Formalization, Testing and Execution of a Use Case Diagram

Wuwei Shen[1] and Shaoying Liu[2⋆]

[1] Dept of Computer Science, Western Michigan University, USA
wwshen@cs.wmich.edu
[2] Department of Computer Science, Hosei University, Tokyo, Japan
sliu@k.hosei.ac.jp

Abstract. Errors in a requirements model have prolonged detrimental effects on reliability, cost, and safety of a software system. It is very costly to fix these errors in later phases of software development if they cannot be corrected during requirements analysis and design. A Use Case diagram, as a requirements model, plays an important role in giving requirements for a software system. It provides a communication tool between software requirements developers and prospective users to understand what requirements of a software system are. However most descriptions of a use case diagram are written in some informal language, leading to possible misunderstanding between developers and users. In this paper, we propose a new rigorous review technique which can be applied to software requirements models. Using this new technique before a software system is fully designed will help us find some potential errors in a requirements model, resulting in reduced time, labor and expenditure in software development.

1 Introduction

The Unified Modeling Language (UML) [12] has been proposed as a modeling language which can be applied in software development from software requirements and specification to software code generation through model design under the same framework. UML has become a standard modeling language in software development. As the first step in software development, the quality of requirements analysis is of great importance to the later phases of software development. High quality of a requirements model can most likely reduce many potential errors occurred in later phases of software development.

According to recent error investigation in software development, researchers have found that more errors are introduced during requirements analysis and design than any other phase in software development. Furthermore, requirements errors have prolonged effects on reliability, cost, and safety of a software system [14, 13]. Requirements errors are more costly to fix during later phases of software development than during the requirements analysis and design phase [2].

Central to software requirements analysis and design is a use case diagram. A use case diagram has been proposed in UML as a notation to describe a software system's requirements and behavior. It has been served as a communication tool between software developers and users. After talking with potential users of a software system, software requirements developers should transform what users expect in a software system into a requirements model given by use case diagrams, representing what the system is expected to do from a perspective of software developers.

⋆ This work is supported by the Ministry of Education, Culture, Sports, Science, and Technology of Japan under Grant-in-Aid for Scientific Research on Priority Areas (No. 15017280).

Use case diagrams play an important role during software development. On the one hand, software users can understand whether the software system satisfies their need at the very beginning in software development when they see use case diagrams. Their complaints about a system can be directly reported to requirements developers and the necessary changes can be made in the requirements model accordingly. Use case diagrams make it possible for users to evaluate the system behavior before code is written. On the other hand, use case diagrams can be used as blueprints during the whole software development. Besides requirements developers, other software developers such as model designers and testers can further design and test the software system based on these use case diagrams.

Most requirements models given by a use case diagram consist of two parts. One is a diagram part and the other is a textual description. The diagram part gives the relationship among use cases and actors. The textual description part informally presents the description for each use case. Most descriptions are written in informal language such as English. Although UML accepts any level of formality for use cases, we think a high level of formality for use cases can reduce many confusion and misunderstanding between software developers and users. The reduction of confusion and misunderstanding can be very helpful in improving software quality.

Formalizing a requirements model has aroused some attention in software research community [2, 7, 15, 1, 17]. Although most research works are based on some UML diagrams except for use case diagrams, we find there are a few research works about use case diagrams. Operation schemas [16] have been proposed to formalize use cases based on the fact that use cases provide the informal description of interactions between a system and its actors, whereas operation schemas precisely describe a particular system action which executes atomically. Since operation schemas are more precise and formal than natural language, they offer some rigorous basis which makes some reasoning possible. We also provided a new formal language (High-level Constraint Langauge) [18] which can be used to give the pre- and post- condition for a use case. We want to use some formal language to describe some requirements model and ultimately to reduce some confusion and ambiguity in software requirements description.

In spite of non-ambiguity in formal languages, many software practitioners complain about the impracticality of many formal languages when the formal verification has been applied in some industry applications. Due to this reason we are looking for some new method, called "rigorous reviews". Instead of some convincing formal proofs, required by most formal methods, to ensure some correctness, rigorous reviews employ some sound and practical review technique to ensure the correctness. Thus rigorous reviews can be done either by means of systematic checking of specification manually or by execution of specification with some tool support. Practitioners can easily learn these techniques without some special training. Especially in this paper we propose a new rigorous reviews technique which can be achieved by means of manual checking and some software tool together, and so practitioners can observe the result directly.

Among the rigorous reviews techniques, testing and execution are the most powerful methods because the behavior of a system can be tested and observed. Execution is a powerful and direct mechanism to observe a system. When practitioners execute a system and find some results which are not what they want in the system, it usually means that errors exist in the system.

However one of the sapient differences between a requirements model and an execution model is a requirements model presents what a system should do, while an execution model presents how a system can do. Because of this difference, it is almost impossible to execute a requirements model during software requirements analysis and design; therefore some requirements errors are really hard to detect when they are first introduced and they usually cannot be found until the software system is tested. Even worse, some of them may not be found after the software system is delivered.

Unfortunately we find there is almost no research work about really using some rigorous reviews techniques to find some requirements errors during requirements analysis and design based on use case diagrams. After observing the lasting impact of requirements errors on software development,

we proposed a new language (High-level Constraint Language) (HCL) [18] to which a requirements model given by use case diagrams can be mapped. In that paper, we used the "execution", the most obvious rigorous reviews technique, to check whether or not a high-level model based on a use case diagram satisfies users' requirements. However, the execution techniques cannot be used to check all high-level models because some specifications are not executable.

Fortunately testing, especially specification testing as a rigorous review technique, can be used to attack the weakness of the execution approach. By specification testing, we mean presentation of inputs and outputs to a specification, and evaluation to obtain a result—usually a truth value, as described in detail in the second author's previous publication [9]. As the post-condition of an operation usually describes the relation between its inputs and outputs, an evaluation of the post-condition needs both input and output values. This is slightly different from program testing in the sense that program testing needs to run the program with test cases only for input variables, while testing specifications (especially for those written in terms of pre and post-conditions) require test cases for both input and output variables, and there is no need to run any program but just to evaluate the related predicate expressions (e.g., pre and post-conditions). When testing an operation, it is necessary to treat the state variables before and after the operation, for example, $\overleftarrow{x}$ and x in VDM [6]; x and x' in Z [3], and $\check{}x$ and x in SOFL [10, 8], as inputs and outputs of the operation, respectively. This will allow the evaluation of the post-condition of the operation and an examination of whether the change of the state made by this operation is satisfactory in its consistency.

The remainder of this paper is organized as follows. Section 2 gives the rigorous reviews techniques used in this paper. In section 3 we first introduce a use case diagram and then a vending machine example is used to show the application of our rigorous reviews techniques. Section 4 draws some conclusions and suggests future work.

2 Requirements Model Testing and Execution

After a requirements model is given by a use case diagram together with HCL specification for each use case, we can use specification testing and execution to find whether any errors exist in the requirements model. Specification testing is used first to check whether there are some inconsistencies in a requirements model. If no inconsistencies are found, then requirements model developers can move to the second stage, i.e. execution of the requirements model.

Although a requirements model can pass specification testing, it cannot guarantee that the model totally satisfies users' needs. One example is that one of the plausible requirements in a vending machine example is the change returned to a customer plus the price of a product the customer would like to buy times the number of the product the customer buys should be equal to the amount of money he pays to the machine. But according to this requirement, the machine can always return all the coins the customer pays to the machine as change when the product is still available. But this solution cannot be acceptable and the specification testing may not find any error in this requirement. Thus, users of the vending machine may not find the problem until they run the prototyping system. Therefore only when users run a prototyping system do they find some more subtle errors in a requirements model.

Therefore the rigorous reviews technique consists of two parts: test a requirements model and execute a requirements model. Fig. 1 gives a use case diagram to show what our technique can do. From this diagram, we can know that software developers are an actor who can get a result from use case *Test a Req. Model* and *Execute a Req. Model.*, while software users are an actor who can observe the system behavior by use case *Execute a Req. Model*. Based on the results, software developers can redesign a requirements model if necessary.

Fig. 2 is a state chart diagram which shows a software development process to which our rigorous reviews can be applied. After designing a use case diagram and giving a pre- and post- condition,

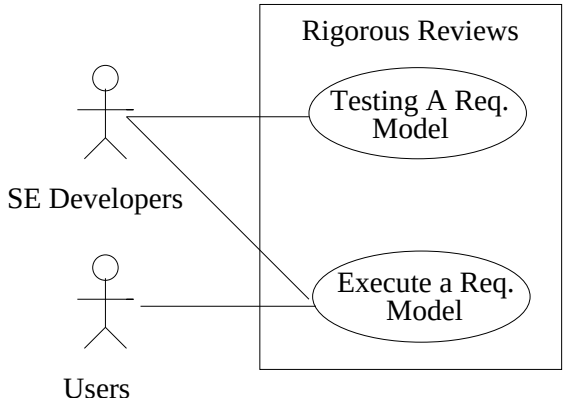

Fig. 1. A use case diagram representing the rigorous reviews presented in this paper.

software developers can use the specification testing and execution proposed in this paper to find errors in a requirements model. If an error is found, software developers can return to the first state, redesigning the requirements model. If no error is found and the requirements model is executable, then the developers can execute the model to find more potential errors. If no error is found and the model is non-executable, then the developers can have two choices. One is they can further refine the model to make the new model executable. In this case the developers return to the first state. The second choice is that the developers can leave the requirements analysis and design phase and go to the next phase to further develop the model. Since software development process is iterative, the process shown in Fig. 2 can always be repeated when a requirements model is designed.

The specific approach to testing a requirements specification (representing a model of the system) consists of two steps. The first step is to derive the properties from the specification as targets for testing. For example, the properties may include the satisfiability of operations, consistency between invariants and operations, and the consistency between different operations when they are integrated. The second step is to test the properties for their consistency. Similar to conventional program testing [19], testing a target property also takes three steps: (1) generate test cases; (2) evaluate the property with the test cases; and (3) analyze the test results to determine whether faults are detected. For the testing of internal consistency of the target properties, test cases are mainly generated based on the structure of the specification. This is similar to *structural testing* for programs where test cases are generated by examining the program structure. In this method, there is no need to provide expected test results, because the nature of the test is interpreted based on the established criteria.

There are five strategies for testing predicate expressions with different objectives, each imposing a different constraint on the selection of test cases. Let $P \equiv P_1 \vee P_2 \vee \cdots \vee P_n$ be a disjunctive normal form and $P_i \equiv Q_i^1 \wedge Q_i^2 \wedge \cdots \wedge Q_i^m$ be a conjunction of relational expressions Q_i^j which are atomic components, where $i = 1...n$ and $j = 1...m$. The first strategy requires that the entire disjunctive normal form be evaluated as true and false, respectively, in order to allow the examination of each case of all the possible evaluations of the expression. The second strategy focuses further attention on each of the disjunctive clause of the form and requires that each disjunctive clause P_i be evaluated as true and false, respectively. Although each disjunctive clause is tested individually using this strategy, there is no guarantee of the independency of the testing due to the possible inter-relation among the disjunctive clauses. Consider the form $x \leq 20 \vee x > 10$ as an example. When the relation

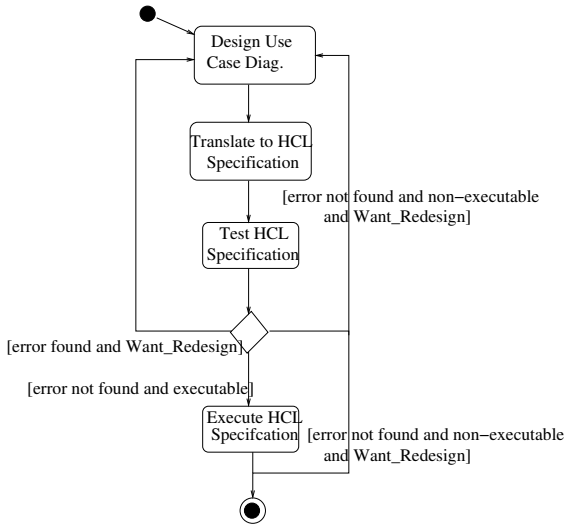

Fig. 2. A state chart diagram showing a software development process where our rigorous reviews can be applied.

$x \leq 20$ evaluates with the test case $x = 15$ as true, another relation $x > 10$ in the disjunction also evaluates as true. Therefore, this test does not examine the case when the truth evaluation of $x \leq 20$ leads to the truth evaluation of the entire disjunctive normal form individually. To overcome this weakness, the third strategy can be adopted, which requires that each disjunctive clause in the disjunctive normal form evaluate as true while all the other clauses evaluate as false. The fourth and fifth strategies give guidelines for the generating of test cases to examine the effectiveness of each atomic expression in each conjunction of the disjunctive normal form, with a little difference in emphasizing the independency of testing each atomic expression.

The next step following the specification testing is to execute the model. Even though not all requirements models can be executable due to high-level specification, the execution of an executable model is important in find errors in a requirements model. We will discuss the executable part in the following. After passing the testing, we will execute the model, trying to find some subtle errors undetected during the specification testing. For those executable part of requirements model, All HCL specifications used to give a requirements model are mapped into AsmL specifications.

AsmL [11] is a formal language based on the theory of Abstract State Machines [5], which was first presented by Dr. Yuri Gurevich more than ten years ago. An ASM is a state machine which computes a set of updates of the machine's variables by firing all possible updates based on the current state. The computation of a set of updates occurs at the same time and the result of computation leads to a new state to be generated. AsmL itself is an executable language and the control flow in AsmL includes many structures such as parallel, sequence, loop, choice and exception. Due to the executability in AsmL, we use AsmL as our target language to execute a use case diagram.

The translation from HCL into ASML consists of two steps. First, each use specification is translated into a method in AsmL. The method body is implemented based on the pre- and post- condition given by HCL. Each pre-condition is translated into a `require` structure in AsmL, which means that

the pre-condition should be satisfied before the method is executed. All the definitions given in the description parts in a requirements model are mapped into declarations in AsmL specifications.

Second, the order of execution of use cases is implied by the inputs and outputs given in a requirements model. Every input which is given in a pre-condition and no relation to other outputs for the rest of use cases is treated as an input in a requirements model. A use case with a model input variable should first be executed. Similarly, any output variable which occurs in a post-condition of a use case but does not appear in any other use case's pre-conditions is regarded as a requirements model output. A use case with a model output should be last executed. The other input and output variables are called internal inputs and outputs respectively. A use case, say `A`, whose output variable is an input of another use case `B`, then the use case `A` should be executed before the use case `B`. Based on the above strategies, we translate a requirement model into an AsmL specification and then execute the specification.

3 Use Case Diagrams and A Vending Machine Example

A requirements model as the first model designed by requirements developers during software development is usually represented by a use case diagram. A use case diagram usually consists of set of use cases, actors and their relationships. An actor is an external user or another system who can initiate a business represented by a use case diagram. A use case represents a sequence of actions which can be performed due to some events triggered by an actor. A use case in a use case diagram only describes one aspect of a software system without presuming any specific design or implementation. A use case only describes what it can do instead of how to do.

There are three relationships which can be defined in a use case diagram, "extend", "include" and "generalization". These relationships try to direct software developers towards a more object-oriented view of the world so as to avoid some duplicate work. But a use case diagram itself does not provide how a requirements model should be structured. Therefore, the description for each use case has become very important in a requirements model. Here we use pre- and post- condition to represent the behavior of each use case. How to use give a pre- and post- condition based on HCL can be found in [18].

In the following we use a vending machine example [4] to illustrate how to give a requirements model by using HCL. A vending machine consists of a money box, a change box, a keypad and a container containing all products to be sold. The vending machine sells sodas, chips and sandwiches whose prices are 60 cents, 50 cents and 100 cents respectively in its container. The keypad provides a mapping between a number and a product. We assume that 0 represents sodas, 1 represents chips and 2 represents sandwiches. We assume that the maximum amounts of the money for the change box, which is used to return change to a customer, and money box, which is used to keep a customer pay, are both 1000 cents. Every purchase only returns one product to a customer.

In the highest level we abstract the problem as follows: a customer can buy a product from the vending machine if (s)he provides a number, which represents the amount of money (coins) to be inserted, and the product code. Then the vending machine can sell the product and return change to the customer if exists. Therefore, the model involves only one actor, i.e. *customer*, and one use case, i.e. *Buy_product*. So the highest use case diagram for the vending machine is shown in Fig. 3, where the stick figure represents the actor who can interact with the system and the oval, named *Buy_product*, represents the use case and provides a service to the actor.

In order to give a complete requirements model, we need to give a pre- and post- condition for each use case in the diagram. Before giving these conditions, we need to find out the requirements of the use case *Buy_product*. After a customer inserts an amount of money and provides a product code, then the vending machine can sell the product and return an integer which represents the amount of change returned to the customer if exists. So the pre- and post- condition for the above use case can

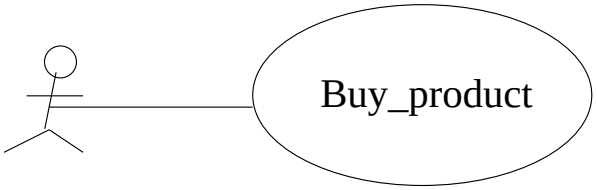

Fig. 3. The highest level model for the vending machine.

```
usecase VendingMachine1 ( in money, product,
                out num_product, changes)
pre: money in Integer, product in Indices(code)
        where money > 0
post: (num_product in RAN, changes in[0..1000]) |
        num_product * price(product) + changes = money
description:
        PRODUCT = {soda, chip, sandiwich}
        RAN = [0..1]
        code: Integer → PRODUCT = {0→ soad,
          1 → chip, 2 → sandwich}
        price: PRODUCT → Integer = { soda → 60,
        chip → 50, sandwich → 100}
```

Fig. 4. The highest requirements model for the vending machine.

be given by the HCL specification in Fig. 4.

The main requirement for this highest level model is that the price of the product a customer buys plus the change if returned should be equal to the amount of money (s)he pays to the vending machine. The other requirement includes the number which represents an amount of money a customer provides should be positive and the product code should be valid. Therefore the *pre-condition* for the use case *VendingMachine1* requires that the amount of money a customer pays be a positive integer and the product (s)he chooses be a valid product stored in the vending machine. The valid products stored in the vending machine are defined by the set *PRODUCT* which is defined in the *description* part. To ensure that a code input by a customer is valid, we use a built-in function $Indices(code)$ which represents the domain for the function *code*.

The post-condition for the use case *VendingMachine1* gives a relation among the amount of money a customer pays, the price of the product (s)he chooses and the change returned to a customer if exists. This is usually what a user of the vending machine expects.

The pre-condition and post-condition of a use case concentrate on some constraints on variables. In order to make the requirements model complete, we use the *description* part to give all the necessary information missed in the pre-condition and post-condition. Thus, we include the definitions for *PRODUCT*, *RAN*, *code* and *price* in the *description* part.

Before we execute the requirements model, we can use specification testing technique to find some inconsistency in the conditions of a use case. According to the criteria mentioned in the previous

money	product	num_product	changes	pre	post	pre => post
20	2	1	20	true	**nil**	**nil**
60	0	0	60	true	**nil**	**nil**
100	2	1	0	true	**nil**	**nil**
50	1	1	0	true	**nil**	**nil**

Table 1. A test for VendingMachine1

---

usecase VendingMachine2 (in money, product,
out num_product, changes)
pre: money in Integer, product in Indices(code)
where money > 0
post: (num_product in RAN, changes in [0..1000]) |
*num_product * price(code(product)) + changes = money*
description:
PRODUCT = {soda, chip, sandiwich}
RAN = [0, 1]
code: Integer → PRODUCT = {0→ soad,
1 → chip, 2 → sandwich}
price: PRODUCT → Integer = { soda → 60,
chip → 50, sandwich → 100}

Fig. 5. The revised highest requirements model for the vending machine.

section, we can automatically generate some test data to check whether there is some inconsistency in the specification for each use case. Table 1 gives a test generated by trying to use the second strategy (i.e., evaluate each disjunctive clause as true and false, respectively) from Fig. 4. The symbol **nil** in tables denotes *undefined*.

Because of the "strange" results when the pre-condition evaluates as true, the post-condition is unable to evaluate as a truth value, we have quickly realized that there must be something wrong in the post-condition. In fact, by this test we have found a type-mismatching fault in the post-condition. In the pre-condition, the variable *product* is defined as an index (natural number or zero) in the set *indices(code)* (the domain of the function *code* {0, 1, 2}), but in the post-condition the function *price*, whose domain is *PRODUCT* = {*soda*, *chip*, *sandwich*} and range is the set of integers {60, 50, 100}, is applied to the variable *product* whose value cannot be a member of the type *PRODUCT*. We correct this mistake by replacing the original function application *price(product)* with the new function application *price(code(product))* and redefine the use case *VendingMachine1* shown in Fig. 5.

To ensure that the modified specification does not introduce new errors and no other kinds of inconsistency problems remain in it, we generate another test given in Table 2 for the modified specification *VendingMachine2*.

In this test the similar phenomena to the test given in table 1 appears again: for the last three test cases in table 2 while the pre-condition evaluates as true the post-condition becomes undefined. After analyzing the reason, we have found that the problem is caused by neglecting that the amount

money	product	num_product	changes	pre	post	pre => post
20	2	0	20	true	true	true
30	1	1	0	true	false	false
60	0	1	0	true	true	true
50	1	1	0	true	true	true
0	0	0	0	false	true	true
200	2	1	100	true	true	true
1100	1	1	1050	true	**nil**	**nil**
1200	2	1	1100	true	**nil**	**nil**
1300	0	1	1240	true	**nil**	**nil**

Table 2. A test for VendingMachine2

of 1000 cents is the maximum capacity for the money box. This problem can be resolved by imposing the further restriction on the range of input variable *money* as *money > 0 and money <= 1000* in the pre-condition. Thus, the use case *VendingMachine2* can be modified into the following specification:

usecase VendingMachine3 (*in money, product,*
out num_product, changes)
pre: money in Integer, product in Indices(code)
where money > 0 and money <= 1000
*... /*the same as that of VendingMachine2 */*

A consistent specification for a use case does not necessarily mean that the specification really satisfies the requirements given by a user. Some design errors in a requirements model cannot be found until users see the execution. Therefore, after checking that a specification for a use case does not include inconsistency, we can execute the specification by translating it into Asml specification. By running the AsmL specification, a user of the system can interact with the prototype system immediately, shown in Fig. 6. The system returns a solution set after execution. Let us assume that a user chooses 0 ("soda") and pays 76 cents. For this given input, there are two solutions which can be observed by this customer. One is to return 76 cents to this customer and the other is return one "soda" and 16 cents as a change to the customer. Obviously, the first solution is not really what a user expects from the vending machine system.

When we return to the HCL requirements model, we find that there exists a problem in the post-condition. The post-condition actually accepts one solution which is that the change to be returned to a customer is equal to the money the customer pays and no product the customer chooses is sold. In any case, this solution should not be accepted. Therefore we should modify the post-condition for the use case *VendingMachine3* and the complete specification is shown in Fig. 7.

In the revised HCL requirements model shown in Fig. 7, we include an `exists` condition in the `if` statement in the post-condition part. The revised post-condition says that if there exists a solution which can return a product to a customer then the vending machine should perform this purchase instead of returning all the money to the customer; otherwise the vending machine should return all the money to the customer.

Before we execute the revised model, we need to check whether there is any inconsistency in our specification. Again this specification can be tested based on the same approach to testing use case *VendingMachine2*. Since giving the specific test data does not add any value in helping explain the principle of our rigorous reviews technique, the test data for testing use case *VendingMachine4* in Fig. 7 are omitted. After the specification testing, we can execute the revised HCL requirements model and find the solution is what a user of the vending machine expects.

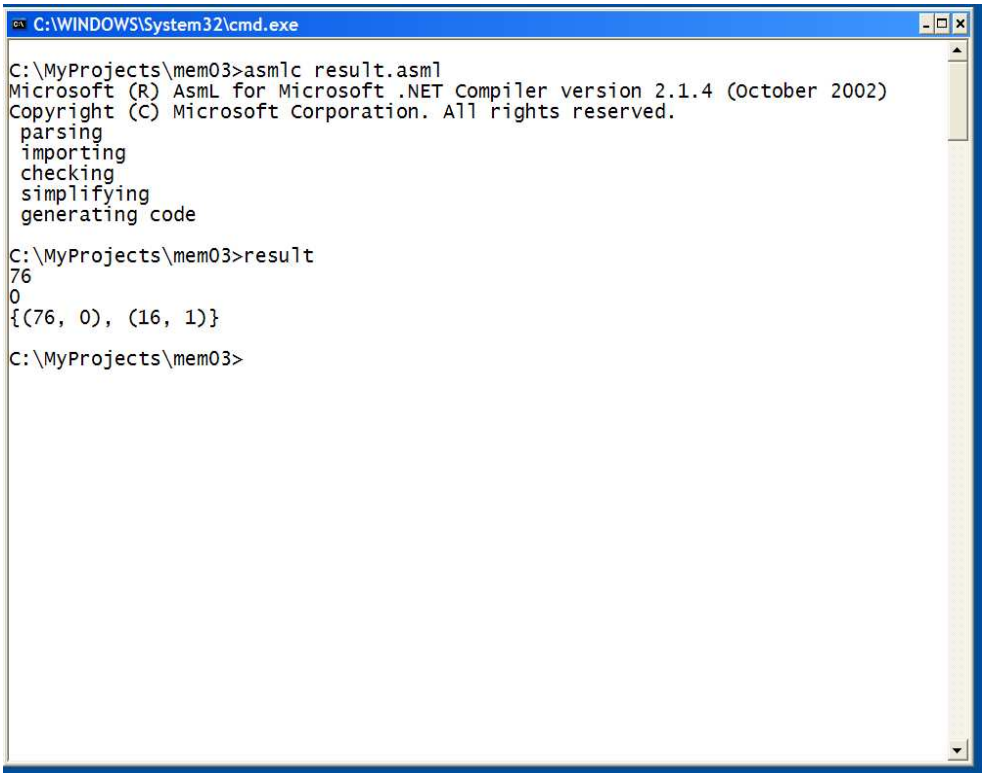

Fig. 6. The result of running the first level model

After giving the highest model for the vending machine example, we can further develop it by refining the diagram shown in Fig. 3. The refined model is presented by a use case diagram shown in Fig. 8. In the refined model, we concentrate on how the model can support the input and output closer to the real vending machine. Usually a customer first provides the code for a product, and then inserts some coins into the machine. The machine figures out and then returns the change to the customer if any according to the money the customer pays and the price of the product.

According to UML, an include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. The included use case never stands alone, but is only instantiated as part of some larger base that includes it. Thus we distribute the responsibility of the use case *deal_order* to its four including use cases. Therefore the new requirements model consists of five different use cases, which are use case *deal_order*, *GetMoney*, use case *GetProduct*, use case *ComputeChanges* and use case *ReturnChanges*; the latter four use cases are included in the use case *deal_order*. The HCL specification for each included use case in the refined requirements model is shown in Fig. 9.

The input, *i.e. money*, to the highest level model has been replaced by a set of numbers, representing a set of numbers of each denomination a customer pays. We assume that the machine can only accept 5-cent coins, 10-cent coins and 25-cent coins. The input parameter n1, n2 and n3 represent

```
usecase VendingMachine4( in money, product,
                         out num_product, changes)
pre: money in Integer, product in Indices(code)
     where money > 0 and money <= 1000
post: (num_product in [0..1], changes in[0..1000]) |
    if (exists (num in [0..1], ret in [0..1000] ) | num > 0
          and num *price(code(product)) + ret = money) then
       num_product > 0 and num_product * price(code(product))
          + changes = money
    else
       num_product = 0 and changes = money
description:
       PRODUCT = {soda, chip, sandiwich}
       code: Integer → PRODUCT = {0→ soad,
         1 → chip, 2 → sandwich}
       price: PRODUCT → Integer = { soda → 60,
       chip → 50, sandwich → 100}
```

Fig. 7. The revised HCL requirements model for the Vending Machine.

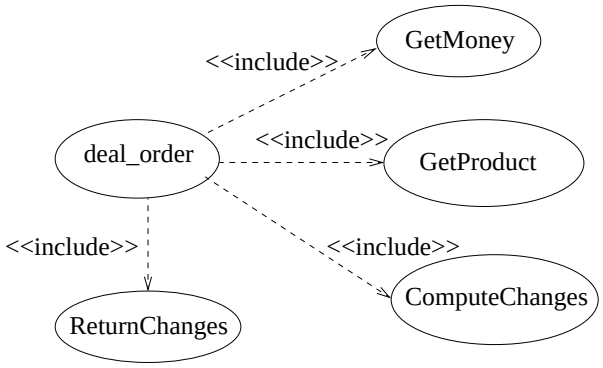

Fig. 8. The refined model for the Vending Machine

the number of 5-cent coins, 10-cent coins and 25-cent coins paid by a customer respectively. These numbers have become inputs to the use case *GetMoney* as well as our incremental model. We deliberately miss the restriction for use case *GetMoney* on the number of coins whose total amount should not exceed 1000 cents. The use case *GetProduct* whose input is a product choice is used to output the price for the product which a customer chooses. Similarly the output *changes* to the highest level model has been replaced by a set of integers, representing a number of each denomination (5-cent coins, 10-cent coins and 25-cent coins) returned to the customer. This set of integers has become outputs in the use case *ComputeChanges*. Also we should follow the restriction on the maximum amount (1000 cents) of money returned to a customer. Here we assume the maximum capacities for 5-cent coins, 10-cent coins and 25-cent coins in the change box are 10, 20 and 30.

```
usecase GetMoney(in n1, n2, n3, out total)
pre: n1 in Integer, n2 in Integer, n3 in Integer where
        n1 > 0 and n2 > 0 and n3 > 0
post: (total in [0..1000]) |
        total = n1 * 5 + n2 * 10 + n3 * 25

usecase GetProduct(in choice, out product_price)
pre: choice in Indices(code)
post: (product_price in [0..100]) |
        product_price = price(code(choice))
description:
        code: Integer → PRODUCT = {0→ soad,
          1 → chip, 2 → sandwich}
        price: PRODUCT → Integer =
          {soda→690, chip→50, sandwich→100}

usecase ComputeChanges(in total, price, out changes)
pre: total in Integer, price in Integer where
        total = GetMoney::total and
        price = GetProduct::product_price
post: ( num in [0..1], changes in [1..1000]) |
        total = price + changes)

usecase ReturnChanges(in c, out o1, o2, o3)
pre: c in Integer where c = ComputeChanges::changes
post: (o1 in [0..10], o2 in [0..20], o3 in [0..30] |
        o1 * 5 + o2 * 10 + o3 * 25 = c)
```

Fig. 9. The refined model for the Vending Machine.

The use case *ComputeChanges* is the main part in the vending machine. It accepts the amount of money paid by a customer, which is returned by the use case *GetProduct*. The constraint *total = GetMoney::total* in the pre-condition of the use case *ComputeChanges* asserts that *total* should be from the output *total* in the use case *GetProduct* and so should p = *GetProduct::product_price*. This gives a potential order to execute the use cases when we want to execute a requirements model. According to these inputs, the use case *ComputeChanges* can have the post-condition shown in Fig. 9 to achieve the main requirements for the vending machine.

The use case *ReturnChanges* computes the number of each denomination which should be returned to a customer as change if exists. The pre-condition shows that the input c comes from the use case *ComputerChanges*. Based on the conditions for each use case, we can use specification testing to check whether any inconsistency exists in these use cases.

Based on the conditions for each use case, we can use specification testing to check whether any inconsistency exists in these use cases. Two tests for the use cases *GetMoney* and *GetProduct* are given in Table 3 and 4, respectively, and tests for another two use cases *ComputeChanges* and *ReturnChanges* are omitted for brevity.

n1	n2	n3	total	pre	post	pre => post
5	10	15	450	true	true	true
10	100	5	1175	true	**nil**	**nil**
100	200	30	999	true	false	false
150	20	10	1200	true	nil	nil

Table 3. A test for GetMoney

choice	product_price	pre	post	pre => post
0	60	true	true	true
1	50	true	true	true
2	100	true	true	true

Table 4. A test for GetProduct

Since the post-condition evaluates as either nil or false while the pre-condition evaluates as true for three test cases in which the value of the variable *total* is beyond 1000, the test shows the possibility of involving an error in the post-condition. After a careful analysis, we have realized that the specification is not satisfiable because of the missing requirements for the maximum amount of money which the machine can accepts. To make this specification satisfiable, we add a further constraint on the pre-condition to ensure that no value of the variable *total* can exceed 1000 according to the formula computing *total* in the post-condition. The modified precondition of use case *GetMoney* is shown in Fig. 10.

As far as the test in Table 4 is concerned, since the post-condition evaluates as true while the pre-condition evaluates as true for all the test cases, there is no indication of potential faults in the specification. After testing, we can further execute the model. For brevity, we skip the execution result, and actually this is a valid model.

The requirements model can be further refined. For brevity, we only refine the use case *Return-Changes*. In this case we consider more requirements in the new refined model while not changing the inputs and outputs for the use case *ReturnChanges*. In the refined use case, the refined requirement is to return the fewest number of denominations to the customer; and therefore we rewrite the post-condition to satisfy the fewest numbers of denominations returned to a customer without changing any input and output in the previous use case. Fig. 11 gives the revised version for the use case *ReturnChanges*. After giving this model, both developers and users can run the HCL specification and check the solution set, which is omitted for brevity.

---

usecase GetMoney(in n1, n2, n3, out total)
pre: n1 in Integer, n2 in Integer, n3 in Integer where
*n1 > 0 and n2 > 0 and n3 > 0 and n1 * 5 + n2 * 10 + n3 * 25 ≤ 1000*
post: (total in [0..1000]) |
*total = n1 * 5 + n2 * 10 + n3 * 25*

Fig. 10. The revised HCL specification for use case GetMoney.

```
usecase ReturnChange2(in c, out n1, n2, n3)
pre: c in Integer where c = ComputeChanges::changes
post: (n1 in [0..10], n2 in [0..20], n3 in [0..30] ) | n1 * 5 +
      n2 * 10 + n3 * 25 = c and
        forall o1 in [0..10] holds
          forall o2 in [0..20] holds
            forall o3 in [0..30] holds
              o1*5+o2*10+o3*25= c implies
                n1+n2+n3 <= o1+o2+o3)
```

Fig. 11. The revised version for use case *ReturnChanges*.

4 Conclusion

After observing the impact requirements errors on software development, we propose a rigorous reviews technique, based on specification testing and execution, to try to find errors in a requirements model during the early phase of software development. Based on our previous work about formalization of a use case diagram, we present in this paper that a requirements model given by a use case diagram together with a pre- and post- condition for each use case can be first tested, trying to find whether there exists any kind of errors such as inconsistency in a requirements model. In order to implement specification testing, we proposed several strategies for these pre- and post- conditions for use cases.

Furthermore, if the pre- and post- condition in a use case diagram are executable, then the requirements model can be executed and some undesirable effects can be directly observed. We also outline the execution model based Abstract State Machine Language. Due to the executability of a requirements model, some necessary changes can be made accordingly. A Vending Machine example has been illustrated to show how the rigorous reviews technique works.

Our rigorous review technique proposed in this paper can also be applied to the later phases of software development, such as a software model design. A design model given by a class diagram which includes a pre- and post- condition for each method defined in classes can also be tested. If the conditions are executable, then the design model can be executed too. Also, we will be looking for some software development process to which our rigorous review technique can be applied. Especially we will investigate some possibility to apply this technique to some use case driven software development processes.

Furthermore, we will work on a tool based on the rigorous reviews technique, which will be very crucial to software practitioners. Furthermore, more examples, especially some industrial applications, will be studied to make our technique more practical in software development in the near future.

References

1. Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, pages 383–395. Springer, 2000.
2. R. Bourdeau and B. Cheng. A formal semantics for object model diagrams. In *IEEE Transactions on Software Engineering*, volume 21 of *No. 10*, pages 799–821, October 1995.
3. Antoni Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 1994.

4. Microsoft FSE Group. Vending machine case study. Technical report, Microsoft FSE Group, June, 2002.
5. Yuri Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
6. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International(UK) Ltd., 1990.
7. Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. In *Software and System Modeling*, volume 1 of *1*, pages 10–42. Springer, 2002.
8. Shaoying Liu. Developing Quality Software Systems Using the SOFL Formal Engineering Method. In *Proceedings of 4th International Conference on Formal Engineering Methods (ICFEM2002), LNCS 2495*, pages 3–19, Shanghai, China, October 21-25 2002. Springer-Verlag.
9. Shaoying Liu. Verifying Consistency and Validity of Formal Specifications by Testing. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Proceedings of World Congress on Formal Methods in the Development of Computing Systems*, Lecture Notes in Computer Science, pages 896–914, Toulouse, France, September 1999. Springer-Verlag.
10. Shaoying Liu, A. Jeff Offutt, Chris Ho-Stuart, Yong Sun, and Mitsuru Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. *IEEE Transactions on Software Engineering*, 24(1):337–344, January 1998. Special Issue on Formal Methods.
11. Microsoft FSE Group. Introducing AsmL: A Tutorial for the Abstract State Machine Language. Technical report, Microsoft FSE Group, Dec, 2001.
12. OMG. Unified Modeling Language Specification, version 1.3. June 1999.
13. R. R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *SIGSOFT '93 Symposium on the Foundation of Software Engineering*, 1993.
14. R. R. Lutz. Targeting safety-related errors during software requirements analysis. In *SIGSOFT '93 Symposium on the Foundation of Software Engineering*, 1993.
15. Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. In *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.
16. S. Sendall, A. Strohmeier. From Use Cases to System Operation Specification. In *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000*, volume 1939 of *LNCS*, pages 1–15. Springer, 2000.
17. Wuwei Shen, Kevin Compton, and James Huggins. A Toolset for Supporting UML Static and Dynamic Model Checking. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*, pages 147–152. IEEE Computer Society, Aug 26-29, 2002, Oxford, England.
18. Wuwei Shen, Kevin Compton, and James Huggins. Execution of a requirement model in software development. March, 2003.
19. Lee J. White. *Software Testing and Verification*, volume 26. Academic Press, ADVANCES IN COMPUTERS, 1987.