Execution of A Requirement Model in Software Development

Wuwei Shen, Mohsen Guizani and Zijiang Yang
Dept of Computer Science, Western Michigan University
{wwshen,mguizani,zijiang}@cs.wmich.edu

Kevin Compton
Dept. of EECS, The University of Michigan
kjc@eecs.umich.edu

James Huggins
Computer Science Program, Kettering University
jhuggins@kettering.edu

Abstract

Latest research results have shown that requirements errors have a prolonged impact on software development and that they are more expensive to fix during later stages than early stages in software development. Use case diagrams in UML are used to give requirements for a software system, but all descriptions for each use case are written in informal language. In this paper, we propose a new language HCL (High-Level Constraint Language) to which any requirement model given by use case diagrams can be mapped. Not only is the language HCL based on a formal language but also the requirement model written in HCL can be executed. Many errors occurring during requirements analysis and design can be detected by means of execution.

1 Introduction and Related Work

Software development usually consists of the following stages: requirements analysis, design, code and testing. A high-level model is usually presented as a result of software requirements analysis and design. Many research studies have shown that many errors can be introduced in a high-level model during the early phases of software requirements analysis and design. These errors can have prolonged effects on reliability, cost, and safety of a software system [7]. Requirements errors are more costly to fix during later phases of software development than during the requirements analysis and design phase [1].

Requirements errors come from two causes. When software developers are developing a software system, they usually are not familiar with the application domain on which the software system is built and they usually acquire requirements by talking to users. Misunderstandings between developers and users can result in requirements errors. Also, although software developers may understand the requirements of a system, they sometimes cannot develop a system correctly according to the requirements because of human errors. For example, some requirement models cannot be designed in a single step due to the complexity of an application. One level model may not fully follow its previous level model. This can result in some erroneous behavior in the final version of the software system. This kind of errors is also quite common during software development.

Therefore, many researchers have proposed different techniques to support software requirements analysis and design. Most of them are based on some formal methods. However, these techniques based on formal methods are rarely used in industry due to the following reasons.

First, the application of these techniques requires high abstraction and mathematical skills to write specifications and conduct proofs, especially when an application becomes very complicated. A software developer cannot grasp these techniques unless one makes a significant commitment to learn them. Usually, the commitment is not equal to the grasp of the proficiency at the necessary skill level. In most cases, a software developer gives up on these techniques based on a formal method.

Second, most existing techniques based on a formal method cannot provide a good marriage between an academic invention and some real applications in industry. The techniques based on formal methods usually concentrate on notation and proof, but fail to help practitioners to apply these techniques in a practical development process. Therefore, these techniques cannot offer usable and effective methods for use in well-established industrial software processes.

Third, almost all of the existing techniques based on formal methods are far from maturity. Many problems such as state explosion in a formal method continue to exist, although some of them can be overcome in some applications. There is no single technique to deal with all problems in software development.

After finding so many problems in formal methods, researchers have provided some new concepts to support software development. One of them is called "formal engineering". A software developer uses rigorous review techniques to validate specifications against user requirements. This ensures that designs satisfy their requirement specifications while programs satisfy designs.

Unlike a formal method, rigorous reviews can be achieved by execution or validation. This is easier to conduct than applying a formal method. They do not need convincing formal proofs, required by most formal methods, to ensure some correctness. But a sound and practical review technique should be achieved by means of some software tools. Among the rigorous reviews,

"execution" is the most powerful method because the behavior of a system can be directly observed. No other special training is needed for practitioners to find whether a requirement of a system is really what they want.

One of the important differences between a requirement model and an execution model is that a requirement model presents what a system should do, while an execution model presents how a system can do it. Because of this difference, it is almost impossible to execute a requirement model during software requirements analysis and design. Therefore, some requirements errors are really hard to detect when they are first introduced. Also 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.

On the other hand, with the introduction of the Unified Modeling Language [6] in industry, it is possible for software development to use the same conceptual framework and the same notation from software requirements and specification through design to implementation. Use case diagrams have become popular to give a requirement model for a software system as the first step in software development.

To give a complete requirement model, software developers sometimes write informal descriptions for some use cases besides a use case diagram. These descriptions usually include a name, a pre-condition, the flow of control, a post-condition, etc., for each use case. However, since these descriptions are written in some informal language (such as English), there may exist errors caused by ambiguity in the langauge. Because it is impossible to execute a high-level requirement model, it is hard for the users of a software system to understand whether the designed requirement model is really what they want.

Although some researchers are trying to formalize use case diagrams in software development, to the best of our knowledge, no research work about first formalizing a requirement model and then executing it. Once this is done, some requirements errors during requirements analysis and design can be found. After observing the lasting impact of requirements errors on software development, we propose a new language HCL (High-level Constraint Language) to which a requirement model given by use case diagrams can be mapped.

HCL is based on a formal language, Abstract State Machines [3]. It overcomes the ambiguity problem when software developers design a requirement model. Furthermore, a requirement model written in this langauge allows developers and users to use "execution," the simplest rigorous review technique, to find potential errors in a requirement model. Requirements errors, that are caused not only by misunderstandings between users and developers but also by developers' mistakes, can be detected in the HCL specification.

On the other hand, the major processes in requirements development are refinement and increment. Due to the executability of the HCL specification, software developers can find whether a model at one level is correctly refined during the development. This provides software developers with a rigorous review technique which can be used in every step during their software development.

In [8] pre- and post- conditions written in OCL (Object Constraint Language) for a use case have been proposed. The authors in propose to use operation schemas to describe pre- and post- condition for a software system so as to avoid problems in a requirement model caused by informal language. The ambitious motivation for their work is similar to ours but they did not consider how to apply the operation schemas in software development. In their work, software developers should include some class diagrams in a requirement model. This is usually not the initial step in software development. There was no discussion of how the operation schemas can be applied in the process of software development. Last, the main difference between their work and ours is that they do not mention the support for any rigorous review technique while we propose to use "execution" to check the correctness of a requirement model.

The object constraint language [4] has aroused researchers' attention in the software community as a supplementary language to UML. But due to many limitations, such as the complicated and inconsistent notations and the stacked structure, OCL has received many criticisms. Being aware of these problems, we chose AsmL as our constraint language over OCL in our requirement model. AsmL is a rather complicated language which can be used to represent many different computer systems. But the subset of AsmL (mostly boolean expression part) we chose to represent constraints in building our requirement model is simple and easy to learn. Software developers are not required to have special training to learn how to use these AsmL expressions to give pre- and post- conditions.

Another related work is done in Project SOFL [9]. Project SOFL proposed a new methodology based on a new language called SOFL, which combines the traditional waterfall development with object-oriented development. But in the high level model design, the authors require one to consider data structures and the flow of control among data. To the best of our knowledge, this information is not usually proposed during software development until some more functional requirement models are designed. In addition, with the popularity of UML in industry, it is not necessary to invent a new notation to replace the current one, as it is often hard to have new notations accepted.

As the first step toward finding a reliable methodology to develop a software system, we investigated the early phases in software development. We concentrated on execution of a prototype system represented by use case diagrams. In the future, we will study the subsequent phases, such as the design and code in software development.

The remainder of this paper is organized as follows. Section 2 gives background about use case diagrams and software requirement development. Section 3 presents our new methodology. A vending machine example is used to show the application of our new methodology in section 4. Section 5 draws conclusions and suggests future work.

2 Software Requirement and Use Case

During software development, requirements analysis and design is the first phase, in which software developers design a requirement model after they talk to users of a software system. Be-

cause the requirement model will be used from design to testing, its quality has an influential impact on the whole software system. However, how to design a requirement model which can closely follow the requirements given by the users of a software system is a challenging topic. In general, users of a software system are not familiar with the jargon used in software engineering. The users cannot find whether the system is exactly what they want until they see the system running.

On the other hand, with the application of software systems to more complicated problems in the real world, software developers cannot design a requirement model in one step. Usually software developers first design an abstract model and then refine all parts of the abstract model in the following steps. If at any abstraction level in software development the users of a system can interact with a requirement model, the quality of a software system can be dramatically improved because software developers can adjust their requirement model right after they receive some feedback from the users instead of waiting until the users use the software system after it has been developed.

A requirement model only describes the functionality of a software system, i.e what to do, instead of implementation details. With the birth of the Unified Modeling Language, parts of requirements can be moved into use case diagrams which are widely used in software requirements analysis and design. Because use case diagrams provide a clear way to represent the structure of the requirements in a software system and therefore they are easy to serve as a communication means between software developers and users, use case diagrams have played an important role in software requirement development.

A use case diagram consists of a set of use cases, actors and some relationships among them. A use case represents a function of business in a system, i.e. what the business does. An actor is an outside user of the system and the actor can interact with use cases defined in the system. The relationship in a use case diagram can be divided into four categories: *generalization, include, extend* and *association*. However, in most cases when describing software requirements, it is not enough to only use the diagrams; therefore some descriptions for each use case such as main flow of events, precondition, post condition and exceptional flow events should be provided as supplements to a use case diagram. All of this graphical and textual information yields a complete requirement model for a software system.

Although a use case diagram which gives requirements for a software system provides an obvious and nice means to get some feedback from some outside users, the way to design software requirements through use case diagrams still makes it hard to have outside users know whether the system does what they want. The best way to have users understand the requirements of a system is to run a prototype of the requirement system. However, with the current requirement model represented by use case diagrams, it is hard to accomplish this goal.

There are two reasons which make it difficult to execute a prototype system for a requirement model. First, most descriptions for a use case are written in an informal language. Second, each use case describes what to do for a system or subsystem instead of how to do it; so most software systems have difficulty supporting the execution of a requirement model.

Furthermore, software development is a process of refinement, increment and iteration. How to guarantee that one level requirement model is correct is a challenging topic. Also how to make sure that one level model correctly refines its higher level model is another important task faced by software researchers. How to iteratively apply the refinement and increment methods to software development has become an important issue when developing a complicated software system.

However, after having seen the importance of software requirements in software development, we present a new language, which can be used to describe a use case given in a software requirement model. Unlike most programming languages, this new language concentrates on structures capable of describing what to do for a system instead of how to do it. The goal behind this new language is that we will provide a methodology which can be applied to different levels of software development. This new methodology not only can be used to describe a requirement model for a software system but also provides a rigorous review technique to let software developers and user observe the dynamic behavior of a requirement model at different levels during software development, shown in Figure 1.

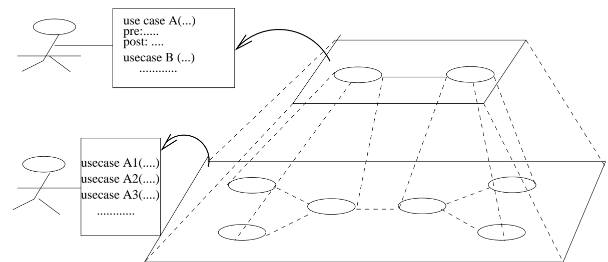

Figure 1: The relationship among a HCL specification, users/developers, and a requirement model at different levels during software development.

In most requirement models, some descriptions such as main flow of event, precondition and post condition are given. In fact, most of these descriptions can be implied by a pre-condition and post-condition. Therefore, the new language we propose in this paper is used as a supplement to use case diagrams by only describing the pre-conditions and post-conditions. We use HCL to formalize and execute a requirement model. After a requirement model given in HCL, software developers and users have an opportunity to observe the dynamic behavior of a requirement model by means of execution.

3 The High-level Constraint Language for Software Requirement Development

During software requirements development, software developers can develop a software system by increment or refinement.

In most cases, the development should be iterative. Use case diagrams play an important role in requirement model development. However, a use diagram itself does not provide too much information about a requirement model except for some use cases and their relationship in the model. Software developers usually use some descriptions to describe the behavior for each use case.

The High-level Constraint Language (HCL) is designed to textually represent use cases in a requirement model. A HCL specification for a requirement model consists of a set of HCL specifications, each of which is used to describe requirements for each use case in the model. Each use case HCL specification includes *pre-condition, post-condition* and *description*. The *pre-condition* for a use case gives the condition which should be satisfied so that the use case can be called. The *post-condition* represents a condition which should be satisfied after the use case is called. The *description* gives some auxiliary information which can make requirement specification for a use case complete.

To reflect the requirements in software development, we choose a subset of Abstract State Machine Language (AsmL) to give the conditions and description. In the HCL specification, we omit the relationship given in the use case diagram. In general, use case diagram describe requirements (dynamic behavior) of a software system. Therefore, we assume that in any given requirement model, all requirements for a use case (*A*) are assigned to its *included*, *excluded* and *children* use cases if they exist in a use case diagram. All requirements for these use cases should achieve the requirements for the use case (*A*). Therefore, only are these *included*, *excluded* and *children* use cases mapped to the HCL specification and the HCL specification for the use case (*A*) is omitted.

After a requirement model given in a use case diagram is mapped to a HCL specification, the HCL requirement model can be executed thanks to the executability of AsmL; therefore the users of a software system can immediately observe the result of the prototype system instead of waiting until the software has been fully developed. Because there is no relation among all use cases in the specification, we use the input and output variables in the HCL specification to find some execution order in a requirement model.

Before introducing the software requirements refinement, we give some introduction to HCL. HCL accepts most types used in AsmL. Besides the predefined types such as Integer, String etc., HCL provides some other types which can be defined by a user. A user defined type can include a set, a sequence or a map.

Because HCL is used to describe the functionality for a system by giving a pre-condition and post-condition for each use case, we choose parts of AsmL syntax to represent these conditions. Most of them accepted in HCL are related to (boolean) expressions. The syntax for AsmL expressions can be found in the AsmL document [5]. In order to give a complete HCL specification for a requirement model, we give the syntax for the HCL specification in Figure 2.

To describe a requirement model given by a use case diagram, we use a use case HCL specification to describe the functionality for each use case. A use case HCL specification consists of a use case name, parameters, a pre-condition and a post-condition. Following the keyword `usecase` is a use case name defined in a use case diagram. Parameters are given in a pair of parenthesis following the use case name. There are two kinds of parameters in a use case HCL specification. One is the input variables following the keyword `in`. The other is the output variables following the keyword `out`. All parameters are separated by "," and all output variables should be defined after the input variables in each use case.

```
HCL_specification :: = usecase_spec {usecase_spec}
usecase_spec ::= USECASE ID "(" parameters ")"
            pre_cond post_cond descrip
pre_cond ::= PRECONDITION ":" AsmLexpression
post_cond ::= POSTCONDITION ":" post_condition
post_condition :: = [ quantifier ] "(" variable_list ")"
              "|" constraints
quantifier :: = exists | all
variable_list :: = variable IN type { "," variable IN }
                  [type where AsmLexpression]
constraints ::= AsmLexpression
descrip ::= DESCRIPTION: description
description ::= set_def | func_def | range_def
set_def ::= ID "=" "{" element { "," element} "}"
func_def ::= ID ":" ID → ID [ "=" "{" ID → ID
            {ID → ID} "}"
range_def ::= ID "[" Integer ".." Integer "]"
```

Figure 2: The syntax for HCL.

The *pre-condition* part includes a valid AsmL boolean expression. The *post-condition* part consists of two parts; one includes all output variables and their ranges related to this use case, while the other part is a valid AsmL boolean expression which gives a restriction on these output variables. The output variable range should have a finite number of elements so that the execution can find all elements, which satisfy the restriction, in this finite range. The *description* part gives complete information about the use case and it usually includes definitions for functions and maps etc. used either in *pre-* and *post- condition* parts.

The development of a software system is a process for tackling a problem in a piece-meal fashion, i.e. describing the problem incrementally. To add more information in our design model as software development is going on is a necessary step and the new model with more details is regarded as a refinement model for the previous one. One example of a incremental model is that the formal input/output of a model will be changed so as to closely follow the real application. But how to make sure that the incremental model still satisfies the requirement is still important.

The development of a software system is also a process of refinement. It means that without changing the behavior given in one level model, software developers add some more requirement details to the next level model which is said to refine the previous level model.

Last the development of a software system is also a process of iteration. In some cases such as when software developers find some problems at one level model or the users of a software sys-

tem change their requirements, they usually return to the previous models designed in the early phases of software development and make some necessary changes. These changes usually can result in some changes in the subsequent models.

The whole software development process is actually a process which repeatedly uses one of the above three processes. Because iteration is the process to return to the previous development phases, we consider the first two processes, i.e. increment and refinement process by using the HCL specification.

Either software developers or software users can observe whether an HCL specification for a requirement model satisfies the software system requirements by executing the HCL specification during any phase in software development. From the set of outputs returned by the HCL specification, software developer or users can find whether all elements in the output set are really what they expect. If there is some element which they do not expect, then some changes are necessary to make in the HCL specification for the requirement model.

usecase Buy_product (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, sandiwich → 100}

Figure 3: The highest level of a software requirement model.

4 An Example: Vending Machine

We use a vending machine example [2] to illustrate how to give a requirement model by means of HCL. A vending machine consists of a money 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 amount of the money (machine money) which can be returned to a customer is 1000 cents. Every purchase only returns one product to a customer. All money is stored in the money box.

In the highest level model for the vending machine, we abstract the model as follows. A customer can buy a product from the vending machine if (s)he provide a number, which represents the amount of money to be inserted, and the product code. We assume that the products are always available in the vending machine and the vending machine returns an integer which represents the amount of changes returned to the customer if exists. So there is only one use case in the highest level requirement model, called `Buy_product`.

To describe a complete requirement model for use case `Buy_product`, we can give the HCL specification in Figure 3.

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 a customer inputs should be positive and the product code should be valid. The *pre-condition* for the use case *Buy_product* 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 the function $Indices(code)$ which represents the domain for the function *code*.

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

The pre- and post-condition of a use case concentrate on some constraints on variables. The description part gives all the necessary information used in the pre-condition and post-condition and the description part makes the requirement model complete. We include the definitions for *PRODUCT*, *RAN*, *code* and *price* in the description part.

After giving the above HCL requirement model, we can execute it and a user of the system can interact with the prototype system immediately, shown in Figure 4. 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 really not what we want for the vending machine system.

When we return to the HCL requirement 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 returned. In any case, this solution should not be accepted. Therefore we should modify the post-condition for the use case shown in Figure 3.

In the revised HCL requirement model shown in Figure 5, 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

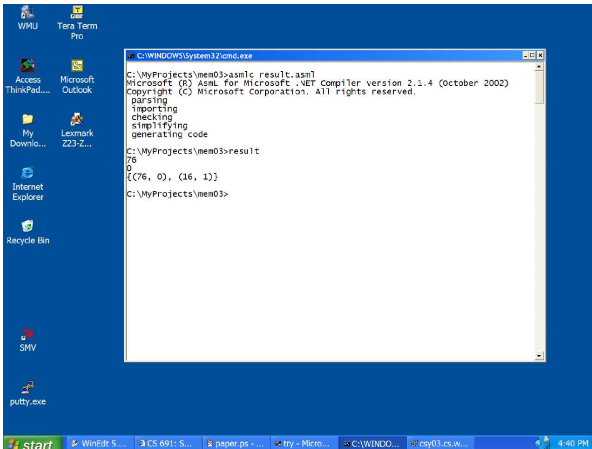

Figure 4: The result of running the first level model

usecase Buy_product (in money, product,
out num_product,
. . . , . . . , . . .
post: (num_product in [0..1], changes in[0..1000]) |
if (exists (num in RANGE1, ret in RANGE2) | num $>$ *0*
*and num *price(product) + ret = money) then*
num_product $>$ *0 and num_product * price(product)*
+ changes = money
else
num_product = 0 and changes = money
. . . , . . . , . . .

Figure 5: The revised HCL requirement model for the Vending Machine.

return all the money to the customer. After we execute the revised HCL requirement model, we find the solution is what a user of the vending machine requires.

5 Conclusion and Future Work

In this paper, we present a new notation which can be used to formally and textually represent a requirement model, usually presented by a use case diagram. Although there are some research work about using pre-condition and post-condition to describe a use case, our goal is to use "execution", the simplest rigorous review technique, to observe a requirement model at a different high level instead of only describing a requirement model.

Thanks to the executability of Abstract State Machine Language, any use case diagrams can be mapped to a set of HCL specifications, each of which includes a pre-condition, a post-condition and description for one use case. Therefore, both software developers and users can observe some execution results about the prototype of the software system being designed. Any error or undesired result observed through the execution can be corrected accordingly.

Besides execution, we will study some other rigorous review techniques which can be applied to software development. While we propose to use HCL specification to give a requirement model for a software system, we believe some of high level models can not be easily executed. Therefore, we should use other rigorous review techniques to evaluate these high level models. As a supplementary rigorous review technique to the execution method proposed in this paper, we will consider specification testing, which has become an important technique applied to evaluate a requirement model, to support the early phase of software development. In short, the focus of this work will further seek for a complete methodology to help software developers to design a correct and reliable software system. Furthermore, some state-of-the art applications such as computer network protocol designs will be studied using this new methodology. We will also validate some computing security issues which are important in computer applications.

References

[1] 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.

[2] Microsoft FSE Group. Vending machine case study. Technical report, Microsoft FSE Group, June, 2002.

[3] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[4] Jos B. Warmer, Anneke G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley, 1998.

[5] Microsoft FSE Group. Introducing AsmL: A Tutorial for the Abstract State Machine Language. Technical report, Microsoft FSE Group, Dec, 2001.

[6] OMG Unified Modeling Language Specification, version 1.3, June 1999.

[7] R.R. Lutz. Targeting safety-ralated errors during software requirements analysis. In *SIGSOFT '93 Symp. on the Foundation of Software Engineering*, 1993.

[8] 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.

[9] Shaoying Liu, Jeff Offutt, Chris Ho-Stuart, Yong Sun, Mitsuru Ohba. SOFL: A Formal Engineering Methodology for Industrial Applications. In *IEEE Transactions on Software Engineering, Special issue on Formal Methods*, volume 24 of *1*, pages 24–45. IEEE Computer Society Press, January 1998.