Observations for Assertion-based Scenarios in the context of Model Validation

Emine G. Aydal, Richard F. Paige, Jim Woodcock

Department of Computer Science, University of York, UK. {aydal, paige, jim}@cs.york.ac.uk

Abstract. Certain approaches to Model-Based Testing focus on test case generation from assertions and invariants, e.g., written in the Object Constraint Language. In such a setting, assertions and invariants must be validated. Validation can be carried out via executing scenarios wherein system operations are applied to detect unsatisfied invariants or failed assertions. This paper aims to improve our understanding of how to write useful validation scenarios for assertions in OCL. To do so, we report on our experiences during the creation and execution of 237 scenarios for validating assertions for the Mondex Smart Card application. We also describe key factors that must be considered in transforming scenarios into test cases.

1 Introduction

In Model-Driven Engineering (MDE), models can be used for systematically deriving other artefacts needed within the engineering process, such as code and test cases. Applying MDE in practice requires well-defined modelling languages, scalable and practical tools for constructing and managing models, as well as means for validating models.

In this paper, we analyse the applicability and suitability of using UML and the Object Constraint Language (OCL) for model validation within the context of Model-Based Testing (MBT). Some of the research in this area focus on determining a set of test targets and translating them into abstract test cases [12, 13]. These studies are valuable in introducing *Requirements Traceability* to MBT and finding test targets against these requirements, however, how model validation is achieved against these criteria is kept outside of the scope.

The main focus, in this study is on validating models by constructing snapshots representing system states at a particular point in time with objects, attribute values, and links. There are other research studies based on the expressive power of snapshots such as [3, 4, 9]. The difference of this work is that we base our scenarios both on invariants and assertions of the system operations. Therefore, the scenarios not only check whether there is a state where all the system invariants are satisfied, but they also check whether there are states that allow system operations to run. For instance, if one of the preconditions of an operation can never be satisfied, then either the operation is redundant or the operation is ill defined. The benefit of this approach in addition to model validation is that, potentially, the scenarios can also be used for abstract test case generation.

1.1 Mondex Smart Card Application

The software system used as the basis of this validation experiment was the Mondex Smart Card Application. Mondex is a global electronic payment scheme that provides digital form of cash [2]. The card holds values in several different currencies and provides direct money transfer without signature, PIN or transaction authorization between card holders [5].

Mondex was the first case study carried out in the implementation of the Grand Challenge program that aims to populate a repository of formally specified and verified codes that are useful in practice and serve as examples for the future applications [8]. Unlike other research studies based on the monograph outlined in [7], we follow a different path in the sense that we created the model of the system from the informal requirements detailed in [5]. In doing so, we covered some of the functional requirements omitted in [7] and in all the other studies that have been based on this monograph.

In this experiment, we modelled the system using UML and OCL in *USE* tool. USE allows users to specify system models, invariants, and pre- and postconditions textually, and allows assertions to be checked. The class diagram of the Mondex Smart Card application populated in this study is given in Figure 1. The system has 30 invariants, the classes given in Figure 1 have 31 operations and 197 assertions were written in order to cover these operations. These numbers exclude utility classes such as *Date* and their associated operations.

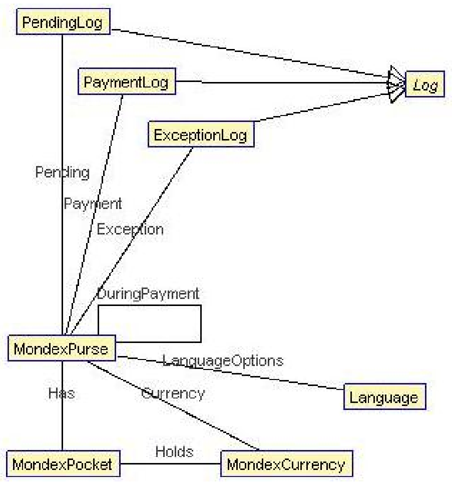

Fig. 1. Class Diagram - Mondex

1.2 Test Scenarios in UML Specification Environment (USE)

The USE tool provides a multi-level platform where the model is defined in a .use file, the generation of an instance of the model is managed by an .assl file, the extra optional invariants are imposed in a .invs file and all these files as well as other USE-related commands are executed by calling .cmd files in command prompt of the tool. An example operation definition written in .use file for creating pending logs in Mondex Smart Card Application is given in Table 1.

context MondexPurse::CreatePendingLog(p_PendingLog: PendingLog) : Boolean
pre CreatePendingLogPre1: self.LockingState = 'Unlocked' pre CreatePendingLogPre2: p_PendingLog.isDefined() pre CreatePendingLogPre3: pendinglog->isEmpty()
post CreatePendingLogPost1: pendinglog->includes(p_PendingLog) post CreatePendingLogPost2: pendinglog->size() = 1 post CreatePendingLogPost3: self.LockingState = self.LockingState@pre

Table 1. CreatePendingLog()

Once the operation definition is defined, there are other tasks to be completed before writing a scenario. An instance of a system can be generated in USE by using ASSL (A Snapshot and Sequence Language) [3, 4] Once the instance is generated, the output can be written into a command (.cmd) file and executed in the command prompt of the tool. Depending on the strategy to be followed, the system and its environment may need adjustment. This is also called *Preamble* in Model-based Testing terminology, i.e. bringing the system into a specific state before executing an operation. The tasks carried out as part of an operation are written in a .cmd file. Finally, by using openter and opexit features of USE, the preconditions and postconditions of the operation are checked. The following is an example scenario written for *CreatePendingLog* operation.

```
open c:/<root>/Mondex.use
gen start -b c:/<root>MondexInstance.assl PurseGenerator{2}
gen result
read c:/<root>/snapshot.cmd
read c:/<root>/pre_CreatePendingLog.cmd
!openter pendingLog1 CreatePendingLog(pendingLog1)
read c:/<root>/CreatePendingLog.cmd
!opexit true
```

Table 2 gives the structure of these scenarios with associated tasks. In Step 1, a basic, valid, stable instance of the system model is created. This step is ideally the same for all the scenarios. Step 2 prepares the system for the operation under investigation. At this point, we determine the set of variables that the operation reads/writes from/to; this set is called Frame Variable Set (FVS). Observation 6 in Section 2.3 explains this concept in detail. During Step 3, the operation call is put into the call stack and the preconditions of the operation are checked. Step 4 is where the actions of the operation are carried out. If the value of an attribute in the set of frame variables should not change, it is also important to explicitly state this at this step. Note that if we aim to write a scenario that creates a conflict with one of the postconditions of the operation under investigation, it is Step 4 where we need to produce this conflict. In the final step, we exit the operation and the postconditions are checked if the preconditions were satisfied.

Step No.	Step	Description
1	Initial loading of object model	An instance of the model is loaded at this stage.
2	Environment Setting (Preamble)	- Determine frame variables - Creation of objects - Setting attribute values
3	Access to operation	The preconditions are checked.
4	Modification of the model	- Creation/Deletion of objects - Setting/Modifying attribute values - Coverage of frame variables
5	Exit from the operation	The postconditions are checked.

Table 2. The steps of a basic scenario

1.3 Validation of Assertions and Generation of Abstract Test Cases

Validation is a crucial phase of the software development process in demonstrating that the system under investigation meets its requirements. Assertion validation ensures that invariants, pre- and postconditions of operations are expressed as intended. There are three ways of accomplishing this:

1. Checking that the preconditions, invariants and postconditions of the *Operation under Investigation (OpuI)* are satisfied at least once.
2. Ensuring that the OpuI does not execute when at least one of the preconditions fails.
3. Checking that at least one of the postconditions of the OpuI fails when a mutant is inserted into the operation, provided that the preconditions and the state invariants are satisfied.

The first approach guarantees that the postconditions of the operation are not too strong and that there is at least one case where preconditions and state invariants are satisfied. The second approach checks the response of the system when the preconditions are not satisfied. Finally, the last approach verifies that the harmful modifications to the operation are caught by the postconditions and that the postconditions are not too weak. Note that the third approach needs mutants to be generated and inserted into the system, and therefore constitutes white-box testing. We included this for the sake of completeness in showing how the assertion can be used in different ways for scenario generation, however, in the context of Model-Based Testing, this approach cannot be used.

Table 3 summarises the statistics for all the scenarios we created in our experiment. 237 scenarios were created of which 32 presented the ideal cases, 94 violated a precondition, 104 violated one postcondition and 7 violated more than one postcondition. Of these 237 scenarios, 64 also contradicted an invariant. This means that if this scenario were to lead to an error in the system, invariant failure would provide an alert as well. In addition to these, 98 scenarios require creation/deletion of objects and 71 scenarios are involved in nested calls. The importance of these numbers is made clear in the rest of this paper.

Scenarios	
Total number of scenarios	237
Scenario presenting ideal case	32
Scenario conflicting preconditions	94
Scenario conflicting postconditions	111
Invariant conflict	
Scenario conflicting one invariant	55
Scenario conflicting two invariants	9
Scenario satisfying all the invariants	170
State Change	
Scenario that requires state change within operation	40
Scenario that requires state change outside the op.	23
Scenario that requires no state change	171
Other information	
Scenario requiring object deletion/creation	98
Scenario with nested calls	71
Scenario that changes the value of at least one attribute	181

Table 3. Statistics for the scenarios created in this study

1.4 Contribution

The *discipline* of writing scenarios that allow developers both to validate models and generate test cases from the scenarios is not yet fully understood. Thus, the main aim of this work is to improve the understanding of how to validate assertions by reporting on the results of a large-scale experiment in which validation scenarios were written and used to check assertions. By doing so, we provide a structured set of observations from this set of experiments that provide advice on how to go about validating assertions in a scenario-based way and to use them as a basis for test case generation. One of the novelties of this work is that it explains the issues encountered from three different perspectives: Tool-related, scenario-related and assertion-related. This classification not only facilitates the understanding the root cause of certain problems, but also foresee the benefits/disadvantagesof the techniques applied.

One of the guidelines followed during the study was to keep the scenarios as general enough as possible, so that they can be applied to different instances of the model with little or no modification. The rationale behind this is to ease the transformation of these scenarios to test cases in a systematic manner with a good degree of automation. This principle has pointed out many different issues that need consideration, but have been overlooked, such as frame variables, test case sequencing, dependency of scenarios, etc. The rest of the paper explains our observations in a structured format.

2 Observations on Scenario Creation in the context of MBT

In this section, we report our experiences in creating the scenarios described above. We structure our observations into three categories: tool-related observations, assertion-related observations, general scenario formation-related observations. The tool-related observations aim to reveal the drawbacks and advantages of using the *USE* tool. The assertion-related observations address the usage of the pre/postconditions as well as the invariants in the process of scenario creation. The observations in this category focus on how to detect some of the errors related to assertions and how to write better assertions. The final category outlines the observations regarding the scenarios that either show the ideal execution of an operation or violate a pre/postcondition. Each observation is described using the template given below.

Short description of the observation
Aim: the aim of the action that produced the observation.
Context: in which environment the observation is made
Issue encountered: the unexpected behaviour
Detailed observation: a clear explanation of the issue
Conclusion: the effects of the problem.
Extra notes: extra comments about the issue and its impacts

2.1 Tool-related observations

In this section, tool-related observations during the validation of assertions are discussed in detail.

Observation 1: How is the call stack managed in USE?

Aim: The aim is to find a scenario that contradicts a precondition of an operation.
Context: In USE, the preconditions of an operation are checked when the command for the operation call -!openter- is executed. If the preconditions of the operation are satisfied, then the operation is put into the *call stack*. If the operation is defined by OCL expressions, these are executed and the expected value is returned when the !opexit command is run. The postconditions of the operation are also verified at this point.
Issue encountered: If the operation is not defined by using OCL, i.e. when the operation changes the value of some variables, creates/deletes some objects, etc., then these changes are performed via the commands introduced in *USE* between the !openter and !opexit commands. !set, !create, !destroy are examples of such commands.

The problematic situation occurs if the operation is defined by *USE* commands and the preconditions of the operation are not satisfied. In such a case, the operation is not put into the *call stack* due to precondition failure, however, the commands after the operation call are still executed, i.e. the changes that the operation would do are still performed. Then, when the !opexit command is called, the tool states that the call stack is empty and do not check the postconditions assuming that the operation is not executed anyway. The concern here is that the system may be in a different state than it was in the beginning although the operation is not executed according to the tool.
Observations: One of the operations in which we observed this behaviour of the tool is given in Table 4. This operation is called when the personal code is entered more than the number of times stated by the attribute *PersonalCodeAttempts*. The attribute *_NumberofIncorrectEntries* counts the number of incorrect entries.

The following generates a contradiction in the second precondition.

context MondexPurse::ChangeTheStateToLockedOut() : Boolean
pre ChangeTheStateToLockedOutPre1: self.LockingState = 'Unlocked' or self.LockingState = 'Locked' pre ChangeTheStateToLockedOutPre2: PersonalCodeAttempts<=_NumberOfIncorrectEntries
post ChangeTheStateToLockedOutPost1: self.LockingState = 'LockedOut' post ChangeTheStateToLockedOutPost2: PersonalCodeAttempts=PersonalCodeAttempts@pre

Table 4. ChangeTheStateToLockedOut

```
<ChangeTheStateToLockedOut_Pre2.cmd>
read basemodel.cmd
!set P1.LockingState := 'Unlocked'  --Setting the Frame Var.
!set P1._NumberOfIncorrectEntries := 1  --Conflict Creation
!set P1.PersonalCodeAttempts := 4
!openter P1 ChangeTheStateToLockedOut() --Enter the operation
!set P1.LockingState := 'LockedOut' --Modification of Frame Var.
!opexit true  --Exit the operation
```

Conclusion: In this example, we managed to reach our aim in the sense that we found a scenario where the second precondition fails, however, the system is now in a different state than it was and this may cause unexpected results when the next scenario is run.
Extra notes: The workaround we followed for this problem is that we executed the commands that *neutralise* the modifications occurred during the execution of such a scenario. Observation 7 explains the process of *neutralisation* in more detail.

Observation 2: Reducing the dependency of scenarios on the model instance

Aim: In the creation of the scenarios, one of the rules we need to follow is that we need to be *reasonably independent* of the instance of the model we created. What we mean by this is that the instance of the model created for the test environment is just one of many possible, and the objects that exist in one instance may not exist in another. In other words, the number of objects, the number of links, the name of the objects, and the attribute values may all be different. Since the scenarios in our approach are based on the instance of the model, it is true that they are somewhat dependent on the instance chosen. Having said that, the level of this dependency can be decreased by using more general statements in the scenarios.
Context: If deleting an object is one of the operations handled by an operation, some of the postconditions of this operation must make sure that the deletion really occurs. The observation that is explained in this section is made whilst trying to find a scenario that is related to one of these postconditions and that deletes an object.
Issue encountered: Some of our attempts to write more general statements in our scenarios have caused exceptions in USE especially during the deletion of an object.
Observations: The EnquireCurrencyInfo() function defined in our system provides information about a given currency. One of the postconditions of this function states that that the operation should not modify currency objects. In our approach, we created scenarios that present the ideal functioning of an operation and that also generates conflicts with each different pre/postcondition of the operation. The scenario that targets the postcondition above deletes a currency object and observes the reaction of the system. Our first attempt to delete the currency object had the following code:

```
let ObjectDel : MondexCurrency = Purse1.avCurrencies->
    select(ISOCurrencyCode ='GBP')->asSequence()->first()
!destroy ObjectDel
```

However, the *ObjectDel* object behaves like a pointer to the actual object and therefore the object requested to be deleted is not removed and we receive a *RunTime Exception - unbound variable* - error. Our next attempt was to define the object to be deleted as defined in ObjectDel:

```
!destroy Purse1.avCurrencies->
    select(ISOCurrencyCode = 'GBP')->asSequence()->first()
```

This expression states that the the first element in the set of Currencies in Purse1 whose ISOCurrencyCode is *GBP* should be deleted. This command deletes the first object in the selected collection. However, when we execute the *!opexit* command to exit the operation, we received a *Null Pointer Exception* and the postconditions are not checked. Our final attempt was to execute the delete command by calling object's name:

```
!destroy C_GBP1
```

After this, the object is deleted, the postconditions are evaluated and no exception is thrown.
Conclusion: When we examine the scenario statements written above, we see that the first two need the same amount of information in order to proceed, i.e. the currency that is under investigation. This information is passed to the function as a parameter anyway and therefore the user does not necessarily have to know a lot about the current instance of the model. The third one, on the other hand, needs knowledge of an object name which makes the scenario extremely dependent on the selected instance of the model. We tried these options in different scenarios and noticed that in some situations the second - and the more general - option works, however we were unable to find a pattern that explains the rationale behind this varying behaviour. The main message we here is that the tool must allow the user to write reasonably general, OCL-based scenarios, and the rules related to creation and deletion of the objects must be clear.

2.2 Assertion-related observations

The observations made during the creation of pre/post-conditions are given in this section. The situations where a scenario highlights the importance of a change in the assertions and in invariants are also presented.

Observation 3: Writing an invariant instead of an assertion tuple

Aim: Some of the most important characteristics our model should possess are clarity, consistency and simplicity. The elements forming the model such as diagrams and OCL expressions must also have these characteristics and should not introduce further complexity.
Context: Whilst creating the pre and postconditions of the operations, it is important to understand the context of the operation and the scope of the frame variables. In certain cases, the change in one variable may affect another variable and these changes must

be presented in postconditions.
Issue encountered: For two conditions X and Y, if Y must be true each time X holds, then this relationship must be shown each time X appears in a pre/postcondition. If X is a widely-used variable, i.e. is an element of frame variable set of many operations, then Y must be repeated as many times as X appears.
Observations: We first noticed this whilst writing scenarios for *ChangeTheStateToUnlocked()* function. Initially, the definition of the function had the pre/postconditions shown in Table 5.

context MondexPurse::ChangeTheStateToUnlocked() : Boolean

pre ChangeTheStateToUnlockedPre1:
LockingState = 'NonLocking' or LockingState = 'Locked'
pre ChangeTheStateToUnlockedPre2:
LockingState = 'NonLocking' implies (not PersonalCode.isDefined or PersonalCode = 0)

post ChangeTheStateToUnlockedPost1: LockingState = 'Unlocked' post ChangeTheStateToUnlockedPost2: PersonalCode.isDefined and PersonalCode <> 0

Table 5. ChangeTheStateToUnlocked - First version

We then realised that each time the state is in *Nonlocking* state or changes from *Nonlocking* state to any other possible state, we have to check the value of *Personal Code* even if the operation itself does not necessarily state a change in the value of Personal Code. This not only creates many duplicates of the same requirement, but also creates a hole in the system when it is forgotten. To avoid this, we created the following invariant:

```
inv iPerCode_Nonlocking:
 (LockingState = 'Nonlocking' implies
 (PersonalCode = 0 or not PersonalCode.isDefined()))
 and (LockingState <> 'Nonlocking' implies
 (PersonalCode <> 0 and PersonalCode.isDefined()))
```

This invariant states that if the system is in *Nonlocking* state, the Personal Code is either zero or not defined. When the system is in any other state, the personal code has a value other than zero. After the introduction of this invariant, the pre/postcondition definition of the above function became as given in Table 6.

context MondexPurse::ChangeTheStateToUnlocked() : Boolean
pre ChangeTheStateToUnlockedPre1: LockingState='NonLocking' or LockingState ='Locked'
post ChangeTheStateToUnlockedPost1: LockingState = 'Unlocked'

Table 6. ChangeTheStateToUnlocked - Second version

Conclusion: If there is a repetition of pre/postconditions, it is worth looking at the relationship between the variables under investigation to see whether such a relationship would hold for all cases. The example given above is a case where we noticed the need for an invariant due to this repetition.

Observation 4: Incorrect invariant detection

Aim: The rationale behind creating scenarios that violates a pre/postcondition is to be able to construct abstract test cases to challenge our system. One of the advantages of the process of scenario-creation is that it also allows us to test a model for correctness and consistency.
Context: It is possible to find a set of scenarios that violates an assertion of an operation. The choice of scenario to be used can be done simply by the user or a set of criteria can be defined and the scenarios may be expected to comply with some/all of these criteria.
Issue encountered: Sometimes, we observed that the set of scenarios that aim to contradict a pre/postcondition of an operation must also violate an invariant. In one of these situations, the invariant that the scenario was supposed to violate seemed to be satisfied. After close examination, we found out that the predicate in the invariant was incorrectly written.
Observations: One of the preconditions of the function *ReadPaymentLogs()* is that the system must either be in state *Unlocked* or *Locked*. There are two possible scenarios that would violate this precondition: the scenario that sets the system state to *Nonlocking* and the scenario that sets it to *LockedOut*. The invariant that deals with the cases where the state is *Nonlocking* -*iPerCode_Nonlocking*- was given in Observation 3. The invariant that checks the suitability of system variables for the *LockedOut* state -*iLockedOutState*- is given below.

```
inv iLockedOutState :
  LockingState = 'LockedOut' implies (PurseExhaustionFlag = true
    or _NumberOfIncorrectEntries >= PersonalCodeAttempts)
```

Both of these invariants require that some other variables of the system are set to certain values. In the scenarios created to contradict the precondition about the state of the system, we first followed the approach where the system state is set to *LockedOut* or *Nonlocking* and all the relevant variables such as Personal Code, PurseExhaustionFlag, etc. are excluded. This is why we expected that each scenario would conflict at least with the relevant invariant. However, to our surprise, the invariant *iPerCode_NonLocking* was satisfied in both cases. When we analysed the problem further, we realised that the invariant in the form of *(p implies q) AND (NOT p implies r)* was written as *(p implies q) AND NOT p implies r* which was interpreted as *((p implies q) AND NOT p) implies r* by the tool and therefore the invariant was satisfied although it should not have been. The issue was resolved when the fault in the invariant was corrected.
Conclusion: The example above presents a case where a scenario written with the aim of violating a pre/postcondition also detects a fault in the model itself. This means that the approach supports the process of correctness and consistency check for the model itself. Further research is needed to assess to what degree this support extends.
Extra notes: Note that, whilst creating the scenarios discussed above, the variables that the invariants are associated to are excluded, i.e. the scenario did not deal with the setting of variables such as *PersonalCode, PurseExhhaustionFlag*, etc. As an alternative, we also created scenarios that add these variables to the FVS of the operation and that handles the correct setting of these extra variables for the given scenario. By doing this, we bring the system in a different stable state and then observe the cases where the

precondition under investigation fails, but all the invariants are satisfied. This issue is briefly discussed again in Observation 6 in Section 2.3.

Observation 5: Overlapping postconditions

Aim: In our experiments, we created the scenarios that ideally violated only one assertion at a time. We believed that the possibility of finding an error increases when each scenario dealt with a different assertion, since the domain of a possible error differs when the subject matter is different in a scenario.
Context: In OCL, there may be several ways of navigating to reach an object. Some operations may create even more links and increase the number of navigation possibilities. If such operations have postconditions, the postconditions may also use different navigation routes.
Issue encountered: We noticed that when postconditions use the links created by the operation under investigation, they happen to assume that the links are created successfully. In other words, they also check the creation of links in addition to their main goal. This conflicts with our initial aim.
Observations: Following assertions were two of the postconditions initially written for the function *SendValue()*.

```
post SendValuePost1 :
  pockets->select(Default = true and
        currency.ISOCurrencyCode = p_ISOCurrCode)->size() = 1
post SendValuePost4 :
  pockets->select(Default = true)->
        asSequence()->first().Value@pre -
  pockets->select(Default = true)->
        asSequence()->first().Value = p_PaymentValue
```

SendValuePost1 states that the pocket that carries the requested currency given by the parameter *p_ISOCurrCode* is the default. This postcondition ensures that the pocket that carries the currency in which the amount will be transferred is set as the default pocket. The postcondition *SendValuePost4* ensures that the value held by the pocket from which the amount is transferred is decreased by *p_PaymentValue*.
Both conditions seem to match their definitions as given above. However, when analysed further, we noticed that when the first condition fails, both postconditions fail even if the money is transferred in correct currency. This is because *SendValuePost4* assumes that the default pocket is set properly and therefore tries to reach the object through a newly defined link. For instance, if the transfer is made in GBP and the pocket that holds GBP is not set as default, but transfer is made in GBP successfully, then we expect *SendValuePost1* to fail and *post SendValuePost4* to pass. We observed that both of the above postconditions fail for such a scenario. This may seem as an advantage at first sight, but when we try to detect the root cause of such a failure, it is more difficult to find and we are unable to say which assertion is the main target of such a scenario.

The solution to this is to separate the concerns as much as possible for each pre / postcondition. This also requires withdrawing the sequential way of thinking to reach the outcome of an operation. In the above example, one may think that the first action is the change of the default pocket and then the value transfer occurs, so the rationale behind writing a postcondition like *SendValuePost4* may be the result of such reasoning. To overcome this issue, we changed the *SendValuePost4* as below:

```
post SendValuePost4 :
  pockets->select(currency.ISOCurrencyCode@pre = p_ISOCurrCode)
    ->asSequence()->first().Value@pre -
  pockets->select(currency.ISOCurrencyCode = p_ISOCurrCode)
    ->asSequence()->first().Value = p_PaymentValue
```

This new version of *SendValuePost* subtracts the previous and current values of the pocket that holds the currency in which the payment is made. In this version, there is no assumption about any of the actions that the function under investigation must take prior to money transfer. As a result of this, the failure in *SendValuePost1* does not necessarily mean a failure in *SendValuePost4*.
Conclusion: There are several conclusions we can reach by looking at this example. The first one is the importance of demonstrating the *independent effect* of each pre/postcondition. This is similar to that of Modified Condition/Decision Coverage (MC/DC), which is a structural coverage criterion that requires that the effect of all conditions in a program are demonstrated and that there is no condition that does not affect the outcome of a decision [11]. Analogous to this criterion, in our approach, we create scenarios - that would then form the abstract test cases - based on the pre/postconditions and therefore, it is important to be able to see the independent effect of each unit. By showing the independent effect, we not only avoid the possibility of an assertion being masked by another assertion, but also let each assertion contribute to the final test suite. In addition to this, separation of concerns also makes *root cause analysis* easier when a fault is detected. In other words, if an error occurs during the execution of a test case that is based on a scenario created by using our approach, we can backtrack and reason about the error by looking at the part of the program that deals with the postcondition under investigation.

2.3 General Scenario Formation-related observations

Scenarios form the core of our approach and therefore it is crucial to investigate carefully the way they are written, how well they achieve the aim of contradicting an assertion, and the factors that affect their execution. This section presents the issues encountered during the scenario creation and execution.

Observation 6: Frame variables and treatments to frame variables

Aim: In most cases, the operations of a system do not have to read/write from/to all the attributes of the model. The set of attributes that an operation is in contact with is called *frame variables* [6]. The Frame Variable Set (FVS) serves as a completeness check for the operation in the sense that it includes all the variables that must appear in the definition of the operation. The elements of FVS can be analysed under two categories: *static* elements (those that are only read), *dynamic* elements (those that are modified by the operation). Correct determination of frame variables is essential in order to be able to systematically monitor the state of the system after the execution of an operation,

therefore in our observations we aim to find guidelines to reach a *reasonably* complete set of frame variables.
Context: By definition, if a variable is read/modified during the execution of an operation, it is considered to be included in the FVS.
Issue encountered: During the course of scenario creation, we noticed that the above definition alone is not sufficient to cover all the frame variables. There are various external factors such as invariants, nested calls, etc. that require the use of other variables than those listed in the initial form of FVS of an operation.
Observations: Following is a list of some of the cases that must be considered in the determination of FVS:

- If an operation calls another operation from within, either the elements of the FVS of the called function is added to that of callee function, or the FVSs do not change. Note that, in theory, a static element of callee function's FVS should not be a dynamic element of the called function's FVS.
- As explained in Observation 4, the change of a variable value may cause an invariant conflict. If the conflict is due to another variable that is not even considered in the scenario, then we have two routes to follow. We either count these scenarios that conflict with invariants as test cases, or frame variable set will be extended in order to cover the variables required by the invariant and new scenarios that do not conflict with the invariants will be created. So, the point to consider is whether to include the variables that are associated to already existing frame variables through an invariant.
- Another issue that must be taken into account in FVS determination is the case of derived attributes. OCL supports the definition of derived attributes that are prefixed with / in UML and we explained briefly how we dealt with derived attributes in [1]. Currently, there is no automatic way of assigning the value of these attributes. In the context of frame variables, this brings extra work since the operation may not directly use the derived attribute, but if at least one of the dynamic variables included in its FVS set is the variable that affects the value of the derived attribute, then the derived attribute should also be modified accordingly. An example to this can be given by explaining the changes in *_NumberOfUnusedExceptions*. This derived attribute is calculated by subtracting the $exceptionlogs- > size()$ value from $cMaxExceptionNo$ constant.

  When an exception occurs, the system calls *CreateExceptionLog()* function. This function creates an extra exception log if the current number of exceptions does not exceed a certain value. Since the value of *CMaxExceptionNo* is constant, but the size of exception logs changes during the course of this operation, we have to include *_NumberOfUnusedExceptions* in the FVS and change the value of the attribute accordingly. This allows us to use it as a counter in other functions.

Conclusion: The list presented above is not a complete list by any means, but we believe it demonstrates the need for the extension of FVS through different channels and if this concept is to be integrated into OCL tools, it is necessary that the aforementioned issues are considered.

Observation 7: Running sequences of scenarios

Aim: It is important to find the patterns that would help us automate our technique and/or integrate it with other techniques. In the context of scenario creation, automation is not only important in forming scenarios, but also in executing them one after the other.
Context: When the instance of the system model is exercised by a scenario, the scenario brings the system into a certain state. This new state may be the same as the old one if the scenario does not perform any changes on any of the existing objects and does not create/delete any objects. In this case, a second scenario can be run straight after the first one, since the system is in its initial, stable state. Clearly, each scenario assumes that the system is in a state that accepts the further requests to be made by itself. This assumption comes from Step 1 in Table 2.
Issue encountered: If the first scenario makes modifications on the initial instance of the model, then the assumptions made by the second scenario may not hold. Thus, the question that arise in the context of scenario execution is that how the process of running several scenarios one after the other can be achieved especially in the presence of those that modify the system. The following are two solutions to this problem:

– System reset: After each scenario, the system can be reset to its initial state by performing Step1 on Table 2.
– Neutralisation: Actions that erase the effects of the last scenario can be carried out.

Observations: During the execution of our scenarios, we used both of the above techniques. Neutralisation requires *undoing* the actions taken by the scenario under investigation. For instance, if the scenario creates an object, the object and all its links to other objects must be deleted during the neutralisation process. This may seem straightforward, but it enforces to analyse all sorts of different actions and executing commands that ultimately has the opposite effect of these actions. This is a rigorous process especially if the scenario has nested calls and object deletions.
In order to apply the *Neutralisation* technique, we need to undo the actions in the reverse order. Adjusting the value in default pocket by addition, deleting the payment log that holds the details for a transaction, changing the default pocket to its previous form, creating the previously existing exception logs are some of the actions carried out in order to neutralise the effect of the scenarios. This list only gives a rough idea about the actions that needs to be done before executing the next scenario. When we examine further, we realised that the elaboration of neutralisation process requires thorough analysis of program semantics and the process itself includes: storage of the previous values (in order to restore them later); implementation of *reverse* functions, i.e., functions that have the opposite effect of existing functions.

On the other hand, *system reset* only requires the re-compilation of the model and the execution of the first line in the scenario -*read basemodel.cmd*-.
Conclusion: Neutralisation and system reset are two solutions to the problem of running scenarios one after another. System reset may mean the initialisation of the whole model and depending on the technology used and the system environment, this may require extra memory space and adjustment of certain environment variables. However, during this research, system reset option has been used extensively in order to save time.

3 Future Work and Conclusion

In this paper, the observations made during the creation and execution of scenarios for validation of assertions have been outlined. The importance of independent scenarios, the effect of overlapping postconditions, the importance of the frame variable set and the ability to carry out successive executions of scenarios are some examples of observations that are not only applicable in the context of the *USE* tool, but are also relevant for other tools that support scenario creation and execution. We believe that these observations explain how scenarios can be defined in a structured manner and what sort of obstacles can be experienced, thus leading us to a better model, and helping to form the basis for test case generation through model artifacts.

The lessons learned during this study lead the work that compares several state-based modelling tools and concretises the tasks performed during modelling and model validation [10]. The outcome of this study also shows that the tasks carried out during these stages of Model-based testing may vary from one tool to another, however, the main concerns about scenarios (validity, generality, multi-platform applicability) that form the base of test cases are similar, if not the same. By using the comparison results and the lessons learned during this study, we now focus on on automatic generation, and concretisation of abstract test cases.

References

1. Aydal E.G., Paige R.F., Woodcock J., Evaluation of OCL for Large-Scale Modelling: a Different View of the Mondex Smart Card Application, *Ocl4All: Modelling Systems with OCL*, Workshop at *MODELS'07*, Nashville, USA, 2007.
2. Clarke R., The Mondex value-card scheme: a mid-term report. Chip-Based Payment Schemes: Stored-Value Cards and Beyond, 1997.
3. Gogolla M., Bohling J., Richters M., Validation of UML and OCL Models by Automatic Snapshot Generation, *Proc. UML 2003*, Springer, LNCS 2863, 2003.
4. Gogolla M., Buettner M., Richters M., USE: UML Specification Environment for Validating UML and OCL, *Science of Computer Programming*, 2006.
5. Introduction to Mondex Purse Operation, Tech. report., Mondex International Limited, 1999.
6. Kassios I.T., Dynamic Frames: Support for Framing, Dependencies and Sharing without Restrictions, *Proc. FM 2006*, Springer, LNCS, 2006.
7. Stepney S., Cooper D., Woodcock J., An Electronic Purse: Specification, Refinement and Proof. Oxford University Computing Laboratory, Tech Report, 2000.
8. Woodcock J., Banach R., The Verification Grand Challenge, CSIC, 2007.
9. Ziemann P., Gogolla M., Validating OCL Specifications with the USE Tool: An Example Based on the BART Case Study, Elsevier Science, http://www.elsevier.nl/locate/entcs/volume80.html, 2003.
10. Aydal E.G., Utting M., Woodcock J., A Comparison of State-based Modeling Tools for Model Validation, TOOLS-Europe'08, Switzerland, July 2008.
11. Miller S.P. and Chilenski J.J., Applicability of modified condition/decision coverage to software testing, Software Engineering Journal, 1994.
12. Bouquet F., Grandpierre C., Legeard B., Peureux F., Vacelet N., Utting M., A subset of precise UML for Model-based Testing, Proceedings of the 3rd international workshop on Advances in model-based testing, ISBN:978-1-59593-850-3, 2007.
13. Bernard E., Legeard B., Requirements traceability in automated test generation: application to smart card software validation process, A-MOST, 2005.