Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 207 (2008) 49–67
www.elsevier.com/locate/entcs

The Verification of rCOS Using Spin[1]

Xiao Yu, Zheng Wang, Geguang Pu,
Dingding Mao and Jing Liu[2]

Software Engineering Institute, East China Normal University, Shanghai, China

Abstract

The rCOS is a relational object-based language with a precise observation-oriented semantics. It can capture key features of object model including subtypes, visibility, inheritance, polymorphism and so on. To analyze the model specified by rCOS, we propose a verification approach to check whether those properties such as the assertion, invariant of class and method contracts hold. The Spin model checker is used in this approach. To enhance the ability of description of concurrency, we extend the original rCOS with parallel structure and synchronization mechanism. The Promela model is constructed from rCOS specification with non-trivial mapping rules. We also present a case study to show how our approach works.

Keywords: Verification, rCOS, Object Model, Spin

1 Introduction

Software development and maintenance are costly endeavors. However, the cost can be reduced if we detect more software defects earlier in the development cycle. To model the behaviors of the software design, especially for the object-oriented programming, researchers proposed some model-based formalisms [15,5,4] for the specifying and capturing of software requirements on the design model. The analysis approaches can be applied to design model to discover the possible inconsistences.

rCOS [8], named after a Refinement Calculus for Object Systems, is also a model for object-oriented programming to focus on a mathematical characterization of object-oriented concepts. It provides a proper semantic basis essential for ensuring the correctness of programs and for developing tool support for formal techniques. rCOS semantics are based on the Hoare and He's Unifying Theories of Programming (UTP) [9] and the its refinement calculus is basically derived from the

[1] The work is partially supported by the NNSF of China (No. 60603033, No. 60673114), STCSM project No.06JC14022 and Qimingxing project No. 07QA14020, 973 project No. 2005CB321904, and 863 project No.2006AA01Z165.

[2] Email: {lesteryu, wangzheng, ggpu, ddmao, jliu}@sei.ecnu.edu.cn

doi:10.1016/j.entcs.2008.03.085

implication between the predicates. The design of intension of rCOS is to specify object-oriented designs by means of capturing the essential object concepts such as subtypes, inheritance, dynamic binding and so on. When the design model is constructed by rCOS, the refinement laws can be applied to add more details or change the relations on the design with preserving the semantics.

Adding notations in program or specification language became practical and important for the property check of the system. JML [3] is a behavioral interface specification language tailored to Java supporting assertion, quantifiers etc. to describe the behaviors of Java program. Eiffel [12] directly integrates the features of specification langauge into it and provides the design by contract techniques to achieve the reliability of the software development. Specsharp [1]is developed my Microsoft for the extension of C sharp language to permit specification and reasoning about programs from easily usable dynamic checking to high-assurance automatic static verification. The design of rCOS follows the idea of mixing specification notations into object programming to support the rigorous software development. The features of rCOS include its precise semantics based on UTP and the supporting of structural and behavioral refinement of object-oriented designs.

To check the specification properties, we provides the verification approach to check whether those properties such as the assertion, invariant of class and method contracts hold. The approach actually is based on reachability analysis and particularly on the use of SPIN model checker [10]. The check procedure is carried out by the following step:

- Construction of the rCOS model in Promela [10] description. The rCOS specification notations are also converted into the corresponding part in Promela.
- Analysis of Promela model to check the properties. It is performed by Spin model checker.
- The error location in the original rCOS model. If errors are founded in Spin, a trace algorithm is applied to find the error location in the rCOS model.

The mapping rules from rCOS and Promela are not trivial and it takes the elaborate efforts to construct the Promela model from rCOS, especially for the concurrency of rCOS model. To capture the concurrent design model, we extend the original rCOS with parallel and synchronization keywords. The use of concurrency will be illustrated by an example later. The contribution of this work is to add the concurrency mechanism into rCOS and provide the verification with rCOS model using Spin. Moreover, we intend to make rCOS become a modelling language for object systems, and provide it with the analysis and verification tool set in the near future. The rCOS parser and rCOS2Spin are under development with this paper.

This paper proceeds as follows. Section 2 gives the overview of rCOS language. Section 3 introduces the approach to converting rCOS model in Promela. Section 4 presents an example to show the verification of rCOS. The last Section gives the conclusion and future work.

2 The Overview of rCOS Specification

rCOS is an object-based modelling language. It is defined with many features such as subtype, visibility, inheritance, type casting, dynamic binding and polymorphism. rCOS uses pre/post conditions on methods and invariants on classes to verify if the program matches the design specification. rCOS can be used to specify object-oriented designs as well as programs and support both structural and behavioral refinement of object-oriented designs. For simplicity, some modern object-oriented language features are not defined in rCOS, including, without limitation, attribute hiding, multiple inheritance, interface implementation, exception handling and garbage collection.

rCOS programming structures are similar to Java language. Thus we do not present all the syntax of rCOS here, and introduce the key features of rCOS for its own. These features are as follows:

- Class definition
- Method definition
- Multi-thread modelling
- Undetermined choice statement
- Predicate expression

2.1 Program structure

In rCOS, a program (object system) contains *Cdecls* and *Main*. *Cdecls* is a finite sequence of class declarations. Every class contains members, methods and invariants. Main is a special method which is not defined in class. It is the entry point of a program. An example of "Hello rCOS" is shown below:

```
class Hello_rCOS {
    invariant (true);

    string s = "Hello rCOS";
    method SayHello (;string result;)
    {
        require(true);
        result = this.s;
        ensure(true);
    }
}
Main()
{
    string s = null;
    Hello_rCOS obj = new Hello_rCOS(;;);
    obj.SayHello(;s;);
    /*Now variable s contains string "Hello rCOS".*/
```

```
}
```

In this example, *Hello_rCOS* is defined as a class with an invariant command, a field and a method. *SayHello* is a method of class *Hello_rCOS*. Keywords *require* and *ensure* are used to represent pre and post condition for which the method call must meet.

In the language definition, the complex data structures such as collection and string are supposed to be encapsulated as predefined libraries. For the tool support of rCOS, those libraries needs to be developed.

2.2 Class definition

The form of Class definition in rCOS is:

```
[modifier] class classname [extends base_classes]
{
    [field_definition]
    [method_definition]
    [invariant_definition]
}
```

where

- A class can be declared as private of public (default). Only public classes can be used in Main.
- Class inheritance rules are the same to Java language.
- Fields can be tagged with private, protected (default) and public. The meaning of these modifiers is the same to Java language.
- All methods in rCOS are treated as public visibility.
- Invariant definition is in the form of:

  ```
  invariant (expression [expression]*)
  ```

  Invariant definition consists of a set of logical expressions, which will be explained later. All instances of a class should fulfil to these expressions. A class may have multiple invariant definitions.

2.3 Method definition

Method definition in rCOS is different from other modern programming language. The form of method definition is like:

```
[synchronized] method methodname( [value parameters];
                                  [result parameters];
                                  [value-result parameters] )
{
    [method body]
```

```
}
```

An rCOS method is led by keyword '*method*' and can be tagged with '*synchronized*' denoting that the current object in which the atomic method lies is locked when the synchronized method is called by clients.

Three types of method parameter are designed in rCOS. There are value , result and value-result parameters. Value parameter is used to pass information to method. Result parameter is similar to return value in other programming languages, but it gives rCOS the ability to return more results after a method call, even there is a class constructor. Value-result parameter is a special form of result parameter. Result and value-result parameters both pass parameters by reference. The difference between them is that value-result parameter can have initial value.

In the method body, users can specify pre and post conditions by using keywords *require* and *ensure*. Before a method call, all expressions within pre-condition declarations must be true. Similarly, all expressions within post-condition declarations must be true after a method call. The violation of pre and post conditions is the manifestation of a bug. If the violation is in pre-condition, the problem is in the caller; if the violation is in post-condition, the problem is in the method body. To refer the value of a variable on method entry, use keyword *origin*. It is only used in pre and post conditions and very useful to retrieve the old value of a variable after the variable has been updated. Here is an example:

```
method foo (;;int val)
{
    require (val >= 100);
    val = val - 100;
    ensure (origin val == val + 100);
    /*The value of origin val is the value before method call.*/
}
```

2.4 Multi-thread modelling

Many object-oriented languages support multi-thread programming. In Java or C# language, there exists a Thread class to provide multi-thread feature. We extend the original rCOS to support multi-thread modelling. As an object-oriented modelling language, rCOS is designed to use declarative syntax to implement multi-thread modelling. Thus users can write multi-thread code more easier. The example is as follows:

```
Main ()
{
    Cook c1 = new Cook (;;);
    Cook c2 = new Cook (;;);
    Cook c3 = new Cook (;;);
    parallel
    {
```

```
            c1.Cook (;;);
            c2.Cook (;;);
            c3.Cook (;;);
        }
    }
```

The keyword *parallel* introduces a parallel block. The parallel block specifies that all method calls within this block are executed in parallel, which means they are running in different threads and scheduled by operating system. How threads are scheduled is an irrelevant implementation detail to the modeling. Although parallel block can specify multi-thread execution, a method is needed to be tagged with atomic usually for dealing with resource conflict in concurrent system.

2.5 Undetermined choice statement

In some situations, the sequence of program execution cannot be foreseen in design time, especially in distributed system. For example, a file sharing server may accept many types of disorder client requests in a short time, such as user login, file copy, folder listing, and file statistics. In many modelling methods, it is impossible to simulate these behaviors.

rCOS has the facility to simulate random behaviors in design time, introduced by *undetermined* notation. This mechanism is corresponding to the demonic choice introduced in [7]. For the example above:

```
undetermined
{
    case:
        server.DoLogon ();
        break;
    case:
        server.ChangeDir ();
        break;
    case:
    .
}
```

Undetermined statement is similar to *switch* statement in which executing path depends on the input state. The difference is *undetermined* statement has no input state; all choices are decided by random functions. *Undetermined* statement and *parallel* statement can be used together to construct complex conditions in object systems. The composition of *undetermined* and *parallel* statements is helpful to verify if the design is correct and to improve the reliability of target system. The example above may be modified to this form:

```
while (true)
{
    parallel
```

```
        {
            undetermined
            {
                case: server.DoLogon (); break;
                case: server.ChangeDir (); break;
                case:..

            }
        }
    }
```

This segment of code gives a prototype of an entry, which can be used to simulate the working condition of a file sharing server. While using verification tool to verify this system, this entry drives the whole system.

2.6 Predicate expression

To specify the properties in rCOS, the predicate expression is used in invariant, pre and post conditions. The logic operators used in other language such as *and, or* exists in rCOS as well. Here we introduce the quantifiers in rCOS predicate expressions. Many programming languages prefer using loop statements to implement quantifier operations. For example, the statement *foreach* in Java language, with a sequence of expressions and statements in it, constructs a code block which implements functions of predicates. As a modelling language, rCOS prefers using declarative syntax to construct predicate operations. There are two quantifiers operators in rCOS, *exists* and *foreach*.

The form of quantifier expression is:

```
quantifier operator value in set [calculates var] where
expressions
```

where

- Keyword *value* is used to provide reference to elements in set. If elements in set are type of primitive type, value is a variable of primitive type; if elements in set are object reference type, value is a reference to the real object.
- Notation *set* is a collection which is iterable.
- Optional. Keyword *calculates* declares a bool variable implicitly, which ought to be the expression result. The scope of var is the expression it belongs to.
- *Expressions* following the keyword where can be another quantifier expression or only a general expression which returns true or false.

Table 1 describes operators used with predicate expressions.

Here are some examples:

- *exists value in Books calculates hasBook*
  where value.Title.Contains(“*rCOSspec*”; *hasBook*;);

Operator	Description
exists	Predicate "exists something in somewhere"
foreach	Predicate "for all elements in somewhere"
=>	Collection element reference
where	Compartmentation keyword. The left hand of it is one or more predicate expressions; the right hand of it is one or more general or predicate expressions.

Table 1
Operators used with quantifier expressions

- $foreach\ value\ in\ Employees\ where\ value.Salary \geq 10000$;
- $exists\ value\ in\ SetA$
  $where\ foreach\ value\ in\ SetB\ where\ SetA => value.m == SetB => value*100$;

3 The Verification of rCOS

In this Section, we introduce how to use model checker Spin to verify the rCOS model. The rCOS2Spin tool is under development at the same time.

3.1 The rules of translating rCOS to SPIN

This section describes the translation of rCOS into PROMELA, which is the modelling language used in SPIN.

3.1.1 Classes and Objects

Each class definition in rCOS introduces both attributes and methods. An object of a class is created with the "new" method. As a modelling language, PROMELA lacks of memory allocation and object reference. Under these limitations, these mechanisms can be emulated using predefined arrays of objects in a convenient size. The term "convenient size" means large enough to allow all object instantiations during the execution of an rCOS program to be performed, but still small, not to exceed the bounds imposed by the state space explosion problem. An array is declared for each class and entries of the array is a record ("typedef" in PROMELA), which represents the attributes of the class. The object reference mechanism uses a pair (c, i) in order to distinguish among several different classes or different instances of the same class. The pair is implemented by an integer value of which value is $(c * 100 + i)$. Here is a segment of rCOS codes:

```
public class Product {
  protected long barcode;
  protected double price;
  protected int amount;
```

```
  invariant (amount >= 0);
  invariant (price > 0);
}
```

There is the corresponding PROMELA codes:

```
#define Index byte
#define ObjRef int
#define MAX_OBJECT 99
#define get_index(x)\
  (x - ((x / 100) * 100))
#define get_class(x)\
  (x/100)
typedef Product_Class {
  int barcode;
  int amount;
  double price;
};

Product_Class Product_Obj[MAX_OBJECT];
Index Product_Class_Next = 0;
```

Where, macro *get_index* is defined to calculate the index of an object record in its record array, while macro *get_class* calculates the class of the object. Two synonyms are defined for convenience, one of which is *Index*, which means *byte* to the index of a object in the record array, another is *ObjRef* that is *int* denoted a reference of an object.

3.1.2 Methods

Methods in rCOS are simply translated into PROMELA macro definitions parameterized with an object reference recording the object on which the method is called. One of the features of PROMELA macro is its lack of local variables. It is a drawback in translating local variables and value parameters in rCOS methods. To remedy this drawback, the identifiers of local variables are prefixed with their class name and method signature. For each value parameter, an extra-variable is introduced as the copy of in parameter. There is no special rule for out and value-result parameters. There is an example for methods:

```
#define Product_get_amount(obj, res)\
  assert(get_class(obj) == Product);\
  res = Product_Class_Obj[get_index(obj)].amount
```

The PROMELA macro *Product_get_amount* is translated from rCOS method *getAmount* in class *Product*. To avoid conflicts, macros *Product_get_LOCK* is used to test if the object manipulated is locked by the thread itself.

```
#define Product_set_amount(obj, value)\
  assert(get_class(obj) ==Product);\
```

```
  (Product_get_LOCK(obj) == null || Product_get_LOCK(obj) == this);\
  Product_Class_Obj[get_index(obj)].amount = value
```

The PROMELA marco *Product_set_amount* is translated from rCOS method *setAmount* in class *Product*. A constructor is also a method. The macro corresponding to constructor of class *Product* is defined below:

```
#define Product_Class_constr(obj, value1, value2, value3)\
  create_object(obj, Product, Product_Class_Next);\
  Product_set_amount(obj, value1);\
  Product_set_price(obj, value2);\
  Product_set_barcode(obj, value3)
#define create_object(obj, c, i)\
  atomic {obj = c * 100 + i;  \
    i++}
```

In macro *create_object(obj, c, i)*, the first parameter "obj" denotes the object reference of the new object created. The second "c" represents the class. The third "i" is a count of class instances and increased one when an instance of this class is created. However, we do not consider garbage collection, so the count will not be decreased.

3.1.3 Control flow

The translating rules for control flow from rCOS to SPIN are shown in the table 2.

3.1.4 Parallel and Synchronization

To deal with the concurrency in rCOS, the synchronization mechanism should be introduced. We follow the way that the shared resources can be locked and released when multiple processes run in parallel together. However, for a designer using rCOS, he does not need to know the lock mechanism behind.

Here we introduce how the synchronized methods work by an example. We assume that the class "Product" has a synchronized method "updateAmount".

```
public class Product {
    ...
    synchronized Method updateAmount(int v; ; ) {
        amount = amount - v;
    }
    ...
}
```

In this case, only one thread at a time may execute this method on the same object. Some other fields are added in the SPIN record to ensure the implement. They are three variables, "LOCK", "WAITING" and "WAIT". The following PROMELA codes is show the extra-fields for parallel and synchronization.

```
typedef Product_Class{
```

	rCOS	SPIN
Sequence	$c_1; c_2$	$c_1; c_2$
Conditional	$c_1 \lhd b \rhd c_2$	if $:: b \rightarrow c_1$ $:: else \rightarrow c_2$ $fi;$
Undetermined choice	$c_1 \sqcap c_2$	if $:: c_1$ $:: c_2$ $fi;$
Loop	$b * c$	do $:: b \rightarrow c$ $:: else \rightarrow break$ $od;$

Table 2
translating rules from rCOS to SPIN

```
    ...
    int LOCK;
    byte WAITING;
    chan WAIT = [0] of {bit};
    ...
}
```

At any time, the "LOCK" variable will be either null (a negative integer) or the thread ID of the thread that currently is executing a synchronized method on the object. Hence, once this field is set to a proper thread ID by a thread that calls a synchronized method, only the method with this thread ID is allowed to access this object. When the call of the synchronized method terminates, the lock is released by setting it to null again. The variables *WAITING* and *WAIT* are used to manage threads that call the *wait()* and *notifyAll()* methods on the object. The macros to access these three variables are defined as follows:

```
#define this _pid
#define continue 0
#define Product_get_LOCK(obj) \
  Product_Obj[get_index(obj)].LOCK
#define Product_set_LOCK(obj,value)        \
  Product_Obj[get_index(obj)].LOCK = value
```

```
#define Product_any_WAITING(obj)           \
  Product_Obj[get_index(obj)].WAITING > 0
#define Product_incr_WAITING(obj)         \
  Product_Obj[get_index(obj)].WAITING++
#define Product_decr_WAITING(obj)        \
  Product_Obj[get_index(obj)].WAITING--
```

A thread that calls method *wait()* will first increase the variable *WAITING* and then try to read a value on the rendezvous channel *WAIT*, if the current object that the thread accesses is locked. The macro to implement operations *wait()* and *lock()* are defined as follows:

```
#define Product_wait(obj)                \
  atomic {                               \
    {                                    \
      Product_incr_WAITING(obj);         \
      Product_get_WAIT(obj)?continue;    \
    } unless {                           \
      Product_get_LOCK(obj) == null      \
    };                                   \
    Product_lock(obj)                    \
  }
#define Product_lock(obj)                \
  atomic {                               \
    Product_get_LOCK(obj) == null ->     \
    Product_set_LOCK(obj,this)           \
  }                                      \
}
```

We use a rendezvous channel in PROMELA to model rendezvous communication for the purpose of ensuring other threads intending to operate this object in parallel not to cause conflicts. When a synchronized method has finished, the corresponding thread sends a value on this channel in order for calling thread to be released. At any time, all threads that are waiting to get access to the object are waiting on this channel. The number of threads waiting on the object is stored in variable *WAITING*. Hence, each time a thread calls the *wait()* method on the object, the variable *WAITING* is increased by one, and decreased by one when released. The variable *WAITING* is used when *notifyAll()* is called by a thread, and all waiting threads have to be released. How many times the *WAIT* channel must be signaled can be known from the variable *WAITING*. The converted rCOS codes of operation *notifyAll* for class *Product* is macro *Product_notifyAll* defined below:

```
#define Product_notifyAll(obj)           \
  atomic {                               \
    do                                   \
    :: Product_any_WAITING(obj) ->       \
      Product_get_WAIT(obj)!continue;    \
```

```
      Product_decr_WAITING(obj)          \
    ::else->break                        \
    od;                                  \
    Product_unlock(obj)                  \
  }
#define Product_unlock(obj)              \
  Product_set_LOCK(obj,null)
```

In the pre-section, we defined the syntax and semantic of parallel. We will propose an example to show how to translate a parallel from rCOS to SPIN. Assume that identifier "o1" and "o2" denotes two objects of class "C" separately and there is a method whose signature is "m1()" in class "C".

Then an rCOS command shown below represents the method call "o1.m1()" and "o2.m1()" will execute in parallel.

```
parallel { o1.m1(); o2.m1(); }
```

The PROMELA codes are followed here:

```
#define C_m1(obj)\
......
proctype proc_C_m1(ObjRef obj) {
    C_m1(obj);
}
......
ObjRef o1 = null; ObjRef o2 = null;

create_object(o1, C, C_Next);
create_object(o2, C, C_Next);
......
run proc_C_m1(o1);
run proc_C_m1(o2);
```

3.1.5 Contract

rCOS supports the method Design by Contract. The principal idea behind Design by Contract is that a class and its clients have a "contract" with each other and the instances of a class should also hold some properties. A client must guarantee certain conditions before calling a method defined by a class and in the same way the class guarantees certain properties that will hold after the call. In the other hand, all the instances of a class should also hold some certain properties. In rCOS, the conditions a client must guarantee before call is pre-condition, the properties a class must guarantee after call is post-condition and the properties all the instance of a class must hold is the invariant of a class. A property is described in first-order predicate logic formula in rCOS. On the other hand, SPIN uses linear temporal logic formula. There are universal quantifier and existential quantifier in first-order predicate logic while no corresponding structure in linear temporal logic. So a loop in an atomic statement is used to simulate an existential or universal quantifier.

When rCOS is translated to SPIN, pre and post conditions are translated into "assert" and invariants are translated into "never" claims. We also introduce one "never" claim for each instance of the class. There is an example for invariant translating to "never" claims. The example is about invariant "$amount >= 0$" in class "Product".

```
/* All the attribute "amount" in class "Product" should be not less
than 0*/
#define Product_Class_inva_0 Product_Class_Obj[0].amount>=0
never {
accept_init: T0_init:
        if
        :: (Product_Class_inva_0) -> goto T0_init
        fi;
}
......
#define Product_Class_inva_4 Product_Class_Obj[4].amount>=0
never {
accept_init: T0_init:
        if
        :: (Product_Class_inva_0) -> goto T0_init
        fi;
}
```

3.2 The translator rCOS2Spin

We have discussed about the steps of verification to rCOS and translating rCOS to Spin is the first step. A tool, named "rCOS2Spin", for the purpose to translate rCOS to SPIN is under developed. A general description about this tool is introduced in this paragraph, and more details can be found in our technical report [16].

The tool input is rCOS codes and the output is PROMELA codes. JavaCC [11] is used to generate a syntax parser to handle rCOS codes. After the parser has analyzed the input, a syntax tree of the rCOS codes and some other related such as variable table are built. Then according to the translating rules described in section 3.1, corresponding PROMELA codes are generated automatically. Then we call the Spin engine to verify the generated PROMELA codes. When the error is reported by Spin, a trace algorithm is applied to locate the corresponding error appeared in the original rCOS code. We can locate which section of PROMELA codes that do not hold the property by the counter-examples given by SPIN if not hold. For the PROMELA codes are translated from rCOS program, the counter-examples can be translated back to the rCOS program through the PROMELA codes.

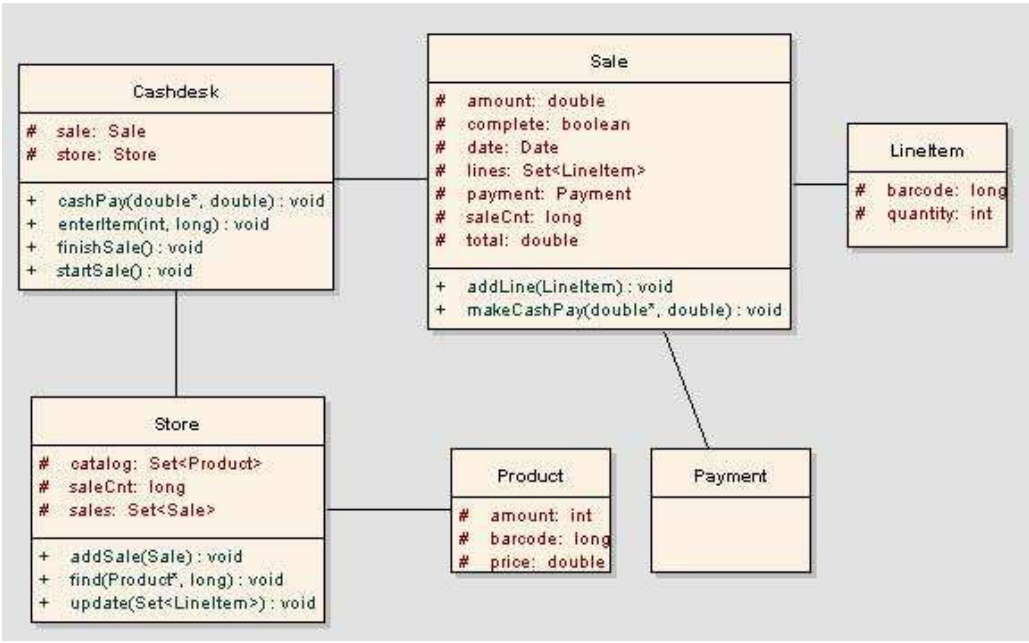

Fig. 1. The class diagram of case

4 Case Study

The case is from CoCoME project [14], which is a shopping system. Here we focus on the sale part of that system. The formalization of the case is given in rCOS. The verification on them will show the potential risk in this case and the remedy will be studied as well.

4.1 Overview of the case

This case is about sale part in shopping systems. When a customer arrives at cash desk, the cashier should check the products the customer wants to buy. The cashier starts a new sale firstly, and then records each product with the price and quantity. When all products are recorded, the payment will be calculated and presented. The customer pays for the products and the cashier makes a change. At last, the sale is ended and inventory data about storage should be update. A class diagram is shown in Figure 1. The diagram is a general version, more details can be found in our technical report [16]. These classes presented in Figure 1 are defined in rCOS followed here. Due to the limited space, we represent key parts in this paper and take class *Store* and its method *update* as an example. The rCOS codes of them are as follows:

```
public class Store {
  protected Set<Product> catalog;
  protected Set<Sale> sales;
  protected long saleCnt;
```

```
  ...
  method update(Set<LineItem> lines; ; ) {
    require(foreach value in lines
      where exsits value in catalog
        where lines=>value.barcode == catalog=>value.barcode)
    Iterator i = catalog.iterator();
    Iterator j = lines.iterator();
    bool iNext, jNext;
    j.hasNext(;jNext;)
    while(jNext==true) {
      i.hasNext(;iNext;)
      while(iNext==true) {
      Product p;
      LineItem l;
      i.next(;p;);
      j.next(;l;);
      long iBarcode, jBarcode;
      i.getBarcode(;iBarcode;);
      j.getBarcode(;jBarcode;);
      if(iBarcode == jBarcode) {
        int q;
        l.getQuantity(;q;);
        p.updateAmount(q;;);
      }   }   }
      ensure(foreach value in lines
        where exsits value in catalog
          where (lines=>value.barcode == catalog=>value.barcode
            && origin catalog=>value.amount ==
                catalog=>value.amount +
                lines=>value.quantity))
  }
}
```

A class named *Cashier* is introduced to drive cases for verification. The class has only one method *doSale*.

```
    public class Cashier {
        method doSale(Cashdesk desk; ;) {
            int quantity;
            long barcode;
            double amount, change;
            desk.startSale(;;);
            desk.enterItem(quantity,barcode;;);
            ...//codes to simulate the sale activity
            desk.cashPay(amount;change;);
            desk.finishSale(;;);
```

```
    }
}
```

4.2 The consistency of contract

In this case, a cashier handles a sale business independently. The program accesses and modifies the data of product storage.

```
Main() {
    Store store = new Store(;;);
    ...//codes to initialize the products stored in the store.
    Cashdesk cashdesk = new Cashdesk(store; ;);
    Cashier cashier = new Cashier(; ;);
    cashier.doSale(cashdesk;;);
}
```

When the SPIN codes converted from rCOS codes of this case are put into to SPIN verifier, there is no assertion violations or any other problems reported. It means that the contracts are satisfied.

4.3 The problem of concurrency

In practice, a system usually allow multiple threads to run in parallel. There are two main good reasons for allowing concurrency:

- Reduced waiting time. There may be a mix of sale business handling on the system, some of which has more products to be sold and some less. If all the methods *doSale()* execute sequentially, a business of selling less may have to wait for a preceding business of selling more to complete, which will lead to unpredictable delays in handling these business. If these methods can run in parallel, the unpredictable delays will be reduced.
- Improved throughput and resource utilization. A thread consists of many commands step by step. Different commands involve different resources. When a command executes, the resources not involved may idle. If other commands can be performed at the same time, the free resources may be employed and the resource utilization is improved. Without doubt, throughput of commands execution in parallel is improved from sequent commands.

However, allowing multiple threads to execute in parallel also cause some problems. For example, Several threads which modify shared data concurrently may lead several conflicts with the consistency of the data. The synchronization is the mechanism to avoid this problem, and in rCOS, keyword *synchronized* introduced in section 2 can be used as a modifier of a method to implement this mechanism.

In the case below, two methods *doSale()*, which are related with object *cashier1* and object *cashier2* respectively, execute in parallel. They both access the data of *Product* which are the elements in set *catalog*, an attribute of object *store* and update the value of attribute *amount* by calling method *synchronized updateAmount(int v; ;)* in class *Product*, which is defined in section 3.1.4.

```
Main() {
    Store store = new Store(;;);
    ...//codes to initialize the products stored in the store.
    Cashdesk cashdesk1 = new Cashdesk(store; ;);
    Cashdesk cashdesk2 = new Cashdesk(store; ;);
    Cashier cashier1 = new Cashier(; ;);
    Cashier cashier2 = new Cashier(; ;);
    parallel {
        cashier1.doSale(cashdesk1;;);
        cashier2.doSale(cashdesk2;;);
    }
}
```

The method *synchronized updateAmount(int v; ;)* has a modifier *synchronized* and can avoid the conflicts with the consistency of the data in multiple threads. If the modifier is omitted, the SPIN tool will report that there exist assert violations when the PROMELA codes translated from the rCOS of the case previous are performed in verification mode. The assertion violated is converted from the post-condition of method *update(Set <LineItem>lines; ;)* of class *Store*, the definition of which can be found in section 4.1.

5 Conclusion and Future Work

In this paper, we give an approach to verify the object model of rCOS using model checker Spin. To enhance the ability of specifying concurrency in rCOS, we add the parallel with synchronization into this language, and demonstrate the usage of concurrency from a case study of CoCoMe Project. The specification property can be denoted by assertion, pre and post conditions, invariant etc. in rCOS. The usage of Spin provides rCOS with a verification engine, and the mapping rules between Spin and rCOS are also constructed as well. The converting method is complicated because of the object nature and concurrency in rCOS. The case study shows this approach can be applied in real life.

For the future work, we hope to develop the analysis and verification tool set for rCOS, including design analysis, design verification and code generation and so on. On the other hand, we consider rCOS as a specification language, and hope to apply it with UML to analyze the consistency of kinds of UML elements. We think this work is the first step for the developing of rCOS tool set.

References

[1] M. Barnett, K. M. Leino, and W. Schulte, *The Specsharp programming system: An overview*, CASSIS04, LNCS 3362, 2004.

[2] M. Beato, M. Barrio-Solorzano, and C. E. Cuesta, *UML Automatic Verification Tool(TABU)*, SAVCBS'04 NewPort Beach, California USA.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. M. Leino, E. Poll, *An overview of JML tools and applicatons*, Journal on Software Tools for Technoloy Transfer 7(1), 2005.

[4] D. Coleman, *et al. Onject-Oriented Development: the FUSION Method*, Prentice-Hall, 1994.

[5] E. Durr and E. M. Dusink, *The role of VDM++ in the development of a real-time tracking and tracing system*, FME93, LNCS 670, pp 64-72, 1993.

[6] K. Havelund and T. Pressburger, *Model Checking Java Programs using Java PathFinder*, International Journal on Software Tools for Technology Transfer, 2(4), April 2000.

[7] J. He, X. Li, and Z. Liu. *A Refinement Calculus for Object Systems*, Technical Report 322, UNU-IIST, P.O.Box 3058, Macau, May 2005. Published in Theoretical Computer Science (2006).

[8] J. He, X. Li, and Z. Liu, *rCOS: A refinement calculus of object systems*, Theoretical Computer Science 365(1-2), pp 173-195, 2006.

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

[10] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004.

[11] JavaCC: A parser scanner generator for java:
https://javacc.dev.java.net/.

[12] B. Meyer, *The start of an Eiffel standard*, Journal of Object Technology 1(2), pp 95-99, 2002.

[13] R. Iosif and C. Demartini and R. Sisto, *Modeling and Validation of Java Multithreading Applications using SPIN*, In Proceedings of the 4th SPIN workshop, Paris, France, November 1998.

[14] Modelling Contest: Common Component Modelling Example (CoCoME),
http://agrausch.informatik.uni-kl.de/CoCoME/.

[15] G. Smith, *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000.

[16] X. Yu, Z. Wang, G. Pu, D. Mao, *The Techonicial Report of the Verification of rCOS Using Spin*, available in the web site:
http://www.sei.ecnu.edu.cn/teacher/ggpu/TR-2007-7.PDF.