 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 approach has the virtue of well-developed underlying frameworks and tools.

3. The Point of Sale System

The point of sale system (POS) was originally used as a running example in Larman's book [11] to study concepts of OO systems design. An extended version is now being used as the case study in the Common Component Modeling Example (CoCoME) [1], and identified as a pilot project at the Asian Working Conference on Verified Software [19].

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 system deals with ordering goods and generates various reports for management purposes. It 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 and has a number of checkout points. The whole system includes hardware components such as computers, bar code scanners, card readers, printers, and 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. Requirements

The requirements are captured by identifying the use cases, which describe the business processes of the application domain, and the constraints on them.

3.2. Use cases of POS

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

This use case can perform either express checkout process *for customers with only a few items to purchase, or a* normal checkout process. *The main courses of interactions between the actors and the system is described as follows.*

1. *The* cashier *sets the checkout mode to express check out or for normal check out. The system then sets the displaylight to* green *or* yellow *accordingly.*
2. *When a* customer *comes to the* checkout point *with their* items *to purchase, the cashier indicate the system to handle a new* sale.
3. *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.*

   In express checkout mode, only a limited number of items are allowed to checkout.
4. *At the end of entering the items, the* total *of the sale is calculated. The cashier tells the customer the total and asks her to pay.*
5. *The customer can pay by cash or a credit card:*

   (a) *If by cash, the amount received is entered. The system records the* cash payment *amount and calculates the change.*

   (b) *If by credit card, the card information is entered. The system sends the credit payment to the* bank *for* validation. *The payment can only be made if a positive validation reply is received.*

   The inventory of the sold items is updated and the completed sale is logged in the store *to complete the process.*

There are exceptional courses of interactions. For example, 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. Systems need 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.

There are many other use cases: **UC2**: *Order products*, that orders products from the supplier; **UC3**: *Managing inventory*, including changing the stocked amount of an item, changing the price of a product, and adding/deleting products; **UC4**: *Produce monthly reports on sales* that is to, say, generate a report of all sales in the last 30 days and information of profit and loss; and **UC5**: *Produce stock reports* , that produces reports on stock items.

4. rCOS Specification of POS

Following our modeling principle in [5], we specify the operations of a use case as methods of a designated class, called the *use-case handler class* of the use case.

UC1: Process sale. We first model the interaction protocol that the system offers the actor, here, the Cashier. This is given as the *use case sequence diagram* in Fig. 1, denoted by SD_{uc1}.

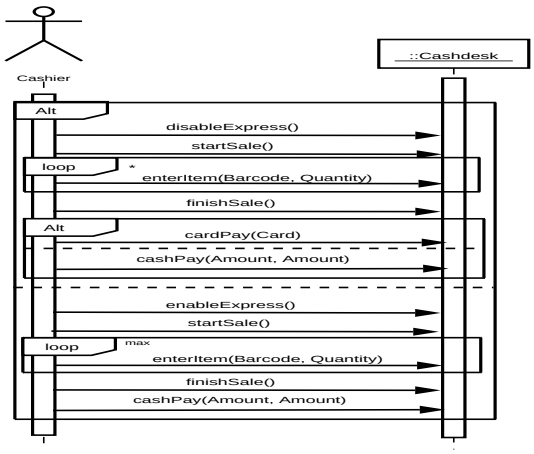

Figure 1. Sequence diagram for UC1

In this figure, *max* is the maximal number of items allowed in express checkout. The set of traces of the diagram is given by the following regular expression.

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

The flow of control of the use case can be given by a state diagram, that we omit here.

We now give the functionality specification of this use case. We show the attributes in the class diagram in Fig.2, and the comments in the specification should help the understanding.

Use Case UC1: Process Sales

Class *Cashdesk*{

Meth *enableExpress*() {
Pre: *true*;
Post: $light.display' = green$ };

Meth *disableExpress*() {
Pre: *true*;
Post: $light.display' = yellow$ }

Meth *startSale*() {
Pre: *true*;
Post: /** a new *sale* is created and its *lines* initialized to empty, and its date correctly recorded **/
$sale' = Sale.New(clock.date/date)$}

Meth *enterItem*(*long c*, *double q*) {
Pre: /**the input barcode *c* is valid **/
$store.cat.find(c) \neq null$
Post: /** a new *line* is created with its *barcode* and *quantity* set to *c* and *q*, **and** **/
$line'$=$LineItem.New(c/barcode, q/quantity)$
/**the *subtotal* of the *line* is set, **and** **/
$\wedge$ $line'.subtotal$=$store.cat.find(c).price \times q$)
/**add *line* to the current *sale* **/
$\wedge$ $sale.lines.add(line')$ }

Meth *finishSale*() {
Pre: *true*;
Post: /** *sale* is set to *complete*, **and** **/
$sale.complete' = true$
/** *sale*'s *total* is calculated **/
$\wedge$ $sale.total'$=$addAll[[l.subtotal|l \in sale.lines]]$ }

Meth *cashPay*(*double a*; *double c*) {
Pre: $a \geq sale.total$ /** amount no less than total **/
Post: /** *Cashpayment* of *sale* is created, **and** **/
$sale.pay' = CashPayment.New($
$a/amount, a - sale.total/change)$
/** the change is returned, **and then** the completed *sale* is logged in *store*, **and** **/
$\wedge$ $c' = a - sale.total$**;**$store.sales.add(sale)$
/** the inventory is updated **/
$\wedge \forall l \in sale.lines, \forall p \in store.cat$: (
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 the bank **/;
Post: the *CardPayment* of the *sale* is created, **and then** the completed *sale* is logged in *store*, **and** **/
$sale.pay' = CardPayment.New(c/card)$**;**
$store.sales.add(sale)$
/** the inventory is updated **/
$\wedge \forall l \in sale.lines, \forall p \in store.cat$: (
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 begins with *startOrder* and then a number of times *orderItem* followed by *makeOrder*. We only specify the functionality of the use case operations, and the related classes, as it is important for the later design. *OrderDesk* denotes the use case handler class.

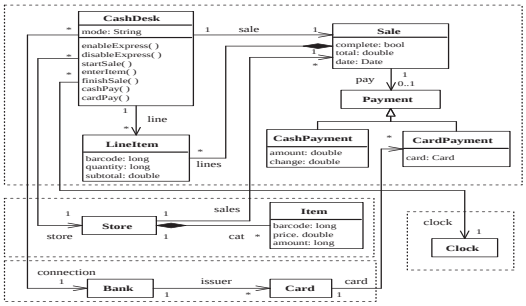

Figure 2. Class Diagram for UC1

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 **/
$order' = Order.New()$ }

Meth *orderItem*(*long c*, *double q*) {
Pre: /**the input barcode exists in the catalog**/
$supplier.cat.find(c) \neq null$;
Post: /** a new *line* is created with its $identifier$ and *quantity* set to *c* and *q*, **and** **/
$line' = OrderLine.New(c/identifier, q/quantity)$
/**add *line* to the current *order* **/
$\wedge$ $order.lines.add(line')$ }

Meth *makeOrder*(*order*) {
Pre: *true* /** validity of order is checked by supplier **/
Post: /** order is logged in *store* and sent to supplier**/
$store.orders.add(order)$
$\wedge store.supplier.receiveOrder(order)$ } }

Class *Store* {
Attr /**add one attribute to that in UC1**/
Supplier supplier
}

Class *Order* {
Attr *bool complete =false,*
$set(LineItem)$ $lines = empty$
}

Class *LineItem* {
Attr /**take the same as the line item in UC1, though in reality it may not be**/
}

Class *Supplier* {
Meth *receiveOrder*(*Order order*){
/**provided**/}
}

UC3: Manage inventory. This use case carries out changes to the inventory items. Here we only specify the use case operations for changing the price of an item and adding a new item. Other operations, such as updating the amount of an item (as specified in UC1) and deleting an item can be easily specified. Also, the protocol of this use case allows any sequences of invocations of these operations.

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: /** new *order* is created and its *lines* initialized to empty **/
$\forall p \in cat : (\textbf{if } p.barcode = code \textbf{ then } p.price' = newPrice\ \}$
Meth *addItem*(*long code*, *amt*, *double prc*, *string spc*) {
Pre: *valid*(*code*) /**not defined here **/
Post: *store.cat.add*(*Item.New*(*code,amt,prc,spc*/ *barcode,amount,price,specification*) } }

Discussion. With an informal description, we are not able to describe the functionality of a use case completely and clearly. It would be too complex to describe both interactions and functionalities using a single notation. Our specification gives a clear separation of these two aspects, and we can change one aspect without the need to change the other. Indeed, we have gone through a number of changes to both the protocol and the functionality separately, the functionality in particular.

Writing the complete (as complete as possible) and precise functionality specification is crucial to identify the classes, their attributes and associations needed for the realization of the use case. Writing the specification of the methods helps to capture the classes, attributes and associations that are needed to support 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 start a proper design.

Existing UML tools, such as Rational Role and MasterCraft [18] support the production of the sequence diagram and the textual specification of the methods. Later, when specifying a use case, some classes specified earlier are used. However, new attributes of those classes and new classes are often introduced. An integrated specification (or class diagram) should be obtained and it must completely support the specification of the functionality of all the use cases.

5. A design model of POS

This section illustrates how refinement rules in rCOS can be effectively used when we work out a correct design of POS. The effectiveness is due to that the rCOS rules prove big-step refinement rules (design patterns), and their direct mappings 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 effectiveness of the refinement rules.

Operation *startSale*(). The specification requires to get the date, *clock.date*() and create a new *sale*. 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*:

CashDesk :: *startSale*() {*Date d* := *clock.date*(); *sale* := *Sale.New*(*d*)};
Clock :: *date*(){*return* := *date*}

In Java, sets are implemented through a class that *implements* 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 in a special case of the expert pattern. According to the specification, the constructor *Sale*() of *Sale* is given as

Sale :: *Sale*(*Date d*){*complete* := *false*; *lines* := *set*(*LineItems*).*New*(); *total* := 0; *date* := *d*}

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

CashDesk :: *bool know*(*long code*) {*return* := *store.know*(*code*)}
Store :: *Set*(*Item*) *cat*; *bool know*(*long code*){*cat.know*(*code*)}
Set(*Item*) :: *bool know*(*long code*) {*return* := *find*(*code*) ≠ *null*}

where we leave further refinement and coding of method *find*() in the *set* class out of the paper.

For the postcondition, we can directly refine each conjunct in the specification following the expert pattern by the following method definitions:

CashDesk :: *enterItem*(*long code*, *long qty*){
/** combine pre and postconditions **/
if *know*(*code*) **then** *makeLine*(*code*, *qty*)
else throw exception handle ¬*know*(*code*)
}
makeLine(*long code*, *long qty*) {
line := *LineItem.New*(*code*, *qty*);
line.subtotal(*store.getPrice*(*code*), *qty*);
sale.addLine(*LineItem line*)
}
Store :: *double getPrice*(*long code*) {*cat.getPrice*(*code*)}
set(*Item*) :: *double getPrice*(*long code*) {*return* := *find*(*code*).*price*}
Sale :: *set*(*LineItem*) *lines*; *addLine*(*LineItem l*){*lines.add*(*l*)}
LineItem :: *LineItem*(*long code*, *long qty*){*barcode* := *code*; *quantity* := *qty*}
subtotal(*double price*, *long qty*) {*subtotal* := *qty* × *price*}

We leave the requirements on what to do when exceptions occurs unspecified. Any specification would be a refinement of the original specification.

The cash desk 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* on the code can introduce more methods or methods to a class so that method calls do not occur in method parameters. For example, we introduce in *makeLine*() the command *double p* := *store.getPrice*(*code*) and then passing *p* as a parameter to the constructor of *LineItem*. Correct refactoring is also formalized as refinement in rCOS.

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

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

We implement the above formula in a method called *updateInventory*() and introduce the following methods to realize this functionality:

CashDesk ::	*updateInventory*() {$\forall l \in sale.line$: *store.updateInventory*(*l.barcode*, *l.quantity*)}
Store ::	*updateInventory*(*long code*, *long qty*){ *cat.updateInventory*(*code*, *qty*)}
set(*Item*) *s* ::	*updateInventory*(*long code*, *long qty*){ $\forall p \in s$: (**if** *p.barcode* = *l.barcode* **then** $p.amount' = p.amount - l.quantity$)}

We introduce a pattern for a Java implementation of $\forall o \in s$: *statement* for *s* being a set of type *set*(*T*):

Iterator *i* := *s.iterator*(); **while** *i.hasNext*(){*statement*}

Applying this pattern to the above refinement specification, 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. We can see the effectiveness of the expert pattern in localizing design of functionalities to the classes involved in the functionality. After the design of the functionalities of classes and interactions among them, encapsulation on attributes, e.g. turning public attributes private, can be refined.

The expert pattern and some other refinements can be realized by automated transformations that generate a detailed design, only leaving the specifications of functions that do not require much inter-object communication to be coded by a programmer. Some of these specifications, such as the greatest common divisor of two integers and shortest paths of a directed graph, require sophisticated algorithms to be designed or specified before coding.

6. Component-Based Architecture of POS

We first show how to refine the OO design model 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 will discuss the design of the GUI components and the control of the hardware devices. The aim is to show the separation of these different concerns.

6.1. 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 is to 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 tightly depends on these objects.

The component decomposition requires the 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* and *deliveries* made. Let *UC* be the set of names of the use case handler classes that are associated with the *Store* class. We need the following invariant:

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

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

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

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

  Notice that we have omitted the parameters and the protocol of the required interface that is needed when the component is wrapped as a black box and to be used by a third party. The protocol of 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*().

This component requires a 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.

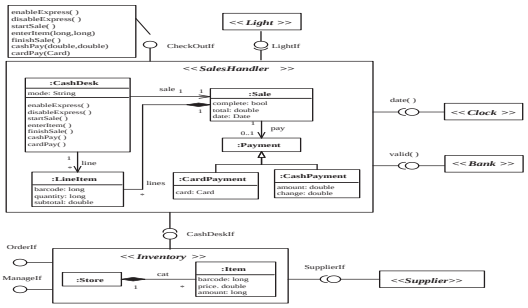

Figure 3. Components of POS

6.2. 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.

The component *Data* can be 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.

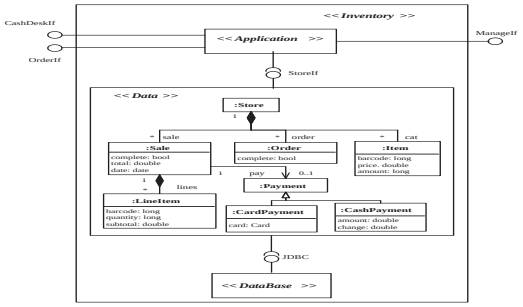

Figure 4. Decomposition of *Inventory*

6.3. Inter-component interaction mechanisms

So far the semantics of the interfaces between the components are 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 and only one *Inventory* instance. We should 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* can be handled, e.g., through JDBC.
- As all the *SalesHandler* instances share the same inventory, we can introduce a connector by which the *SalesHandler* instances get product information or request the inventory to update a product by passing a product code. This can be implemented using an event channel.
- The interaction between the *SalesHandler* instances and the bank or the product supplier can be made via RMI or CORBA.

6.4. GUI components

In our approach, we keep the design of the application independent from the design of the GUI, so that we do not need to change the application. The GUI design is only concerned about how to link methods of GUI objects to the interface methods of the application components to delegate the requested operation and to get the information that are needed to display on the GUI. In general, the application components should not call methods of the GUI objects. Also, no references should be passed between application components and GUI components (so called service-oriented interfaces should be used). This requires that information that is

displayed on the GUI should be available in the application components and corresponding interface operations should be provided by them.

6.5. 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 create a sale and the printer controller should 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 *LightDisplayController*, *CardReaderController* and the GUI. The component, the device controllers and the GUI components have to register at their *checkOutChannel* and event handlers have to be implemented. A message middleware, such as JMS, can be used to call the event handlers. All the channels can be organized as a component called *EventBus*.

We should note that a controllers is active and thus corresponds to a *process component* in rCOS whose 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.

6.6. Discussion

The presented component-based design shows that an OO design model is very useful to identify components and their interfaces. We can also see the clear separation of concerns on functionality, interaction mechanisms, GUI and controllers of hardware devices. There are very mature techniques and existing designs for different interaction mechanisms and GUIs. RMI, CORBA and JMS implement interaction mechanisms and middlewares. The design of the controllers and communication with the GUI and the application components are carried out in an event based model. Automated tools such as FDR can be used for verification and analysis.

7. Verification and Analysis

Various verifications and analysis 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) is 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 can obviously be syntactically checked or ensured with 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 case study, we carried 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 to detect possible mistakes made when applying the rules manually. However, we have translated some of the design into JML [12] and carried out runtime checking and testing which we discuss in the following.

Runtime checking and testing in JML. We translate each rCOS class *C* into two JML files, one is *C.jml* that contains the JML specification translated from the rCOS specification, and the other is a Java source file *C.java* containing code that implements the specification. During translation, the variables used in the rCOS specification are taken as specification-only variables in *C.jml*. They are mapped to program variables in *C.java*.

The JML files can be compiled by the JML Runtime Assertion Checker Compiler (*jmlc*). Then, test cases can be executed to check satisfaction of the specification by the implementation. The automatic unit testing tool of JML (*jmlunit*) can be used for generating unit testing code, which 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 5 is translated and stored in 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 5. If the same testing was taken against a *.jml* file containing the functionality specification of *enterItem*() given in Section 4, 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 missing 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. Testing reveals 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; it is also desirable to carry out static analysis with tools like ESC/Java [2].

```
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 an embedded 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.

8. Conclusions

We have presented the 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 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 we do not need much time to construct a model for them. However, making the functional specification of the use cases is both crucial and time consuming. Also, this is not a task feasible for a software engineer without the experience of using formal specification. An experienced formal rCOS expert spent one and a half day on working out the specification of UC1, and the other six use cases took a 4-man week 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 automate 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 did not take much time and can be aided by automated transformation tools. Existing interaction mechanisms, such as RMI, CORBA and JMS can be used to implement 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 glue these interfaces 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, including learning basic CSP. 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 clear integration of a number of formal theories. It is effective in the specification of 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 automata-based approaches.

Our future work includes development of automated transformations for OO refinement and component-based architecture design from an OO design, a formal syntax for rCOS, and to automate the translation from rCOS to JML so that JML tools can be used.

References

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

[2] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111, pages 342–363, 2006.

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

[4] 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.

[5] X. Chen, Z. Liu, and V. Mencl. Separation of concerns and consistent integration in requirements modelling. In *Proc. Current Trends in Theory and Practice of Computer Science, LNCS*. Springer, 2007.

[6] Yoonsik Cheon and Gary T. Leavens. The JML and JUnit way of unit testing and its implementation. Technical Report 04-02a, Department of Computer Science, Iowa State University, April 2004.

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

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

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

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

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

[12] J.L. Leavens. JML's rich, inherited specification for behavioural subtypes. In *Proc. 8th Intl. Conf. on Formal Engineering Methods (ICFEM06). Lecture Notes in Computer Science Volume 4260*. Springer, 2006.

[13] X. Liu, Z. Liu, and L. Zhao. 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, Published in Proc. International Workshop on Refinement, ENTCS. An extended version is submitted for Journal Publication.

[14] 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.

[15] M. Möller, E-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A case study. In *Proc. Integrated Formal Methods (IFM'04). LNCS Volume 2999*. Springer, 2004.

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

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

[18] Tata Consultancy Services. Mastercraft. http://www.tata-mastercraft.com/.

[19] UNU-IIST. 1st Asian Working Conference on Verified Software, 2006. http://www.iist.unu.edu/www/workshop/AWCVS2006/.