Evaluation of OCL for Large-Scale Modelling: A Different View of the Mondex Smart Card Application

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

University of York, UK

Abstract. OCL is used to add rigour to UML/MOF models, and in particular can be used to express behavioural details (e.g., operation pre- and postconditions, class invariants) of such models. The applicability and utility of OCL can be assessed by applying it to realistic applications and by investigating its capabilities both in terms of language characteristics and tool support. With this in mind, in this paper we model functional requirements for the Mondex Smart Card Application using UML Diagrams, demonstrate how system invariants as well as operation pre- and post-conditions are specified in OCL, and explore the degree to which OCL tool support can be used to create and validate these models. Moreover, we discuss how these pre- and post-conditions can be validated, in part by discussing how test cases can be selected from the OCL specifications created.

1 Introduction

The Unified Modeling Language (UML) is accepted as a de facto standard for software and system modelling. It offers a rich set of notations for modeling both the static and dynamic aspects of an object-oriented system [6]. The Object Constraint Language (OCL), developed by Warmer as a business modeling language within IBM [12], is a declarative language that can be used to describe model behaviour, as well as metamodel constraints.

In this paper we present an exemplar of system modelling, using OCL together with UML. This is carried out for the purposes of evaluating the utility of OCL for realistic systems modelling. The evaluation is in terms of both the language and its existing tool support. Moreover, we also consider how the constraints, pre- and postconditions specified in OCL can be validated, via an overview of strategies for selecting test cases from models.

The system we model using UML and OCL is the Mondex Smart Card Application. We thus commence with an overview of the system under investigation.

1.1 Mondex Smart Card Application

The Mondex Smart Card Application, also known as Mondex Electronic Purse, is a global electronic payment scheme providing an alternative form of cash to traditional notes and coins [2]. It offers immediate transfer of value without signature, PIN or transaction authorization between card holders in currencies allowed [3].

Mondex is an important step also in the implementation of the Grand Challenge Programme, i.e., a multi-national, long-term, research programme that aims to create a substantial and useful body of code that has been verified to the highest standards [4]. One of the main objectives of this programme is to populate a repository of formally specified and verified codes that are useful in practice and serve as examples for the future applications. The first case study proposed to be included in this repository is the Mechanical Verification of Mondex. For this Mondex Challenge, various groups from different organisations have produced distinct versions of the software by using different modeling/specification languages. A group at the Massachusetts Institute of Technology used Alloy; the University of Southampton applied Event-B; a group at the University of Bremen used OCL; Escher Technologies chose Perfect Developer; RAISE is used at the University of the United Nations Macao and the the Technical University of Denmark; and finally Z is used by a group at the University of York [4]. The work from all these groups was based on the monograph that outlined the specifications, refinement and proof details of Mondex in Z [5]. It is important to note that this monograph focused on a subset of the actual requirements, so as to concentrate on security/mission-critical requirements.

In this study, we have followed a different path than the studies mentioned above and started by creating the model of the system from informal requirements detailed in [3]. By doing so, we covered some of the functional requirements that were omitted in [5] as well as all the other studies based on this monograph.

The main goal of our study is to test some of the aforementioned versions of Mondex by using model-based testing techniques and tools. Figure 1 shows the testing process we plan to follow during this study [17]. The versions of

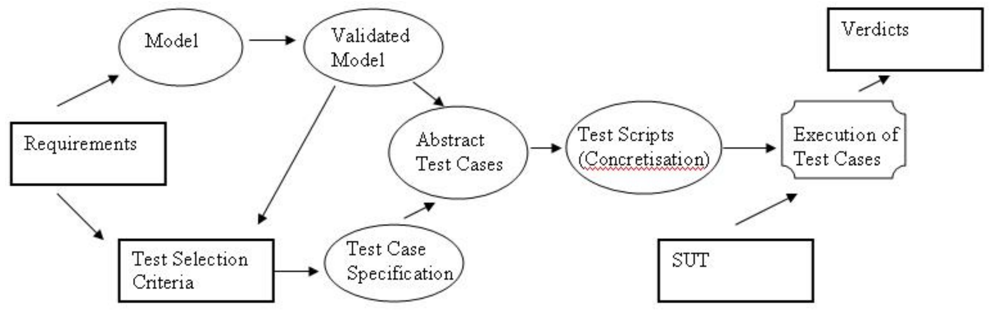

Fig. 1. Model-Based Testing

Mondex Purse we aim to test are formally verified. Formal verification brings a significant amount of trust to the produced code, but it is a very long and rigorous process. Significant amount of time is spent in writing the formal specifications and verifying the systems correct, therefore it would be really beneficial if the testing time can be reduced by generating effective test cases by using the models created early in the process. Successful implementation of this study would also demonstrate the invaluable contribution of model-based testing in a more formal context.

1.2 Contribution

In this paper, we focus on *Model, Validated Model* and *Test Selection Criteria* items shown in Figure 1. The main contribution of the paper lies within the complicated details of the modelling and the validation stage of a real life software application, Mondex Smart Card Application. We present the difficulties encountered whilst specifying the model behaviour with OCL by using the USE tool. Frame variables, messaging between objects, derived attributes, constants are some of the concepts we discuss in detail. We also explain how the system invariants and pre-/post-conditions of operations are handled, and how the system is validated by using the scenario-based technique. This technique is based on automatic snapshots technique introduced by Gogolla et al in [8] and [9]. Automatic snapshots technique has only been applied to invariants so far.

In this paper, we also explore the relationship between testing and the use of OCL, and explain our plans in extending the technique mentioned above to cover pre-/post-conditions with the aim of test data selection. When achieved, this new approach may allow the testers to carry out the test data selection process during modelling instead of implementation stage, thus reduce the time for testing and provide a language-independent form for test cases.Whilst explicating these, we demonstrate how the tool chosen for this study is used and what sort of improvements are required to make the process more efficient.

In the rest of this report, the modelling stage of the experiment is explained in Section 2. Section 3 focuses on how scenario-based validation of the model is carried out in this study and how techniques used in validation can be applied to test data selection. Finally, Section 4 outlines the lessons learned and provide concrete suggestions to the issues mentioned throughout the paper.

2 Modelling Mondex Smart Card Application

In this experiment, we have modeled the system by making use of UML diagrams. We have created use case diagrams, use case scenarios, class diagrams and state diagrams of the system. To strengthen the meaning of our diagrams and to specify the system constraints as well as pre/postconditions of operations, we used OCL expressions. Our current system consists of 8 classes, 31 operations, 30 invariants and 166 pre/postconditions. This number excludes the utility classes such as *Date* and their associated operations.

In the first phase of modelling, we separated the system into broadly defined modules that address different functions of the system. Table 1 gives brief description of these modules.

Module. No	**Module Name**	**Module Description**
M1	Payment	deals with the functional requirements that are related to normal value transfers.
M2	Logging	records the transactions and the errors.
M3	Recovery	handles the exceptions and the actions to be taken in case of a failure during a payment.
M4	Currency Management	deals with the currency-related features.
M5	Operational Control	manages the authentication-related issues.
M6	Data display and customisation	lets the user to view and customise certain data held by the Purse.

Table 1. Module Descriptions

We then created use case diagrams and complemented each diagram with well-defined scenarios. Each scenario is also linked to relevant modules and requirements. Before creating further UML Diagrams, we searched for modeling tools that allowed defining the invariants and pre-/post-conditions in OCL. We considered the following tools [15] [14]:

– *OCL Compiler by University of Dresden (OCLCUD)* is a tool that can be used independently as the OCL compiler or as part of the free UML modeling tool Argo/UML.
– *UML Specification Environment (USE)*, implemented by Mark Richters in University of Bremen, is an application where the models, invariants, pre-/post-conditions can be specified textually.
– *OCL Compiler*, produced by Cybernetic Intelligence GMBH, is an application for analysing OCL expressions which appear in the UML model. The current version (1.5) of the tool can perform syntactic checking of OCL constraints and consistency check between classes and associated objects.

There are also other OCL-compatible tools such as *Octopus* by Klasse Objecten; *KeY* by University of Karlsruhe, Chalmers University of Technology, Gothenburg, and the University of Koblenz; *OCLE* by Babes Bolyai University; *ModelRun* by BoldSoft(Borland).
Whilst choosing the tool used in this study, the following factors have been taken into account:

a. Syntactic checking of OCL Expressions (type checking,etc.)
b. Semantic checking of OCL Expressions (multiplicity checking for a given model, class-object compliancy checking,etc.)
c. Verification of Invariants for a given instance of the model
d. Verification of pre-/post-conditions for a given scenario
e. Consistency checking for invariants
f. Visual representation of models
g. Dependency on other applications
h. Capability in automating re-usable artifacts

Most of the tools listed are capable of the first two items in the list given above. In addition to these, they all have additional features addressing different needs such as generation of Java code from OCL expressions, providing compliancy with MOF, etc. After careful consideration, we decided to use the tool *USE* especially due to its capabilities in generating automatic snapshots of the system and in validating pre-/post-condition through scenarios. It also allows the creation of re-usable artifacts through series of commands saved in files.

After having selected the tool, we have defined the class diagram of the system under investigation. Once the class diagram is completed, we focused on the invariants of the system. Table 2 lists some of the invariants and respective OCL expressions considered during this stage.

Inv. Name	**Invariant Desc.**
iCurrList	The currency assigned to a pocket must be included in the currency list of the purse. inv iCurrList: avCurrencies->includesAll(pockets.currency)
iTransferLimit	The transfer limit cannot exceed the transfer limit ceiling value. inv iTransferLimit: self.TransferLimit <= self.TransferLimitCeiling
iNoException	Number of exceptions logs is fixed. inv iNoException: exceptionlogs->size() <= cNoException
iDefPocket	At a given time, number of the default pockets is at most one. inv iDefPocket: pockets->select(Default=true)->size() = 1
iNoPocket	Each purse can hold several currencies. Each currency is held in a different pocket and the number of pockets is fixed for a given purse. inv iNoPocket: pockets->size() <= cNoPocket

Table 2. Examples to Invariants

Invariants

The USE tool does not provide a consistency check for the invariants. In other words, there is no way of verifying whether there are conflicting and inconsistent invariants. However, one of the features the tool provides is that it can check whether all the invariants hold for an instance of the model. In UML terms, an instance of a class diagram is called an object diagram. The USE tool does not allow the user to create incorrect bindings between objects. For instance, if the user attempts to create a link between two objects where no association is defined or if the link being created conflicts with the multiplicity rules, then the tool gives immediate warning. The invariant check list is also updated after each action, and therefore the user gets immediate feedback from the tool about the current status of the system.

In this study, we created instances of our model where all the invariants are satisfied. This gave us the confidence that there are no conflicting invariants. However, we are aware that there may still be *restricting* invariants. By restricting invariants, we mean that the effect of some invariants may be stronger than others in which case further refinement may be required. For example, if invariant x states that $a > 0 \wedge a < b$ and a second invariant reads $a < c$ where $c < b$, then clearly second invariant actually restricts the borders defined by the first invariant and the variable a can only be in the range [0,c] when these two invariants are combined. One may think that as long as the invariants are satisfied, the *restricting* invariants would not cause a problem. However, one of our plans in testing our model is to use the invariants as our test data selection criterion and clearly the range of variables is of crucial value in such a process.

After having created the invariants, we have formed a traceability matrix in order to trace the relation between requirements, modules, use cases, constants and invariants. The traceability matrix has revealed the requirements that have been missed out or those that need to be addressed in a later stage.

Pre-/Post-conditions

In the next stage of system modelling, we focused on pre- and post-conditions of the operations. The states of the system are also taken into account since not all the operations are allowed in different states of the system. One down side of the tool used is that it is not possible to draw the state-chart diagrams. To introduce the states of the system to our model, we created a *state* variable to our main class and checked the value of this variable each time we needed to check the state of the system. It is assumed that each transition that causes a state change would also change the value of this variable. Below, we list two versions of a precondition that checks the state of the system where the second version is the one used in this study due to restrictions of the tool.

pre Version1: self.oclInState(Unlocked)
pre Version2: self.LockingState = 'Unlocked'

Another issue encountered at this stage is the distinct set of variables used in the definition of the pre- and post-conditions, because each operation require or restrict the modification of different variables. These variables are also called the *frame variables* [6]. Post-conditions must not only describe all the changes to frame variables, but also make sure that frame variables that do not change are mentioned as unchanged. This second factor revealed some of the missing post-conditions. For instance, the fourth and the fifth post-conditions written for the operation *EraseExceptionLog*, listed in Table 3, are found as the result of this consideration.

One of the assumptions taken at this point is that all the variables except frame variables stayed unchanged during the course of the operation. For the example given in Table-3, we only mentioned the frame variables in the post-conditions and did not create post-conditions in the form of $a = a@pre$ for the rest of the variables. We believe the recognition of this concept in OCL (and UML) tools would prevent the post-condition-related errors caused due to this assumption from arising.

Another point of concern addressed during the determination of post-conditions has been the *messaging* issue. OCL provides *HasSent (' ˆ ')* operator to allow

context MondexPurse:: EraseExceptionLog(p_SequenceNumber : Real) : Boolean

pre EraseExceptionLogPre1:
    self. PurseProviderFlag = true
pre EraseExceptionLogPre2:
    self.LockingState = 'Unlocked' or self.LockingState = 'Locked'
pre EraseExceptionLogPre3:
    exceptionlogs->select(SequenceNumber = p_SequenceNumber)->size()= 1

post EraseExceptionLogPost1:
    exceptionlogs->size() = exceptionlogs@pre->size() - 1
post EraseExceptionLogPost2:
    exceptionlogs->select(SequenceNumber = p_SequenceNumber)->size()= 0
post EraseExceptionLogPost3:
    self._NumberOfUnusedExceptions = self._NumberOfUnusedExceptions@pre + 1
post EraseExceptionLogPost4:
    self.LockingState = self.LockingState@pre
post EraseExceptionLogPost5:
    self.PurseProviderFlag = self.PurseProviderFlag@pre

Table 3. Operation : EraseExceptionLog()

communication between operations [7]. This operator is used when an operation x is called during the execution of another operation y and postcondition of y returns true only if the operation x returns true. The USE tool does not recognise the *HasSent* operator, therefore, instead of mentioning the called operation as the post-condition of the callee operation, we mentioned the post-conditions of the called operation. In other words, we extend the frame of the callee operation by adding the frame of the called operation. Table 4 presents an example to how *HasSent* operator is used in our case study.

In this example, the operation *ChangePersonalCode* returns true if the personal code is changed and required actions are carried out. However, if at the end of the operation, the personal code does not seem to have changed, then the operation expects *ChangeTheStateToLockedOut* to return true and as a result, *ChangePersonalCode* returns false. The reason is if the personal code is not changed, this means that the purse user has entered the code incorrectly more than the times allowed. In such a case, the purse is expected to lock itself out. In our model, we reflected this by inserting the post-conditions of *ChangeTheStateToLockedOut* operation into the post-conditions of *ChangePersonalCode* operation. This technique, as mentioned above, enlarges the frame variables set for *ChangePersonalCode* by adding the variables *_NumberOfIncorrectEntries* and *PersonalCodeAttempts*.

We are aware that the solution proposed above does not ensure the successful execution of the called functions since there is no verification of pre-conditions for the called functions. However, we could not find any evidence of *HasSent* operator handling this issue, either. Instead, the workaround applied in this study is that in the scenarios written for the validation of pre-/post-conditions of operations, we created nested calls to operations, therefore when a new operation

```
context MondexPurse::ChangePersonalCode() : Boolean
...
post ChangePersonalCodePost1:
        % Personal Code changes, desired affects are applied and
        % operation returns true.
        or
        (PersonalCode = PersonalCode@pre and
                self ˆ ChangeTheStateToLockedOut() and
                result = false)
...
```

```
context MondexPurse::ChangePersonalCode() : Boolean
...
post ChangePersonalCodePost1:
        % Personal Code changes, desired affects are applied and
        % operation returns true.
        or
        (PersonalCode = PersonalCode@pre and
                self._ NumberOfIncorrectEntries = self.PersonalCodeAttempts and
                self.LockingState = 'LockedOut' and
                result = false)
...
```

Table 4. An example of how *HasSent* operator is handled

is called from within an operation, the preconditions of called operation are checked automatically. The validation of pre-/post-conditions is further discussed in Section 3.

3 The Validation and Test Data Selection

Pre- and post-conditions determine the accessibility and validity rules for a given operation. Their contribution to a software development process is invaluable not only in modeling and implementation phases, but also in testing.

In [10], Korel et al introduced a test data generation technique where the test case specification is defined in terms of *assertion violation*. According to [10], finding an assertion violation may reveal a fault in the program, a faulty precondition or an erroneous assertion. They tackle the problem of finding program input on which an assertion is violated by reducing it to finding program input on which a selected statements is executed. In practice, they augment the program with the negation of the assertions, and then they used the existing test data generation techniques to find program input to execute these inserted statements in the augmented program. In this study, one of our aims is to use pre- and post-conditions as test data selection criterion early in modelling phase instead of waiting for the code to be produced in implementation phase.With this idea in mind, the following section presents how the validation of pre- and post-conditions is being done in our experiment. We then discuss our plans in extending these validation activities to test data selection.

Scenario-based Validation of Pre/Post-conditions

The objective of this process is to present that given the right inputs, the system can enter and exit operations successfully. In other words, the strength of the pre- and post-conditions are well balanced, so that the access/exit to/from operations are not prevented by too strong pre-/post-conditions.

To address this issue, we make use of the operation execution capability of UML Specification Environment (USE) tool. The tool allows the user to check pre-/post-conditions by calling an operation on an instance of the model, i.e. on an object diagram. The pre-conditions are checked when the *openter* command is executed. If the pre-conditions are satisfied, the operation is put into call stack. After executing the statements between enter and exit points, the tool also checks the post-conditions and the return value when the *opexit* command is run. Further details about how to enter and exit operations are given in [1].

At first, we formed a base object model of the system that satisfies all the invariants. This is to ensure that the scenarios are built on top of a valid state of the system. We then examined the structure of a scenario that is used to validate the pre- and post-conditions of an operation. Following list summarises the command set in such a scenario:

- Setting/Creation of frame variables/objects/operation parameters
- Access to the operation with the correct list of parameters on a given object
- Modification/Deletion of frame variables/objects
- Exit the operation with return value

Note that the term *scenario* is used instead of a *snapshot* in this study. We use the term snapshot for a randomly created object model of the system where as the scenarios are defined in such a way that they serve a purpose. The scenarios can also be seen as tuned versions of snapshots that satisfy a property. Table 5 gives the scenario created to satisfy the pre- and post-conditions of the *EraseExceptionLog()* function given in Table 3.
As shown in Table 5, we first created a new exception log record and linked it to Purse1. Other variables related to state and user of the purse are also set according to the preconditions of the operation. After entering the operation, the log is erased and the number of unused exceptions is incremented by one.
The commands in Table 5 are collected in EraseExceptionLog.cmd file. Analogous to this operation, the scenarios for the rest of the operations are also created as .cmd files and these *command* files are executed in USE command line. After each execution, the invariants of the system are also checked in order to avoid any plausible conflict.
For more effective creation of these .cmd files, we implemented a small VB application that takes the variable names and values from the user and writes into a file in a correct format.

The drawback of this process was that we had to find the set of frame variables as well as the values for these variables that satisfy pre-/post-conditions of each operation. The next section explains our plans in making this process more automatic and using the pre/post-conditions as a way of selecting test data for the system.

<EraseExceptionLog.cmd>
–Setting the Frame Variables ! create exLog1 : ExceptionLog ... ! set exLog1.SequenceNumber := 100 ... ! insert(Purse1,exLog1) into Exception ! set Purse1.PurseProviderFlag := true ! set Purse1.LockingState := 'Unlocked' ! set Purse1._ NumberOfUnusedExceptions := 6
–Enter the operation ! openter Purse1 EraseExceptionLog(100)
–Modification of Frame Variables ! destroy Purse1.exceptionlogs->select(SequenceNumber = 100) ! set Purse1._ NumberOfUnusedExceptions := 7
–Exit the operation ! opexit true

Table 5. Scenario for the validation of EraseExceptionLog()

Test Data Selection

It is important that the system is in the right state when entering an operation and the operation makes the right effect on the system, so the aim of writing pre-/post-conditions is to ensure the correct functioning of the system by checking the status of the system on the entry and exit point of the operations. Within this context, in this section,we explain our plans using pre-/post-conditions in the selection of test data with the aim of exercising the system in crucial points to find faulty situations.

In [8] and [9], Gogolla et al present how to integrate ASSL (*A Snapshot and Sequence Language*) elements to generate scenarios, i.e. snapshots that satisfy a property. They also generate test cases that exercise certain invariants and *validation* cases that proves that no scenario can be found that satisfies the negated version of an invariant.

We plan to apply this technique to the pre-/post-conditions of our system. One big advantage is that there will be scenarios that satisfy a negated version of a pre-/post-condition and these scenarios will still be valid for the system. For instance, the user may attempt to run an operation that can not be run in the current state of the system. In such a case, the pre-condition of the operation must fail and the system must inform the user of the situation. This is a perfectly valid scenario that can be generated by negating a pre-condition and introducing it as a property to be satisfied.

In the generation of scenarios for invariant checking and validation, [8] classifies the commands as those that generate objects and links, those that load invariants and those that introduce negations of the invariants. In addition, there is a command to load the class model with its invariants and related pre-/post-conditions. Moreover, a separate command exists that calls the final scenario satisfying any specification that has been loaded. In the application of the above technique to pre-/post-conditions, the main difference seem to appear in mapping pre-/post-conditions to the scenarios. Table 6 shows an example to this mapping in the case of *EraseExceptionLog* operation. The second column presents the *Operation View* where the pre-/post-conditions of the operation are defined where as the first column, *Scenario View*, gives the statements that match the respective pre-/post-conditions in a scenario.

SCENARIO VIEW	OPERATION VIEW
! insert(Purse1,exLog1) into Exception	pre: exceptionlogs-> select(SequenceNumber = p_SequenceNumber)->size()= 1
! set Purse1.PurseProviderFlag := true	pre : self.PurseProviderFlag = true
! set Purse1.LockingState := 'Unlocked'	pre : self.LockingState = 'Unlocked'
! destroy Purse1.exceptionlogs-> select(SequenceNumber = 100)	post : exceptionlogs->size() = exceptionlogs@pre->size() - 1 post : exceptionlogs->select (SequenceNumber = p_SequenceNumber) ->size()= 0
! set Purse1._NumberOfUnusedExceptions := 7	post : self._NumberOfUnusedExceptions = self._NumberOfUnusedExceptions@pre+1

Table 6. Mapping between Scenario View and Operation View

As seen in Table 6, the mappings concerning value check, such as the ones in the second and third row, are straightforward. On the other hand, the ones that involve addition or deletion of an association or an object require more thorough understanding of the semantics which makes the mapping rather complicated. This issue is further discussed in Section 4.

In addition to this, the concern of invariants being disjoint and mutually exclusive does not necessarily apply to pre-/post-conditions. In other words, it is acceptable to find scenarios where more than one post-conditions fail. Table 7 shows a trivial example of this where a *leaf* object is linked to a *root* object by the *branch* association. First post-condition ensures that the *leaf* elements linked to root element includes the new leaf element and the second post-condition ensures that no leaf element is deleted during this operation, and therefore, the size of *branch* links is incremented by one. This explains that both post-conditions contribute, but in the case of failure in linking the new leaf to the root, both would fail, so in scenarios that are generated with the aim of failing pre-/post-conditions, it is not an obligation to target one pre- or post-condition at a time.

```
–Association
association branch between
    RootObject[1..*] role root
    LeafObject[0..*] role leaf
end
```

```
–Pre-/Post-condition Definition
context RootObject::CreateNewLeaf( p_leaf: LeafObject ) : Boolean
...
post CreateNewLeafPost1 : branch->includes(p_leaf)
post CreateNewLeafPost2 : branch->size = branch@pre->size + 1
...
```

Table 7. Semantically overlapping post-conditions

The following section summarises the lessons learned so far and outlines the suggestions addressing the improvements of OCL tools, particularly USE tool, to cater for the requirements mentioned in this paper.

4 Conclusion

In this paper, we have presented our experience with the USE tool in modeling and validating Mondex Smart Card Application by using UML diagrams and OCL expressions. We believe that OCL is indispensable for use with UML in describing model behaviour and finding evidence for model validation. Especially after having monitored the scenario-execution capabilities for pre-/post-conditions provided by USE, based on OCL, we see great potential for OCL to be also used in test data selection context.

Other OCL- and tool-related observations made during this study are listed below under various subtitles.

Invariants The invariants are well integrated into the model structure in USE tool. We believe the existence of a *invariant consistency check* component in OCL tools will greatly improve the process of validation by removing the necessity to create the instances of a model. This feature would also shorten the process of re-checks when invariants are modified. As mentioned in Section 2, *OCL Compiler* produced by Cybernetic Intelligence GmbH, is expected to have this feature in its next release (version 2.0), but it would be useful if this feature could be integrated to all OCL-related tools/components.

Constants The constants for a system may be fixed in a later stage of the development or even during application loading. It would be helpful to be able to differentiate constants from variables in the modeling stage and to have a user interface similar to the invariants screen, where system constants are listed.

Derived Attributes OCL supports the definition of derived attributes that are prefixed with '/' in UML, but this concept is hardly integrated into the tools. To overcome this problem, we have created invariants ensuring the right values of derived attributes. These attributes are prefixed with '_' character instead of '/' due to the constraints in USE. However, to remove the task of creating extra, invariant-like structures, the tools must handle the automatic setting/modification of derived attributes.

Frame Variables We believe that the integration of the concept of frame variables into OCL tools would enhance the process of pre-/post-condition determination. This is also supported by Kassios on his recent work that focuses on dynamic frames and dependencies [16]. Implementation and integration of dynamic frames to current OCL tools may introduce a new era for pre-/post-condition handling. The idea of including all the variables involved in an operation also supports the idea of completeness of the pre/post-conditions for an operation. Once this new scheme is introduced, OCL tools must also guarantee that variables except the frame variables do not change during the course of an operation by checking the pre and post states of the objects without the user intervention.

HasSent (ˆ)Operator As mentioned in Section 2, *HasSent* operator is used when an operation x is called during the execution of another operation y and postcondition of y returns true only if the operation x returns true. This operator is beneficial for both keeping track of function flow and for checking the pre-/post-condition consistency between two operations. Especially to reveal the infeasible/unreachable functions, the addition of this feature to OCL tools has utmost importance. We believe the tool developers can make use of the *Inline function* concept in implementing this operator in OCL tools.

Pre-/Post-conditions and Scenarios It is our belief that the technique introduced for finding scenarios for invariants in [8] and [9] has great potential for adjustment to pre-/post-conditions in the context of test data selection. However, there are several issues that needs careful consideration.

As observed in Table 6, mapping from Operation View to Scenario View is not a trivial task in many cases. In fact, there is a third layer where Pascal-like ASSL procedures are used in generating several objects, links by using loops, and a fourth layer where these procedures, invariants and scenarios are actually executed. Figure 2 is a simplified version of a deployment diagram given in [8]. In the components given Figure 2, we can observe three syntactically different languages. The *.use* file contains the definition of classes, associations, invariants, pre-/post-conditions. The definition of classes and associations are specific to USE tool, but the definition of invariants and pre-/post-conditions are written in OCL. The *.invs* file also contains invariants written in OCL. These invariants do not hold for all the states of the system, but are necessary in generating scenarios with specific properties. The *.assl* files provide the procedure definitions

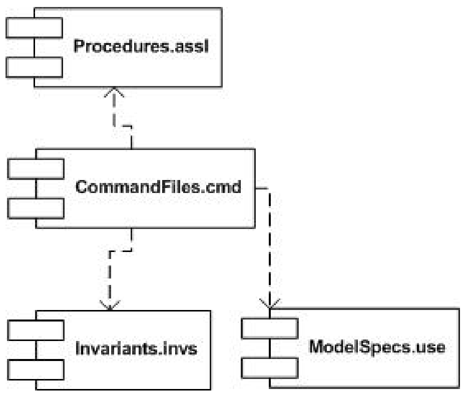

Fig. 2. Deployment Diagram for USE

used in generating n number of objects that may be linked to other objects. As the name suggests, the syntax used in these files is ASSL, which is a Pascal-like language. Finally, the *.cmd* file provides the means of communicating with the tool by loading/reading other files and is specific to USE. It is also possible to create/delete/modify objects by using USE commands written in a .cmd file.

Although these components seem to target different objectives, their functionalities overlap. For instance, in ASSL, if an instance of the association *paymentlogs* will be created between the objects MondexPurse1 and PaymentLog1, the syntax reads *Insert(paymentlogs,[MondexPurse1],[PaymentLog1]*, where as the same effect would be produced if the following command is run in USE Command line: *!insert(MondexPurse1,PaymentLog1) into paymentlogs*. We believe that it is absolutely crucial to minimise such syntactical differences between semantically-similar languages especially if they are used in collaboration with each other. The alternative and the ideal solution would be to create a uniform integrated language that is compliant with UML, OCL and other related OMG standards. For instance, we are aware that there is current work focusing on *MOF object lifecycle operations* submitted to OMG to be a standard [11]. When completed, the standard is expected to include specifications for operations such as object creation, object deletion, object comparison, etc., thus the uniform language suggested above must also comply with such a standard when it is established.

In addition, in formalising the *scenario description language*, assertion-accepting programming languages can also be taken into account. Examples to these languages include Spec#, JML toolset, etc. These languages are used in implementation level, but it would definitely be beneficial for model-based software development and testing if this level of compliancy and compatibility between the scenario description language (ASSL and USE commands in our case), constraint language (OCL) and modeling language (UML) can be achieved in the modeling stage.

Last words As mentioned in [13], besides improved tool support and a clear and concise language description, OCL would benefit from more convincing examples and application scenarios. We believe our experience and findings will help to shed light to future users of UML and OCL, as well as tool developers, and we will continue to report further observations on the matter.

References

1. Gogolla, M., Richters, M., Using the UML Specification Environment(USE). http://citeseer.ist/psu.edu/531031.html , 2002.
2. Clarke, R., The mondex value-card scheme, a mid-term report. Chip-Based Payment Schemes:Stored-Value Cards and Beyond, 1997.
3. Introduction to Mondex Purse Operation, Tech. report., https://mol.mastercard.net/mol/molbe/public/login/ebusiness/smart_cards/mondex/, Mondex International Limited, 1999.
4. Woodcock J., Banach R., The Verification Grand Challenge, Computer Society of India Communications, 2007.
5. Stepney S., Cooper D., Woodcock J., An Electronic Purse: Specification, Refinement and Proof. Oxford University Computing Laboratory, Tech Report, 2000.
6. Sendall S., Strohmeier A., Specifying System Behaviour in UML, Technical Report DI/00/343, EPFL, 2000.
7. Warmer J., Kleppe A., The Object Constraint Language, *Getting Your Models Ready for MDA*, Wesley, 2003.
8. Gogolla M., Bohling J., Richters M., Validation of UML and OCL Models by Automatic Snapshot Generation, Proc. 6th Int. Conf.Unified Modeling Language (UML'2003), Springer, LCND 2863, 2003.
9. Gogolla M., Buettner M., Richters M., USE:UML Specification Environment for Validating UML and OCL, Science of Computer Programming, 2006.
10. Korel B. and Al-Yami A. M., Assertion-oriented automated test data generation, Proceedings of the 18th International Conference on Software Engineering, (ICSE), pages 71-80. IEEE, 1996
11. Adaptive, Compuware Corp, Sun Microsystems, MOF 2.0 Facility and Object Lifecycle Specification, 4th Revised Submission to OMG, 2007.
12. The Object Constraint Language, http://www.um.es/giisw/ocltools/ocl.htm, viewed 06/2007.
13. Baar T.,Chiorean D.,Correa A.,Gogolla M.,Hussmann H.,Patrascoiu O.,Schmitt P.H.,Warmer J., Tool Support for OCL and Related Formalism - Needs and Trends, Technical Report, http://www.db.informatik.uni-bremen.de/publications/Baar_2005_OCLWS.ps, MODELS'05, 2005.
14. Toval A., Requena V., Alemn J.L., Emerging OCL Tools, Software and System Modeling (SoSyM), vol. 2 num., 2003.
15. OCL Tools, http://www.um.es/giisw/ocltools/index.htm#use , viewed 06/2007.
16. Kassios I.T., Dynamic Frames: Support for Framing, Dependencies and Sharing without Restrictions, FM2006, 208-283, 2006.
17. Utting M., Pretschner A., Legeard B., A taxonomy of model-based testing, Tech. report, http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf, University of Waikato, 2006.