Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

# Assembling Scenario Patterns for Checking Model Behavior

Nisha Desai[a] Martin Gogolla[a]

a. University of Bremen, Department of Mathematics and Computer Science, D-28334 Bremen, Germany

Abstract

Model validation and verification are crucial tasks in model-driven software engineering. For checking behavioral model properties, operation call sequences, i.e., dynamic scenarios, can be used, and developers must identify relevant scenarios. Dependent on the model under consideration, many different dynamic scenarios must be taken into account for comprehensive model testing, and it is a challenging task to identify them. In order to give advice and to develop guidelines for constructing dynamic scenarios, we assemble a catalogue of different scenario patterns that can be applied in all models. We explain the catalogue applicability through a collection of exemplary models and validate our proposal through a study conducted with UML and OCL experts.

Keywords Behavioral model validation and verification; Test case; Scenario pattern.

## 1 Introduction

In Model-Driven Engineering (MDE) [BCW17], models are the essential development artifacts. For the MDE design of complex software systems, validation and verification of models are central tasks. Modeling languages like the UML (Unified Modeling Language) [RJB99] applied together with the OCL (Object Constraint Language) [WK99] can be used to describe for a system the structural aspects, e.g., in terms of invariants, and the behavioral aspects, e.g., in terms of operation contracts (i.e., pre- and postconditions). Models considered here utilize UML class models and in particular OCL invariants and OCL contracts for determining structure and behavior.

To check and to test behavioral model properties for a UML model with associations, attributes and operations, a developer can create dynamic scenarios. A scenario can be static (no operation calls, only construction of object models that populate classes, attributes and associations) or dynamic (with additional operation calls). More detailed, the term dynamic scenario refers to a modeling unit consisting of (a) an initial system state (an object model), (b) a finite sequence of operation calls, and

(c) (ideally) system states (object models) after each operation call. If a collection of dynamic scenarios shows common structures, the scenario collection may be abstracted in form of a scenario pattern that puts the common aspects into the foreground.

The notions 'scenario pattern' and 'dynamic scenario' are related as follows: one 'scenario pattern' is an abstraction of many 'dynamic scenarios'; a 'dynamic scenario' may turn out to be a positive or negative test case; with 'positive', we refer to a test case in which all operation calls are executable and yield a resulting object model, and with 'negative' we refer to a test case in which at least one operation call cannot be executed without abnormalities, for example, due to a failing pre- or postcondition, invariant or model-inherent constraint (e.g., a multiplicity constraint); in negative test cases there may be operation calls that do not have a resulting object model.

The developer has to be careful to have in mind and to cover all necessary dynamic scenarios for checking essential model properties. Our aim is to develop guidelines and advice for the construction of dynamic scenarios by offering a catalogue of scenario patterns with which dynamic scenarios can be built. Each single scenario pattern may be understood to be a general template that can applied in concrete modeling situations to yield scenarios. This contribution extends our previous incomplete and short work [DG19a], where we sketched a preliminary catalogue of scenario patterns. In the current contribution, we significantly extend the previous catalogue by introducing new patterns and for the first time show detailed descriptions and explanations of all patterns using different UML and OCL models. We additionally also describe the nature and purpose of all scenario patterns, in order to ease the decision process for a developer when to use which pattern for particular models or model fragments. We also extend the related work.

The patterns proposed in the catalogue consider different class operations and provide suggestions in calling these operations in different orders and in different frequencies. The goal of the catalogue is to reduce the burden of identifying all necessary and crucial scenarios by giving a systematic guideline to the developer for constructing operation call sequences (i.e., dynamic scenarios) that check dynamic model properties. The catalogue is not claimed to be complete, future work can add new patterns.

All scenario patterns are formulated using our model behavior validation and verification technique called filmstripping [GHH+14]. However, the scenario patterns are described independent of our filmstripping approach and can be applied in other tools and approaches as well [AAFR13] [TB05] [CDJ11]. In a validation study of our approach that we have conducted with UML and OCL experts, the experts applied the pattern approach and developed plain operation call sequences for a UML model without invariants and without operation contracts, but in which the operations had an imperative SOIL implementation [BG14]. In the filmstripping approach, our tool USE (UML-Based Specification Environment) [GHD18] is employed in order to transform a UML and OCL application model into an equivalent so-called filmstrip model. Using our model validator [GHD18], we automatically construct test cases in form of (filmstrip) object models for the given dynamic scenarios. By analyzing the state transitions in the resulting filmstrip object models, behavioral properties of a UML and OCL model can be checked. The dynamic scenarios are determined by so-called configurations (specifications that determine how classes, associations and attributes are populated in the filmstrip object models) and additional OCL invariants that are not part of the application model, but serve to direct and help the process of finding relevant test cases.

The paper organization is as follows. Section 2 provides the background of our filmstripping approach. Section 3 describes the idea behind the scenario pattern catalogue. Detailed pattern explanations are discussed in Sect. 4. In Sect. 5 we report on a study that we have conducted with UML and OCL experts with the aim of checking whether the pattern catalogue can be applied in practice. Related work is addressed in Sect. 6. The paper is closed with conclusions and future work in Sect. 7.

## 2 Background: Filmstripping

A UML and OCL model can describe structural properties in terms of OCL invariants and behavioral properties in terms of operation pre- and postconditions. The model validator in the tool USE is specifically designed for structural analysis of models including only invariants. Therefore, in order to validate behavioral aspects, our filmstrip transformation approach [GHH$^{+}$14] is used. As pictured in Fig. 1, filmstripping transforms a given UML and OCL model, that we call an application model and that includes invariants and pre- and postconditions, into an equivalent model, that possesses only invariants and that we call filmstrip model. The resulting filmstrip model involves only structural OCL constraints in form of invariants and can be validated with the USE model validator.

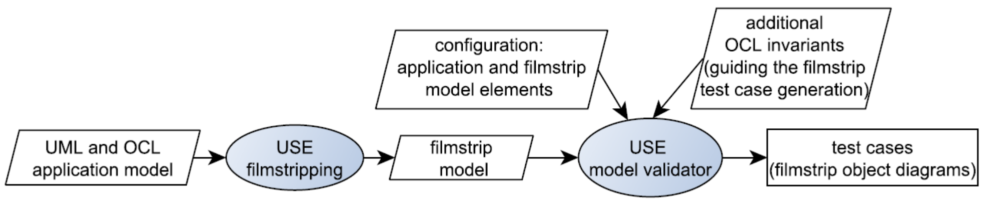


Figure 1 – Overview on Filmstrip Model Validation Process in USE

The filmstripping approach can be explained best in terms of an example. A simple *SocialNetwork* model in which a `Profile` can create, accept or reject a friendship request is chosen as an example. The upper part of Fig. 2 shows the class model of the filmstrip model. The original application model, consisting of the classes `Profile` and `Friendship`, is contained in the filmstrip model and indicated in a gray-shaded style. The small sequence diagram also represents application model elements. The application model is automatically transformed into the filmstrip model: the non-gray shaded classes and invariants (not shown in the figure) are added. In essence, an application model sequence diagram becomes a filmstrip model object model. `Snapshot` objects explicitly allow to capture single system states from the application model. Operation call objects (suffix OpC) describe operation calls originally expressible in the application model. Basically, each operation is transformed into an `OperationCall` class with attributes for a `self` object on which the operation is called and for the operation parameters. Thus, for example, the call `profile3.accept(profile1)` (dotted box in the sequence diagram) is represented by the object `accept_profileopc1` in the filmstrip object model. The effect of the operation call is represented by the differences between the left and the right snapshot: the `accept` operation call changes the attribute `status`. The four `Profile` and the two `Friendship` objects represent different object states before and after the operation call: E.g., the object `profile2` is a later incarnation of the object `profile1`.

Figure 1 gives an overview on the existing filmstrip transformation and model validation process. The filmstrip model along with a so-called configuration (specifying

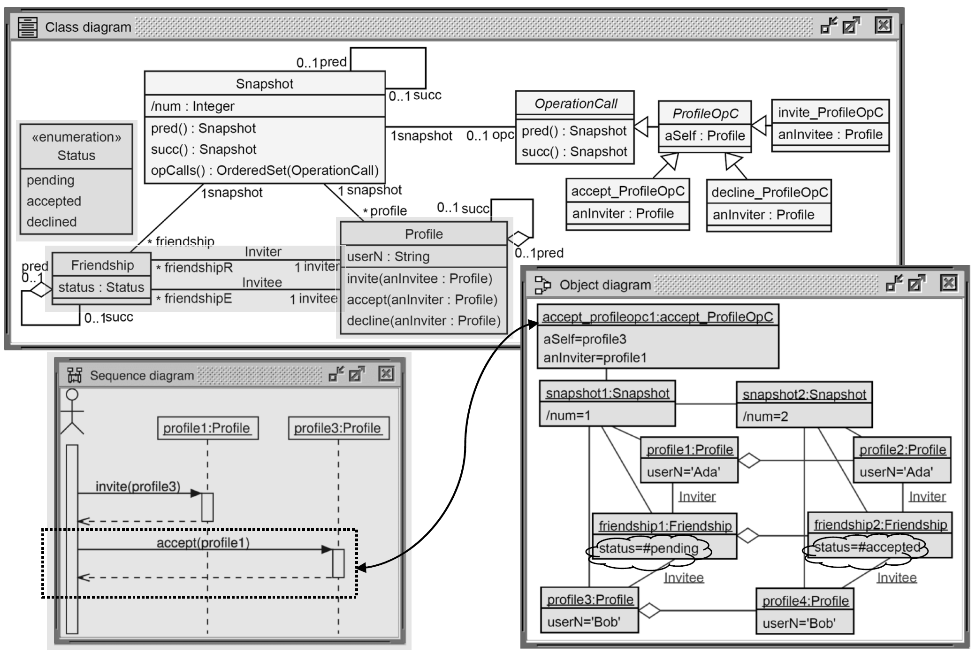


Figure 2 – Application Model (gray-shaded) and Filmstrip Model (Operation Calls become Operation Call Objects).

how the class model is populated; fixing in particular maximal upper bounds for the objects in a class) and additional invariants are given to the model validator. As an outcome, the model validator automatically generates test cases (filmstrip object models like the one in the lower right of Fig. 2) using SAT-based techniques. By analyzing snapshot changes induced by operation calls in the filmstrip model, properties about model behavior can be checked [GH16].

## 3 Scenario Pattern Catalogue

The aim of this chapter is to present the basic idea of our pattern catalogue. We give an overview on all patterns that range from basic simple ones directly applicable to advanced ones needing additional specifications, e.g., in form of OCL expressions. The aim of our scenario patterns is to give methodological support for developers in validating and verifying model properties about dynamics.

We start with a given class model (say, `k` classes named `A, B, C, ...`) with operations (`n` operations in total named `A::opX(), B::opY(), C::opZ(), ...`). We assume a descriptive behavioral model with OCL class invariants and OCL operation contracts is present. The model may optionally contain explicit frame conditions, i.e., postconditions that completely describe an operation by stating for an operation the changed *and* unchanged model parts. A dynamic scenario may then consist of a 'start object model' and a sequence of operations calls '`c.opZ(); b1.opY(); a.opX(); b2.opY()`' on particular objects. In general, different results from executing a dynamic scenario are possible: (a) the operation call sequence (the dynamic scenario) may in total be executable and yield an 'end object model', or (b) it may in between violate some operation contract, invariant or model-inherent constraint and thus will not be executed completely. The aim of the present approach is to develop systematic, rich criteria and guidelines for building such dynamic scenarios that are explained by operation call sequences. Our catalogue currently contains eleven different patterns and is described in tabular form in Fig. 3. The table shows in the columns (1) the pattern name, (2) an informal description of the pattern, (3) the potential number of scenarios induced by the pattern, and (4) one or more example scenarios illustrating the pattern. The patterns have been developed based on our work experience, and based on dynamic scenarios that we have encountered in scenarios from example models and student projects. The catalogue is open for extensions, and one could discover and add more scenario patterns.

All scenario patterns proposed have their own importance and usefulness. But not all patterns may be applicable to all models. The usefulness and applicability of each single pattern can be shortly described as follows. The pattern *ONCE* can be technically applied to all types of models as it checks whether each single operation is executable at all. The pattern *PAIR* can be useful for those models in which operations have a relationship to each other, e.g., one depends on the other. The patterns *ALLOPS* and *SML* can also be applied to all types of models, but it would be more practical to use them for models which have a higher number of operations; ALLOPS may be impractical on very large models. The pattern *REPEAT* is useful for checking whether a single operation can be called repeatedly without another operation in between. Sometimes, an operation which is followed by a specific sequence of operation calls is called again, so that a cycle of operation calls is created. For such models, the pattern *CYCLE* can be applied. The pattern *MOUNTAIN* is useful for models in which operations add and remove model elements (e.g., objects or links). The pattern *DEADEND* applies in particular to those models which have an end object

| Pattern Name | Informal Description | Number of Scenarios | Example Scenario(s) |
|---|---|---|---|
| ONCE | Each single operation once | n | A::opX()<br>B::opY()<br>C::opZ() |
| PAIR | Each operation pair | n*n | A::opX(); A::opX()<br>A::opX(); B::opY()<br>A::opX(); C::opZ()<br>... |
| ALLOPS | All operations from all classes in single operation call sequence | 1 | B::opY(); A::opX(); C::opZ() |
| SML | Short/middle-sized/long (s/m/l) operation call sequences with ~1/4 n, ~1/2 n, ~3/4 n calls | 3 | A::opX()<br>A::opX(); B::opY()<br>A::opX(); B::opY(); C::opZ() |
| REPEAT | Repeatable operation | at most n | A::opX(); A::opX(); ... |
| CYCLE | Operation cycles | at least n*n | A::opX(); B::opY(); ...; A::opX() |
| MOUNTAIN | Adding and successively removing objects or links | no bound expressible | A::opX(); A::opX(); ... ; B::opY(); B::opY(); ... |
| DEADEND | Dead-end call sequences; no further call possible at end | no bound expressible | A::opX(); B::opY(); C::opZ() |
| ABSENCE | Checking absence of an operation in operation call sequence | n | A::opX(); B::opY()<br>A::opX(); C::opZ()<br>B::opY(); C::opZ() |
| INIT2FIN | Call sequences leading from an initial to final condition/state | no bound expressible | {no links present}<br>C::opZ(); B::opY(); A::opX() {all objects linked} |
| REGULAR | Regular expressions over operation calls | no bound expressible | ( A::opX(); B::opY(); ) + ( A::opX() \| C::opZ() ) |

Figure 3 – Catalogue of Scenario Patterns

model in which no operation can be executed any more. The pattern *ABSENCE* can be applied to any model, and it helps the developer to analyze whether some operation in a model is more important than the other operations. In the patterns *INIT2FIN* and *REGULAR*, developers can create initial and final conditions on states or regular expressions over operation calls, respectively, for checking particular behavior of the model. These two patterns can also be applied to any model. In summary, in each single pattern a question is stated about whether particular behavior scenarios have been considered or not. These questions are shown in the table in Fig. 4

| Pattern | Question |
|---|---|
| ONCE | Is each single operation executable? |
| PAIR | Are all possible operation pairs executable? |
| ALLOPS | Is there an operation call sequence that includes all operations from all classes? |
| SML | Are there short, middle-sized and long operation call sequences that in sum include as much operations and classes as possible? |
| REPEAT | Are there operations that can be directly repeated in an operation call sequence without any interrupting other operation? |
| CYCLE | Are there operations that can be repeated in a single behavior scenario with other operations in between? |
| MOUNTAIN | Are there operation call sequences that in the first half build up objects or links and in the second half dismount these objects or links? |
| DEADEND | Are there behavior scenarios such that at the scenario end no operation call is possible any more? |
| ABSENCE | Are there operation call sequences such that a particular operation is not included? |
| INIT2FIN | Are there operation call sequences that lead from a given initial condition or state to a final condition or state? |
| REGULAR | Are there behavior scenarios that follow a given regular expression over operation calls? |

Figure 4 – Questions Stated by Scenario Patterns

The intention of the catalogue, as said, is to give methodological support and advice, in the sense that, after a set of dynamic scenarios has already been constructed in a project, the catalogue can be applied (a) to check whether all relevant scenarios for all relevant properties have been formulated and (b) to give the developer suggestions for further scenarios. In the following, we will instantiate all eleven patterns in an exemplary way and show that all patterns can be applied fruitfully for smaller and larger models.

The table in Fig. 5 summarizes the patterns and the complexity of the used models. The table shows that the models that we have used to instantiate each pattern show partly a highly complex constraint structure, for example, the SML model has 255 pre- and postconditions (4th example row, 5th column). The table displays for each used example model its complexity in terms of the (a) number of classes, associations, operations, invariants, and pre- and postconditions, and (b) number of constructed scenarios. The columns with suffix 'coverage' give a measure for the complexity of the used OCL invariants, preconditions and postconditions: the columns indicate the number of class model elements (attribute, association end) that are accessed in the particular OCL constraints.

| | Application model | | | | | | | | Filmstrip model | | | | | Scenarios | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #Classes | #Invariants | #Non-query Ops | #Pre- and postconditions | #Associations | #Invariants coverage | #Preconditions coverage | #Postconditions coverage | #Classes | #Invariants | #Operations | #Associations | #Invariants coverage | #Test cases | #Objects (in single state) | #Operation calls |
| ONCE - Scheduler | 2 | 3 | 4 | 15 | 3 | 15 | 8 | 157 | 9 | 30 | 9 | 9 | 300 | 4 | 3 | 1 |
| PAIR - CivilStatus | 1 | 1 | 3 | 9 | 1 | 15 | 33 | 103 | 7 | 18 | 9 | 5 | 208 | 9 | 2 | 2 |
| ALLOPS - TollCollect | 2 | 3 | 13 | 55 | 2 | 21 | 53 | 229 | 14 | 73 | 18 | 7 | 588 | 1 | 4 | 8 |
| SML - HealthSystem | 5 | 0 | 17 | 255 | 5 | 0 | 0 | 1328 | 29 | 288 | 22 | 17 | 2634 | 3 | 5 | 4,8,12 |
| REPEAT - PersonCompany | 2 | 4 | 2 | 8 | 1 | 18 | 5 | 48 | 7 | 24 | 7 | 7 | 148 | 2 | 6 | 5 |
| CYCLE - Library | 3 | 8 | 7 | 41 | 2 | 31 | 26 | 289 | 15 | 67 | 12 | 10 | 573 | 2 | 3 | 5 |
| MOUNTAIN - Library | | | | | | | | | | | | | | 1 | 7 | 6 |
| DEADEND - SocialNetwork | 2 | 3 | 3 | 12 | 2 | 11 | 14 | 95 | 8 | 27 | 8 | 8 | 217 | 2 | 3 | 2 |
| ABSENCE - SocialNetwork | | | | | | | | | | | | | | 3 | 3 | 2 |
| INIT2FIN - StudentReport | 3 | 4 | 5 | 23 | 2 | 18 | 25 | 110 | 12 | 43 | 10 | 9 | 386 | 1 | 5 | 9 |
| REGULAR - Account | 1 | 2 | 4 | 18 | 0 | 8 | 8 | 53 | 8 | 27 | 9 | 3 | 156 | 1 | 1 | 6 |

Figure 5 – Evaluation: Patterns together with Evaluated Models, their Complexity and Constructed Scenarios

Let us explain the details for one row in an exemplary way. The first row in the table in Fig. 5 expresses the following: the behavior pattern *ONCE* is explained with a UML and OCL example application model *Scheduler* that has 2 classes, 3 invariants, 4 operations with 15 pre- and postconditions (a single operation can have more than 1 precondition or more than 1 postcondition; for example, 2 separate preconditions could describe independent requirements) and 3 associations; the complexity of the invariants, pre- and postconditions, i.e., the coverage, is measured as 15, 8, 157 (e.g., the value 15 for 'Invariant coverage' means that in the invariants for the model Scheduler, 15 accesses to the attributes or the association ends from the Scheduler class model are made); the automatically transformed filmstrip model has 9 classes, 30 invariants, 9 operations, 9 associations; the coverage of the filmstrip invariants is 300; 4 test cases are constructed, each having 3 objects in each filmstrip snapshot; in each test case there is 1 operation call. Further details about the coverage determination can be found in [DG19b](pp. 77-78).

## 4 Scenario Pattern Explanations

Model validation and verification are conducted using our filmstripping approach, and the proposed catalogue is applied for constructing different test cases. Complete explanations for all eleven scenario patterns with different UML and OCL models and the constructed filmstrip object models are shown in this section. The short paper [DG19a] only discussed two patterns (*PAIR*, *SML*).

### 4.1 Each operation once (ONCE)

*Nature and purpose.* In the pattern *ONCE*, each single operation of the model should be executed as the only operation in a scenario, i.e., if the model has in total `n` operations then `n` different scenarios are considered. The pattern checks the behavior of each single operation separately, before possibly checking further behavior in the context of other operations with other patterns. With this pattern, a necessary condition for the

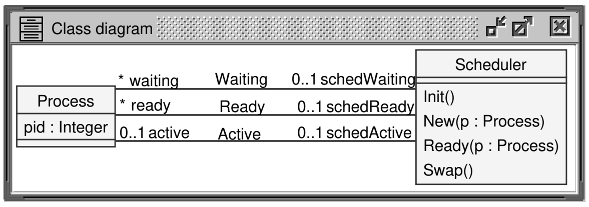
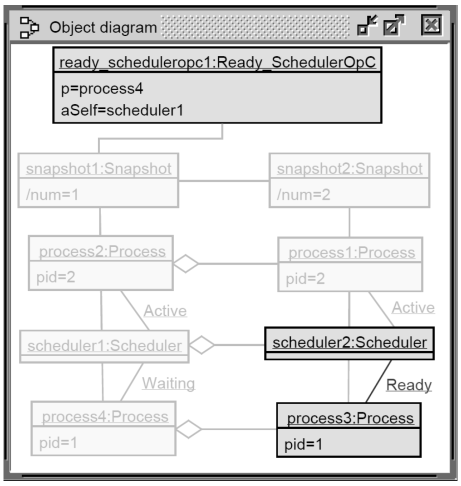


Figure 6 – Class Model of the *Scheduler* Application Model

consistency of the model can be verified by checking whether each single operation is executable at all.
*Example.* The *Scheduler* model [SA05] is chosen to explain this pattern. The model (Fig. 6) has classes `Process` and `Scheduler`. The class `Scheduler` has four operations: (a) the operation `Init` to initiate a new scheduler, (b) the operation `New` to add a new process to the waiting queue, (c) the operation `Ready` to put a process that is currently in the waiting queue either into the ready queue (if the other process is active) or directly activate the process, and (d) the operation `Swap` to swap the currently active process by putting it into the waiting queue. These four operations are separately executed, and the scenarios are constructed in the form of filmstrip object models using the model validator. The filmstrip object models for the operation `Ready` and the operation `Swap` are shown in Fig. 7 and Fig. 8, respectively, and the others can be found in [DG19b].



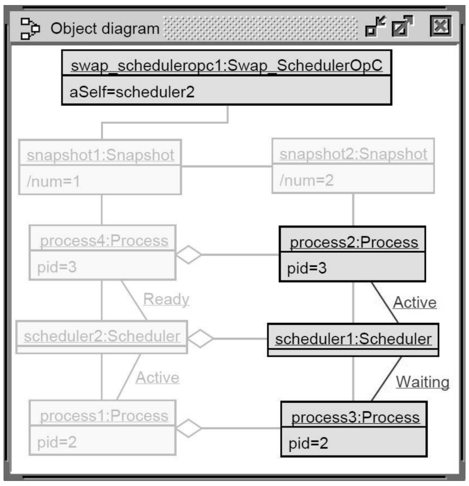


Figure 7 – Filmstrip Object Model - Operation `Ready`

Figure 8 – Filmstrip Object Model - Operation `Swap`

## 4.2 Each operation pair (PAIR)

*Nature and purpose.* The nature of this pattern is to consider all the pairs of operations, i.e., if the model has `n` operations, then `n*n` pairs are considered. It is checked whether a pair can be executed or not. The result has to be compared with the expected behavior. The intention is to detect relationships between operations.
*Example.* The *CivilStatus* model (Fig. 10) is chosen for the demonstration. The model has three non-query operations, namely `birth`, `marry` and `divorce`. Thus, there are

nine possible pairs. In the configuration, two `Person` objects are provided for each snapshot. The model validator tries to execute all operation pairs in order to construct scenarios as filmstrip object models using the constructed filmstrip model and the configuration that in particular fixes maximal upper bounds for the objects in a class.

| Operations | birth(aGender:Gender) | marry(aSpouse:Person) | divorce() |
|---|---|---|---|
| birth(aGender:Gender) | yes | yes | no |
| marry(aSpouse:Person) | no | no | yes |
| divorce() | no | yes | no |

Figure 9 – Executable and Non-Executable Operation Pairs

In Fig. 9, the pairs that could be executed, indicated with [yes], and that could not be executed, indicated with [no], are shown. For example, the `birth-divorce` operation pair cannot be executed because the operation `divorce` cannot be executed without marrying. In the same way, other executable and non-executable pairs can be understood, and the behavior of the operation pairs is validated. The constructed filmstrip object model for the executable `marry-divorce` pair is shown in Fig. 11, and the filmstrip object models of other executable pairs can be found in [DG19b]. In this pattern, not only the executable pairs are interesting, but also the non-executable pairs give feedback to the developer with regard to negative test cases.

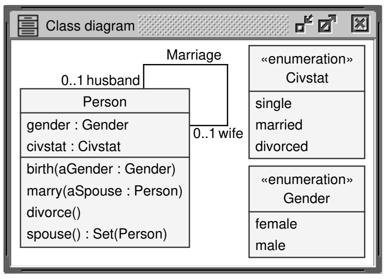


Figure 10 – Class Model of the *CivilStatus* Application Model

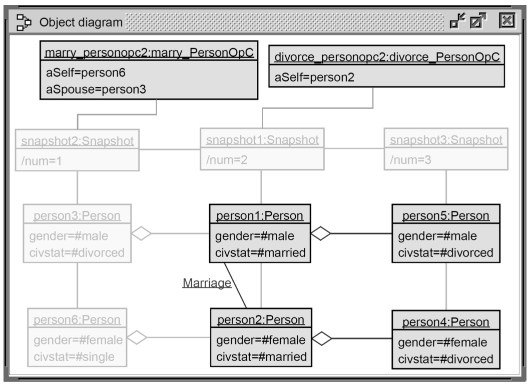


Figure 11 – Filmstrip Object Model - `marry-divorce` Operation Pair

## 4.3 All operations in single operation call sequence (ALLOPS)

*Nature and purpose.* The *ALLOPS* pattern proposes that all operations from all classes should be executed within one single scenario. The purpose is to show that all operations can be successfully executed in the sense that all pre- and postconditions and invariants can be satisfied in a single scenario. For better understanding, we explain this with the abstract example in Fig. 12.

The abstract example model in Fig. 12 has three operations, and there are three operation calls, namely, `o1.op1()`, `o2.op2()` and `o3.op3()`. In the figure, `INVS`, `PRE` and `POST` refer to all invariants, preconditions and postconditions of the model, respectively. Before and after executing each single operation all invariants are satisfied. Before and after the individual operation calls the respective pre- and postconditions are satisfied as well (e.g., before the call `o1.op1()` the preconditions `PRE1` and after the call the postconditions `POST1` are valid in the respective object model). Earlier in the introduction, we have called such a unit consisting of an object model sequence and

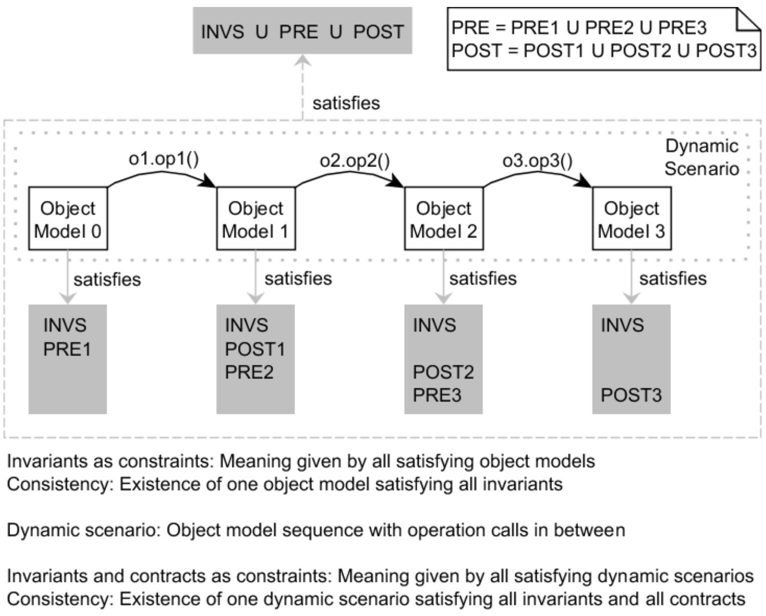

INVS U PRE U POST
PRE = PRE1 U PRE2 U PRE3
POST = POST1 U POST2 U POST3
satisfies
Dynamic Scenario
o1.op1()
o2.op2()
o3.op3()
Object Model 0
Object Model 1
Object Model 2
Object Model 3
satisfies
INVS PRE1
INVS POST1 PRE2
INVS POST2 PRE3
INVS POST3
Invariants as constraints: Meaning given by all satisfying object models
Consistency: Existence of one object model satisfying all invariants
Dynamic scenario: Object model sequence with operation calls in between
Invariants and contracts as constraints: Meaning given by all satisfying dynamic scenarios
Consistency: Existence of one dynamic scenario satisfying all invariants and all contracts

Figure 12 – Abstract Scenario - All Operation Calls

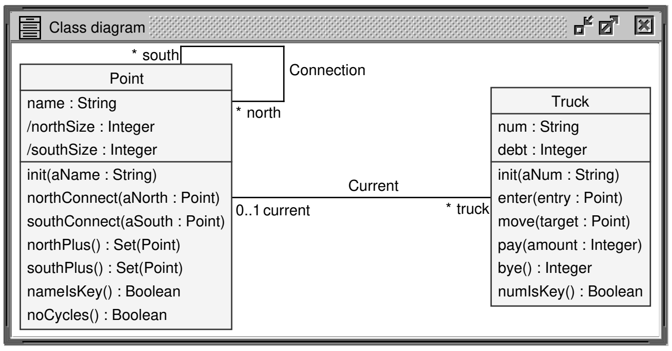


Figure 13 – Class Model of the *TollCollect* Application Model

operation calls in between a *dynamic scenario.* When all invariants and the pre- and postconditions are satisfied, we call it a *satisfying* dynamic scenario. Such a satisfying dynamic scenario is a constructive proof for the consistency of the invariants with the pre- and postconditions, in the sense that there exists an operation call sequence satisfying all pre- and postconditions and invariants and in which all model operations occur. This is analogous to a constructive proof for satisfiability of a set of invariants by providing a single object model in that all invariants are satisfied. A satisfying dynamic scenario including all operations shows that the pre- and postconditions (contracts) and invariants are not contradicting each other, because they can be satisfied.
*Example.* This pattern is demonstrated with the *TollCollect* model (Fig. 13). It has two classes `Point` and `Truck` and in total eight operations. The class `Point` has three operations, namely `init` to initialize a point, `northConnect` to connect to the north point, and `southConnect` to connect to the south point. The class `Truck` has five operations, namely `init` to initialize the truck, `enter` to enter the point connection

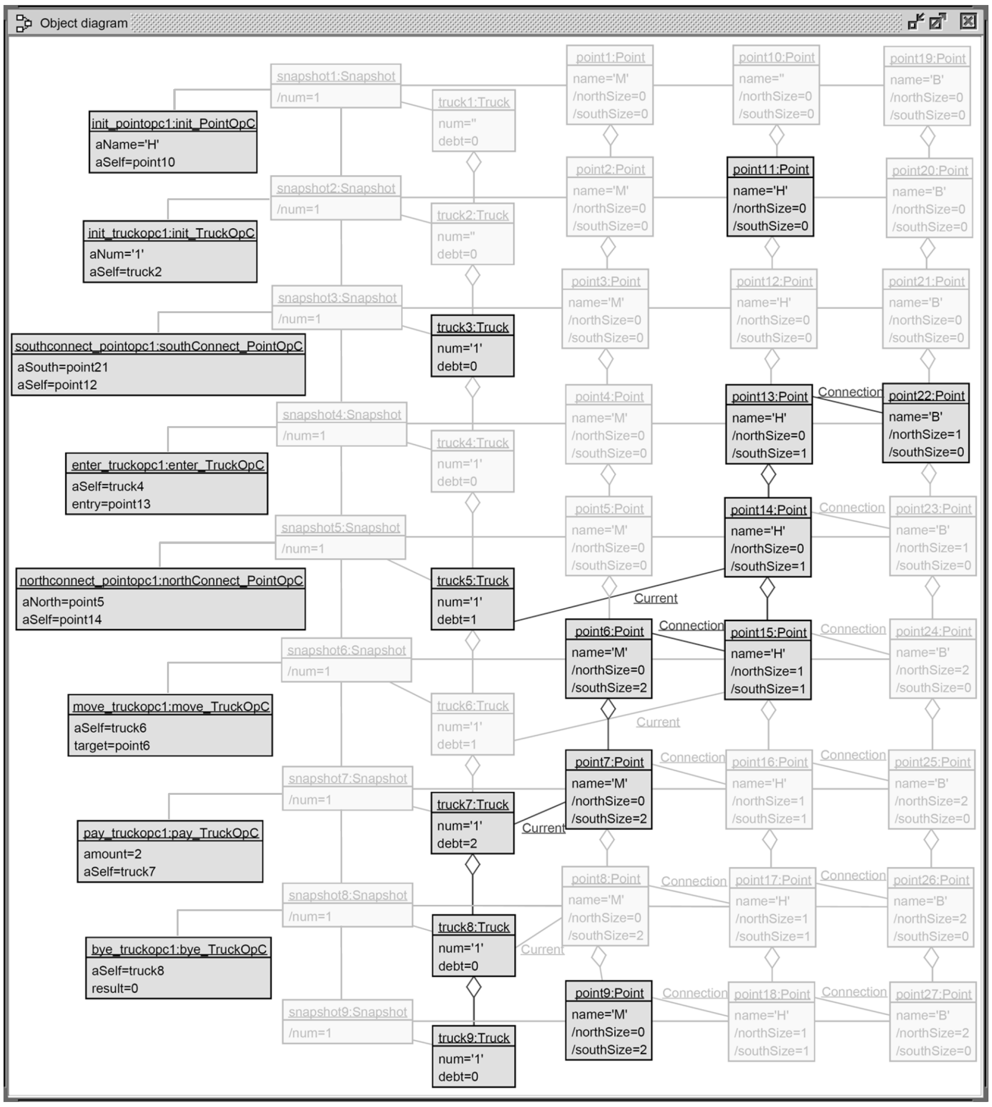


Figure 14 – Filmstrip Object Model - All Operation Calls

network at a particular point, `move` to drive from the current point to another point, `pay` to pay a particular amount, and `bye` to exit the point connection network and to obtain back a possible over-payment. In this pattern, all operations are executed once in a single scenario. In the configuration, for all operations the range is given as 1..1, which makes sure that all operations show up exactly once in the scenario. The filmstrip object model for this scenario is shown in Fig. 14.

## 4.4 Short/middle-sized/long operation call sequences (SML)

*Nature and purpose.* In this pattern, three scenarios with a short, middle-sized and long sequence of operation calls are considered. The goal of this pattern is to approach system behavior incrementally. Here we have decided that a developer starts with a scenario which has a short sequence, then a middle-sized one is considered, and in the end a long sequence is constructed. Instead of taking exactly *three* scenarios (SML), one could construct, if desired, *four* or *five* scenarios with increasing number of operation calls. In order to make the general goal concrete, we decided for taking three scenarios. Step by step, the level of scenario complexity is increased which can help for a better analysis of model behavior. Ideally, if possible, the involved operations in the small, medium-sized and long scenario are disjoint to each other, so that a broader spectrum of operation call sequences is gained.

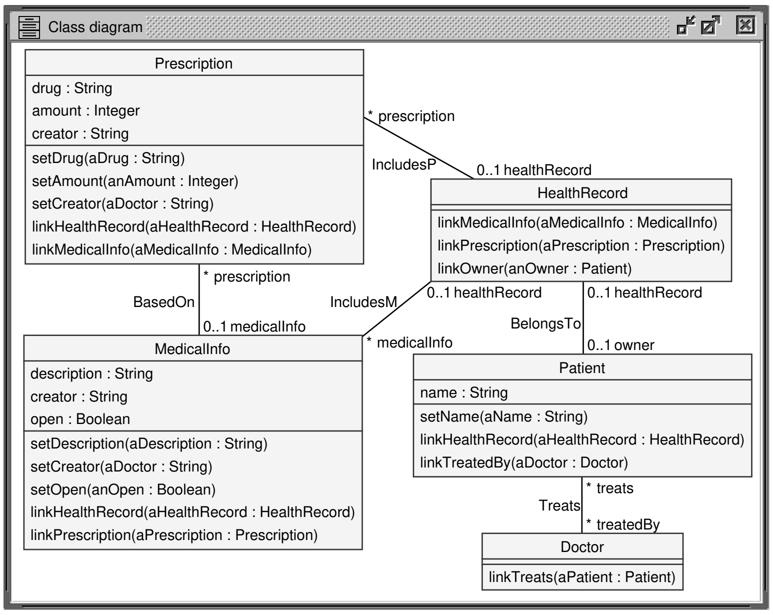

Class diagram
Prescription
drug : String
amount : Integer
creator : String
setDrug(aDrug : String)
setAmount(anAmount : Integer)
setCreator(aDoctor : String)
linkHealthRecord(aHealthRecord : HealthRecord)
linkMedicalInfo(aMedicalInfo : MedicalInfo)
* prescription
IncludesP
0..1 healthRecord
HealthRecord
linkMedicalInfo(aMedicalInfo : MedicalInfo)
linkPrescription(aPrescription : Prescription)
linkOwner(anOwner : Patient)
* prescription
BasedOn
0..1 medicalInfo
IncludesM
0..1 healthRecord
0..1 healthRecord
BelongsTo
* medicalInfo
0..1 owner
MedicalInfo
description : String
creator : String
open : Boolean
setDescription(aDescription : String)
setCreator(aDoctor : String)
setOpen(anOpen : Boolean)
linkHealthRecord(aHealthRecord : HealthRecord)
linkPrescription(aPrescription : Prescription)
Patient
name : String
setName(aName : String)
linkHealthRecord(aHealthRecord : HealthRecord)
linkTreatedBy(aDoctor : Doctor)
* treats
Treats
* treatedBy
Doctor
linkTreats(aPatient : Patient)

Figure 15 – Class Model of the *HealthSystem* Application Model

*Example.* For this pattern, a *HealthSystem* model is chosen, being an adaptation of the model developed in [Bru17]. The model has been modified according to our requirements, as it originally did not have any operations or contracts. We have used the approach from [DG19a] for schematically constructing operations in classes: for each attribute, an operation *set'AttributeName'* is constructed, which manipulates the attribute value; for each role name, an operation *link'RoleName'* is constructed, which adds a specific link. For example, in Fig. 15, the class `Patient` has the attribute `name`, so we have the operation `setName`, which will manipulate the `name` attribute of a `Patient` object. `Patient` is associated with the classes `HealthRecord` and `Doctor` with the role names `healthRecord` and `treatedBy`, resp. Therefore, we have the operations `linkHealthRecord` and `linkTreatedBy`, which will construct links for `BelongsTo` and `Treats`, resp. The post- and frame conditions of the operations are specified using our developed *PCDL* approach [DG19c]. The model is non-trivial and has many operations making it suitable for this pattern.

The model has seventeen operations, and in order to fulfill the pattern criteria, three scenarios with four, eight and twelve operation calls are constructed using the model validator. To assure that each called operation occurs at most once in the scenario, the operations specification is given in the range 0..1 in the configuration. One additional invariant (detailed in [DG19b]) is specified which makes sure that initially (a) there are no links between objects, (b) all String-valued attributes are *empty ('')*,

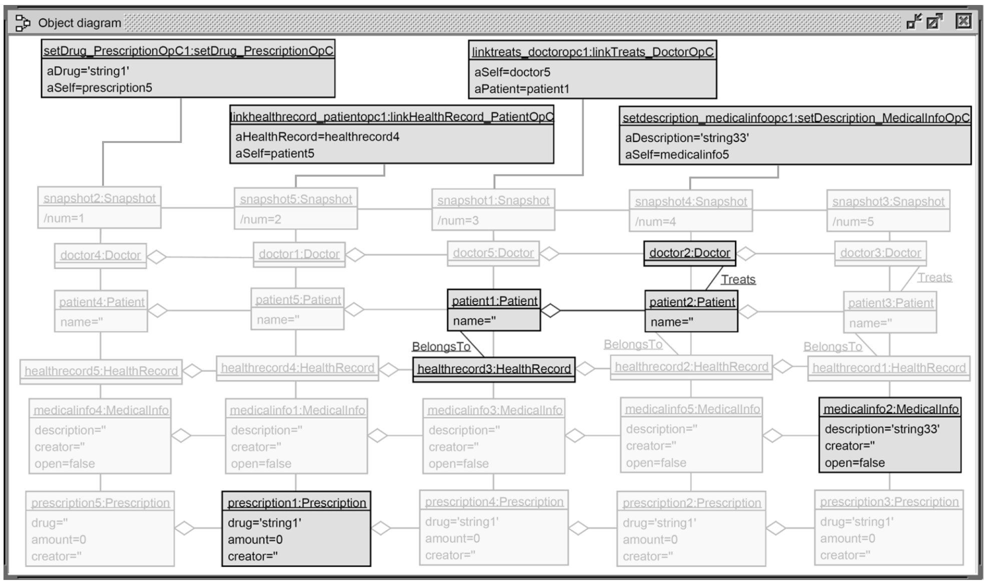


Figure 16 – Filmstrip Object Model - Short Operation Call Sequence

(c) Boolean attribute values are *false* and (d) Integer attribute values are *0*. Figure 16 is the filmstrip object model with the short operation call sequence (4 calls; the call sequences with 8 and 12 calls can be found in [DG19b]).

## 4.5 Repeatable operation (REPEAT)

*Nature and purpose.* In the scenario pattern *REPEAT*, the same operation should be executed repeatedly without any other interrupting call to a different operation. This pattern analyzes the operation properties by checking whether each single operation can be executed repeatedly or not, according to the specific operation behavior.

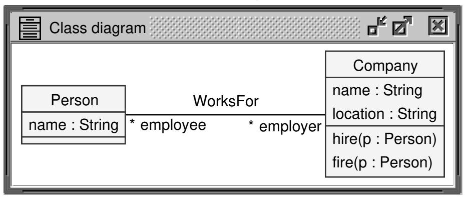


Figure 17 – Class Model of the *PersonCompany* Application Model

*Example.* To demonstrate the pattern, the *PersonCompany* model (Fig. 17) is chosen. In the model, a company can hire and fire people, and to check the behavior, two scenarios are constructed. In the first scenario (Fig. 18), the operation `fire` is repeatedly executed (5 times), and in the second scenario (in [DG19b]), the operation `hire` is called repeatedly.

## 4.6 Operation cycles (CYCLE)

*Nature and purpose.* The first and last operations in the operation call sequence should be the same in this scenario pattern. The task of this pattern is to validate model

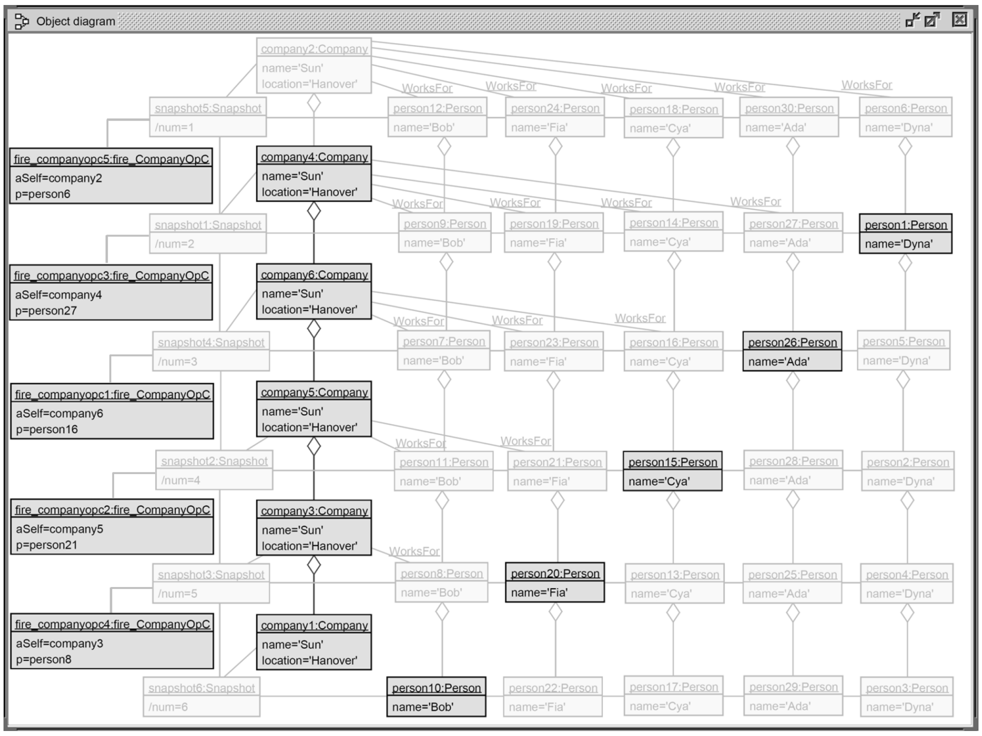

Object diagram
company2:Company
name='Sun'
location='Hanover'
snapshot5:Snapshot
/num=1
fire_companyopc5:fire_CompanyOpC
aSelf=company2
p=person6
company4:Company
name='Sun'
location='Hanover'
snapshot1:Snapshot
/num=2
fire_companyopc3:fire_CompanyOpC
aSelf=company4
p=person27
company6:Company
name='Sun'
location='Hanover'
snapshot4:Snapshot
/num=3
fire_companyopc1:fire_CompanyOpC
aSelf=company6
p=person16
company5:Company
name='Sun'
location='Hanover'
snapshot2:Snapshot
/num=4
fire_companyopc2:fire_CompanyOpC
aSelf=company5
p=person21
company3:Company
name='Sun'
location='Hanover'
snapshot3:Snapshot
/num=5
fire_companyopc4:fire_CompanyOpC
aSelf=company3
p=person8
company1:Company
name='Sun'
location='Hanover'
snapshot6:Snapshot
/num=6
WorksFor
person12:Person name='Bob'
person24:Person name='Fia'
person18:Person name='Cya'
person30:Person name='Ada'
person6:Person name='Dyna'
person9:Person name='Bob'
person19:Person name='Fia'
person14:Person name='Cya'
person27:Person name='Ada'
person1:Person name='Dyna'
person7:Person name='Bob'
person23:Person name='Fia'
person16:Person name='Cya'
person26:Person name='Ada'
person5:Person name='Dyna'
person11:Person name='Bob'
person21:Person name='Fia'
person15:Person name='Cya'
person28:Person name='Ada'
person2:Person name='Dyna'
person8:Person name='Bob'
person20:Person name='Fia'
person13:Person name='Cya'
person25:Person name='Ada'
person4:Person name='Dyna'
person10:Person name='Bob'
person22:Person name='Fia'
person17:Person name='Cya'
person29:Person name='Ada'
person3:Person name='Dyna'

Figure 18 – Filmstrip Object Model - Operation `fire`

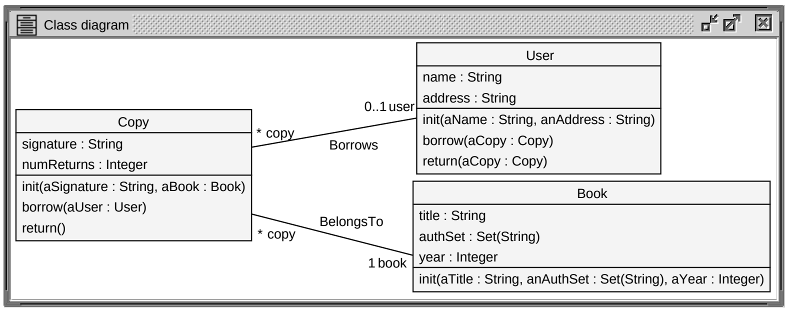


Figure 19 – Class Model of the *Library* Application Model

behavior by checking whether a cyclic call of operations is possible or not, allowing possibly other operation executions to occur between the first and last operation call. *Example.* The *Library* model (Fig. 19) in which users can borrow and return book copies, is used to demonstrate this pattern. To construct a cyclic scenario, the first and last operation calls in the scenario are specified by additional invariants. The invariants along with the configuration are given to the model validator. The cyclic scenarios for the operations `borrow` and `return` of the class `User` are constructed. The scenario in the form of a filmstrip object model for the operation `borrow` of the class `User` is shown in Fig. 20. For the operation `return` of the class `User` the constructed scenario is in [DG19b]. The additional invariants for the operation `borrow` are shown below.

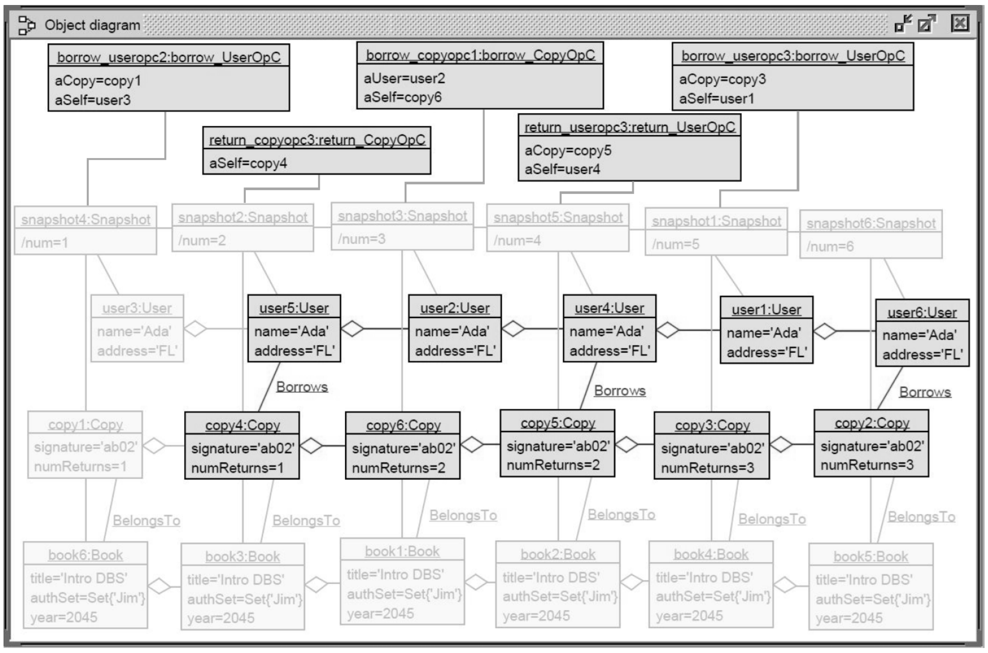


Figure 20 – Filmstrip Object Model - Cyclic `borrow` Operation Calls

```
context Snapshot inv firstOpC:
  self.pred()=null implies self.opc.oclIsKindOf(borrow_UserOpC)

context Snapshot inv lastOpC:
  self.succ()=null implies self.pred.opc.oclIsKindOf(borrow_UserOpC)
```

## 4.7 Adding and successively removing objects or links (MOUNTAIN)

*Nature and purpose.* Here, in the pattern MOUNTAIN, an operation call sequence should be considered that initially has no objects or links, then incrementally adds objects or links until a peak number is reached and afterwards successively removes the objects or links until no more objects or links are present anymore. This scenario will use additional invariants that require respective objects or links to exist.
*Example.* To explain this pattern, the *Library* model (Fig. 19) is used. The `User` class of the *Library* model includes the operations `borrow` and `return`, which respectively add and remove the `Borrow` links between the `User` and `Copy` objects. The three invariants shown below ensure that the constructed scenario first increments and then decrements the number of `Borrow` links.

```
context Snapshot inv initNoLink:
  self.pred()=null implies self.user.copy->size()=0

context Snapshot inv increments:
  (self.pred()<>null and self.pred().num<=3) implies
    self.user.copy->size()=self.pred().num
```

```
context Snapshot inv decrements:
  (self.pred()<>null and self.pred().num>=4) implies
    self.user.copy->size()=self.opCalls()->size()-self.pred().num
```

In the invariants, the expression `self.user.copy->size()` determines the number of `Borrow` links in the snapshot. The integers `3, 4` in the expressions `self.pred().num <= 3` and `self.pred().num >= 4` care for the size of the mountain, i.e., until the fourth snapshot (third operation call) a link will be added, and from the fifth snapshot (fourth operation call) a link will be removed.

Furthermore, the invariants are reusable in the sense that the developer only needs to change the expression which defines the number of links, and the integers which define the top of the mountain. In the configuration, the snapshot is specified in the range 7..7. Figure 21 shows the constructed filmstrip object model, in which first three `Borrow` links are added (operation `borrow`) and then three `Borrow` links are removed (operation `return`). The filmstrip object model also contains the `Book` objects not shown in Fig. 21.

## 4.8 Dead end call sequences (DEADEND)

*Nature amd purpose.* In the scenario pattern *DEADEND*, no further operation call should be possible at the end of the operation call sequence. In a model, sometimes operations require a specific call sequence and further execution of an operation call is forbidden. Not all models are applicable to this pattern, as an end object model is required in which no operation can be called.
*Example.* The pattern is demonstrated using the *SocialNetwork* model (Fig. 2), in which a user can initiate a friendship with an invite, and then another user can accept or decline the friendship request. To realize the pattern, in the configuration, two `Profile` objects are specified, and the operation specification is in the range 2..3. In the constructed scenarios (filmstrip object models), only two operation calls exist. The first operation call is an `invite`, and the other operation call is an `accept` (Fig. 22) or

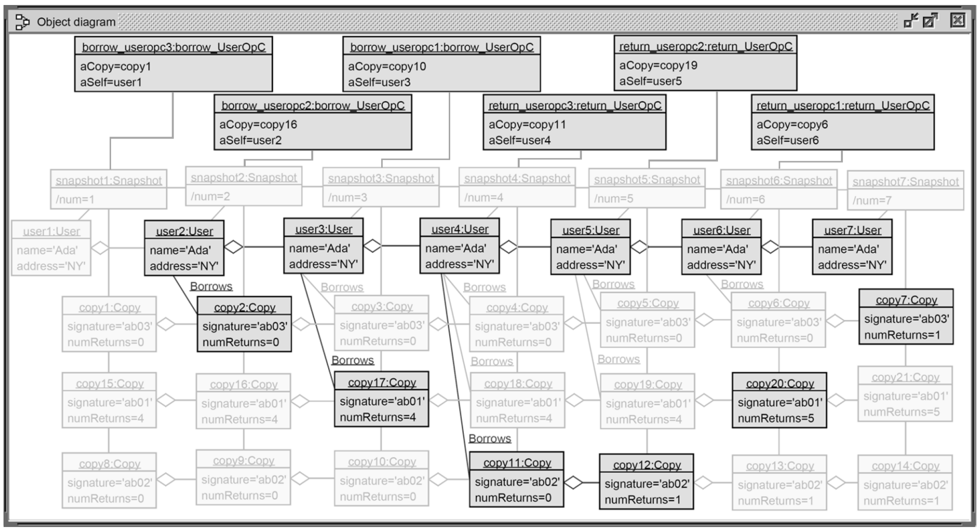


Figure 21 – Filmstrip Object Model - Number of `Borrow` Links: 0,1,2,3,2,1,0 (Mountain)

a `decline` (Fig. 23). As there are only two `Profile` objects, no further operation call is possible. If the operation specification is given in the range 3..3, then the filmstrip object model generation for this scenario is unsatisfiable. This implies that in this example, after two operation calls, no further operation call is possible, and there is a dead end.

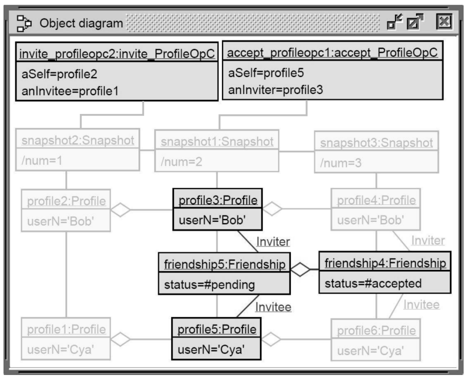


Figure 22 – Filmstrip Object Model - `invite-accept` Operation Call

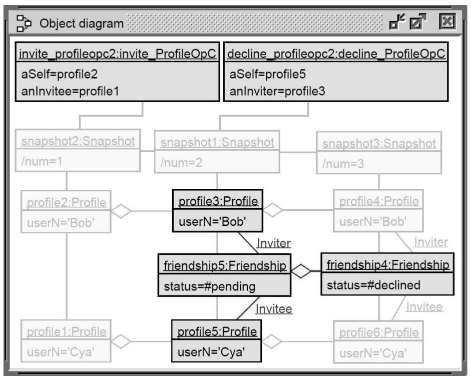


Figure 23 – Filmstrip Object Model - `invite-decline` Operation Call

## 4.9 Absence of an operation in an operation call sequence (ABSENCE)

*Nature and purpose.* The pattern *ABSENCE* requires that all operations should be called in the scenario except one which is intentionally left absent. This pattern helps to identify crucial operations that have to be present in all scenarios. From the operation signature viewpoint, all operations may look similar, but by using this pattern, one can analyze whether some operations are more important than others. In particular, feedback in the sense that no operation sequence can be found may occur for this pattern. The pattern tries to give a judgement on operation importance and tries to answer the question: Is the operation really needed?
*Example.* The *SocialNetwork* model (Fig. 2) is again used to demonstrate this pattern. It has three operations (`invite`, `accept`, `decline`). According to the pattern, three scenarios are possible by keeping each single operation absent, one at a time, i.e., absence of `invite`, absence of `accept` or absence of `decline`. In the configuration, two `Profile` objects are specified in each snapshot. The constructed scenario with absence of `accept`, shows the operation call sequence `invite-decline` (Fig. 23), and absence of `decline` shows the operation call sequence `invite-accept` (Fig. 22). The scenario generation for absence of `invite` is unsatisfiable because without a friendship invite there cannot be an accept or a decline for a friendship request, i.e., the operation `accept` or `decline` cannot be called without the operation `invite`. The conclusion is that the operation `invite` plays somehow a more important role in the model than the operations `accept` or `decline`.

## 4.10 Operation call sequences leading from an initial to a final condition (INIT2FIN)

*Nature and purpose.* This pattern provides an opportunity to the developer to give specific initial and final conditions so that an operation call sequence is constructed that leads from an initial to a final state or condition. This pattern helps to check particular

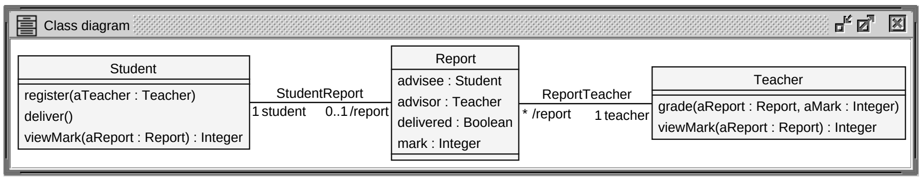


Figure 24 – Class Model of the *StudentReport* Application Model

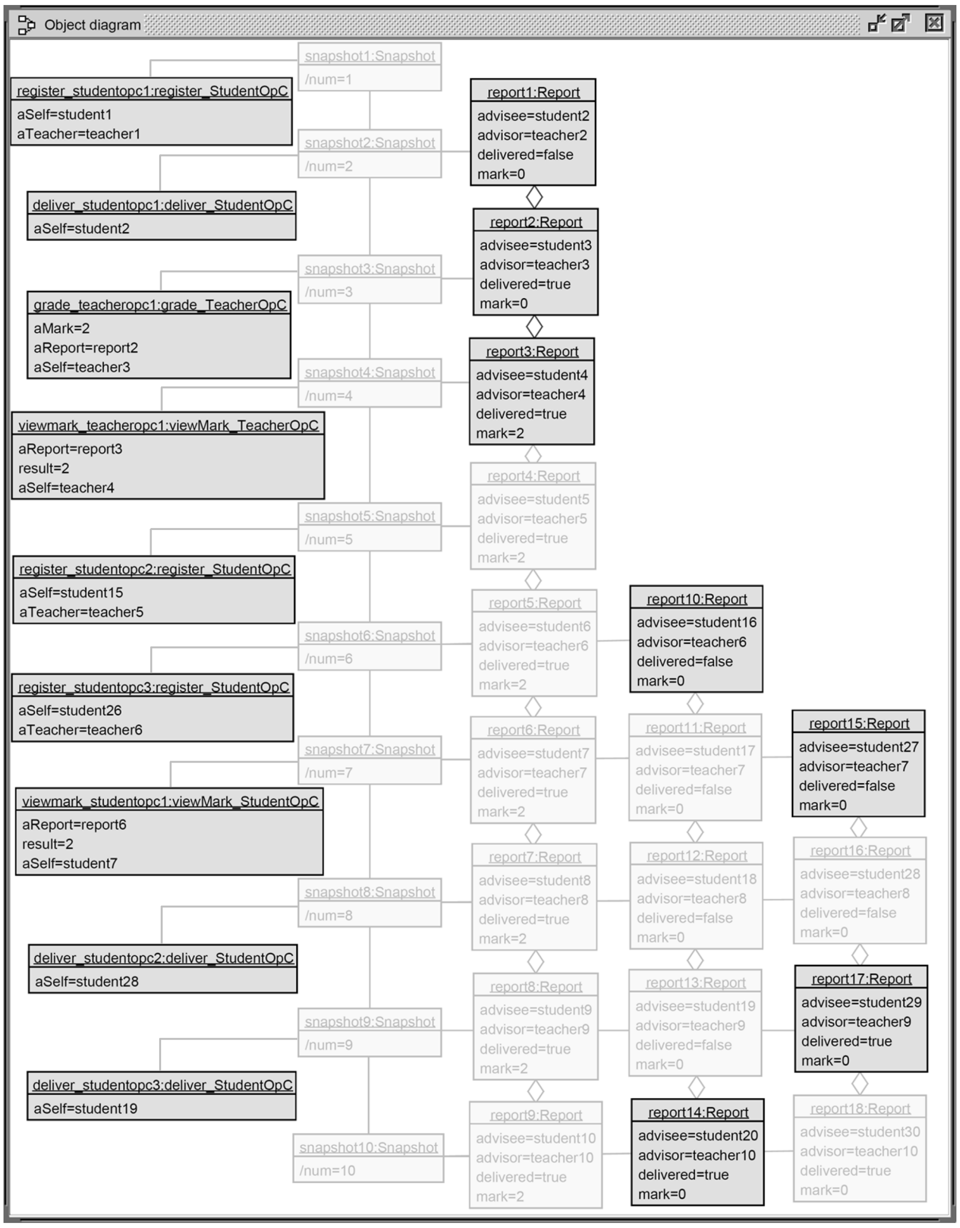


Figure 25 – Filmstrip Object Model - Initial and Final State Conditions

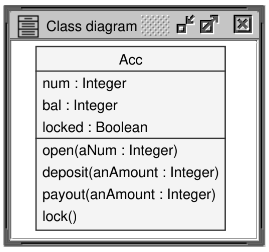


Figure 26 – Class Model of the *Account* Application Model

properties of operations by accordingly specifying initial and final requirements.
*Example.* The pattern is demonstrated using the *StudentReport* model (Fig. 24), in which a student can register and deliver a report, a teacher can give grades, and both can view the marks. To construct the scenario using this pattern, the initial and final state conditions are given to the model validator together with the configuration. In the initial state condition (`initNotReport`), no reports exist, and in the final state condition (`finAllReportDelivered`), there exist at least one report, and all existing reports must be delivered. In the configuration, the operation specification is in the range 9..9. The scenarios in the form of filmstrip object models are constructed, and one is shown in Fig. 25. The figure shows that initially there is no report and in the final snapshot, three reports exist and all are delivered, i.e., the given initial and final state conditions are satisfied. The filmstrip object model also contains the `Student` and `Teacher` objects not shown in Fig. 25.

```
context Snapshot inv initNotReport:
  self.pred()=null implies self.report->size() = 0

context Snapshot inv finAllReportDelivered:
  self.succ()=null implies self.report->size() <> 0 and
    self.report->forAll(r|r.delivered)
```

## 4.11 Regular expressions over operation calls (REGULAR)

*Nature and purpose.* In the scenario pattern *REGULAR*, an operation call sequence should be constructed according to a given regular expression over operations, which can be expressed in terms of additional OCL invariants. The intention of this pattern is to check whether a scenario that satisfies the given regular expression can be constructed or not.
*Example.* In order to explain this pattern, the *Account* model (Fig. 26) is used which has one class `Acc`. The class `Acc` has four operations: (a) `open` to specify the account number and to unlock the account, (b) `deposit` to deposit a specific amount in the account which increases the balance accordingly, (c) `payout` to payout the amount from the account which decreases the balance accordingly, and (d) `lock` to lock an open account. To fulfill the criteria for this pattern, the regular expression `open();[deposit();payout();]+lock();` is specified, and the corresponding OCL invariants are shown below.

```
context Snapshot inv openFirst:
  open_AccOpC.allInstances->forAll(open|open.pred().pred()=null)
```

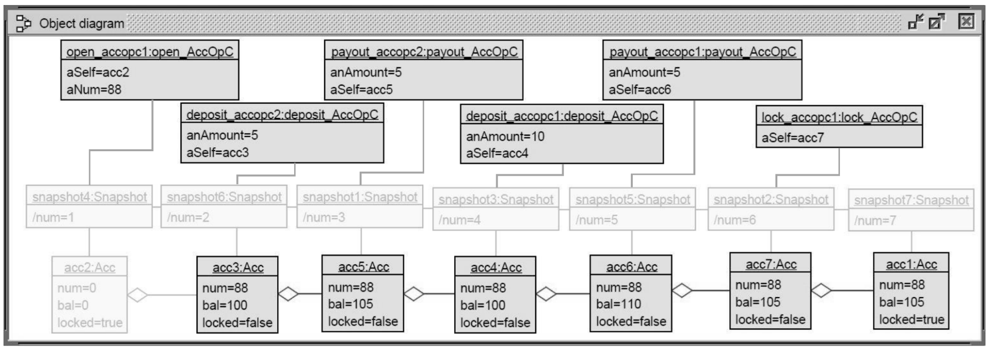


Figure 27 – Filmstrip Object Model - Regular Expression

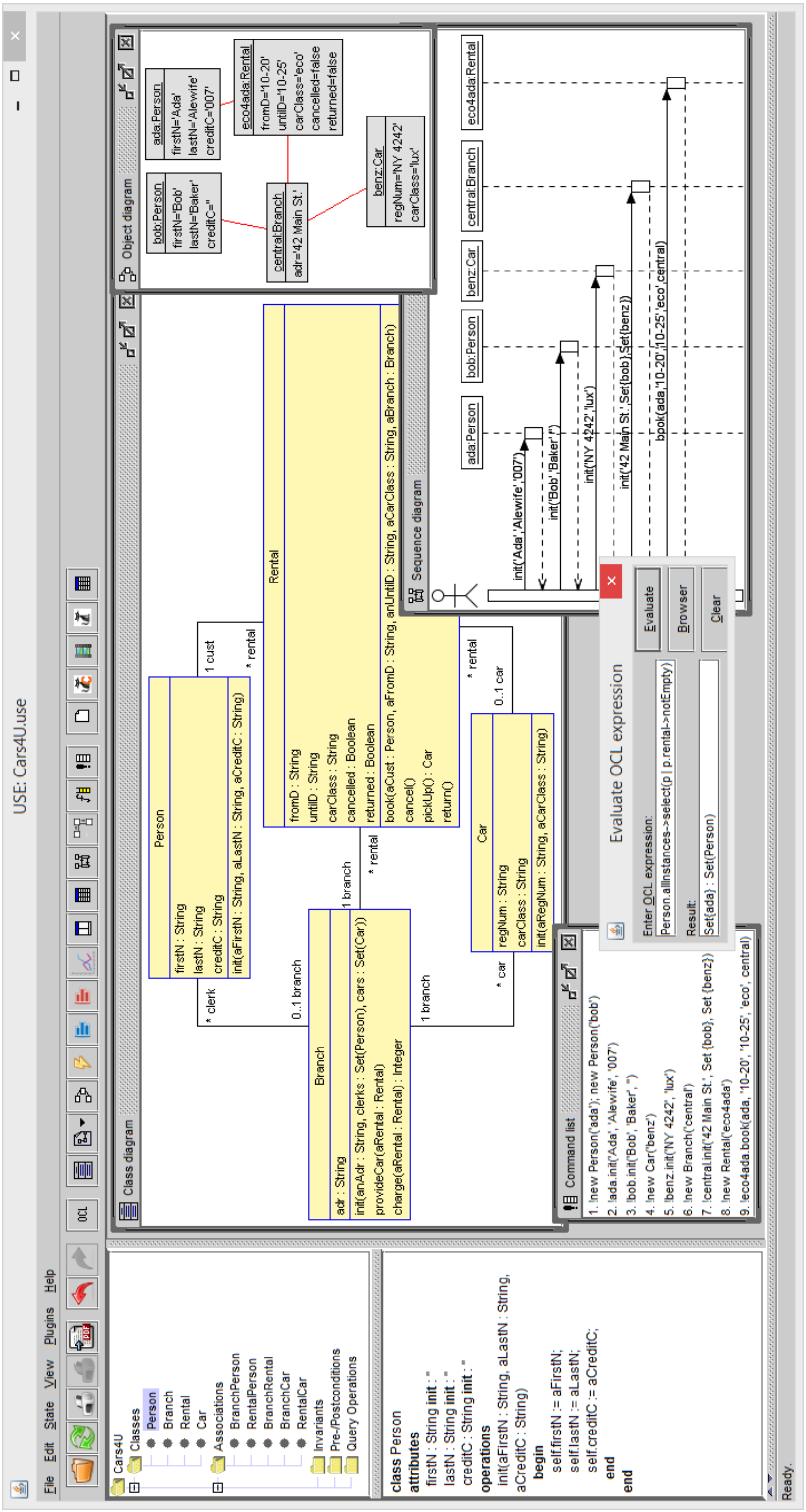
```
context Snapshot inv lockLast:
  lock_AccOpC.allInstances->forAll(lock|lock.succ().succ()=null)

context Snapshot inv depositPayout_evenUnevenSnapshot:
  deposit_AccOpC.allInstances->forAll(d|d.pred().num.mod(2)=0) and
    payout_AccOpC.allInstances->forAll(p|p.pred().num.mod(2)=1)
```

In the configuration, the operation specification is in the range the 6..6. Based on the given invariants and the configuration, the scenario in the form of a filmstrip object model is constructed and shown in Fig. 27. As seen in the figure, the first and last operation call are `open` and `lock`, respectively, and in between two times a `deposit-payout` operation call sequence is constructed. This complete operation call sequence satisfies the given regular expression.

## 5 Approach Validation by a Study with Specialists

**Study goal:** The goal of the study was to validate whether our approach utilizing patterns for developing behavioral test scenarios can be practically applied and whether developers would accept the pattern idea and use the patterns for creating new test cases in situations where some test cases are already present in a project.
**Study subjects and process:** In total, 11 subjects participated in the study. All subjects (a) posses a master in computer science (8 subjects; among them 5 subjects working at University of Bremen, 3 subjects working in industry; subjects at University of Bremen with 1-4 years work experience; subjects in industry 0.5-7 years work experience) or (b) are as students in the process of writing their master thesis (3 subjects). All subjects possess detailed know-how in object-oriented modeling, UML and OCL and our deployed tool USE through participation in a 14 week, 3 x 90 minutes per week university course. The study took place in two rounds.

In Round 1, the topic of the study was introduced by a modeling example: A car rental UML model (see Fig. 28) with 4 classes, 11 data-valued attributes and 9 operations realized in SOIL (Simple Ocl-like Imperative Language) [BG14]. The subjects were told that: "The model is (purposefully) not completed yet. Some important parts like OCL invariants, pre- and postconditions are missing. However, all operations have a preliminary SOIL implementation that explains the intended



```
class Person
attributes
  firstN : String init : ''
  lastN : String init : ''
  creditC : String init : ''
operations
  init(aFirstN : String, aLastN : String,
aCreditC : String)
  begin
    self.firstN := aFirstN;
    self.lastN := aLastN;
    self.creditC := aCreditC;
  end
end
```

```
1. !new Person('ada'); new Person('bob')
2. !ada.init('Ada', 'Alewife', '007')
3. !bob.init('Bob', 'Baker', '')
4. !new Car('benz')
5. !benz.init('NY 4242', 'lux')
6. !new Branch('central')
7. !central.init('42 Main St.', Set {bob}, Set {benz})
8. !new Rental('eco4ada')
9. !eco4ada.book(ada, '10-20', '10-25', 'eco', central)
```

Enter OCL expression:
`Person.allInstances->select(p | p.rental->notEmpty)`

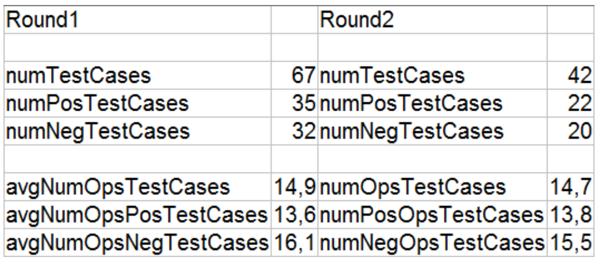
Result:
`Set{ada} : Set(Person)`

Figure 28 – UML Model Used in the Study with UML and OCL Specialists

functionality of the operation (preliminary means that some operations might not operate as one would expect, for example in exceptional cases, like empty set, empty string or the like). Your task is to develop behavioral test cases (tests for the operations and operations sequences) in form of SOIL command sequences (positive test cases, in the sense that everything works in a proper way as expected; negative test cases, in the sense that on execution an error is reported). Your test cases should check whether the current model meets your expectations, i.e., whether operation calls induce the respective system changes, return the desired result (if applicable), or show an error message in case of illegal operation calls."

In Round 2, we have been offering to the subjects the pattern approach in form of a paper, a condensed 14-page version of the current contribution. We asked to read the paper and let the subjects develop further test cases. In addition, we included a questionnaire about whether subjects without knowing the scenario pattern approach applied one or many patterns implicitly in Round 1 and whether subjects developed new test cases with previously unknown patterns.

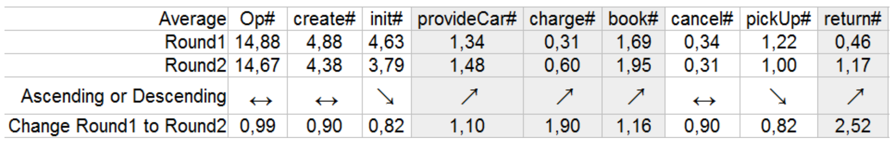
**Study results:** The table in Fig. 29 gives an overview on the number and structure of developed test cases by the subjects, i.e., the number of positive and negative cases and the number of tested operation calls and commands. Basically there is no difference between Round 1 and Round 2. The length of negative test cases is slightly greater than for positive ones.



| Round1 | | Round2 | |
|---|---|---|---|
| | | | |
| numTestCases | 67 | numTestCases | 42 |
| numPosTestCases | 35 | numPosTestCases | 22 |
| numNegTestCases | 32 | numNegTestCases | 20 |
| | | | |
| avgNumOpsTestCases | 14,9 | numOpsTestCases | 14,7 |
| avgNumOpsPosTestCases | 13,6 | numPosOpsTestCases | 13,8 |
| avgNumOpsNegTestCases | 16,1 | numNegOpsTestCases | 15,5 |

Figure 29 – Overview on Obtained Test Cases Developed by Subjects

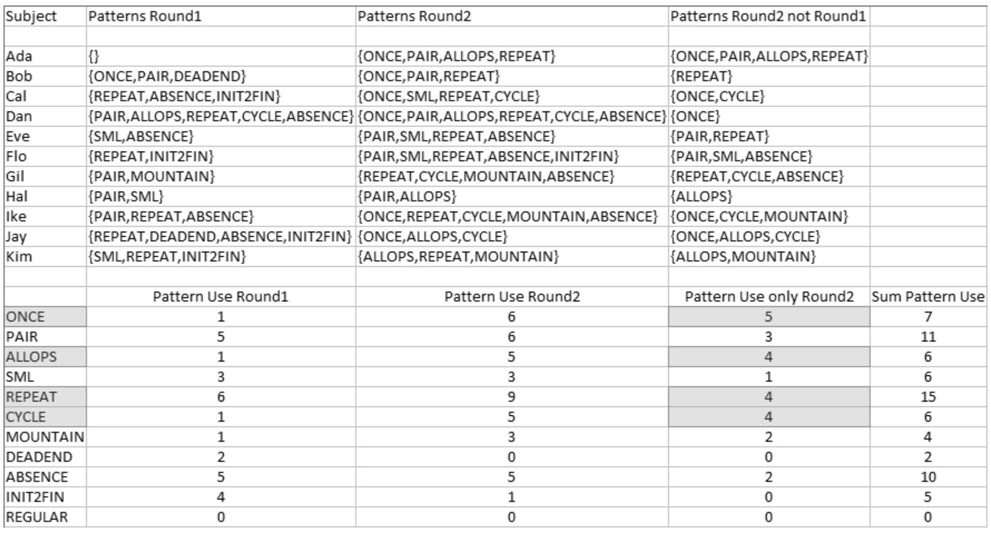
The table in Fig. 30 compares the tested operations in the two rounds. It is interesting to see that the patterns lead to a slightly different use of operations. In Round 2 the degree of operations more related to the end of the 'natural' operation cycle increased. We conclude that the patterns gave rise to fully consider all options in operation call sequences whereas in Round 1 some cornerstones were missing in the test cases.



| Average | Op# | create# | init# | provideCar# | charge# | book# | cancel# | pickUp# | return# |
|---|---|---|---|---|---|---|---|---|---|
| Round1 | 14,88 | 4,88 | 4,63 | 1,34 | 0,31 | 1,69 | 0,34 | 1,22 | 0,46 |
| Round2 | 14,67 | 4,38 | 3,79 | 1,48 | 0,60 | 1,95 | 0,31 | 1,00 | 1,17 |
| Ascending or Descending | ↔ | ↔ | ↘ | ↗ | ↗ | ↗ | ↔ | ↘ | ↗ |
| Change Round1 to Round2 | 0,99 | 0,90 | 0,82 | 1,10 | 1,90 | 1,16 | 0,90 | 0,82 | 2,52 |

Figure 30 – Comparison Round 1 vs. Round 2: Tested Operations

The table in Fig. 31 shows the subjects' opinion on implicit use of the patterns in Round 1 and on explicit use of the patterns in Round 2. In fact, some patterns proved to be more popular than others. It is also remarkable, that the marked patterns ONCE, ALLOPS, REPEAT, and CYCLE were applied by a relative high number of subjects for the first time in Round 2. So these patterns did de facto have an impact on the constructed test cases.



| Subject | Patterns Round1 | Patterns Round2 | Patterns Round2 not Round1 | |
|---|---|---|---|---|
| | | | | |
| Ada | {} | {ONCE,PAIR,ALLOPS,REPEAT} | {ONCE,PAIR,ALLOPS,REPEAT} | |
| Bob | {ONCE,PAIR,DEADEND} | {ONCE,PAIR,REPEAT} | {REPEAT} | |
| Cal | {REPEAT,ABSENCE,INIT2FIN} | {ONCE,SML,REPEAT,CYCLE} | {ONCE,CYCLE} | |
| Dan | {PAIR,ALLOPS,REPEAT,CYCLE,ABSENCE} | {ONCE,PAIR,ALLOPS,REPEAT,CYCLE,ABSENCE} | {ONCE} | |
| Eve | {SML,ABSENCE} | {PAIR,SML,REPEAT,ABSENCE} | {PAIR,REPEAT} | |
| Flo | {REPEAT,INIT2FIN} | {PAIR,SML,REPEAT,ABSENCE,INIT2FIN} | {PAIR,SML,ABSENCE} | |
| Gil | {PAIR,MOUNTAIN} | {REPEAT,CYCLE,MOUNTAIN,ABSENCE} | {REPEAT,CYCLE,ABSENCE} | |
| Hal | {PAIR,SML} | {PAIR,ALLOPS} | {ALLOPS} | |
| Ike | {PAIR,REPEAT,ABSENCE} | {ONCE,REPEAT,CYCLE,MOUNTAIN,ABSENCE} | {ONCE,CYCLE,MOUNTAIN} | |
| Jay | {REPEAT,DEADEND,ABSENCE,INIT2FIN} | {ONCE,ALLOPS,CYCLE} | {ONCE,ALLOPS,CYCLE} | |
| Kim | {SML,REPEAT,INIT2FIN} | {ALLOPS,REPEAT,MOUNTAIN} | {ALLOPS,MOUNTAIN} | |
| | | | | |
| | Pattern Use Round1 | Pattern Use Round2 | Pattern Use only Round2 | Sum Pattern Use |
| ONCE | 1 | 6 | 5 | 7 |
| PAIR | 5 | 6 | 3 | 11 |
| ALLOPS | 1 | 5 | 4 | 6 |
| SML | 3 | 3 | 1 | 6 |
| REPEAT | 6 | 9 | 4 | 15 |
| CYCLE | 1 | 5 | 4 | 6 |
| MOUNTAIN | 1 | 3 | 2 | 4 |
| DEADEND | 2 | 0 | 0 | 2 |
| ABSENCE | 5 | 5 | 2 | 10 |
| INIT2FIN | 4 | 1 | 0 | 5 |
| REGULAR | 0 | 0 | 0 | 0 |

Figure 31 – Comparison Round 1 vs. Round 2: Applied Patterns by Subjects

As a summary, we conclude that the idea of the patterns was on the one hand already familiar to the subjects, on the other hand some patterns were accepted very well in order to construct new test cases for situations not considered before.

## 6 Related Work

In order to compare our contribution to similar works, we first present related approaches that propose or use different patterns for model specification and model checking, and then we discuss related work considering approaches which aim at helping in designing and generating test cases.

**Pattern-based approaches:** The authors in [AT06] present a library of reusable OCL specification patterns to simplify the constraint definition in a UML/OCL behavioral model. The work [FHKS18] introduces a model and a scenario-based pattern catalog to ensure the quality of specifications for the Modal Sequence Diagrams (MSD) requirement language. The paper [GKC07] introduces *Cobra* patterns, which provide model templates to assist developers in constructing goal and UML models that capture system requirements and their constraints. In contrast to [AT06], [FHKS18] and [GKC07], we propose scenario patterns to guide developers in constructing dynamic scenarios in order to build test cases. In [BMSJ15], the authors propose a catalog of anti-patterns that analyze a typical constraint interaction that cause correctness or a quality problem in UML class models and suggest possible repairs. Our catalogue does not contain anti-patterns, however, negative test cases can be constructed. In [DFB19], the authors propose a temporal property language based on property patterns for a model-based testing approach using UML/OCL models. However, our patterns can be directly applied for checking model behavior properties.

**Test case generation approaches:** In [CDJ11], the authors propose a scenario expression language in order to describe scenarios as operation sequences to generate functional test cases. [LS06] presents an algorithm which is based on the formal operational semantics for deriving tests from sequence diagram specifications. In [MLMK13], a test specification language is introduced for specifying a test suite that validates the correct behavior of UML activities based on fUML (Foundational UML). In [LBZ$^+$09], the authors describe the use of a genetic algorithm to generate test cases from finite state machines for class behavioral testing. In contrast to [CDJ11], [LS06], [MLMK13] and [LBZ$^+$09], we do not propose or use any explicit algorithms or languages for generating test cases. In [BKLW10], the authors put forward a method to generate test data on a higher-order representation of OCL models. However, we are not transforming OCL constraints into any other representation. The authors in [SHS03] and [VT98] described an approach to generate test cases automatically from state charts. In [PM18], a tool is presented that is called TCGen in order to automatically generate test cases using different testing criteria starting from UML models. In contrast to [SHS03], [VT98] and [PM18], our work provides methodological suggestion and advice for developers to construct the test cases. [AAFR13] proposes an approach for specifying and analyzing temporal properties expressed in TOCL (Temporal logic extension of OCL), and these properties are checked against behavioral scenarios. In contrast, our approach focuses directly on behavioral scenario construction. [TB05] presents a prototype generator for generating traces to test model behavior.

In our recent paper [DG19a], a preliminary catalogue of scenario patterns is sketched, and in the current paper, we significantly extend the catalogue, show the explanations of all patterns and describe their nature and purpose. In contrast to other mentioned works, our approach proposes a catalogue of patterns for developing dynamic scenarios in order to construct test cases independent of testing techniques and shows how behavioral patterns are applied.

## 7 Conclusion and Future Work

In this contribution, we have extended our existing catalogue for scenario patterns. The catalogue can be applied in a UML and OCL model for developing operation call sequences that check dynamic model properties. We have described the nature and purpose of all scenario patterns in order to ease the process of applying the patterns for the considered model. We have explained the complete catalogue by different UML and OCL models. The approach was validated by a study with UML and OCL experts.

As future work, we intend to apply the catalogue to other approaches for checking dynamic model properties and to include more patterns. Furthermore, we plan to give more support on the technical level for particular interesting patterns. For example, the work with the pattern *INIT2FIN* could be automated in a tool by merely specifying an initial and a final OCL condition and upper class bounds, or the pattern *MOUNTAIN* could be specified only by the peak number of objects or links from a fixed class or association. Last but not least, we plan to provide support for determining the patterns that can be applied in the model under consideration.

## References

[AAFR13] Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, and Indrakshi Ray. An approach to analyzing temporal properties in UML class models.

In Frédéric Boulanger, editor, *Proc. 10th Int. Workshop MoDeVVa*, pages 77–86. CEUR-WS.org, 2013.

[AT06] Jörg Ackermann and Klaus Turowski. A library of OCL specification patterns for behavioral specification of software components. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering, 18th Int. Conf., Proc.*, pages 255–269. Springer, 2006.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice, Second Edition.* Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.

[BG14] Fabian Büttner and Martin Gogolla. On OCL-Based Imperative Languages. *Journal on Science of Computer Programming, Elsevier, NL*, 92:162–178, 2014.

[BKLW10] Achim D. Brucker, Matthias P. Krieger, Delphine Longuet, and Burkhart Wolff. A specification-based test case generation method for UML/OCL. In Jürgen Dingel and Arnor Solberg, editors, *Proc. MODELS*, pages 334–348. Springer, 2010.

[BMSJ15] Mira Balaban, Azzam Maraee, Arnon Sturm, and Pavel Jelnov. A pattern-based approach for improving model quality. *Software and Systems Modeling*, 14(4):1527–1555, 2015.

[Bru17] Achim D. Brucker. OCL examples, 2017. `https://git.logicalhacking.com/HOL-OCL/ocl-examples/src/branch/master/health_system`.

[CDJ11] Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. Scenario-based testing from UML/OCL behavioral models - application to POSIX compliance. *STTT*, 13(5):431–448, 2011.

[DFB19] Frédéric Dadeau, Elizabeta Fourneret, and Abir Bouchelaghem. Temporal property patterns for model-based testing from UML/OCL. *Software and Systems Modeling*, 18(2):865–888, 2019.

[DG19a] Nisha Desai and Martin Gogolla. A Catalogue of Scenario Patterns for Validating and Verifying Model Behavior. In Michel Chaudron and Joerg Kienzle, editors, *Proc. IEEE/ACM MODELS 2019 Satelite Events (MODELS 2019).* IEEE, 2019.

[DG19b] Nisha Desai and Martin Gogolla. *Addendum to: Assembling Scenario Patterns for checking Model Behavior.* https://tinyurl.com/y5qdjkt9, 80 pages, 2019.

[DG19c] Nisha Desai and Martin Gogolla. Developing comprehensive postconditions through a model transformation chain. *Journal of Object Technology*, 18(3):5:1–18, July 2019.

[FHKS18] Markus Fockel, Jörg Holtmann, Thorsten Koch, and David Schmelter. Formal, model- and scenario-based requirement patterns. In Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic, editors, *Proc. 6th MODELSWARD*, pages 311–318. SciTePress, 2018.

[GH16] Martin Gogolla and Frank Hilken. Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In Andreas Oberweis and Ralf Reussner, editors, *Proc. Modellierung (MODELLIERUNG'2016)*, pages 203–218. GI, LNI 254, 2016.

[GHD18] Martin Gogolla, Frank Hilken, and Khanh-Hoang Doan. Achieving Model Quality through Model Validation, Verification and Exploration. *Journal on Computer Languages, Systems and Structures, Elsevier, NL*, 54:474–511, 2018. Online 2017-12-02.

[GHH+14] Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B. France. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Proc. Modellierung (MODELLIERUNG'2014)*, pages 273–288. GI, LNI 225, 2014.

[GKC07] Heather Goldsby, Sascha Konrad, and Betty H. C. Cheng. Goal-oriented patterns for UML-based modeling of embedded systems requirements. In *Tenth IEEE Int. Symposium on High Assurance Systems Engineering*, pages 7–14. IEEE Computer Society, 2007.

[LBZ+09] Jinhua Li, Wensheng Bao, Yun Zhao, Zhibing Ma, and Huangzhen Dong. Evolutionary generation of unique input/output sequences for class behavioral testing. *Computers & Mathematics with Apps*, 57(11–12):1800–1807, 2009.

[LS06] Mass Soldal Lund and Ketil Stølen. Deriving tests from UML 2.0 sequence diagrams with neg and assert. In Hong Zhu, editor, *Proc. Int. Workshop Automation of Software Test AST*, pages 22–28. ACM, 2006.

[MLMK13] Stefan Mijatov, Philip Langer, Tanja Mayerhofer, and Gerti Kappel. A framework for testing UML activities based on fUML. In Frédéric Boulanger, editor, *Proc. 10th Int. Workshop MoDeVVa*, pages 1–10. CEUR-WS.org, 2013.

[PM18] Constanza Pérez and Beatriz Marín. Automatic generation of test cases from UML models. *CLEI Electron. J.*, 21(1), 2018.

[RJB99] James E. Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language reference manual.* Addison-Wesley-Longman, 1999.

[SA05] Percy Antonio Pari Salas and Bernhard K. Aichernig. Automatic test case generation for OCL : a mutation approach. In *Technical Report, Int. Institute for Software Technology*, 2005.

[SHS03] Dirk Seifert, Steffen Helke, and Thomas Santen. Test case generation for UML statecharts. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th Int. Conf. PSI*, pages 462–468. Springer, 2003.

[TB05] Anastasia Tircuit and Boumediene Belkhouche. Object-oriented behavioral testing through trace generation. In Mário Guimarães, editor, *Proc. 43nd Annual Southeast Regional Conf.*, pages 306–310. ACM, 2005.

[VT98] Marlon Erthal Righi Vieira and Guilherme Horta Travassos. An approach to perform behavior testing in object-oriented systems. In *TOOLS 1998: 27th Int. Conf.*, pages 318–327. IEEE, 1998.

[WK99] Jos B Warmer and Anneke G Kleppe. *The object constraint language : precise modeling with UML.* Reading, Mass. : Harlow : Addison-Wesley, 1999.

## About the authors

**Nisha Desai** is a former Ph.D. student at the University of Bremen in the Department of Mathematics and Computer Science in Germany. She has worked on improving and optimizing quality assurance techniques for behavioral models. Besides the interest in UML/OCL modeling, she is fond of software design and development. Currently, she is working as a research engineer at Institut für Angewandte Systemtechnik Bremen GmbH, and engaging herself in different European research projects. Contact her at `nisha@informatik.uni-bremen.de`.

**Martin Gogolla** is professor for Computer Science at University of Bremen, Germany and is the head of the Research Group Database Systems. His research interests include software development with object-oriented approaches, formal methods in system design, semantics of languages, and formal specification. Martin Gogolla is actively participating in the MODELS community and is involved in the organisation of the OCL workshops. Martin Gogolla is Associate Editor of the Springer journal on Software and Systems Modeling. In his group, foundational work on the semantics of and the tooling for UML, OCL and general modeling languages has been carried out. The group develops the OCL and UML tool USE (UML-based Specification Environment) since about 15 years. The tool is internationally and nationally widely accepted and employed for research and teaching and in software production. Contact him at `gogolla@informatik.uni-bremen.de`

**Acknowledgments** Reviewers and ECMFA program chairs kept us running. They provided critical, very fruitful contributions that helped a lot to improve the work. Naturally, the remaining deficiencies are due to the authors.