USE CASE BASED REQUIREMENTS VERIFICATION
Verifying the consistency between use cases and assertions

Stéphane S. Somé, Divya K. Nair
School of Information Technology and Engineering (SITE), University of Ottawa, 800 King Edward, Ottawa, Canada
ssome@site.uottawa.ca, dnair@site.uottawa.ca

Keywords: Requirements Analysis, Use cases, Assertions, Verification.

Abstract: Use cases and operations are complementary requirements artefacts. A use case refers to operations and imposes their sequencing. Use cases templates usually include assertions such as preconditions, postconditions and invariants. Similarly operations are specified using contracts consisting in preconditions and postconditions. In this paper, we present an approach aiming at checking the consistency of each description against the other. We attempt to answer questions such as the following. Is the use case postcondition guaranteed by the operations ? Are all operations possible according to their preconditions ? We provide answers to these questions by deriving state predicates corresponding to each step in a use case, and by showing the satisfaction of assertions according to these predicates.

1 INTRODUCTION

Requirements engineering is a critical task in the software life cycle. Different studies have shown requirements as the most important source of software defects (Johnson, 2006). Moreover, finding and fixing a software defect after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phases (Boehm and Basili, 2001). It is therefore critical to perform requirements verification and validation early in the life cycle.

Several artefacts are usually used to fully specify requirements (textual list of requirements, use cases, diagrams, operation contracts, ...). One aspect of verification is to ensure the consistency of these artefacts. In this paper we focus on use cases (Jacobson et al., 1993) and assertions (Hoare, 1969).

Use cases (Jacobson et al., 1993) describe interactions involving systems and their environments. Each use case is the specification of a sequence of actions, including variants, that a system can perform, interacting with actors of the system (OMG, 2003). Use cases have become one of the favorite approaches for requirements capture.

The notion of assertion based software specification and verification originates from C.A.R. Hoare (Hoare, 1969). Assertions are logical statements about the behavior of a software component. Assertions include pre-conditions, post-conditions and invariants. The use of assertions for software specification is exemplified by B. Meyer's *design by contract* approach (Meyer, 2000).

Use cases and assertions are complementary. A use case description is usually supplemented with pre-conditions, post-conditions and invariants. A use case pre-condition specifies the necessary state for the use case application, the post-condition specifies the resulting state at the end of the use case, and the invariant states minimal guarantees throughout the use case. Use case operations are also elaborated using assertions. For instance, an operation may be specified using a *contract*, which consists on a precondition stating a necessary condition for the operation, and a post-condition stating the condition resulting from the operation. Software development approaches such as Fusion (Coleman et al., 1994) and the Responsibility Driven Design approach (Larman, 2004) combine use cases and assertions.

In this paper, we present an approach for the verification of the consistency between use cases and assertions. Given a requirements specification consisting in use cases and operations elaborated with contracts, one verification concern is to determine if the sequencing of all operations as mandated by the use cases is possible according to their contracts. Another concern is to determine whether use case assertions are guaranteed by the operations. We propose a logic based approach where use case events are considered sequentially as part of scenarios.

The remainder of this paper is organized as follow. We discuss some related work in the next section. Background information related to our approach is presented in section 3. This information consists on a notation for use cases and domain operations. In section 4, we present our verification approach. Finally, section 5 concludes the paper and discusses some future works.

2 RELATED WORK

The work presented in this paper concerns use cases verification against operation contracts. A work with a similar goal is presented in (Giese and Heldal, 2004). Giese and Heldal propose an approach aiming at bridging the gap between informal and formal requirements. The informal requirements are represented as use cases written in natural language with pre and postconditions. The approach assumes a corresponding formal representation of each use case as a statechart. The postcondition of the operations in the statechart are specified using OCL (OMG, 2003). By considering each path in the statechart, the combination of the operation postconditions is checked against the informal postcondition of the use case. Our approach bares similarities with Giese and Heldal's in the way that we combine conditions and perform assertion checking. A major difference between the two approaches is that Giese and Heldal propose a manual approach, while one of our primary goals is to provide an automated approach. The necessity to develop a statechart and provide detailed OCL specification also makes Giese and Heldal approach applicable later in the development life-cycle than ours. Finally beside postconditions, we consider other assertions such as preconditions and invariants.

Another approach related to ours is proposed by Toyoma and Ohnishi (Toyama and Ohnishi, 2005). This approach proposes rules for the verification of scenario based requirements. As in our work, scenarios are described using a restricted form of natural language. The verification checks for errors such as the lack of events, extra events, and the wrong sequence of events in scenarios. Verification rules are used to specify the correct occurrence times of events as well as the time sequence among events. These rules are stored in a "rule database" and retrieved based on preconditions and post-conditions. The focus of our approach is different to Toyoma and Ohnishi's as we focus on assertions rather than events time sequence.

3 USE CASES AND OPERATIONS

Figure 1 shows an example of use case. We consider a use case as a tuple [*Title*, *Inv*, *Pre*, *Steps*, *Post*] with *Title* a label that uniquely identifies a use case, *Inv* an invariant, *Pre* a use case precondition, *Steps* a sequence of steps, and *Post* a postcondition. Each step in *Steps* includes a use case operation with an optional step condition and a set of alternatives. For instance

```
Title: cash withdrawal
Invariant: ATM is ON
Precondition: ATM is ON
STEPS
1. The USER inserts card in the ATM card slot
2. The ATM asks User for her pin
3. The USER enters her pin
4. The ATM displays an operation menu
5. The USER selects cash withdrawal operation
6. The ATM asks the withdrawal amount
7. The USER specifies the withdrawal amount
8. The ATM checks the amount entered
9. IF USER amount is okay THEN, the ATM
   updates the USER's account
10.The ATM dispenses the cash in the ATM
    cash dispenser slot
11.The ATM ejects the USER's Card
Postcondition: USER Account is updated AND
               Cash is dispensed
ALTERNATIVES
1.a.The USER Card status is not valid
 1.a.1. The ATM alerts the Security Branch
 Alternative Postcondition: USER Card is
   in card slot AND Security Branch is alerted
3.a.The USER identification is not valid
    AND The USER numbers of attempts < 3
 3.a.1.The ATM displays a wrong
       identification error message
 3.a.2.Goto Step 2.
3.b.The USER identification is not valid
    AND The USER numbers of attempts is == 3
 3.b.1.The ATM displays a wrong
      identification error message
 Alternative Postcondition: USER Card is
                           in card slot
8.a.The USER amount is not okay
 8.a.1.The ATM displays an amount not okay
       error message
 8.a.2.Goto Step 6.
```

Figure 1: Cash Withdrawal Use Case.

step *9* in use case "Cash Withdrawal" includes condition "USER amount is okay" and operation "ATM updates the USER's account". We distinguish three types of use case operations: use case inclusion directives, Goto statements as in step *3.a.2*, and instances of domain operations. A step may include alternatives. Each alternative is introduced by a condition and consists of a set of steps. An alternative may also include a postcondition.

A use case consists of a set of *scenarios* (Jacobson et al., 1993). Each scenario being a *sequence of steps* ($step_1$ - $step_2$ - $\cdots$ $step_n$) in the use case. The set of a use case scenarios includes a *primary scenario* and zero or more *secondary scenarios*. The primary scenario captures the "normal" or "most common" behavior. It is written as if everything goes right without any error. The completion of the primary scenario fulfills the goal of the use case. A secondary scenario describes an alternative outcome that may result from an error. Each secondary scenario is written by defining diverging behaviors from a specific point of a primary scenario. The set of scenarios corresponding to use case "cash withdrawal" is listed in Table 1. The

Table 1: Set of scenarios from use case "cash withrawal".

Scenario	Sequence	Type
1	1-2-3-4-5-...11	primary
2	1-1.a.1	secondary
3	1-2-3-3.a.1-2	secondary
4	1-2-3-3.b.1	secondary
5	1-2-3-..8-8.a.1-6	secondary

primary scenario is the sequence steps *1* to *11*. Examples of secondary scenarios are sequences *1 - 1.a.1* and *1 - 2 - 3 - 3.a. -2*. Notice that for secondary scenarios ending with Goto statements, the last step listed is the target of the statement.

The different assertions in a use case are as follow. A use case precondition specifies a condition[1] that must hold for the behavior defined in the use case to be guaranteed. A use case invariant specifies a condition that must hold at each step of the use case. A use case postcondition specifies a condition that holds at the end of a use case primary scenario. Each secondary scenario may specify an alternative postcondition.

We use a restricted form of natural language as a concrete syntax for use case description (Somé, 2006). For instance, simple conditions are *predicative sentences* in the form [*determinant*][2] *entity verb value*, where "*entity*" is a reference to a domain element and "*verb*" a conjugated form of to be. Simple conditions can be negated and combined using operators "AND" and "OR" to form compound conditions.

We formally define simple conditions as predicates on entities of the application domain. Each predicate is a pair [*E*,*V*] where *E* is an entity and *V* is a value. For the predicate to evaluate to true, *E* must have the property *V*. For instance the use case precondition "ATM is ON" is formally the predicate [*ATM*,*ON*]. It evaluates to true if the entity *ATM* has the property of being *ON*. A compound condition corresponds to a compound predicate with sub-predicates and logical operators.
Preconditions, postconditions and other conditions in use cases are predicates on *classes* and *attributes* from the domain model. Figure 2 shows a domain model for our ATM application example. Domain models

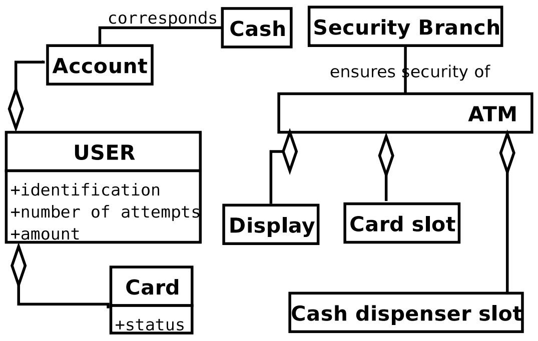

Figure 2: ATM example domain model.

are depicted as a UML class diagram (OMG, 2003).

Domain operations are elaborated using pre and postconditions. An operation precondition specifies a condition under which the operation may be invoked such that its results are as expected, while an operation postcondition describes how the system state is changed by the operation. Figure 3 shows pre and postconditions of the operations in use case "cash withdrawal". Preconditions and postconditions result from assumptions made by software specification developers. Our goal is to verify their adequacy in relation to use cases.

4 VERIFICATION APPROACH

A use case scenario dictates a certain sequencing of operations. This sequencing is possible only if each operation precondition is verified by the system's state at the moment of the operation. After each operation, a new system state is obtained by taking the operation postcondition into consideration. The

[1]The term "constraint" is used to refer to conditions in the UML specification.

[2]Elements between "[]" are optional.

```
insert card
 pre: ATM is ON AND
      USER Card is not in Card slot
 post:USER Card is in Card slot AND
     (USER Card status is valid OR
      USER Card status is not valid)
enter pin
 pre: ATM Display is pin enter prompt
select cash withdrawal operation
 pre: ATM Display is operation menu
specify withdrawal amount
 pre: ATM Display is withdrawal amount
alert security branch
 post: Security Branch is alerted
ask user pin
 pre: USER Card is valid
 post: ATM Display is pin enter prompt
display wrong identification error message
 post: ATM Display is wrong identification
display operation menu
 post: ATM Display is operation menu
ask withdrawal amount
 post: ATM Display is withdrawal amount
display amount not okay error message
 post: ATM Display is wrong amount
check amount entered
 post: USER amount is okay OR
       USER amount is not okay
dispense cash
 post: Cash is dispensed
eject user's card
 pre: USER Card is in Card slot
 post: USER Card is not in Card slot
```

Figure 3: Operations in use case "cash withdrawal".

objectives of our verification approach are to ensure: (1) each operation is possible, (2) use case invariants are not violated, and (3) the use case postconditions hold at the end of their respective scenario.

Figure 4 shows an algorithm that summarizes our verification approach. We assume the following functions.

- Given a use case *uc*, inv(uc) returns the predicate corresponding to the use case *uc* invariant.
- Given a step in a use case scenario $step_i$, prec($step_i$) returns a predicate corresponding to the precondition of the domain operation invoked in step $step_i$.
- Given a step in a use case scenario $step_i$, post($step_i$) returns a predicate corresponding to the postcondition of the domain operation invoked in step $step_i$.
- Given a step in a use case scenario $step_i$, cond($step_i$) returns a predicate corresponding to step $step_i$ condition. An alternative condition is returned if $step_i$ is the first step of an alternative. For instance function cond() would return the predicate `NOT([USER.Card, valid])` for step *1.a.1* of use case "cash withdrawal".

  The description of a use case primary scenario generally omits step conditions implicitly assumed from alternatives. For instance, step *2* in use case "cash withdrawal" has as implicit condition, the negation of alternative *1.a* condition (*User Card is valid*). This condition follows from the fact that step *2* would not be possible if alternative *1.a* is executed. In general, the condition of a step *i* in a primary scenario includes the conjunction of the negation of all the conditions of step *i-1* alternatives.

```
VerifyScenarios(uc: Use Case)
  FOR EACH Scenario $sc$ = $step_1$ - $step_2$ - $\cdots$ $step_n$ in uc
  1.   $i = 0$
  2.   $state_i$ = prec(uc)
  3.   $i = i+1$
  4.   FOR EACH Step $step_i$ in sc
  4.1.   IF (!CheckAgainstState(inv(uc),$state_{i-1}$)
         return
  4.2.   IF (!CheckAgainstState(prec($step_i$), $state_{i-1}$) return
  4.3.   IF (!CheckAgainstState(cond($step_i$), $state_{i-1}$) return
  4.4.   $state_i$ = DetermineState($state_{i-1}$, post($step_i$))
  4.5.   $i = i+1$
  5.   IF (!CheckAgainstState(post(uc),$state_{i-1}$)
       return
```

Figure 4: Verification algorithm.

- Given two predicates p_1 and p_2, function **CheckAgainstState(p_1,p_2)** returns true if $p_2 \Rightarrow p_1$. The function returns false otherwise.
- Given a predicate p_{old} and a predicate p_{chg}, function **DetermineState(p_{old},p_{chg})** returns a predicate p_{new} such that p_{new} is the logical conjunction of p_{old} and p_{chg}. All sub-predicates in p_{old} on an entity E are replaced with sub-predicates from p_{chg} on the same entity if any. For instance, suppose p_{old} = [E_1, V_1] AND [E_2, V_2] and p_{chg} = [E_1, V_1'] AND [E_3, V_3], **DetermineState(p_{old},p_{chg})** would return p_{new} = [E_1, V_1'] AND [E_2, V_2] AND [E_3, V_3].

For each of a use case scenario $sc = step_1 - step_2 - \cdots step_n$, we attempt to generate a sequence of states $state_0 - state_1 - state_2 - \cdots state_n$ such that: each state corresponds to a predicate, $state_{i-1}$ is the state before $step_i$, and $state_i$ is the state after $step_i$.

Table 2: State predicates corresponding to the primary scenario of use case “cash withdrawal”.

State	Predicate
$state_0$	[ATM, ON] AND NOT([USER.Card, in_Card_slot])
$state_1$	([USER.Card.status, valid] AND [ATM, ON] AND [USER.Card, in_card_slot]) OR (NOT([USER.Card.status, valid]) AND [ATM,ON] AND [USER.Card, in_card_slot])
$state_2$	[ATM.Display, pin_enter_prompt] AND [USER.Card.status, valid] AND [ATM, ON] AND [USER.Card, in_card_slot]
$state_3$	[ATM.Display, pin_enter_prompt] AND [USER.Card.status, valid] AND [ATM, ON] AND [USER.Card, in_card_slot]
$state_4$	([USER.Card.status, valid] AND [ATM.Display, operation_menu] AND [ATM, ON] AND [USER.identification, valid] AND [USER.Card, in_card_slot]) OR (**[USER.Card.status, valid] AND [ATM.Display, operation_menu] AND [ATM,ON] AND [USER.number_of_attempts,** >3**] AND [USER.Card, in_card_slot]**)
...	...
$state_{11}$	([Cash, dispensed] AND [USER.Card.status, valid] AND [USER.identification, valid] AND [ATM, ON] AND [USER.amount, okay] AND NOT([USER.Card, in_card_slot]) AND [ATM.Display, withdrawal_amount]) OR (**[Cash, dispensed] AND [USER.Card.status, valid] AND [ATM, ON] AND [USER.amount, okay] AND NOT([USER.Card, in_card_slot]) AND [USER.number_of_attempts,** >3**] AND [ATM.Display, withdrawal_amount]**)

In step *4.1* of the algorithm, the use case invariant is checked against each generated state. In step *4.2*, we check the precondition of the current use case step operation, and in step *4.3*, we check the condition of the current use case step. The verification stops whenever a check results in a failure. This helps avoid an accumulation of verification errors and favors an iterative approach for verification. A new state predicate is generated in step *4.4*. Upon a successful sequence of state predicates generation, we check in step *5*, the scenario postcondition against the last state of the sequence of states.

As an example, consider use case “cash withdrawal” primary scenario. The first generated state corresponds to the use case precondition; predicate $state_0$ = [ATM, ON]. The verification of the use case invariant against $state_0$ in step *4.1* proceeds successfully since use case “cash withdrawal” invariant is exactly the same predicate as $state_0$. In step *4.2* of the verification algorithm we check the precondition of operation “insert card”, the first operation of the scenario against $state_0$. According to the definition in Figure 3, operation “insert card” precondition is predicate [ATM, ON] AND NOT([USER.Card, in_Card_slot]). The verification fails because the implication [ATM, ON] $\Rightarrow$ ([ATM,ON] AND NOT([USER.Card, in_Card_slot]) can not be established. This verification failure is indicative of an under-specification of the use case precondition. The error can be corrected by adding the missing condition “*USER Card is not in Card slot*” to the use case precondition. Suppose the precondition of use case “cash withdrawal” is changed to “*ATM is ON AND USER Card is not in Card slot*”. Table 2 shows some of the states corresponding to the primary scenario. We obtain $state_1$ by applying operation “insert card” from $state_1$. According to the definition in Figure 3, operation “insert card” postcondition is [USER.Card, in_card_slot] AND ([USER.Card.status, valid] OR NOT([USER.Card.status, valid])). The conjunction of this predicate with $state_0$ = [ATM,ON] AND NOT([USER.Card, in_Card_slot], results in $state_1$ = ([USER.Card.status, valid] AND [ATM, ON] AND [USER.Card, in_card_slot]) OR (NOT([USER.Card.status, valid]) AND [ATM, ON] AND [USER.Card, in_card_slot]). Before operation “insert card”, the USER’s Card is not in the Card slot and therefore its validity is not determined. After the operation, the USER’s Card becomes in the Card slot and a determination to its validity is made. The remaining state predicates are obtained similarly. The use case postcondition, which is defined as predicate [USER.Account, updated] AND [Cash, dispensed] is then checked against state $state_{11}$, which is the final state of the scenario. The verification fails because the predicate component [USER.Account, updated] is not satisfied. For this inconsistency to be corrected, at least one of the operations in the scenario needs to include [USER.Account, updated] as a postcondition. Operation “update user’s account” appears as the most natural choice.

An examination of the state predicates in Table 2 shows another inconsistency. $State_4$ predicate is a disjunction consisting in sub-

predicates [USER.Card.status, valid] AND [ATM.Display, operation_menu] AND [ATM, ON] AND [USER.identification, valid] AND [USER.Card, in_card_slot] and [USER.Card.status, valid] AND [ATM.Display, operation_menu] AND [ATM, ON] AND [USER.number_of_attempts,$>$ 3] AND [USER.Card,in card slot]. The second sub-predicate does not include predicate [USER.identification, valid]. Therefore, $state_4$ allows operation “ATM displays an operation menu” in a situation where the validity of the USER identification is not established. This comes from the fact that step *4* condition ($cond_4$) is the conjunction of the negation of the conditions of alternatives *3.a* and *3.b*.
Therefore,

$cond_4$ = NOT(NOT([USER.identification, valid]) AND [USER.number_of_attempts,$<$3]) AND NOT(NOT([USER.identification, valid]) AND [USER.number_of_attempts,$==$3])

Which after simplification gives

$cond_4$ = [USER.identification, valid] OR [USER.number_of_attempts,$>$3].

The inconsistency is propagated through the scenario as shown by $state_{11}$. It might be the case that the use case embeds the assumption that [USER.number_of_attempts,$>$3] will never hold because of the user interface. However, this assumption could constitute a serious safety vulnerability if the implementation solely relies on the documented requirements and does not safeguard against the possibly that the *number_of_attempts* could become greater than 3 (for instance by bypassing the user interface). It is possible to correct the problem by adding condition *USER identification is valid* as a precondition to operation “display operation menu”. This would remove the second part of the disjunction from $state_4$ and from all subsequent states including $state_{11}$. The precondition also constitutes a documented record that helps ensures that the implementation would consider the necessary checks.

5 CONCLUSIONS

We have presented an approach for checking use cases against operation contracts. This approach is currently implemented in a prototype tool for use cases based requirements engineering. The verification approach helps refine use cases in conjunction with a domain model. It supplements a full validation based on simulation. We believe this approach can be applied in any circumstance where use cases are combined with contracts.

The limitations of the approach depend on the strength of the underlying logic. In this paper, we illustrated the approach with a simple predicate logic without quantification. The approach does not preclude from using a stronger form of logic. However, the stronger the logic, the more sophisticated the proof engine needs to be. We are currently experimenting with theorem proving approaches (Duffy, 1991). Beside the need for more sophisticated proof mechanisms, more complex logic systems involve languages farther from natural language.

This approach could be extended beyond use cases. For instance, as a future work, we are considering the possibility to use the same verification approach to check design level interaction diagrams against operations specified in OCL (OMG, 2003).

REFERENCES

Boehm, B. and Basili, V. R. (2001). Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. (1994). *Object-Oriented Development the Fusion Method*. Prentice Hall.

Duffy, D. A. (1991). *Principles of Automated Theorem Proving*. John Wiley & Sons.

Giese, M. and Heldal, R. (2004). From Informal to Formal Specifications in UML. In *UML 2004 - The Unified Modelling Language: Modelling Languages and Applications*, pages 197–211.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communication of the ACM*, 12(10).

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1993). *Object-Oriented Software Engineering*, A Use Case Driven Approach. Addison-Wesley, ACM Press, 2 edition.

Johnson, J. (2006). *My Life is Failure*. The Standish Group International, Inc.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR.

Meyer, B. (2000). *Object-Oriented Software Construction*. Prentice Hall, 2 edition.

OMG (2003). UML 2.0 Superstructure. Object Management Group.

Somé, S. (2006). Supporting Use Cases based Requirements Engineering. *Information and Software Technology*, 48(1):43–58.

Toyama, T. and Ohnishi, A. (2005). Rule-based Verification of Scenarios with Pre-conditions and Post-conditions. In *13th IEEE International Conference on Requirements Engineering (RE 2005)*, pages 319–328.