From Informal to Formal Specifications in UML

Martin Giese and Rogardt Heldal

Chalmers University of Technology
Gothenburg, Sweden
{giese|heldal}@cs.chalmers.se

Abstract. In this paper, we consider a way of bridging informal and formal specification. Most projects have a need for an informal description of the requirements of the system which all people involved can understand. At the same time, there is a need to make some of the requirements more formal. We present a way to relate informal requirements, in form of use cases, to more formal specifications, written in the Object Constraint Language (OCL). Our approach gives the customers of software systems a way of guiding the development of formal specifications. Conversely, the formal specification can improve the informal understanding of the system by exposing gaps and ambiguities in the informal specification.

1 Introduction

The development of software systems using the Unified Modeling Language (UML) [14, 13] has become the *de facto* standard for modeling object-oriented software systems. There are several reasons for this: it is relatively easy to understand and learn, it permits several views of software systems, and it gives a good overview of the software's architecture.

The simplicity of UML has its cost: it is less precise than many other specification languages, for example that of the B-method [1]. Some of the strength of these other specification languages can be obtained by adding Object Constraint Language (OCL) [13, 17] constraints to UML models. The added precision of a formal specification can greatly enhance the quality of the produced software. Tools like the KeY system [2] can be used to assist the authoring of OCL constraints, to check their consistency, and even to verify that an implementation adheres to the constraints.

On the other hand, it is not at all clear at which point in the development process OCL constraints should be written. Who is going to add them to the UML diagrams: customers, analysts, or designers? Who should understand OCL? This missing integration into the development process might be a reason why OCL is hardly used in industry.

We believe that the customers of the software system to be built need to be one of the driving forces behind producing OCL constraints. After all, only the customers can know what behavior they want from the system. However to expect the customer or even all of the developers to know OCL is unrealistic.

Thus, there is a need for an informal description that everybody involved in the project can understand.

This is indeed the purpose of use cases. Use cases describe the system behavior in an informal way, usually using only text. In contrast to some research [7] which tries to add formal specifications directly to use cases, we will stick to text, because we believe that there needs to be an informal description of the system *somewhere*. Even for informal, textual use cases however, templates are used to structure the descriptions, e.g. the base use case description template in the Rational Unified Process (RUP), and these templates normally include pre- and post-conditions. In contrast to the OCL constraints usually attached to operations in a class diagram, these textual constraints can be understood by and negotiated with the customer.

In this paper, we investigate the *relationship* between the informal, textual pre- and post-conditions of use cases and the formal OCL pre- and post-conditions of operations in the class diagram. Apart from being a step from informal to formal specification, this is also a step from the analysis phase (use cases) to the design phase (sequence and class diagrams). The primary goal will be to make sure that all the behavior specified together with the customer in the use cases is also captured by the formal OCL specification attached to operations in the class diagrams.

Another important aspect when going from informal to formal description is the need to become more precise. It is often necessary to add contextual information in a formalization that was implicitly assumed, or simply forgotten in the textual use case description. We will consider ways to cope with such additional information, either by enhancing the use case text or at by acknowledging contextual information as such, thus making the formalization process more transparent. In other words, not only does the informal specification help in writing the formal one, but the formalization can also help in improving the informal one. Including customers in the development of the formal specification in this way gives them more power, but also more responsibility. We believe that it can add significantly to the value of a formal specification.

We should note that our method does not, in general, give OCL specifications for all operations in the model. In fact, there will usually be two categories of constraints: The first kind directly reflects the customer's requirements, and it is only this kind we consider in this paper. The other kind relates to the internal structure of the system, and is dependent on the design. Conditions on the type and range of method arguments belong to this category, as well as specifications of auxiliary methods used to implement the required functionality. Although this second kind of constraints is also important, they cannot easily be linked to the information in use cases.

This work is concerned with three types of UML diagrams: use case diagrams, state chart diagrams and class diagrams. Before we explain the approach itself, we briefly discuss use cases and state chart diagrams. In Sect. 2, we present the theoretical basis of our approach, which we then use in a case study in Sect. 3. The case study suggests a refinement of the theory which is presented in Sect. 4.

1.1 Use Case diagrams

Use cases were invented by Jacobson [10]. They are part of the UML [13] and supported by UML tools, and widely used in the industry. There are a number of books dedicated only to use cases and countless papers discussing possible interpretations of use cases. Some of the more recent papers which have influenced our understanding of use cases are [5, 6, 9] and the book [3].

The informal nature of use cases allows more or less any textual description of a system to be categorized as a use case. This is both a strength and a weakness: on one side they can be used in many contexts, but on the other side their informal nature creates problems like choosing the right level of abstraction and ensuring inconsistency both inside and between use cases. Writing good use cases is a far from trivial task, but they can make a valuable contribution to the understanding of a project.

In our work, the style of writing use case descriptions is less important, as long as they create a common understanding between customers and developers, in such a way that sensible pre- and post-conditions can be obtained. In our case study, we describe the flows of the use cases in an abstract way, keeping out details like the user interface, in a style often referred to as essential use cases [5, 4]. This style of use cases was enough to obtain the required pre- and post-conditions. The text of pre- and post-conditions should be precise, but there is no need for formal or mathematical notation at this level.

Alternative flows [3] are crucial for our usage of use cases. There is nearly always more than one path through a use case, and the user should specify what happens in these alternative scenarios. Technically, alternative flows may be split into separate use cases, but we think having a use case for what is actually an error condition goes against the intention of use cases as satisfying a goal.

On the other hand, we do not explicitly consider 'include' and 'extends' [3] relationships for the time being. They may be treated by a transformation to 'flat' use cases that moves the extensions, resp. inclusions into the extended, resp. including use case.

1.2 State chart diagrams

We want to consider all scenarios, i.e. all possible paths through a use case. To describe all paths, we use a state chart diagram [13]. We are only interested in a limited part of the state chart: the final call event for each path, together with all the branching conditions for that path. In Sect. 2, we will show that this information suffices to capture the properties we want to hold.

Sequence or collaboration diagrams are commonly used to illustrate a scenario of a use case. The problem with these diagrams for our purpose is that they generally show only one path through a use case.[1]

Even though we use state chart diagrams to model all possible paths through a use case, it is useful to start with a sequence diagram for some of the important

[1] It might be possible to use generic interaction diagrams[16] instead, including conditional behavior and iteration.

scenarios. This simplifies the process of creating the state chart diagram, since it helps to identify the important calls.

2 From Use Cases to Operation Post-Conditions

We start from a use case UC equipped with a post-condition $Post_{UC}$ formulated in natural language. We will not consider pre-conditions of use cases in this work, because the usual understanding of a pre-condition is that the post-condition need only be guaranteed if the pre-condition is met before the use case. In other words, having "the customer entered the correct PIN" as a pre-condition means that we will not specify what happens if the customer enters a wrong PIN. This is a much weaker statement than saying that the use case cannot be executed if the pre-condition is not met. It is easy to get confused by these two readings. To avoid this trap, we leave use case pre-conditions completely empty.

In the implementation, we expect the use case UC to correspond to a sequence of calls into the system, which we model as a single system object s. This is the only object we will talk about, and all OCL constraints are to be understood in the context of s, i.e. `self` $= s$. Depending on what exactly the user does, the sequence of operations might be different. Whatever the sequence is however, $Post_{UC}$ should finally hold, so ultimately it needs to be guaranteed by the post-condition of the last operation.

A post-condition $Post_{UC}$, given in natural language by the customer, will probably refer not only to the final state. It is natural to refer to the *sequence* of events, like in "when the customer entered the wrong PIN three times, ...". We will capture the possible sequences of events in a state chart which we attach to the system object s. Paths through the state chart correspond to possible sequences of events. The important aspects of this state chart are the events that correspond to operation calls from the outside, i.e. from the agent. The state chart can also contain conditions which refer to the arguments of these events.

The goal of this section is to match the post-conditions of the operations and conditions from the state chart to the use case post-condition. While the use case post-condition $Post_{UC}$ is an informal artifact of the analysis phase, and thus necessarily informal, the conditions in the state chart and the post conditions of the operations are written in OCL as part of the design.

Given a state chart, we can collect the set Σ of all paths π from the initial state to some final state. Depending on the state chart, there may be infinitely many of these. For every path π with events $op_1(args_1), \ldots, op_k(args_k)$ let

$$final(\pi) := op_k$$

be the last operation called. For correctness, we want every final state that the post condition of this method allows to also be permitted by the post condition of the use case. As a first approximation, we want the post-condition of the last operation to *imply* the post condition of the use case:

$$Post_{final(\pi)} \rightarrow Post_{UC}$$

Now, as $Post_{UC}$ includes requirements for all possible different paths, this implication will usually hold only if we add information about the path taken. As OCL does not allow to refer to the values of attributes in different states, we have to impose the restriction that the conditions (guards) in the state chart do not refer to attributes, and that the names for event arguments are all distinct. Let $Cond(\pi)$ be the conjunction of the conditions encountered on the path π, together with an expression of the form `oclInState@pre(x)` which expresses the state before the call of the last operation. What we want is then the weaker statement

$$Cond(\pi) \rightarrow (Post_{final(\pi)} \rightarrow Post_{UC})$$

or equivalently $(Cond(\pi) \wedge Post_{final(\pi)}) \rightarrow Post_{UC}$. Finally, this should be the case for all paths $\pi \in \Sigma$:

$$\bigwedge_{\pi \in \Sigma} \big((Cond(\pi) \wedge Post_{final(\pi)}) \rightarrow Post_{UC}\big) \qquad (I)$$

Note that we do not require $Post_{UC}$ to be given in formal syntax. Instead, we will consider all paths π, *formally* calculate the conjunction $Cond(\pi) \wedge Post_{final(\pi)}$ and then *informally* ask ourselves whether the formal post-condition does enough and whether all it does is intended. We shall see in the case study that it is valuable to ask this question, even without fully formal reasoning.

So far, we have considered only correctness, in the sense that every behavior allowed by the formal specification is also allowed according to the informal one. We shall see that it is also interesting to investigate the opposite direction, namely whether every behavior the informal specification allows is still allowed by the formal one. This would be the case if

$$Post_{UC} \rightarrow Post_{final(\pi)}$$

holds on every path, so

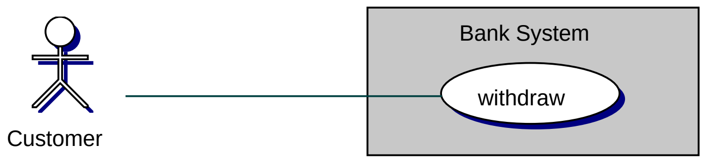
$$\bigwedge_{\pi \in \Sigma} \big((Cond(\pi) \wedge Post_{UC}) \rightarrow Post_{final(\pi)}\big) \qquad (II)$$

Typically, as we will see in the case study, information is added when an informal specification is formalized. This might be domain knowledge that is taken for granted by the authors of the informal specification, or even simple everyday knowledge (*red* $\neq$ *green*). In such cases, requiring (II) to hold would make the informal post specification $Post_{UC}$ unnecessarily verbose. After all, $Post_{UC}$ should be something that can be negotiated with a customer. We could however keep track of the added information $Extra(\pi)$ (which might be formal or informal) for every path and require

$$\bigwedge_{\pi \in \Sigma} \big((Cond(\pi) \wedge Post_{UC} \wedge Extra(\pi)) \rightarrow Post_{final(\pi)}\big) \qquad (III)$$

This would ensure that every requirement expressed in the formal post condition could be traced either to $Post_{UC}$ or to $Extra(\pi)$.

3 Case Study



- If the customer entered the PIN stored on the Card, and the customer's balance was greater or equal to the requested amount, then the customer got the requested amount and the amount was deducted from the balance.
- If the customer entered the wrong PIN three times, the card was retained.
- If the customer requested too much money, the card was returned to the customer.

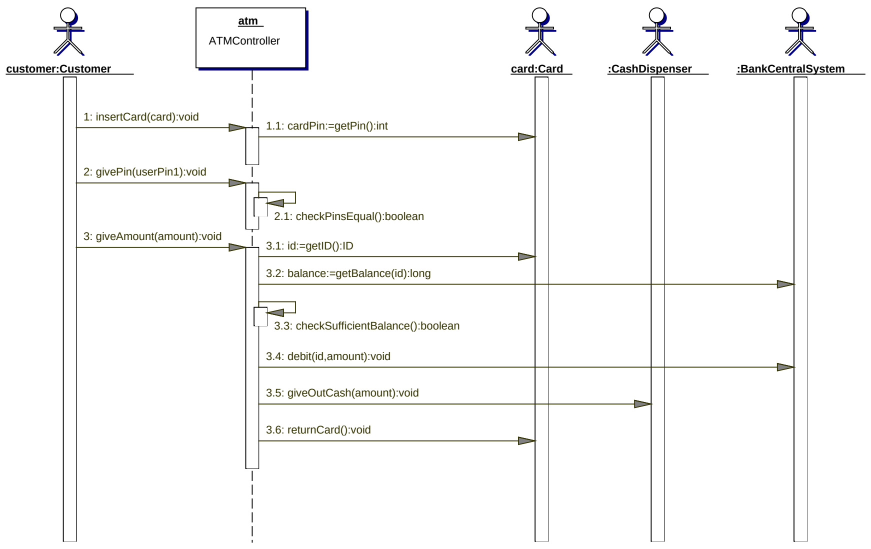
Fig. 1. The 'withdraw' use case for the ATM example, with its textual post condition

We take a simple automated teller machine (ATM) scenario as a case study. We will look at only one use case, named "withdraw", in which the customer attempts to withdraw money from a bank account, see Fig. 1. For this, a PIN has to be entered and the requested amount must not exceed the balance on the customer's bank account. If the wrong PIN is entered three times, the card should be retained.

We can phrase these requirements as a *post-condition* for the withdraw use case, as shown in Fig. 1. Note that this postcondition is stated in natural language, in accordance with the informal nature of Use Cases. Also, it refers to the course of events in the past tense, that is from the perspective of having gone through the use case.

The normal flow of control is illustrated in Fig. 2. The whole sequence of events is initiated by the user presenting the card to the ATM, which is modeled by the call `insertCard(card)` to the system object `atm`. To be able to check the PIN entered by the user, the card is queried for the correct PIN, `cardPin`.

We neglect the modeling of other ATM operations, for example balance inquiry. As is customary, we do not model the control of the user interface, which would of course request the user to enter the PIN number at this point. We just assume that the completed PIN is given to the ATM controller with the call `givePin(userPin)`. For the normal flow of control, the subsequent call of `checkPinsEqual()` would return `true`.



customer:Customer
atm
ATMController
card:Card
:CashDispenser
:BankCentralSystem
1: insertCard(card):void
1.1: cardPin:=getPin():int
2: givePin(userPin1):void
2.1: checkPinsEqual():boolean
3: giveAmount(amount):void
3.1: id:=getID():ID
3.2: balance:=getBalance(id):long
3.3: checkSufficientBalance():boolean
3.4: debit(id,amount):void
3.5: giveOutCash(amount):void
3.6: returnCard():void

Fig. 2. Sequence diagram for the normal flow of the "withdraw" use case

Now, the user interface would be updated to ask the customer for the desired amount, which is passed to the controller in the call `giveAmount(amount)`. This amount needs to be compared to the customer's bank account balance. For this purpose, an identification `id` of the bank account is fetched from the card, and used to query `balance` from the bank's central system. After checking that the balance is sufficient, the `CashDispenser` unit is instructed to deliver the required amount to the user. Finally, the card is returned to the user.[2]

Comparing the post condition in Fig. 1 with this sequence diagram, one immediately notices the shortcoming of the latter: it does not say anything about the alternative flows of control. What happens if the call to `checkPinsEqual` or `checkSufficientBalance` returns `false`? For the normal flow of control, it is obviously the operation `giveAmount` which is responsible for ensuring the post-condition of the use case, simply because it is the last operation called. But what if the user enters the wrong PIN three times? Then the customer will never be

[2] It can be questioned where the call of `returnCard()` should be directed. We chose `card`, because the controller obviously communicates directly only with the card reading device and not with the card itself. If `card` stands for the card reader, than it is the right goal for the `returnCard()` call. One might also argue that the initial `insertCard()` call should then come from the card reader and not from the `customer` actor. However, this confuses more than it helps, and it is not really essential to our investigation.

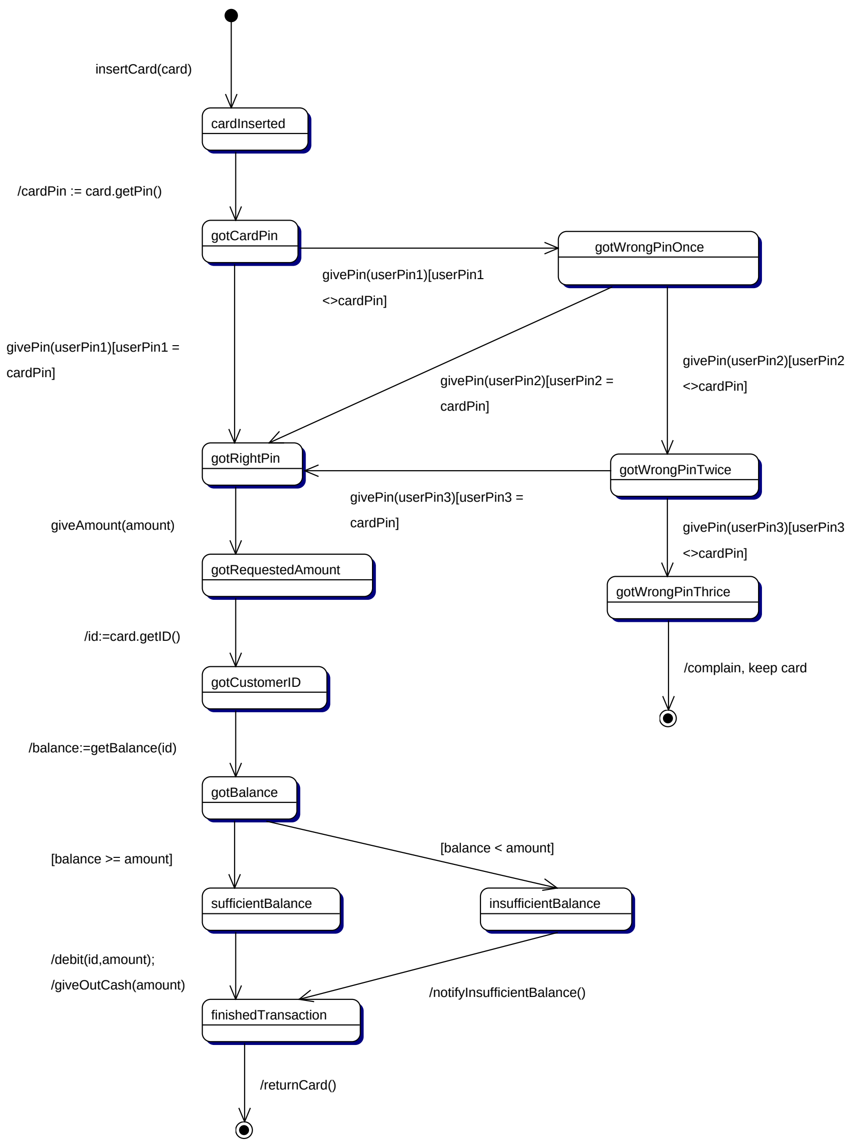


Fig. 3. State Chart for the “withdraw” use case of the ATM example

asked to enter an amount. In that flow of control, the `giveAmount` operation will never be called, so it can no longer be responsible for ensuring the post-condition.

In order to cope with all the different possible flows of control, we attach a state chart to the system object `atm`, which we use essentially as a condensation of all sequence diagrams for all the flows of control of this use case. The following parts are captured here:

1. The *events* which correspond to the calls into the system in the sequence diagram. These are the calls `insertCard(card)`, `givePin(userPin)` and `giveAmount(amount)`. Some transitions have no event associated to them. This is convenient to capture a sequence of actions (e.g. get the `id`, get the balance, check the balance) triggered by a single event. In some paths, the `givePin` method gets called several times, so we call the argument `userPin1`, `userPin2`, etc. to ease later reference.
2. The *conditions* which determine which transitions will be chosen, depending on the data coming with the events. These are the items in square brackets, like `[userPin=cardPin]` or `[balance<amount]`.[3]

Due to the structure of the state chart, we can now see at a glance that there are at most 7 possible flows of control, one for each path from the initial state to some final state. In fact, one can check that the conditions on each of these paths are non-contradictory, so there are actually exactly 7 flows of control. One also sees immediately that the last event, and thus the last operation called for 6 of these is `giveAmount` and for one flow it is `givePin`.

According to the plan outlined in Sect. 2, we want the post condition of the last operation on every path, together with the accumulated conditions, to imply the post condition of the use case. So the post condition of `givePin` will have to take care of the situation that the wrong PIN was entered three times, while the post condition of `giveAmount` will cover all other cases.

To start with the simpler case, here is a possible OCL formulation of a post condition for `givePin`:

```
Context ATMController::givePin(userPin:int):void post:
  if (userPin = card.getPin()) then
    oclInState(gotRightPin)
  else if (oclInState@pre(gotCardPin)) then
    oclInState(gotWrongPinOnce)
  else if (oclInState@pre(gotWrongPinOnce)) then
    oclInState(gotWrongPinTwice)
  else
    not card^returnCard()
```

Note that we use the expression `card.getPin()` for the PIN stored on the card. Using an operation in an OCL constraint requires it to be free from

[3] In this diagram, the conditions unambiguously determine which transitions will be taken. This need not be the case in every diagram. Our method applies without changes even for indeterministic state charts.

side effects, i.e. a *query*. We assume that this is no problem for `getPin()` of `Card`. Finally, we use the 'has Sent' operator `^` of OCL 2.0 to express that the `returnCard()` method of `card` has not been called.

The post condition of `giveAmount(amount:long)` has to take care of all paths through the state chart where the correct PIN was eventually entered. Here is a possibility:

```
Context ATMController::giveAmount(amount:long) post:
  if ( amount <= bank.getBalance(card.getID()) ) then
        cashDispenser^giveOutCash(amount)
    and   bank.getBalance(card.getID())
        = bank.getBalance@pre(card.getID()) - amount
    and card^returnCard()
  else
        not cashDispenser^giveOutCash(?)
    and   bank.getBalance(card.getID())
        = bank.getBalance@pre(card.getID())
    and card^returnCard()
```

We see that there are two cases, according to whether the required amount is below the bank account balance or not. The methods `getID()` of `Card` and `getBalance(id:ID)` of `BankCentralSystem` are both queries.

Now we can investigate the relationship of these post conditions to that of the use case, given in Fig. 1. We will only do this for four of the paths:

1. The "normal flow" with no wrong PIN entered and a sufficient balance.
2. First PIN entered is correct, but insufficient balance.
3. First PIN entered is incorrect, then like in normal flow
4. Three wrong PINs entered.

The three remaining flows are very similar to these four.

Normal flow of control. Let π_1 be the normal flow of control. The last operation $\mathit{final}(\pi_1)$ is `giveAmount()`. The conditions gathered on the normal path are

$$\mathit{Cond}(\pi_1) \quad = \quad (\texttt{userPin1 = cardPin} \land \texttt{balance >= amount})$$

Using the logical tautology $A \land (\texttt{if}\ A\ \texttt{then}\ B\ \texttt{else}\ C) \leftrightarrow A \land B$, we can simplify the conjunction

$$\mathit{Cond}(\pi_1) \land \mathit{Post}_{\mathit{final}(\pi_1)}$$

to

```
      userPin1 = cardPin
  and balance >= amount
  and cashDispenser^giveOutCash(amount)
  and   bank.getBalance(card.getID())
      = bank.getBalance@pre(card.getID()) - amount
  and card^returnCard()
```

One can now easily check that this implies *all three* of the points expressed informally in Fig. 1: the first point because the actions prescribed (give out cash, deduce amount from balance) have taken place. The other two, because the conditions ("If the customer...") of $Post_{UC}$ are contradicted by the gathered conditions $Cond(\pi_1)$. Of course, they will receive proper treatment when we look at the other flows of control. One difficulty here is that expressions using `@pre` refer to the state before the execution of `giveAmount`, instead of that before the use case. We have to convince ourselves that the balance has not been changed by the previous operations. Alternatively, one might introduce a means of referring to the state before the use case in the post condition. How to avoid this problem is a topic for future research.

So, for the normal flow of control, the post condition of `giveAmount` is powerful enough: if the implementation of that operation fulfills the post condition, then it will also correctly fulfill the post condition of the use case. But there is another aspect: in the informal post condition $Post_{UC}$, there was no mention of the card being returned. The clause `card^returnCard()` is something we added when we formulated the OCL constraints. Maybe the author of the informal specification forgot about this information, in which case it should probably by added to $Post_{UC}$. But it might also be information that may be taken for granted, and would only encumber the informal specification. It is hard to decide in general what to do with such added information, but our methodology does not depend on the choice taken. If the instruction to return the card is not added to $Post_{UC}$, it might still be added to $Extra(\pi_1)$, and then (*III*) holds, as one easily sees. The main point here is that a careful comparison of $Cond(\pi_1) \wedge Post_{final(\pi_1)}$ and $Post_{UC}$ can reveal such extra information added during the formalization, and that tool support could help to keep track of it.

Correct PIN but insufficient balance. Let π_2 be the flow of control where the correct PIN is entered at the first attempt but the amount requested is too high. The final operation is still `giveAmount`, but the gathered conditions are now:

$$Cond(\pi_2) \quad = \quad (\texttt{userPin1 = cardPin} \wedge \texttt{balance < amount})$$

For $Cond(\pi_2) \wedge Post_{final(\pi_2)}$, we get

```
    userPin1 = cardPin
and balance < amount
and not cashDispenser^giveOutCash(?)
and   bank.getBalance(card.getID())
    = bank.getBalance@pre(card.getID())
and card^returnCard()
```

Now the condition of the first two items in $Post_{UC}$ are not fulfilled, but that of the third ("If the customer requested too much money") is. And indeed, the conjunction of conditions and post condition states that the card was returned. Again, there is additional information that was missing in $Post_{UC}$, namely that no cash was delivered to the customer, and that the balance on the bank account did not change.

Correct PIN at second attempt. Let π_3 be the flow of control where the correct PIN is entered at the first attempt but the amount requested is too high. The final operation is `giveAmount` again, the conditions are

$$Cond(\pi_3) \quad = \quad \left(\begin{array}{l} \quad \texttt{userPin1 != cardPin} \\ \wedge\ \texttt{userPin2 = cardPin} \\ \wedge\ \texttt{balance >= amount} \end{array} \right)$$

$Cond(\pi_3) \wedge Post_{final(\pi_3)}$ is

```
    userPin1 != cardPin
and userPin2 = cardPin
and balance >= amount
and cashDispenser^giveOutCash(amount)
and   bank.getBalance(card.getID())
    = bank.getBalance@pre(card.getID()) - amount
and card^returnCard()
```

Except for the slightly changed conditions, this is the same as for the normal flow π_1. Indeed, $Post_{UC}$ is implied for the same reason as in π_1, as it says "If the user entered the PIN on the card..." and does not specify in which attempt the correct PIN was entered. We will come back to this observation in Sect. 4.

Wrong PIN three times. We call π_4 the flow of control where the customer enters the wrong PIN three times in a row. The final operation $final(\pi_4)$ is the fatal third `givePin`. The gathered conditions are

$$Cond(\pi_4) \quad = \quad \left(\begin{array}{l} \quad \texttt{userPin1 != cardPin} \\ \wedge\ \texttt{userPin2 != cardPin} \\ \wedge\ \texttt{userPin3 != cardPin} \end{array} \right)$$

$Cond(\pi_4)$ includes the information that `oclInState@pre(gotWrongPinTwice)` is true, so $Cond(\pi_4) \wedge Post_{final(\pi_4)}$ reduces to

```
not card^returnCard()
```

This obviously implies $Post_{UC}$, the second point being the relevant one in this case. Note that we have to assume that the condition "If the customer requested too much money" is not met. In fact, the customer did not get a chance to request any amount of money at all. That there is a problem here can be seen by considering the condition "If the requested amount lies below the customer's balance", which on one hand looks like the negation of the one above, but on the other hand might *also* be considered false in the present case where no amount was requested. One sees that the handling of undefined values is by no means trivial.

There are three more paths in the state chart, but they are very similar to the ones discussed. We think that this case study shows the validity and relevance of the theoretical concepts developed in Sect. 2. There is however a question of practicality remaining, which we will discuss in the following section.

4 Grouping Similar Paths

In our case study, there were only seven possible paths through the state chart. But in general, there might be very many, even infinitely many paths. In that case, it becomes impractical to consider each path separately. On the other hand, we already noticed in the previous section, that similar paths can require similar reasoning. It is therefore worthwhile to divide the set of possible paths into partitions which require similar reasoning. How this is best done is research in progress, but we want to give some ideas in this section.

We first remark that the post condition $Post_{UC}$ is actually composed of a number of clauses, each of which contains some condition on the path taken. The structure in the case study is

$$Post_{UC} = (C_1 \to P_1) \wedge (C_2 \to P_2) \wedge (C_3 \to P_3)$$

To show condition (I) of Sect. 2, it is sufficient to show

$$\bigwedge_{\pi \in \Sigma} \big((Cond(\pi) \wedge Post_{final(\pi)} \wedge C_i) \to P_i \big)$$

for $i = 1, 2, 3$. We now have three conditions to show, but for each of them, many paths are immediately excluded by C_i. For instance

$$C_3 = \text{“If the customer requested too much money”}$$

is only fulfilled on the three paths that go through the state ‘insufficientBalance’ in the state chart. Further, note that the last operation $final(\pi)$ is `giveAmount` on all these paths and that the condition `balance < amount` is on all three paths. It is thus sufficient to show

$$(\texttt{balance < amount} \wedge Post_{\texttt{giveAmount}} \wedge C_3) \to P_3$$

which can easily been seen to hold, because neither $Post_{\texttt{giveAmount}}$ nor P_3 require the information about the PIN verification process that was lost in taking only the conditions that were present on all three paths.

We believe that this approach can be generalized to reduce the number of paths that need to be considered in many practical cases.

5 Related Work

Grieskamp and Lepper [7] used executable Z to describe use cases. The benefit is that this gives a complete formal specification which can even be run. This is good for testing use cases. The drawback is the lack of informal description of the system to be made. It is hard for people without formal specification skills to understand the use cases. This makes use cases difficult to use as communication between customers and developers.

There have been a number of papers about relating the formal specification language B to UML [11, 12, 15]. In the papers [11, 15] the focus is on deriving a B model from a UML class description. Neither of these papers has a customer focus. The work of Levy, Marcano and Souquières [12] considers the mapping from informal to formal specification in a way similar to ours, using the B specification language instead of OCL. It is a short paper which gives a very brief overview of their process. They have not made clear the relationship between informal and formal specification, and no theory is stated in the paper.

6 Conclusion

Both informal and formal specification has its merits. A formal specification is more precise and therefore better suited for supporting the production of code, testing and proving. On the other hand, formal specifications are harder to read, and therefore informal specifications are needed. But for both the informal and formal specification to make sense, there has to be consistency between them – at least in those places where the system is described in both ways. In this paper, we have shown a process of relating informal specifications on use cases to formal OCL specifications on operations. Without a process like the one presented in this paper it is hard to justify the forces behind the writing of OCL constraints related to the customer's requirements. We have shown how to obtain formal specifications from informal ones, but also how the formal specification can improve the informal one.

One important field for further research is to find ways to handle state charts with infinitely many paths, extending the ideas from Sect. 4. Also, ways to handle `@pre` in post conditions correctly need to be investigated.

We also plan to provide a certain extent of tool support within the KeY system [2], to help developers in documenting and tracing the process of formalizing the customers requirements.

Finally, we want to try to automatically translate OCL constraints back to natural language: To automatically go from informal descriptions to formal descriptions is very hard. The other way around is more feasible, since a formal description is unambiguous. There is work by Hähnle, Johannisson, and Ranta [8] in going from OCL constraints to text descriptions. Having OCL pre and post conditions make it possible to obtain better textual description for use cases. The text produced might be used to validate the original description – even the customer can take part in this process since no OCL constraints are involved. Furthermore, the original text might be strengthened or replaced by the new text. This kind of round-trip engineering, involving a formally untrained customer on one side and a precise, formal specification on the other, would add tremendously to the value of formal specifications.

Acknowledgments

We would like to thank Wolfgang Ahrendt, Peter Gammie, and the anonymous referees for their helpful comments on drafts of this paper.

References

1. J. R. Abrial. *B-Book.* Cambridge Univ. Press, 1996.
2. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 3, 2004. To appear.
3. F. Armour and G. Miller. *Advanced Use Case Modeling.* Addison-Wesley, 2001.
4. A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2001.
5. L. L. Constantine and L. A. D. Lockwood. *Structure and Style in Use Cases for User Interface Design*, chapter 7, pages 245–279. Object Technology. Addison-Wesley, 2001.
6. G. Génova, J. Llorens, and V. Quintana. Digging into use case relationships. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 115–127. Springer Verlag, 2002.
7. W. Grieskamp and M. Lepper. Using use cases in executable Z. In *ICFEM*, pages 111–120, 2000.
8. R. Hähnle, K. Johannisson, and A. Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 233–248. Springer Verlag, 2002.
9. S. Isoda. A critique of UML's definition of the use-case class. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003*, volume 2863 of *LNCS*, pages 280–294. Springer Verlag, 2003.
10. I. Jacobson, M., Christerson, P. Johnsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.
11. R. Laleau and F. Polack. Coming and going from UML to B : A proposal to support traceability in rigorous IS development. In *ZB'2002 – Formal Specification and Development in Z and B*, pages 517–534. Springer Verlag, 2002.
12. B. N. Levy, R. Marcano, and J. Souquières. From requirements to formal specification using UML and B. In *International Conference in Computer Systems and Technologies, CompSysTech'2002*, Sofia, Bulgaria, 2002.
13. OMG. *Unified Modeling Language Specification.*
14. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Object Technology. Addison-Wesley, 1999.
15. C. Snook and M. Butler. Verifying dynamic properties of UML models by translation to the B language. In *Proceedings UML 2000 WORKSHOP Dynamic Behaviour in UML Models: Semantic Questions*, York, October 2000.
16. P. Stevens and R. Pooley. *Using UML: software engineering with objects and components.* Object Technology Series. Addison-Wesley, 2000. Updated edition for UML1.3: first published 1998.
17. J. Warmer and A. Kleppe. *The Object Constraint Language.* Object Technology. Addison-Wesley, 2003.