On UML2.0's Abandonment of the Actors-Call-Use-Cases Conjecture

Sadahiro Isoda

Toyohashi University of Technology
Toyohashi 441-8580, Japan
isoda@tutkie.tut.ac.jp

Abstract. UML2.0 recently made a correction which effectively means that they finally abandoned the *ACU conjecture*, admitting it was wrong. The ACU conjecture is a fallacious statement that makes people believe that actors call operations of a use case. It originated in OOSE and was implicitly employed by UML. It caused serious defects in UML's specification of the use-case class. Although the correction itself could shed light on the long-troubled use-case specification, UML2.0 does not recognize its significance. This is proved by the fact that it still retains those defects. The paper states how the ACU conjecture caused them. It then presents a model of a designer's simulation of a use case, derives a use-case class/object, and specifies its static and dynamic properties. In relation with this, it clarifies the meaning of a use-case diagram that has been left ambiguous.

1 Introduction

OOSE, or Object-Oriented Software Engineering, is a use-case driven, object-oriented software development methodology developed by I. Jacobson et al. Use cases are part of requirements specification and they drive the entire system development in that they play a major role in the analysis, design, and testing stages. Since the publication of his prominent work [4], use cases got gradually accepted in the software community, and are now part of UML (Unified Modeling Language), which is a de facto international standard. This fact shows that the idea of use cases is fine and we should recognize our deep debt to I. Jacobson.

The development of UML was started by the three Amigos (G. Booch, J. Rumbaugh, and I. Jacobson) who were the prominent OO methodologists and founders of Booch method [1], OMT [12], and OOSE, respectively. UML was first called the Unified Method, which aimed to be a standard of OO methodology. But soon they decided to restrict the coverage of their product to how to draw diagrams, changing its name to UML. The first version, UML 1.0, was released in 1997, followed by 1.1, 1.2, 1.3, 1.4, and 1.5 [5–10]. Currently, UML2.0 [11] is open to the public. UML was adopted as a standard by the Object Management Group (OMG) in 1997.

UML2.0 recently made a correction about *the relationship between use cases and actors* [11](section 16.3.6, page 523), which effectively means that they finally abandoned the *ACU conjecture*[3], admitting it was wrong. The ACU conjecture is a fallacious statement that makes people believe that actors call operations of a use case. It originated in OOSE and was handed over to UML. The ACU conjecture caused serious defects in UML's specification of the use-case class. Although the correction could shed light on the long-troubled use-case specification, UML2.0 does not recognize its significance; this is proved by the fact that it still retains those defects.

The paper states how the ACU conjecture caused the defects. It then makes a model of a designer's simulation of a use case, derives a use-case class/object, and specifies its static and dynamic properties. In relation with this, it clarifies the meaning of a use-case diagram that has been left ambiguous.

2 ACU conjecture

UML1.5 specifies that "an actor instance calls use-case operations" in the execution procedure of a use-case instance [10](page 2-137). In connection with this, OOSE says that "we may view every interaction between an actor and the system as the actor invoking new operations on a specific use case" [4] (section 6.4.1). OOSE, however, does not present any reasons for this statement. Hence, the paper [3] named it *Actors-Call-Use-Cases* or *ACU conjecture.* While UML1.5 defines that a use case is a description of the communication between actors and a system (i.e., subject) [10] (section 2.11.2.5, page 2-132), the ACU conjecture specifies that actors call use-case operations; a "system" in the definition is replaced with a "use case." Obviously, the ACU conjecture is against the definition of a use case. Although UML1.5 never refers to OOSE's statement mentioned above, i.e., the ACU conjecture, we can assume that it adopts it; otherwise, it could not specify the execution procedure that is in line with the conjecture.

UML2.0 recently made a correction concerning use cases; deleting the expression "an actor communicates with a *use case*" [10](section 2.11.2.1, page 2-131), and replacing it with a new expression "an actor interacts with a *subject*" [11](section 16.3.1, page 512). This correction effectively means that they finally abandoned the ACU conjecture, admitting it was wrong. UML2.0, however, explains the reason for the correction as: "The relationship between a use case and its subject has been *made explicit*" [11](section 16.3.6, page 520). Although this description sounds like saying that the correction is a minor one, [1] the ACU conjecture actually caused serious defects in the use-case specification as is pointed out in the next section and therefore the correction is indeed a significant one; but UML2.0 does not recognize it.

[1] This sort of 'quiet' change is a second one in the histry of UML's use case. The first one was when they changed the category to which the include and extend relationships belong from a kind of generalization to dependency.

3 Consequences of ACU conjecture

3.1 Operations of a use case and its generalization

Because of the ACU conjecture, a use-case class is believed to have illusory operations and hence the generalization relationship comes to have also illusory meaning as shown below.

1. It is actually a system (i.e., subject) that an actor sends a message to and hence invokes the system's action sequence. The ACU conjecture, however, makes people believe that an actor calls a use-case's operation. Consequently, they (wrongly) believe that the client of a use-case class were actors and that the operations that a use-case class provides were equivalent to system functions. This makes a use case to have operations that inherently belong to the system; that is, "a use case is like a system operation" [13] (page 65).
2. Use-case's operations are (wrongly) recognized as equal to the use-case's behavior description. [2]
3. Based on the two items above, a use-case class is (wrongly) recognized to be able to inherit behavior description from a parent use-case class through generalization relationship. The use-case generalization thus defined could add behavior to an existing use case. This brings about the result that a use-case generalization relationship could have a similar effect as a use-case extend relationship in that both relationships can add "behavior" to a use case.

Figure 1 shows a part of a figure derived from UML2.0 [11](Figure 406, page 521). Apparently, it is meant to be an example of generalization between use-case classes; two use-case classes, "Withdraw" and "Transfer Funds," are specializations of "Perform ATM Transaction." However, this does not hold; these three are actually functional modules of a target system. The example is really caught in a trap built by the ACU conjecture.

3.2 "Execution" of a use case

UML2.0 describes the execution of a use case as follows: *This functionality, which is initiated by an actor, must always be completed for the use case to complete. It is deemed complete if, after its execution, the subject will be in a state in which no further inputs or actions are expected and the use case can be initiated again or in an error state* [11](section 16.3.6, page 519).

We have three points to make concerning the description. Firstly, it says that an actor initiates a use case, which is exactly what the ACU conjecture states. Now that UML2.0 effectively abandoned it, the description should be revised.

[2] This in fact is based on an inaccurate reasoning. The behavior description of a use case is a repetition of actors' sending messages and a subject's responses. The operations that a use case comes to have correspond to the subject's responses. That is, the reasoning disregards the actors' sending messages.

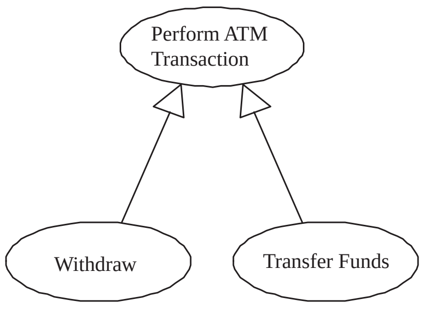

Fig. 1. UML's generalization between use-case classes

Secondly, the description seems to be inadequate because it says how to start and how to end, but does not say anything about how the execution goes on.

Thirdly, we can understand that UML2.0 assumes that an actor is an active agent and a use case executes of itself. We, however, think the assumption is questionable because of the following reasons.

– **Control of use-case "execution."** In order to generate a scenario from a use case, one or more actors send messages to a subject and the subject responds to them all based on the use-case behavior description. Because a use case can generate a variety of scenarios, there may be chances when an actor has to select a message out of alternatives or the subject has to select an action also out of alternatives. If an actor were an active agent and it could do the selection at its own free will, how could we control which scenario to generate?
– **High-level description.** UML2.0 assumes that a use case can execute of itself, but a use case is a high-level description of a functionality and therefore it is generally impossible to do so.

4 Solution

4.1 Modeling a use case

As shown in the last section, the current use-case specification is severely influenced by the ACU conjecture. Now that UML2.0 has effectively abandoned it, we have to build another use-case specification afresh, discarding the current one. So, let us go back to the starting point, i.e., modeling a use case, which is a thing in the real world, as a use-case class/object [3].

A typical situation of using a use case is the analysis and design stages of software development. A designer simulates a system function while reading a use case, making an image of actors and a system in his head, and selecting actors' messages and system's responses from the alternatives the use case provides. Needless to say, it is the designer that decides which use case and which scenario of the use case to simulate (Fig.2(a)).

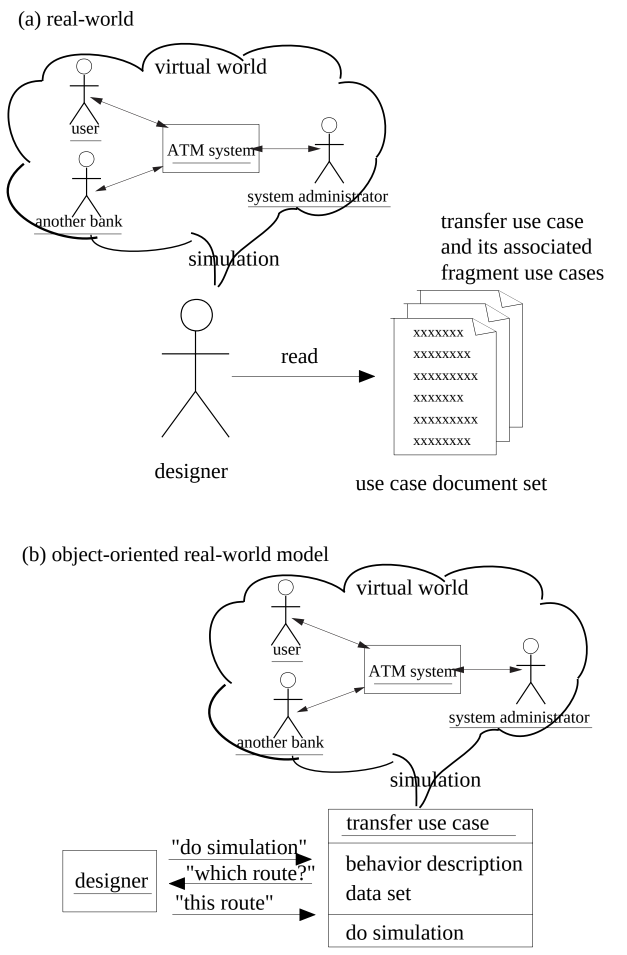

Fig. 2. Real-world use-case simulation and its model

We directly map this situation into an object-oriented real-world model [2]. Because a use case is a thing called document in the real world, we identify a use-case object in the real-world model. In the real world a designer reads a use case and simulates a system function, while in the real-world model we anthropomorphize the use-case object and think that the object itself simulates. The use-case object, however, can not itself decide which scenario to simulate, but the designer object does it just as in the real world.

4.2 Properties of a use-case class

UML2.0 does not explicitly specify what properties (i.e., attributes and operations) a use-case class/object has. Probably this is due to its ambiguous definition

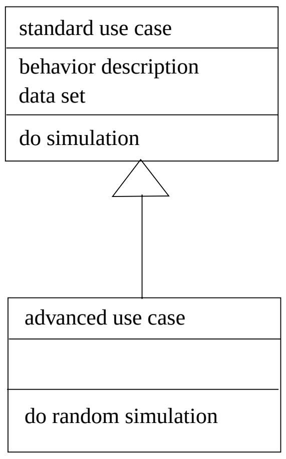

Fig. 3. Example generalization between use-case classes

of a use-case class. [3] Let us define the properties of a use-case class based on the model described in section 4.1. The primary attribute of a use-case object is the behavior description. It is encapsulated into a use-case object together with an operation "do simulation" that simulates its behavior description interactively getting commands from a designer object(Fig.2(b)).

When a generalization relationship is defined between two use-case classes, a use-case subclass can inherit those properties mentioned above from its super class. Figure 3 is to show how a generalization relationship between use-case classes is used. The "advanced use case" class has an advanced simulation operation, "do random simulation," which overrides the standard simulation operation, "do simulation" of the parent class "standard use-case." The operation "do random simulation" is supposed to be an "advanced" one in that it can perform an intelligent simulation; when the advanced use-case object comes to a branching point, it can randomly select a particular route out of alternatives.

4.3 Use-case simulation procedure

The two problems mentioned in section 3.2 can easily be solved when we recognize that use-case "execution" means simulation by a designer (section 4.1). Let us analyze how the simulation goes on. We first assume that a use case consists of a single (primary) use case because we do not know how to do it when fragment use cases are involved. [4]

Following is the simulation procedure when a designer (a designer object, to be precise) conducts simulation using a use-case object in the real-world model.

[3] OOSE claims that a use case is a class just becuase it can 'create' instances (i.e., scenarios)[4](page 128) and UML inherits this idea. However, this is questionable [3].

[4] We will know how to do it in section 4.5.

1. The designer sends a start message to a use-case object.
2. The use-case object makes a list of messages that its primary actor can send to the subject based on its behavior description, and sends it back to the designer.
3. The designer selects a message out of the list and then sends it back to the use-case object, commanding it to simulate the primary actor's sending the message.
4. The use-case object performs the simulation step "The primary actor sends a certain message to the subject" in the virtual world (This is the initiation of the functionality).
   - 4a. If the response of the subject is uniquely determined from the behavior description of the use case, the use-case object performs a simulation step "The subject does something in response to the message."
   - 4b. If the response can not be uniquely determined, the use-case object makes a list of possible responses that the subject can do and then sends the list to the designer. The designer selects one and sends it back to the use-case object, commanding it to simulate the subject's response. On receiving a command from the designer, the use-case object performs a simulation step "The subject does something."
5. The use-case object makes a list of pairs of an actor and its possible message, and then sends it to the designer. If there are no such pairs available, the use-case object terminates the simulation.
6. Go to step 3.

In the light of the use-case simulation procedure above, we can understand what UML2.0's description of the use-case execution is (section 3.2); It describes what is happening in *the virtual world* (Fig.2(a)). Actually, in the virtual world everything happens only under the designer's control; he decides everything including which use case to simulate, which message an actor sends to the subject, which action the subject performs. However, those who watch only the virtual world could not recognize the designer's control and therefore they could assume that actors were active agents and the subject could execute of itself.

4.4 Use-case diagram

A use-case diagram is very popular but its meaning has been left ambiguous. Let us analyze it to clarify what it really means.

UML up to 1.5 explains that a solid line connecting an actor icon and a use-case icon in a use-case diagram means that the actor and the use case communicate. [5] This means that a use-case diagram is nothing but an embodiment of the ACU conjecture. Now that UML2.0 effectively abandoned it, they could have abolished the use-case diagram as well. If they want to retain the diagram, they have to give it a different interpretation.

This triggers us to analyze a use-case diagram. Let us try to answer these questions:

[5] This description is deleted in UML2.0.

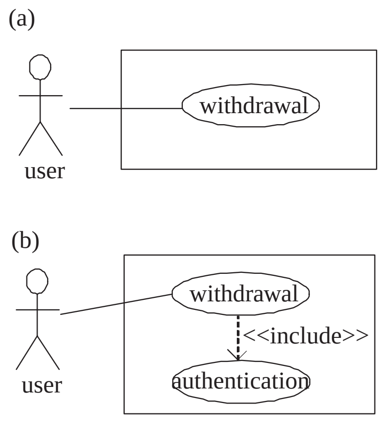

Fig. 4. Simple use-case diagram

- What is a use-case diagram?
- What does it represent?
- What does a connecting line between an actor and a use-case icon represent?

In the simulation procedure stated in section 4.3, we confined ourselves to the case of a single primary use case. In order to complete the use-case simulation procedure, let us here analyze a more general case where a use-case diagram consists of many use cases.

When we draw more than one use case in a use-case diagram, some of the use cases may look like connected through fragment use cases. Because a use case does not have any interaction with other use cases [11](section 16.3.6), we can consider, as far as the simulation of a particular use case is considered, only those use cases (use-case icons, to be precise) that are led by a primary use case; i.e., a primary use case together with its relevant fragment use cases (if any) that are connected by include or extend relationships directly or indirectly to the primary one.

Let us first take a simple use-case diagram that contains a single ellipse icon, i.e., a primary use case (Fig. 4(a)). People might think the icon represents a use case, but we doubt it. Now, let us add another ellipse icon representing a fragment use case to the diagram and connect it to the first use case with an include relationship ((b)). Here are two additional questions:

- Does the fragment use case also represent a use case just like the first one does?
- Does the first one really represent a use case?

We can understand that a fragment use case, be it an addition or extending use case, is part of the behavior description of a use-case object [3]. Extending this idea, we find it is appropriate to think that the first use case and its fragment use case altogether represent the behavior description of a use-case object, *not* *that each ellipse icon represents a use case.* Extending this idea further, we find that actors and a subject that appear in a use-case diagram are also part of the behavior description because they appear in it. Therefore, we now know that a use-case diagram represents the internal structure of a use-case's behavior description; a use-case diagram consists of *actors and a subject* as well as a primary use case and fragment use cases (if any).

Now the questions can be answered. We can explain a use-case diagram as follows.

- A use-case diagram is a special purpose diagram [6] that shows the internal structure of the behavior description of a use-case object.
- An ellipse icon represents whole or part of the behavior description of a use-case object.
- A solid line connecting an actor icon and a use-case icon shows that the actor appears in the behavior description the use-case icon represents.
- A primary use case and its relevant fragment use cases altogether represent the behavior description of a use-case object.
- A general use-case diagram is a superimposition of its relevant use-case objects.

In the definition above, we intentionally use the term "use-case object" instead of "use case." This is because the term "use case" has been carelessly used and therefore is ambiguous; it may mean a "whole" use case, i.e., a use-case object, or part of behavior description that is represented by an ellipse icon in a use-case diagram.

4.5 Completion of the use-case simulation procedure

The definition of a use-case diagram above leads us to solve the problem we met in section 4.3, i.e., how to perform simulation when we are given a use case that has fragment use cases. Generally, when we are given a use case, irrespective of whether it contains fragment use cases or not, we read in the behavior description of all the primary and fragment use cases relevant to the use case. This is because we have to preprocess the behavior description; to acquire all the extension points and their conditions before starting simulation.

Therefore, we add a preparatory process to step 2 of the simulation procedure (section 4.3). Step 2 becomes like this:

2. The use-case object reads in the behavior description of all the primary and fragment use cases relevant to the use-case object to be simulated, and then preprocesses it. Then the use-case object makes a list of messages that the primary actor can send to the subject based on its behavior description, and sends it back to the designer.

We did not notice the process because we assumed a simple use case in section 4.1.

[6] UML2.0 explains that a use-case diagram is a special case of a class diagram. However, a specific use case has to be modeled as an object (section 4.1) and therefore a use-case diagram can not be a class diagram.

5 Conclusion and future work

The paper first points out that UML2.0 recently made a correction which effectively means that they finally abandoned the *ACU conjecture* [3], admitting it was wrong. The ACU conjecture is a fallacious statement that makes people believe that actors call operations of a use case. It was embedded in OOSE and implicitly employed by UML. The paper states what defects the ACU conjecture caused to the UML's use-case specification, and then, to solve the problems, it makes a model of a designer's simulating a use case, derives a use-case class/object, and specifies its static and dynamic properties. In relation with this, it clarifies the meaning of a use-case diagram that has been left ambiguous.

UML's use-case specification has many a problems; too many for a single or two conference papers to cover all of them. In fact the current paper and its preceding one [3] putting together had to leave some items untouched or could not discuss wholly; some of them are the extend relationship semantics and the analysis of why the problems were made.

The UML people in charge of the use-case specification are very slow in getting rid of the problems and rectifying the specification as it should be. Even though, we could still have a hope, seeing that they have made two good dicisions since when UML was started. The first one came with UML1.3; they changed the include and extend relationships that used to be 'a kind of' generalization into stereotypes of the dependency relationship. The abandonment of the ACU conjecture that came with UML2.0 is the second one. It is much more important than the first one, even if they do not understand its significance, because the ACU conjecture has caused the serious defects on the use-case specification. Now we sincerely hope that in the near future the use-case specification will totally be revised by accepting the real-world model of use-case simulation (Fig.2) as the basis of use-case formalization.

References

1. G. Booch, Object-Oriented Analysis and Design with Applications, 2nd edition, Addison-Wesley, Menlo Park, CA, 1994.
2. S. Isoda, "Object-Oriented Real-World Modeling Revisited." The Journal of Systems and Software, vol.59, no.2, pp.153-162, 2001.
3. S. Isoda, "A Critique of UML's Definition of the Use-Case Class," Proceedings of 6th International Conference on the Unified Modeling Language, pp.280-294, 2003.
4. I. Jacobson, M. Christerson, P. Jonsson, and G. Oevergaard, Object-Oriented Software Engineering — A Use Case Driven Approach, Addison Wesley, Reading, 1992.
5. OMG UML Partners, Unified Modeling Language v. 1.0. Document ad/97-01-14, Object Management Group, 1997.
6. OMG UML Partners, Unified Modeling Language v. 1.1. Document ad/97-08-11, Object Management Group, 1997.
7. OMG UML Revision Task Force, OMG Unified Modeling Language Specification v. 1.2. Document ad/98-12-02, Object Management Group, 1998.
8. OMG UML Revision Task Force, OMG Unified Modeling Language Specification, v. 1.3. Document ad/99-06-08, Object Management Group, 1999.

9. OMG UML Revision Task Force, OMG Unified Modeling Language Specification, v. 1.4. Document formal/01-09-67, Object Management Group, 2001.
10. OMG UML Revision Task Force, OMG Unified Modeling Language Specification, v. 1.5. Document formal/03-03-01, Object Management Group, 2003.
11. OMG UML Revision Task Force, Unified Modeling Language: Superstructure, version 2.0, Final Adopted Specification ptc/03-08-02 Object Management Group, 2003.
12. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, 1991.
13. J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, Reading, Massachusetts, 1999.