Quality Assurance for Behavioral UML and OCL Models using Filmstripping

Nisha Desai

University of Bremen
Germany
January 2021
Quality Assurance for Behavioral UML and OCL Models using Filmstripping

by Nisha Desai

Cumulative dissertation
in partial fulfillment of the requirements for the degree of
Doctor of Engineering
- Dr.-Ing. -

Submitted to Fachbereich 3
Department for Mathematics and Computer Science
University of Bremen

Referee
Prof. Dr. Martin Gogolla (University of Bremen)
Prof. Dr. Antonio Vallecillo (University of Málaga)

Colloquium: 26. January 2021
Abstract

In Model-Driven Engineering (MDE), models are used as an abstraction of a system. The Unified Modeling Language (UML) along with Object Constraint Language (OCL) is used to describe for a system the structural aspects, e.g., in terms of invariants, and the behavioral aspects, e.g., in terms of pre- and postconditions. As the size and complexity of models grow, there is an increasing need for testing their correctness. Therefore, a developer-friendly and efficient testing technique is essential that concentrates not only on the structural properties but also on the behavioral properties of the model. This thesis deals with enhancing and optimizing the behavioral testing technique available in the tool USE (UML-based Specification Environment). It allows the modeler to validate models and to verify properties by building test scenarios.

In this thesis, we propose a new method for developing comprehensive OCL postconditions of an operation in UML and OCL models, including so-called frame conditions. Frame conditions define the elements which remain unchanged during the transition from one system state to another and are essential for model validation and verification methods. The method is realized by a transformation chain from an initial user-developed model into a semi-automatically derived test case model for checking the model quality.

Configurations and additional OCL invariants are essential to construct a test scenario for model validation. In this thesis, we extend our tool USE by providing an option that automatically generates the configurations needed for the model behavioral testing. Furthermore, an approach is proposed to automatically transform a diagrammatic test case schema into a corresponding OCL invariant, reducing the overhead of a developer of writing it manually. The schema is a visual representation of a behavioral test scenario constructed by the developer. We also introduce a catalogue of different patterns to help the developer in constructing different dynamic scenarios for model testing.
Acknowledgements

First and foremost, I would like to extend my sincerest gratitude to my supervisor Prof. Dr. Martin Gogolla for giving me the opportunity to work in his research group and for his encouragement and enthusiasm throughout my work. His constant support and valuable guidance not only related to work made him a special person in my life. I am also very grateful to Prof. Dr. Antonio Vallecillo for being the co-supervisor and examining my thesis.

My heartily thanks go to the University of Bremen for providing me the scholarship during my Ph.D. study. I would also like to thank my colleagues Khanh-Hoang Doan and Andreas Kästner for their support and inspiring working atmosphere. Also, I wish to thank the members of my dissertation committee, Prof. Dr. Hans-Jörg Kreowski, Prof. Dr. Jan Peleska, Dr. Sabine Kuske and Umesh Sevra for generously offering their time and suggestions in general.

Furthermore, I am very grateful to Dr. Vinay Kulkarni at Tata Research Development and Design Centre (TRDDC), India for giving me a wonderful opportunity to gain industrial research experience. My thanks go to Deepali Kholkar and Dushyanthi Mulpuru for their support and help throughout my internship at TRDDC.

Last but not least, I find no words to express gratefulness to my family and friends who stand as almighty for me and my husband Jwalant Kagathara for all his love, support, and always encouraging me to excel in my career, a path that made me gain precious life experiences.
# Contents

1 Introduction .......................................................... 1
   1.1 Thesis Goal and Contributions ........................................ 3
   1.2 Thesis Structure ...................................................... 4

2 Background .............................................................. 5
   2.1 Unified Modeling Language (UML) ..................................... 5
       2.1.1 Class Diagram ................................................... 6
       2.1.2 Object Diagram .................................................. 7
       2.1.3 Sequence Diagram .............................................. 8
   2.2 Object Constraint Language (OCL) .................................. 9
   2.3 UML-based Specification Environment Tool (USE) .................. 10
       2.3.1 Software Development with UML and OCL ..................... 11
       2.3.2 Coverage of a UML/OCL Model by OCL Expresssions ........ 11
       2.3.3 USE Model Validator ............................................ 13
       2.3.4 Filmstripping ................................................... 13
           2.3.4.1 Filmstrip model transformation ......................... 14
   2.4 Semantics of Business Vocabulary and Business Rules (SVBR) Model 16
       2.4.1 SVBR Business Vocabulary ..................................... 17
       2.4.2 SVBR Business Rules ............................................ 17
   2.5 Answer Set Programming ............................................. 18

3 Developing Comprehensive Postconditions .......................... 19
   3.1 Post-/frame Condition Determining Language (PCDL) .............. 19
       3.1.1 Distinction between Deleted, Sustained and Added Objects ... 20
       3.1.2 Representing Postconditions in Tabular Form ................ 21
   3.2 Transformation of PCDL into Postconditions ........................ 23
## 6 Constructing Dynamic Scenarios

6.1 Dynamic Scenario Pattern Catalogue

6.2 Scenario Pattern Manifestations

6.2.1 Each Operation Once (ONCE)

6.2.2 Each Operation Pair (PAIR)

6.2.3 All Operations in Single Operation Call Sequence (ALLOPS)

6.2.4 Short/middle-sized/long Operation Call Sequences (SML)

6.2.5 Repeatable Operation (REPEAT)

6.2.6 Operation Cycles (CYCLE)

6.2.7 Adding and Successively Removing Objects or Links (MOUNTAIN)

6.2.8 Dead-end Call Sequences (DEADEND)

6.2.9 Absence of an Operation in Operation Call Sequence (ABSENCE)

6.2.10 Operation Call Sequences Leading from an Initial to a Final Condition (INIT2FIN)

6.2.11 Regular Expressions over Operation Calls (REGULAR)

6.3 Combination of Patterns

6.4 Evaluation

6.5 Related Work

## 7 Exploring Usability of ASP for Checking SBVR

7.1 Formalizing SBVR Structured English (SE) Rules

7.2 Generating Scenarios using ASP

7.2.1 Checking Model properties

7.3 Negative Scenario Generation

7.4 Quality Assurance

## 8 Conclusion and Future Work

8.1 Conclusion

8.2 Future Work

## Bibliography
## List of Figures

1.1 Overview on the Filmstrip Model Validation Process with the New Proposed Approaches .......................................................... 3

2.1 Class Diagram of the SocialNetwork Model .......................... 6
2.2 Object Diagram of the SocialNetwork Model ......................... 7
2.3 Sequence Diagram of the SocialNetwork Model ..................... 8
2.4 Transformation of the Application Model to the Filmstrip Model . . . 15

3.1 Deleted, Sustained and Added Objects ........................................ 21
3.2 Abstract Example: Class Diagram (Left) and Tabular PCDL Model (Right) ................................................................. 22
3.3 Model Transformation Chain for Developing Comprehensive Postconditions .............................................................. 24
3.4 Application Model and Filmstrip Model .................................. 25
3.5 Example Sequence Diagram ..................................................... 29
3.6 Generated Filmstrip Object Diagram using Developer-defined Elements (A) .............................................................. 30
3.7 Generated Filmstrip Object Diagram using Developer-defined Elements (B) .............................................................. 32
3.8 Class Diagram of the ContentSharingPlatform Model .................. 33
3.9 Generated Filmstrip Object Diagram .......................................... 35

4.1 Overview on the Filmstrip Model Validation Process with the New Approaches ............................................................... 38
4.2 Class Diagram of the CivilStatus Filmstrip Model ..................... 39
4.3 Abstract Filmstrip Template Example ....................................... 41
4.4 Generated Filmstrip Object Diagram - CivilStatus Model ........... 42
4.5 Fragment of the Kodkod Plugin Framework ............................... 43
4.6 Model Configuration Option Window ....................................... 45
4.7 Filmstrip Model Configuration Window - Basic Types ............... 45
4.8 Filmstrip Model Configuration Window - Model Elements .......... 46
4.9 Message Window after a Successful Filmstrip Template Construction 49
4.10 Class Diagram of the TollCollect Filmstrip Model ................. 50
4.11 Filmstrip Configuration Window - TollCollect Model ............... 50
4.12 Constructed Filmstrip Template - TollCollect Model ............... 51
4.13 Generated Filmstrip Object Diagram (Test Case) - TollCollect Model 51
4.14 Class Diagram of the SocialNetwork Filmstrip Model ............... 52
4.15 Filmstrip Configuration Window - SocialNetwork Model ............. 53
4.16 Constructed Filmstrip Template - SocialNetwork Model ............. 54
4.17 Generated Filmstrip Object Diagram (Test Case) - SocialNetwork Model 55

5.1 Overview on Filmstrip Validation (Gray Box) with TC Schema
   Approach (Dotted Box) ............................................. 57
5.2 Class Diagram of the ContentSharingPlatform Filmstrip Model ...... 57
5.3 Scenario: a ContentItem Unshared by a User ........................ 58
5.4 Class Diagram of the SocialNetwork Filmstrip Model ............... 59
5.5 TC Schema of Test Scenario 1 ..................................... 60
5.6 TC Schema of Test Scenario 2 ..................................... 61
5.7 An Abstract Example: Open vs. Closed Snapshots ................... 62
5.8 Automatically Generated Test Case 1 of the Scenario 1 ............. 65
5.9 Automatically Generated Test Case 2 of the Scenario 1 ............. 66
5.10 Automatically Generated Test Case 3 of the Scenario 1 ............. 66
5.11 Automatically Generated Test Case 1 of the Scenario 2 ............. 67
5.12 Automatically Generated Test Case 2 of the Scenario 2 ............. 68

6.1 Catalogue of Scenario Patterns .................................... 72
6.2 Class Diagram of the Scheduler Application Model .................. 74
6.3 Filmstrip Object Diagram - Operation Init .......................... 74
6.4 Filmstrip Object Diagram - Operation New .......................... 74
6.5 Filmstrip Object Diagram - Operation Ready ........................ 75
6.6 Filmstrip Object Diagram - Operation Swap ........................ 75
6.7 Class Diagram of the CivilStatus Application Model ................ 76
6.8 Filmstrip Object Diagram - birth-birth Operation Pair .............. 76
6.9 Filmstrip Object Diagram - birth-marry Operation Pair ............. 76
6.10 Filmstrip Object Diagram - divorce-marry Operation Pair .......... 77
6.11 Filmstrip Object Diagram - marry-divorce Operation Pair .......... 77
6.12 Class Diagram of the TollCollect Application Model .............. 77
6.13 Filmstrip Object Diagram - All Operation Calls ........................................... 78
6.14 Class Diagram of the HealthSystem Application Model ................................ 80
6.15 Filmstrip Object Diagram - Short Operation Call Sequence .......................... 80
6.16 Filmstrip Object Diagram - Middle-sized Operation Call Sequence .............. 81
6.17 Filmstrip Object Diagram - Long Operation Call Sequence ........................... 82
6.18 Class Diagram of the PersonCompany Application Model ............................ 83
6.19 Filmstrip Object Diagram - Operation fire .................................................... 84
6.20 Filmstrip Object Diagram - Operation hire ..................................................... 84
6.21 Class Diagram of the Library Application Model .......................................... 85
6.22 Filmstrip Object Diagram - Cyclic borrow Operation Calls ............................. 86
6.23 Filmstrip Object Diagram - Cyclic return Operation Calls .............................. 86
6.24 Filmstrip Object Diagram - Number of Borrow Links: 0,1,2,3,2,1,0
(Mountain) .......................................................................................................... 88
6.25 Class Diagram of the SocialNetwork Application Model .............................. 88
6.26 Filmstrip Object Diagram - invite-accept Operation Call ................................. 89
6.27 Filmstrip Object Diagram - invite-decline Operation Call ................................. 89
6.28 Class Diagram of the StudentReport Application Model ................................ 90
6.29 Filmstrip Object Diagram - Initial and Final State Conditions .......................... 91
6.30 Class Diagram of the Account Application Model .......................................... 92
6.31 Filmstrip Object Diagram - Regular Expression .............................................. 93
6.32 Class Diagram of the ContentSharingPlatform Application Model ... 93
6.33 Filmstrip Object Diagram - generate-remove Operation Pair Repitition ............ 94
6.34 Filmstrip Object Diagram - remove-generate Operation Pair Repitition .......... 95
6.35 Filmstrip Object Diagram - share-unshare Operation Pair Repitition ............... 95
6.36 Filmstrip Object Diagram - unshare-share Operation Pair Repitition ............... 96
6.37 Evaluation: Patterns together with Evaluated Models, their
Complexity and Constructed Scenarios ............................................................. 97
List of Tables

3.1 Generic Transformation of PCDL Elements into OCL Postconditions 23
3.2 PCDL Model in Tabular Form with Initial Elements ............ 26
3.3 PCDL Model with Gray-shaded Developer-defined Elements (A) ... 27
3.4 Example Configuration ........................................... 29
3.5 PCDL Model with Gray-shaded Developer-defined Elements (B) ... 31
3.6 ContentSharingPlatform PCDL Model with Gray-shaded
   Developer-defined Elements .................................. 34

6.1 Executable and Non-Executable Operation Pairs .................. 76
6.2 Executable and Non-Executable Operation Pairs and Repetition of Pairs 94

7.1 Negation of Negative Predicates of Relations and Attributes ..... 108
7.2 Negation of Predicates linked with AND and OR ................. 108
7.3 Test case 1: Generated Scenarios with Positive and Negative ASP
   Programs .......................................................... 111
7.4 Test case 2: Generated Scenarios using Different Boolean Attribute
   Values with Positive and Negative ASP Programs ............... 112
Chapter 1

Introduction

Recent years have seen a lot of research and activities in the field of software development that provides essential services to society and eases the life of human beings. To do so, the software systems have to be reliable and should be eligible to adapt to real-time. Despite of the new generation and advanced software languages, it is still difficult and painstaking tasks to develop a complex software system.

Model-Driven Engineering (MDE) [56] has emerged as a promising field in the area of software development for developing complex software systems. The software development process has become more model-centric and less code-centric. In MDE, models are considered as an important component in the development phase rather than just the documentation of the design process. They are used as an abstraction of a system to deal with the growing complexity of large software systems and have become a central part of software development.

Modeling languages such as the UML (Unified Modeling Language) [58] are used to describe the aspects of software systems, starting from requirements analysis to the implementation and maintenance phase. However, a UML model alone cannot express enough so as to cover all the relevant aspects of a specification of the system. Therefore, the UML together with formal specification languages such as the OCL (Object Constraint Language) [60] are used to describe structural and behavioral aspects of the system [81]. Structural properties can be described in terms of OCL invariants and behavioral properties in terms of operation pre- and postconditions in a UML and OCL model.

For any given application model, it is of utmost importance to verify and validate the system properties during the design phase so as to avoid any unwanted errors for the next step. As Barry Boehm says [11], “Errors are most frequent during the requirements and design activities and are the more expensive the later they are
removed”. So it is of significant importance in the software development to use quality assurance techniques such as validation and verification on the modeling language.

There are variety of verification and validation techniques available which mainly concentrate on static characteristics, i.e., class diagrams and invariants. These techniques partly transform UML models including OCL invariants into validation and verification platforms (like SAT or SMT solvers or relational logic) allowing the developer to check relevant structural properties of the UML model. However, UML and OCL models can involve dynamic characteristics in the form of OCL pre- and postconditions for operations. Changes in object states are triggered by operation calls. The model validator in the tool USE [79] is designed for structural analysis of UML class diagrams. In order to validate dynamic aspects of the model, our filmstrip approach is used [42].

Filmstripping transforms a given UML and OCL model which is comprised of invariants and pre- and postconditions into an equivalent model which possesses only invariants. The transformed model is called a filmstrip model. Also, as the dynamic behaviour properties are transformed into structural aspects in the filmstrip model therefore, it can then be validated with the technique designed for structural analysis. Thus, it can be validated with the available USE model validator. The system dynamics are expressed in the filmstrip model by different explicit objects representing both the application model and the operations calls. A sequence of operation calls and object diagrams of an application model corresponds to a single object diagram of the filmstrip model. Thus, a complete scenario is available in a single object diagram making it easily accessible to complete information of the system [36].

The gray highlighted part in Fig. 1.1 gives an overview on our existing filmstrip model validation approach. The process starts with the application model which transforms into the filmstrip model using the USE filmstrip plugin. The filmstrip model along with the specified configurations and additional OCL invariants are given to the USE model validator. The filmstrip configuration determines how classes and associations are populated and additional OCL invariants which guide the test case generation in a particular direction, and are essential to construct the scenario for model validation. As an outcome, the model validator automatically generates valid test cases in the form of the filmstrip object diagrams according to the given scenario, and by analyzing the state transitions in the diagram, properties for model dynamics can be validated.
1.1 Thesis Goal and Contributions

Modern software systems are getting bigger and complex, which makes the abstract model and necessary test cases also larger, and therefore the complete validation process becomes crucial. The goal of this thesis is to optimize and enhance the validation process by simplifying and making it more developer-friendly. To achieve this goal, different approaches have been proposed and are shown in Fig. 1.1 (Dotted rectangles). The combination of the proposed different approaches strengthen the validation process, makes it simpler and easier and also reduces the overall model validation time. These approaches are briefly explained as follows.

![Figure 1.1: Overview on the Filmstrip Model Validation Process with the New Proposed Approaches](image)

**Developing Comprehensive Postconditions:** For validation and verification methods, OCL pre- and postconditions along with so-called frame conditions are used to describe the behavior of any UML/OCL model, however they are formulated manually. So, to reduce the overhead, a so-called post-/frame condition determining language (PCDL) is proposed, which systematically generates combined post- and frame conditions (comprehensive postconditions).

**Generating Filmstrip Configurations and Templates:** An approach has been developed and implemented in the tool USE for automatic generation of filmstrip configurations (which configures the model validator search space) and so-called filmstrip templates (identifies recurring model parts). Developers specify the configuration for application model elements and accordingly, a filmstrip model configuration and a filmstrip template are automatically generated.

**Generating OCL Invariants:** The additional OCL invariants are specified to guide the object diagram generation into a particular direction, however, they are written
manually. To reduce the burden of a developer, a so-called test case (TC) schema approach has been introduced, in which developers can express the scenario in the form of a partial object diagram. This diagrammatic representation is then transformed automatically into OCL invariants.

**Constructing Dynamic Scenarios:** A developer can create dynamic scenarios for validation and verification of different models. However, selection of such scenarios can be a difficult task. So, a catalogue of different scenario patterns has been proposed to give advice and develop guidelines for constructing dynamic scenarios.

### 1.2 Thesis Structure

The thesis is structured as follows. The basics of UML/OCL models and the filmstrip transformation process are explained in Chapter 2. Chapter 3 describes the proposed PCDL approach and shows the generation of combined post- and frame conditions. Chapter 4 shows the concept and implementation of the approach for automatically generating filmstrip configurations and filmstrip templates in the tool USE. In Chapter 5, the generation of OCL invariants using the TC schema approach is presented. Chapter 6 presents a catalogue of different scenario patterns that help to construct dynamic scenarios. Chapter 7 presents my industrial internship work in the direction of validation and verification of Semantics of Business Vocabulary and Business Rules (SBVR) models by exploring Answer Set Programming (ASP). Chapter 8 concludes this thesis and cites some possible future work.
Chapter 2

Background

This chapter presents fundamentals about concepts and artifacts essential to this thesis. The modeling languages such as, Unified Modeling Language (UML), together with Object Constraint Language (OCL), is used to describe structural and behavioral aspects of the system and is briefly described in this chapter. A description of the modeling and specification tool called UML-based Specification Environment (USE) is presented. The basic about the filmstripping is illustrated, and the filmstrip model transformation is explained in detail. Apart from these, the Answer Set Programming (ASP) and Semantics of Business Vocabulary and Rules (SBVR) are described in this chapter, which were used during my industrial internship.

2.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting software-intensive systems. This unified language is maintained by the Object Management Group (OMG) [58].

The UML has become one of the most widely used modeling language that can be used with all significant object and component methods for describing real-world application domains. Software systems, in today’s world, are growing in size, complexity, distribution and importance. As a result, building and maintenance of software have become a more complex and challenging task. Therefore, the deployment of languages such as UML reduces the complexity and difficulty by providing a high level of abstraction, which describes precise and essential information for designing and developing the software system [50].

The graphical representation of the UML includes a set of diagrams, each focusing on different aspects of a design. The UML Notation Guide states all the notation of diagram elements [58]. These diagrams can be classified into two groups (a) a structural
diagram that represents the static aspect of the system, and (b) a behavioral diagram that describes the dynamic aspect of the system. Altogether, fourteen different model types can be found in the Unified Modeling Language Reference Manual [71]. In this thesis, three diagrams, i.e., class diagram, object diagram and sequence diagram, have been extensively used and are explained further in this section.

2.1.1 Class Diagram

Class diagrams are the most common diagram found in object-oriented modeling systems. It illustrates the static design view of the system model and shows classes of the system, their attributes, operations and associations [64]. The classes reflect a set of objects. The attributes describe values that the objects may contain, and an operation specifies the result of the behavior of objects. The associations describe connections among the different class objects, and they can refer to each other through role names. The number of objects of one class linked to other class objects depends on the multiplicity attached to an association end.

![Class Diagram of the SocialNetwork Model](image)

Figure 2.1: Class Diagram of the SocialNetwork Model

Figure 2.1 shows an example class diagram of a social network system. The system can have many profiles, they can invite friendship requests to other profiles, and the invitee profiles can accept or reject the invited friendship requests. The class diagram consists of the classes Friendship and Profile with the associations Inviter and Invitee. The class Profile has attribute userN (data type String) for unique identification, and the class Friendship has attribute status (data type Enumeration) for showing the friendship relation between the profiles. The attributes can have different data types such as String, Integer, Real and Enumeration. The Enumeration is a user defined data type. For example, Status is the Enumeration data type, and the values can be pending, accepted or declined. The class Profile has the operation invite, which sends the friendship request with the attribute status pending. Further, the attribute status can be changed to accepted with the execution of the
operation accept, can be changed to declined with the execution of the operation decline.

The class diagram can also have other elements such as aggregation, composition and generalization to show the relationship between classes. Aggregation and composition are a whole-part relationship. In aggregation, the relationship between a child class and parent class is independent, whereas, in composition, they are dependent. A generalization is a relationship between classes in which one class is identified as the general class and the others as the specialization of it. A specialized class inherits all the properties and characteristics from the general class.

2.1.2 Object Diagram

Object diagrams are also a type of structural diagram. It represents the entities of the real world, or of the modeled system, as instances of the classes described in the class diagram, and their relationships as links, which are instances of the corresponding associations. Objects are defined by concrete attribute values and a link connects the objects participating in the association. In general, an object diagram provides a snapshot of a system at a particular point in time showing objects, their attribute values, and links connecting the objects [77].

Figure 2.2: Object Diagram of the SocialNetwork Model

Figure 2.2 shows the example object diagram, where profile1 and profile2 objects are instances of the class Profile, and friendship1 object is the instance of the class Friendship from the class diagram shown in Fig. 2.1. Between profile1 and friendship1, there exists the link Inviter, and between profile2 and friendship2 exists the link Invitee. Also, the attribute status of friendship1 object is pending, which implies that profile1 has sent the friendship request to profile2.

The object diagram represents only the single state of a class diagram. So, the previous state information can be lost due to the change in the system state through the operation call. Therefore, a single object diagram cannot represent any flow of information of a state.


2.1.3 Sequence Diagram

A sequence diagram is a behavioral diagram that represents the message interaction between system components [71]. Furthermore, it shows the objects participating in the interaction and the order in time in which messages are exchanged. The creation (construction) and the removal (destruction) of an object can also be sequentially portrayed in a diagram.

![Sequence diagram profile1:Profile profile2:Profile](image)

Figure 2.3: Sequence Diagram of the SocialNetwork Model

Figure 2.3 shows the example sequence diagram of the running example. In the diagram, two message interactions are shown (a) profile1 invites or sends the friendship requests to profile2 (b) profile2 accepts the friendship request of profile1. Each instance in a sequence diagram is represented by an object rectangle which shows the name and the class of the instance (the Profile class objects profile1 and profile2 in Fig. 2.3). The lifeline (vertical dotted line) which is connected to the rectangles represents individual participant in the interaction. The vertical rectangles on the lifeline highlight the period during which the operation execution of the object is performed.

The sequence diagram does not provide any information regarding the system state, so no information regarding the change in the state or the final state can be extracted. Therefore, combining the static information covered by an object diagram and dynamic information covered by a sequence diagram could describe such information.
2.2 Object Constraint Language (OCL)

As explained in Sect. 2.1, UML is a graphical language for visualization of the system state. But visual modeling with the UML alone is not enough for the development of accurate and consistent software model. For this reason, Object Management Group (OMG), in 1997, developed Object Constraint Language (OCL), which describes expressions on UML models and thus extends further the functionality of UML [60]. For example, in the case of the SocialNetwork model, a profile cannot be inviter and invitee at the same time, i.e., the friendship request between two Profile objects should be asymmetric. Such a condition is not possible to be implemented within the UML alone as it shows only the relation between the classes. So, for implementing such constraints, OCL has to be used in addition to the UML.

OCL provides two kinds of descriptions, (a) OCL expression, which is defined in the context of the constrained elements and evaluates some values, and (b) OCL constraint, which is restrictive statements concerning some underlying model that should evaluate to true. Basically, an OCL constraint is referred to as an OCL expression of the Boolean type.

The OCL provides variables and operations which can be combined in various ways to build expressions. There are many OCL expression types. The simplest types are the basic type which consists of the types Boolean, Real, Integer and String. The concrete collection types are Bag, Sequence, Set and OrderedSet. The other types include tuples, which can contain one to many values of other types and special types such as OclAny, OclType, OclExpression, and OclState.

There are three different types of OCL constraints, namely invariants of classes, preconditions and postconditions of operations, which are used to describe model properties. An invariant is a restriction in the form of expression which is applied to all instances of the class diagram and that must be true for all instances, i.e., an invariant must be satisfied after the creation of an instance of the class for which the invariant is defined. The following OCL expression shows one invariant of the SocialNetwork model, which ensures that the profiles have the asymmetric friendship relationships.

\[
\text{Context Profile inv asymmetricFriendship:}
\]
\[
\text{self.friendshipR.invitee} ->
\]
\[
\text{intersection(self.friendshipE.inviter} -> \text{isEmpty}())
\]

The pre- and postconditions are prerequisites in the form of Boolean expressions that must be fulfilled before and after the execution of an operation, respectively. In
the case of postconditions, there is a possibility to access the system state before an
operation was called using the keyword @pre. The following OCL expressions show
one pre- and postcondition of the operation accept of the SocialNetwork model. The
precondition ensures that there exists the pending friendship request between the
profiles, and the postcondition makes sure that the pending request is accepted.

Profile :: accept(anInviter : Profile)
  pre pendingFS: self.friendshipE->
    select(e | e.status = #pending).inviter->includes(anInviter)
  post acceptedFS: self.friendshipE->
    select(e | e.status = #accepted).inviter->includes(anInviter)

2.3 UML-based Specification Environment Tool (USE)

The UML-based Specification Environment (USE) is a system tool that is used for
the specification, modeling and validation of software models based on UML and
OCL. The development of this tool started in 1998 in the working group Database
Systems of the University of Bremen [35]. Since then, USE is continuously extended
and improved. The tool is programmed mostly in Java and since 2009 a plugin API
allows for an easy extension of the functionality [6].

The UML/OCL models are specified and presented in textual form (.use file), and
it contains classes and their attributes and operations as well as associations. The
pre- and postconditions of the operations and invariants of classes are defined using
OCL expressions within this file. The tool USE accesses this file, and the model is
evaluated graphically, which allows examination of class diagrams, object diagrams,
sequence diagrams amongst others [35].

The USE also provides the option for the user to write all the data currently
available in the system state in a .soil (simple OCL-like imperative programming
language) script [16]. This script contains all the commands necessary to create the
current state. This allows object diagrams to be saved in order to be able to process
them at a later time point. The language reuses OCL to describe the implementation
details. The manual, with descriptions of all functions implemented in the USE up
to 2007, can be found under [79].
2.3.1 Software Development with UML and OCL

Modeling in the field of software engineering is a process through which models of software systems are created. In traditional software development, models are used only as documentation or architectural descriptions of the system to be developed. In contrast to traditional software development, in model-driven engineering (MDE) [56], models are used as an abstraction of a system to deal with the growing complexity of large software systems and have become a central part for software development.

In the software design phase, models can be developed using a modeling language that permits the creation of components of a system. In this context, the software systems are specified by models at an abstract level and comprise only relevant information of the system. Thus, it reduces the complexity and helps to accelerate the design process [36] [37] and is also independent of programming languages and platforms. These software models can be validated and verified with the MDE technologies, which can increase the quality of the system to be developed in the design phase. The MDE, therefore, promises a potential increase in productivity and quality in software development.

The models are built using modeling languages such as the UML together with formal specification languages such as the OCL to describe structural and behavioral aspects of the system [81]. An ordinary UML/OCL application model is comprised of a class diagram with any number of classes, attributes, associations, and operations. The structural properties of the model can be described in terms of OCL invariants and behavioral properties in terms of operation pre- and postconditions. The invariants impose the possible system state to create valid object diagrams. The operation pre- and postconditions determine the valid system dynamics in the form of state transitions with intermediate operation calls. The complete application is described in one single model, and a transition is induced by a single operation call [36].

2.3.2 Coverage of a UML/OCL Model by OCL Expressions

The complexity of the software model can be understood and explained using a so-called “coverage” of OCL expressions. The coverage indicates the complexity of OCL invariants and pre- and postconditions in the sense that it indicates the number of class model elements (class, attribute, association end) that are used in the particular OCL formulae.

In the tool USE, the model coverage by invariants, preconditions and postconditions are automatically generated using the commands coverage -sum -invariants.
coverage -sum -pre and coverage -sum -post, respectively. To explain the model coverage by OCL expressions, the SocialNetwork model is continued as an example. The automatically generated coverage of the model by the invariants, preconditions and postconditions employing the tool USE is shown below.

USE version 5.1.0, Copyright (C) 1999-2019 University of Bremen

use> coverage -sum -invariants
Classes covered by invariants: 2/2
  Profile: 6
  Friendship: 5

use> coverage -sum -pre
Classes covered by pre-conditions: 2/2
  Profile: 8
  Friendship: 6

use> coverage -sum -post
Classes covered by post-conditions: 2/2
  Profile: 48
  Friendship: 47

The SocialNetwork model has total 11 (Profile: 6, Friendship: 5) coverage by the invariants. The invariants of the model and the generation of the coverage by them are shown and explained below.

context Profile inv asymmetricFriendship:
  self.friendshipR.invitee ->
    intersection(self.friendshipE.inviter) -> isEmpty()

context Profile inv uniqueUserName:
  Profile.allInstances -> isUnique(p | p.userN)

context Friendship inv fsBetweenTwoProfiles:
  self.inviter -> notEmpty() and self.invitee -> notEmpty()

  In the first invariant, once the class Profile and twice the class Profile association ends (friendshipR, friendshipE), and in the second invariant, twice the class Profile (once in general and once in the isUnique expression) and once the class
Profile attribute name are used. So, the coverage of the class Profile by the invariants is 6. Also, in the first invariant, twice the class Friendship association ends (invitee, inviter) and in the third invariant once the class Friendship and twice the class Friendship association ends (invitee, inviter) are used. So, the coverage of the class Friendship by the invariants is 5. Analogously, it can be understood for the pre- and postconditions as well.

2.3.3 USE Model Validator

In USE, a plugin called model validator has been developed and using it, a developer can automatically generate different object diagrams for a class diagram in a pre-configured search space [66]. The search space of the model validator is configured based on the given configuration and sometimes also on the specified additional invariants. The model validator uses the external tool Kodkod [78] and SAT-based methods for searching the solution. Using Kodkod, the model is expressed in relational logic [44]. Then it represents the model expressed in relational logic as a SAT problem. Finally, a SAT solver checks its validity. The first solution found is immediately instantiated as an object diagram and made visible by the USE, however, the developer can also explore the other available solutions [49].

The configurations are written in text files with the extension .properties. In the configuration file, a lower and upper limit can be specified for each configurable base type (Integer, Real, String), each class, each attribute of a class, and associations, which specifies how many instances should be there in the object diagram. In addition, it is possible to specify a set of possible values of the field as a set. These specifications can speed up the validation process, as the model validator only needs to search for object diagrams that meet the model constraints. In addition to the configuration, the developer can also provide additional OCL invariants to guide the model validator to generate specific scenarios in the form of object diagrams [37].

The validate command in the tool USE initializes the model validator to look for a solution that meets the requirements of the given search space. If there is a solution, the model validator gives the output showing SATISFIABLE. If there is no valid state that satisfies all the conditions, UNSATISFIABLE is reported by the validator.

2.3.4 Filmstripping

In MDE, it is necessary to check that an ordinary UML/OCL model meets the formal and informal described requirements [42]. The UML and OCL application models
can involve structural aspects in the form of OCL invariants and dynamic aspects in the form of operation pre- and postconditions. The changes in object states are triggered by operation calls. A variety of validation and verification techniques are available [21] [44] [18] [78] [15] [5] [67] which usually concentrate on checking the structural aspects of the model. The model validator in the tool USE is also specifically designed for structural analysis of models.

In order to validate dynamic aspects of the model, a so-called filmstrip approach has been developed and integrated in the tool USE [36]. Filmstripping transforms a given UML and OCL model, which is comprised of invariants and pre- and postconditions into an equivalent model that possesses only invariants. The transformed model is called a filmstrip model. The filmstrip model is comprised of a regular application model but with additional classes to supplement the system state information. This means the original model is present and unchanged with its full functionality in a filmstrip model. The additional classes create new objects called snapshots, which represent a single system state of an application model. This makes it possible to read the progression information in a single object diagram of a filmstrip model, i.e., a sequence of operation calls and object diagrams of an application model corresponds to a single object diagram of the filmstrip model. The pre- and postconditions of the application model are transformed into invariants and removed from the operations. So the complete dynamic behavior is integrated in a single structure.

The transformed model involves only structural aspects and can be validated with the available USE model validator. So, this transformation approach validates and verifies both the static and dynamic properties of any given application model.

### 2.3.4.1 Filmstrip model transformation

The model transformation with filmstripping can be explained best in terms of an example and the SocialNetwork model in which a user can invite, accept and decline a friendship request is continued. The upper part of Fig. 2.4 shows the class diagram of the filmstrip model. The original application model, consisting of the classes Profile and Friendship with the associations Inviter and Invitee, is contained in the filmstrip model and indicated in a gray-shaded style. The example sequence diagram also represents elements of the application model.

The application model is automatically transformed using a plug-in in the tool USE into the filmstrip model: the non-gray shaded classes and invariants are added. In essence, the application model sequence diagram becomes a filmstrip model object diagram. Snapshot objects explicitly allow to capture single system states from the
application model. Operation call objects (suffix \text{OpC}) describe operation calls from the application model. Basically, each operation is transformed into an \text{OperationCall} class with attributes for the self objects and for the operation parameters. Thus, for example, the call \text{profile3.accept(profile1)} (dotted box) from the sequence diagram is represented by the object \text{accept\_profileopc1} in the filmstrip object diagram. The effect of the operation call is represented by the differences between the left and the right snapshot: the \text{accept} operation call changes the attribute \text{status}. The four \text{Profile} and the two \text{Friendship} objects represent different object states before and after the operation call. So, one could say that the object \text{profile2} is a later incarnation of the object \text{profile1}.

As shortly explained, during the transformation process, all the application model elements are transformed into the filmstrip model elements, and also new filmstrip elements are introduced. The description of this transformation process is shown below [41]:

\textit{Transformation of classes:} Every class and attribute from an application model is transformed into a class and attribute in a filmstrip model. Furthermore, two new classes are added, namely, \text{Snapshot} and \text{OperationCall} in the filmstrip model. The \text{Snapshot} class assigns each object to a snapshot. The \text{OperationCall} class represents the operation calls that characterize the changes happening in the snapshot.
class utilizing the pre- and postconditions. The parameters of the operation become attributes with respective types in the filmstrip model. The filmstrip operation call classes uses a generalization relationship to inherit with class OperationCall at the top.

Transformation of associations: All associations of the application model become directly part of the filmstrip model. Apart from that, a ternary association is added, which links the Snapshot classes and the OperationCall class. This association navigates from a pre-snapshot to a post-snapshot through an intermediate operation call. Also, other associations and aggregations are added. The associations link Snapshot and application classes, which represent that each application object from the filmstrip model will be linked to exactly one Snapshot object, and the rebirth of application objects in newer snapshots will be expressed by aggregation links.

Transformation of operation definitions and invariants: The operation definition and invariants of the application model become part of the filmstrip model without changing itself.

Transformation of pre- and postconditions: The pre- and postconditions of the application model are transformed into invariants of the filmstrip model and assigned to the concrete operation call class. The invariants of the filmstrip model are called once for every operation call invocation. So, the behavior of these invariants is the same as pre- and postconditions of the application model.

Different architectures are available in the tool USE to connect the class Snapshot and OperationCall [29]. In this thesis, apart from the ternary association, the binary association in which the first association connects the Snapshot sequence and the second connects the OperationCall to Snapshot is also used.

2.4 Semantics of Business Vocabulary and Business Rules (SVBR) Model

The Semantics of Business Vocabulary and Business Rules (SBVR) is a standard specification developed by Object Management Group (OMG) [59], which provides a meta-model to create semantic models covering static nature of business. The SBVR is represented based on (a) business vocabulary, which is a set of concepts used while communicating and describing business, (b) business rules, which constrain or guide business actions (behavior).

The use of natural language representation of business knowledge is usually emphasized by SBVR to help business people in describing knowledge using their own
language. In SBVR 1.5 document [59], the Structured English is proposed, in Annex C, as a possible notation to formulate SBVR rules. Structured English provides a standardized representation for formalizing the syntax of natural language representation [46].

The basic idea of business rules supported by SBVR is “Rules are build on facts, and facts are based on concepts” [59] [68]. It divides knowledge of business in concepts, fact types and rules. These notions can be related to each other as follows: an elementary unit of knowledge in SBVR is a concept (for example, bank, customer, account, document), facts make assertions about these concepts (for example, bank opens account), rules constrain and support these facts (for example, it is necessary that customer submits documents to open an account). The two major elements, i.e., business vocabulary and business rules of SBVR, are briefly described below.

2.4.1 SBVR Business Vocabulary

SBVR business vocabulary is comprised of the specialized terms and concepts used in the definition of a business domain in a particular organization [59]. There are several types of SBVR concepts such as object type, individual concept and fact type.

An object type is a noun concept that categorizes things on the basis of their common properties. For example, ‘customer’ is an object type. An individual concept corresponds to only one object. In English, proper nouns or quantified nouns are classified as individual concepts, for example, ‘Banned Customer’, ‘Commercial Bank’ etc. A fact type is based on a verb phrase that involves one or more noun concepts and whose instances are all real. For example, ‘customer should not be banned customer’ is a fact type. A fact type can have one (characteristic), two (binary) or more fact type roles.

2.4.2 SBVR Business Rules

SBVR business rules are used to represent the action of a business in a logical form. A SBVR rule can be formally defined as “an element of guidance that introduces an obligation or necessity” [59]. The SBVR vocabulary are generally used to form a SBVR rule, and the relation between them are represented by the rule. Basically, a SBVR rule represents structural or behavioral information of the business. Thus the main types of rules defined in the SBVR standard are structural business rules and behavioral business rules.
A structural (also called definitional) rule is used to define a business setup, and it represents the structure of the business and cannot be violated. It is typically a claim of necessity. A behavioral (also called as operative) rule describes the business processes and can be either ignored or violated. It typically expresses the behavior of an entity [59].

2.5 Answer Set Programming

Answer-Set Programming (ASP) is a declarative problem-solving paradigm that uses logic programming [34]. The ASP is a collection of statements that describe the objects of a domain and model relations between them. The basic elements of an ASP program are the rules, facts and constraints. The ASP program is passed onto an answer-set solver, which generates answer-sets to the given program, that are used to obtain solutions to the problem. In general, there are two steps to solve the ASP program (a) grounding in which the input ASP program is transformed into a propositional representation, and (b) solving in which the solution (answer sets) for the transformed variable-free ASP program is computed [33].

An ASP program is comprised of a collection of the rules. Generally, a rule has two parts, i.e., Head and Body, separated by the consequence operator “:-”. An example of a rule is x:- y, z. A head (i.e., x) is a predicate on the left side of the rule, and a body (i.e., y, z) is a set of predicates on the right side of the rule. In such a rule, the head succeeds only if every predicate in the body succeeds. The head or the body in a rule can be empty. A rule with an empty head is called a constraint, whereas a rule with an empty body is called a fact. The constraints are applied in places where we know that certain rules are always false and should not be part of the answer-set. For example, the rule (constraint) “:- y(a)” states that y(a) is always false.

The ASP rules can be negated using two available ASP approaches, i.e., Classical Negation and Negation by Failure. The Classical Negation is a pattern in which negative predicates are used to show the fact that the predicate under consideration has been proved to be false. For example, the rule “-x(a) :- y(a)” states that if y(a) succeeds then x(a) is false or -x(a) is true. Default Negation or Negation by Failure is used to make conclusions based on the absence of information. This type of negation is used to conclude about default rules and assume it to be true in case of the absence of enough information. For example, the rule “x(a):- not y(a)” states that if we are not able to prove that y(a) succeeds, then x(a) succeeds. So, in the example rule, we assumed that x(a) has succeeded based on the absence of information about y(a).
Chapter 3
Developing Comprehensive Postconditions

This chapter summarizes the approach, published in [26], of describing and defining so-called postconditions and frame conditions together. In this contribution, a new tabular, so-called post-/frame condition determining language (PCDL), method is proposed for developing comprehensive OCL postconditions (including frame conditions) for operations in UML and OCL models. Furthermore, this method is realized by a transformation chain for developing comprehensive behavioral models and checking their quality.

3.1 Post-/frame Condition Determining Language (PCDL)

In UML and OCL models, OCL pre- and postconditions describe the functionality of an operation in a declarative way. They limit system states in which an operation may be performed and describe properties that the resulting system state must meet. However, sometimes they may not comprehensive enough to describe what may or may not be changed in a transition between two system states and could lead to unexpected behavior of an operation. Also, for validation and verification methods, it is important to define model elements which may be changed or may not be changed in addition to the elements which are covered by pre- and postconditions.

As the solution of this problem, so-called frame conditions [14, 47] have been proposed in addition to pre- and postconditions. However, the process of generating frame conditions for any UML and OCL model is a complex and unwieldy task, as a significant amount of model elements, as well as their relations, has to be consid-
ered [23, 43]. Also, frame conditions are usually written manually and this can lead to inconsistent and flawed constraints.

To address this problem, we have proposed a so-called post-/frame condition determining language (PCDL) [26]. It systematically generates combined post- and frame conditions (comprehensive postconditions) and is grounded on a table format. In this approach, **PCDL elements** are introduced which are based on a new formal distinction for operation behavior between deleted, sustained and added objects, and simplify the postcondition specification in the PCDL table for a developer. Here, a developer *only* needs to define the model elements which are affected by the execution of an operation in the context of PCDL elements and the other elements by default are considered as unaffected. We have also proposed a method to automatically transform these PCDL elements into OCL postconditions. The aim of this approach is to give developers the ultimate chance to reduce the burden of formulating post- and frame conditions by offering an option to express the behavior of an operation systematically through the PCDL elements using effort reducing, supportive defaults.

The PCDL determines a specific tabular structure consisting of developer-defined elements which are typically OCL expressions written in a particular format (PCDL format). The tabular structure is transformed from a given application model (without postconditions) and additionally consists of proposed elements (Sect. 3.1.1). We call this complete schema a **PCDL model** which is then automatically transformed into the application model with postconditions. In this section, details about the tabular structure and elements of the PCDL including representation of postconditions in the PCDL format are described.

### 3.1.1 Distinction between Deleted, Sustained and Added Objects

Comprehensive postconditions for an operation should define which elements of a model are changed and more importantly which are sustained during the execution of an operation. Also, sometimes during the execution of an operation, new elements are added or existing elements are deleted which should also be defined by postconditions. So, to cover all aspects, we have distinguished operations execution into three distinct partitions, describing deleted (del), sustained (sus) and added (add) objects. In PCDL, if the execution of an operation requires object deletion or object generation, then the elements del or add of the object class should be specified, respectively. Also, if the execution of an operation expects changes in object attribute values or links, then the element sus of the object class should be specified. If the execution
of the operation does not require object deletion, object generation or changes in the object, then initial default PCDL elements remain as they are.

Figure 3.1 explains the basic idea of deleted, sustained and added objects. In OCL postconditions, one can refer with \texttt{C.allInstances} to the objects in class \texttt{C} at postcondition time, and with \texttt{C.allInstances@pre} one can reach the objects at precondition time. Having these two sets available, it is easy to formally define (a) the deleted objects as those which are present at \textit{precondition but not at postcondition time}, (b) the sustained object as those present at \textit{precondition and postcondition time}, and (c) the added objects as those which are present at \textit{postcondition but not at precondition time}.

This distinction of object elements allow developers to precisely formulate postconditions and frame conditions in a single unit of thought in an effective and systematic way. The distinction between deleted, sustained and added objects has not been studied in the literature before.

### 3.1.2 Representing Postconditions in Tabular Form

As previously stated, a tabular structure in the PCDL is generated from a given user-developed model. The model operations are transformed into columns and classes (further sub-categorized with the elements \texttt{del}, \texttt{sus} and \texttt{add}) are transformed into rows of the table. In the table, a developer can specify the elements \texttt{del}, \texttt{sus} and \texttt{add} for classes according to the expected execution of an operation. The small abstract
example from Fig. 3.2 is taken to understand the tabular structure and the PCDL format.

\[
\begin{array}{|c|c|c|}
\hline
\text{ClsA} & \text{ClsB} & \text{ClsA} \\
\text{attr1} & \text{attr3} & \text{roleA} \\
\text{attr2} & \text{roleB} & \\
\hline
\text{ClsA} \{ \text{attr1,attr2,roleB} \} & \text{ClsB} \{ \text{attr3,roleA} \} & \\
\text{ClsA: \text{opX}(param:Type)} & \text{ClsA: \text{opY}(param:Type)} & \text{ClsB: \text{opZ}(param:Type)} \\
\hline
\text{ClsA.del} & \text{isEmpty()} & \text{isEmpty()} \\
\text{ClsA.sus} & U = \text{ClsA.sus} & \text{C = Set\{self\}} \\
& & =\text{ClsA}\{U, \text{U,expr1}\} \\
& & U = \text{ClsA.sus} - C \\
\text{ClsA.add} & \text{ClsA}\{\text{const1,const2,null}\} & \text{isEmpty()} \\
\text{ClsB.del} & \text{isEmpty()} & \text{isEmpty()} \\
\text{ClsB.sus} & U = \text{ClsB.sus} & \text{ClsB}\{\text{const3,expr2}\} \\
\text{ClsB.add} & \text{isEmpty()} & \text{isEmpty()} \\
\hline
\end{array}
\]

Figure 3.2: Abstract Example: Class Diagram (Left) and Tabular PCDL Model (Right)

In this example, the model consists of two classes namely \text{ClsA} and \text{ClsB} with one association between them. \text{ClsA} has attributes \text{attr1} and \text{attr2}, and \text{ClsB} has attribute \text{attr3}. The generic tabular form of PCDL for the abstract example model is shown in the right of Fig. 3.2. In this table, the model operations (\text{opX}, \text{opY}, \text{opZ}) are transformed into the columns and the classes (\text{ClsA}, \text{ClsB}) are transformed into rows. Two letters \text{U} and \text{C} are used which refer to the set of unchanged and changed objects, respectively. Initially, each operation with elements \text{del} and \text{add} is defined with \text{isEmpty}() which means no objects are deleted or added, and elements \text{sus} are specified as \text{U = className_sus} which means all elements (attribute values or links) of the class are sustained without change. The gray-highlighted elements indicate changes from the initial default and are defined by the developer to generate postconditions according to the expected operation execution. The topmost row shows all classes with their attributes and role names in order to have an option to completely catch an object of the class. The developer follows this PCDL format to write the PCDL elements for constructing postconditions. For example, in operation \text{opX}, the element \text{ClsA_add} is defined by specifying \text{ClsA}\{\text{const1, const2, null}\} because it is assumed that the execution of operation \text{opX} will add an object of \text{ClsA}. Here \text{const1} refers to \text{attr1}, \text{const2} refers to \text{attr2} and \text{null} refers to \text{roleB} of \text{ClsA}. This format of defining elements is the same for the elements \text{del} and \text{sus}. However, in the element \text{sus}, a developer can also define the set of changed (\text{C}) and unchanged (\text{U}) objects. So, for each operation, the postconditions will be generated based on the defined elements and the default (initial) elements of the PCDL table.

The number of postconditions for each operation depends on the number of classes in the model. For example, in the abstract model, for three operations and two
classes (six PCDL class elements), a total of eighteen postconditions will be generated. With this approach, we have a clear and systematic case distinction for describing postconditions: the initial, default setting define all operations to do nothing (nothing deleted, nothing changed, nothing added); the developer then has to explicitly specify all operations that delete, change or add items with appropriate PCDL entries.

3.2 Transformation of PCDL into Postconditions

As explained in the previous section, a PCDL table consists of operations as columns and classes as rows further sub-categorized with elements del, sus and add. The postconditions for an operation are developed for the developer-defined and default PCDL elements from the table. In this section, we have proposed a method to automatically transform the PCDL elements into OCL postconditions. To better explain the transformation process, we take an abstract example. In Table 3.1, a generic representation of postconditions transformed from the default settings together with developer-defined elements of a class ClsA is presented.

<table>
<thead>
<tr>
<th>ClsA[{attr1,role1}]</th>
<th>PCDL elements</th>
<th>Generic OCL postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClsA_del</td>
<td>isEmpty()</td>
<td>ClsA_del-&gt;isEmpty()</td>
</tr>
<tr>
<td></td>
<td>ClsA{const1,expr1}</td>
<td>ClsA_del-&gt;size()=1 and (let v=ClsA_del-&gt;any(true) in v.attr1=const1 and v.role1=expr1)</td>
</tr>
<tr>
<td>ClsA_sus</td>
<td>U=ClsA_sus</td>
<td>ClsA_sus-&gt;forall(v</td>
</tr>
<tr>
<td></td>
<td>self=ClsA{U,expr2}</td>
<td>self.attr1=self.attr1@pre and self.role1=expr2 and (ClsA_sus-Set{self})-&gt;forall(v</td>
</tr>
<tr>
<td>ClsA_add</td>
<td>isEmpty()</td>
<td>ClsA_add-&gt;isEmpty()</td>
</tr>
<tr>
<td></td>
<td>ClsA{const2,expr3}</td>
<td>ClsA_add-&gt;size()=1 and (let v=ClsA_add-&gt;any(true) in v.attr1=const2 and v.role1=expr3)</td>
</tr>
</tbody>
</table>

Table 3.1: Generic Transformation of PCDL Elements into OCL Postconditions

In the table, the gray-highlighted parts are the developer-defined elements and their transformed postconditions, and others elements are default elements. In the generic OCL postconditions, the ClsA_del, ClsA_sus and ClsA_add refer to OCL expressions ClsA.allInstances@pre - ClsA.allInstances, ClsA.allInstances@pre->intersection(ClsA.allInstances) and ClsA.allInstances - ClsA.allInstances@pre, respectively (explained in Fig. 3.1). Using these OCL expressions, and based on default and developer-defined elements, postconditions are formed. For example, the element ClsA_del is isEmpty(), so the postcondition generated for this element is let ClsA_del = ClsA.allInstances@pre - ClsA.allInstances in ClsA_del->isEmpty(), and this assures that no ClsA object is deleted by the respective
operation execution. The transformation of any PCDL elements into OCL postconditions follows the specific format described in Table 3.1.

With this method, PCDL elements can be automatically transformed into operation postconditions, a complete behavioral UML and OCL model is generated, and the workload of writing postconditions manually is reduced. As described earlier, typically most of the (automatically generated) default elements will survive in the final PCDL table and that the developer has only to fix some spots in the table according to the expected operation behavior.

### 3.3 Transformation Chain

To realize the method introduced in Sect. 3.1, and to generate comprehensive postconditions effectively and precisely, we have proposed a transformation chain which starts with an application model without postconditions (user-developed model) and yields an automatically generated model with postconditions (test case model), using the PCDL concepts. This test case model is then transformed into the filmstrip model, and using our model validator, the test cases in the form of filmstrip object diagrams are generated for checking the model properties [36]. By analyzing the object diagrams, a developer can check behavioral model properties and accordingly, if required, can modify the model or the PCDL elements. Thus, the transformation chain offers iterative improvement steps with the help of test cases to develop a comprehensive behavioral model. The proposed transformation chain is depicted in Fig. 3.3.

![Figure 3.3: Model Transformation Chain for Developing Comprehensive Postconditions](image)

In Fig. 3.3, the gray-highlighted part shows newly introduced transformation steps that are integrated into our existing filmstripping and validation process. In the textual to tabular transformation step, a given UML and OCL application model
without postconditions (user-developed model) is transformed into a PCDL model which is basically a tabular structure consisting of default (initial) PCDL elements and is based on the desired operation execution. The developer modifies only necessary elements. In the tabular to OCL transformation step, we have introduced a method to automatically transform those elements into OCL postconditions. With help of our filmstripping approach for validating model behavior, the model with newly generated postconditions (test case model) is transformed into the filmstrip model and along with a configuration is given to the model validator. As an outcome, the model validator automatically generates a valid object diagram, and by analyzing the state transitions in the diagram, properties for model dynamics can be validated [37]. Overall, the transformation chain starting from the user-developed model into a semi-automatically derived test case model helps the developer to check the model quality.

To demonstrate the approach, a CompEmp application model in which a system can have many companies and employees, and a company can hire and fire an employee, is chosen as an example and shown in Fig. 3.4. The original application model is indicated in a gray-shaded style, namely the classes Sys, Emp, and Comp with the associations SysEmp, SysComp and Job in the class diagram. In the following sections from 3.3.1 to 3.3.6, each step shown in the transformation chain are demonstrated.

![Figure 3.4: Application Model and Filmstrip Model](image-url)
3.3.1 Transformation to PCDL

The first step in the chain is the transformation of textual to tabular model. Here, the CompEmp application model without postconditions is taken as the input model, and it is automatically transformed into the initial (default), tabular PCDL model, which is shown in Table 3.2.

<table>
<thead>
<tr>
<th>Sys::newEmp(aName:String)</th>
<th>Sys::newComp(aName:String,anAdr:String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_del</td>
<td>isEmpty()</td>
</tr>
<tr>
<td>Emp_del</td>
<td>isEmpty()</td>
</tr>
<tr>
<td>Sys_del</td>
<td>isEmpty()</td>
</tr>
<tr>
<td>Emp_del</td>
<td>isEmpty()</td>
</tr>
<tr>
<td>Emp_sus</td>
<td>U=Emp_sus</td>
</tr>
<tr>
<td>Emp_add</td>
<td>isEmpty()</td>
</tr>
<tr>
<td>Sys_sus</td>
<td>U=Sys_sus</td>
</tr>
<tr>
<td>Sys_add</td>
<td>isEmpty()</td>
</tr>
</tbody>
</table>

Table 3.2: PCDL Model in Tabular Form with Initial Elements

In Table 3.2, all four operations of the CompEmp model are represented as columns and all three classes with elements del, sus, add are represented as rows. As explained in Sect. 3.1.2, initially, the elements del and add are isEmpty(), and the elements sus are unchanged (U). Then the developer defines the PCDL elements according to the expected operations execution. In the example model, the intention of the execution of the operations is as follows: newEmp and newComp should add a new Emp and Comp objects, respectively, and link them with the Sys object; Similarly, the operations hire and fire should add or delete a Job link between the Comp and Emp objects, respectively.

3.3.2 Improving the PCDL Model

The developer modifies the default elements according to expected operation execution in this step. In Table 3.3, the elements edited by the developer are highlighted with a gray background. The short form incl, excl and allIns are used for the OCL terms including, excluding and allInstances, respectively. For the operation newEmp(aName:String) of the class Sys, the elements Emp_add and Sys_sus
can be understood as follows: (a) **Emp_add**: one new **Emp** object is added which has **aName** as attribute **name**, has empty **employer** set (**Set{}**) and is linked with the **self** (**Sys**) object; (b) **Sys_sus**: the **emp** set of the **self** (**Sys**) object includes the newly generated **Emp** object. For the operation **hire(anEmp:Emp)** of the class **Comp**, the elements **Comp_sus** and **Emp_sus** can be understood as (a) **Comp_sus**: the **employee** set of the **self** (**Comp**) object includes the **anEmp** object; (b) **Emp_sys**: the **employer** set of the **anEmp** object includes the **self** (**Comp**) object. The **fire** operation elements can be understood analogously to the **hire** operation elements except that here the **anEmp** and **self** (**Comp**) objects exclude each other. Here, the **newComp** operation elements are intentionally not correctly defined in order to explain the functioning of the transformation chain to generate precise postconditions.
3.3.3 Transformation to OCL Postconditions

The tabular PCDL model (Table 3.3) is transformed into the model with postconditions using the method introduced in Sect. 3.2. The generated postconditions for the newEmp operation from Table 3.3 are shown in Listing 3.1.

Listing 3.1: Postconditions Generated from PCDL Elements for Sys::newEmp(...)

```
context Sys::newEmp(aName: String)
post Comp_del_COND:
  let Comp_del=Comp.allInstances@pre−Comp.allInstances in Comp_del->isEmpty()
post Comp_sus_COND:
  let Comp_sus=Comp.allInstances@pre->intersection(Comp.allInstances) in Comp_sus->forAll(v |
    v.address=v.address@pre and v.name=v.name@pre
    and v.employee=v.employee@pre and v.sys=v.sys@pre)
post Comp_add_COND: ...
post Emp_del_COND: ...
post Emp_sus_COND: ...
post Emp_add_COND:
  let Emp_add=Emp.allInstances−Emp.allInstances@pre in
  Emp_add->size()=1 and (let v=Emp_add->any(true) in
  v.name=aName and v.employer=Set{} and v.sys=self)
post Sys_del_COND: ...
post Sys_sus_COND:
  let Sys_sus=Sys.allInstances@pre->intersection(Sys.allInstances) in
  self.comp=self.comp@pre and
  self.emp=self.emp@pre->including(
    (Emp.allInstances−Emp.allInstances@pre)->any(true)) and
  (Sys_sus−Set{self})->forAll(v | v.comp=v.comp@pre and
  v.emp=v.emp@pre)
post Sys_add_COND: ...
```

The postcondition is generated for all PCDL class element of the newEmp operation. The postconditions Comp_del_COND and Comp_sus_COND are transformed from default PCDL elements Comp_del and Comp_sus, and Emp_add_COND and Sys_sus_COND are transformed from developer-defined elements Emp_add and Sys_sus, respectively. During the execution of the newEmp operation, Comp_del_COND and Comp_sus_COND make sure that a Comp object is not deleted and all Comp sustained objects remain unchanged, respectively. Also, Emp_add_COND makes sure that a new Emp object is added with developer-specified model elements, and Sys_sus_COND assures that the new Emp object is linked with the Sys (self) object; other model elements of all Sys sustained objects remain unchanged. The remaining postconditions are handled analogously. This complete postcondition set assures correct behavior of the newEmp
operation. In the same way, comprehensive postconditions for other operations can be understood and generated.

3.3.4 Transformation to Filmstripping

The newly generated model with postconditions is now transformed into the filmstrip model (Fig. 3.4) for the validation purpose. Table 3.4 states the configuration needed to define the search space for the model validator. In the configuration, 5 Snapshot and 5 Sys objects are specified, and the specification of Emp and Comp objects is in range 5..9. So, the initial system state (snapshot) has 1 Sys object with 1 Comp and 1 Emp object. Also, the operations specification is in range 1..1. Here, the operations can be executed in any sequence, and based on the operations execution, other objects and links will be populated.

![Figure 3.5: Example Sequence Diagram](image)

<table>
<thead>
<tr>
<th>Classes and Associations [min..max]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snapshot</td>
</tr>
<tr>
<td>Emp</td>
</tr>
<tr>
<td>Comp</td>
</tr>
<tr>
<td>Sys</td>
</tr>
<tr>
<td>newComp_SysOpC</td>
</tr>
<tr>
<td>newEmp_SysOpC</td>
</tr>
<tr>
<td>hire_CompOpC</td>
</tr>
<tr>
<td>fire_CompOpC</td>
</tr>
<tr>
<td>Job</td>
</tr>
</tbody>
</table>

Table 3.4: Example Configuration

3.3.5 Transformation to Object Diagram

The USE model validator, based on the configuration and the generated filmstrip model, constructs solutions in the form of filmstrip object diagrams. From the various solutions (filmstrip object diagrams), one is shown in Fig. 3.6. The sequence diagram from the application model corresponding to the generated filmstrip object diagram is displayed in Fig. 3.5. In Fig. 3.6, the aggregation links show the incarnations of the objects. For example, the objects sys2, sys3, sys1, and sys4 are respectively the first, second, third and fourth incarnation of the object sys5. The newEmp operation execution adds the new object emp8 which is linked with the object sys2, the hire operation execution adds the Job link between the objects comp5 and emp2, and the
The fire operation execution deletes the Job link between the objects comp2 and emp3, as expected.

One would expect that the newComp operation execution generates a new Comp object and link it with the object sys3. However, in Fig. 3.6, it is to be noted that there are no changes in the system state (snapshot) after the execution of the operation. Also, some random values for the attributes of the newComp operation call object are generated. These oddities are highlighted with the gray background in Fig. 3.6, and are generated due to the fact that in the PCDL model (Table 3.3), the elements for the newComp operation are not correctly defined. To generate effective postconditions for the newComp operation, the PCDL elements need to be defined according to the
expected operation execution. By analyzing filmstrip object diagrams, developers can check operation behavior and according to that, if necessary, they can modify the PCDL elements.

### 3.3.6 Improving the PCDL Model

The developer, according to the validation results, modifies the PCDL table. So now, Table 3.5 includes the corrected PCDL elements for the **newComp** operation. Here, the elements **Comp_add** and **Sys_sus** are re-defined and can be understood as explained above for the **newEmp** operation elements. The transformations are performed again for the corrected PCDL model (Table 3.5), and with the same configuration (Table 3.4), the model validator generates the new filmstrip object diagram and is shown in Fig. 3.7.

![Diagram](image)

<table>
<thead>
<tr>
<th>Comp{address:String,name:String,employee:Set{Emp},sys:Sys}</th>
<th>Sys::newComp(aName:String,anAddr:String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_del isEmpty()</td>
<td>Sys::newEmp(aName:String)</td>
</tr>
<tr>
<td>Emp{name:String,employer:Set{Comp},sys:Sys}</td>
<td>U=Comp_sus</td>
</tr>
<tr>
<td>Sys{comp:Set{Comp},emp:Set{Emp}}</td>
<td>Comp(add,aName,Set{},self)</td>
</tr>
</tbody>
</table>

| Emp_del isEmpty()                                        | isEmpty()                                |
| Emp_sus U=Emp_sus                                        | isEmpty()                                |
| Emp_add Emp{aName,Set{},self}                            | isEmpty()                                |

| Sys_del isEmpty()                                        | isEmpty()                                |
| Sys_sus C=Set{self}                                      | C=Set{self}                              |
| self=Sys(U,employee->incl((Emp.allsEmp.allsEmp@pre)->any(true))) | self=Sys(comp->incl((Comp.allsComp.allsComp@pre)->any(true)),U) |
| U=Sys_sus-C                                              | U=Sys_sus-C                              |

<table>
<thead>
<tr>
<th>Sys_add isEmpty()</th>
<th>isEmpty()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp::hire(anEmp:Emp)</td>
<td>Comp::fire(anEmp:Emp)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>isEmpty()</td>
</tr>
<tr>
<td>C=Set{self}</td>
<td>C=Set{self}</td>
</tr>
<tr>
<td>self=Comp[U,employee-&gt;incl(anEmp),U]</td>
<td>self=Comp[U,employee-&gt;excl(anEmp),U]</td>
</tr>
<tr>
<td>U=Comp_sus-C</td>
<td>U=Comp_sus-C</td>
</tr>
</tbody>
</table>

| Emp_del isEmpty()                                        | isEmpty()                                |
| Emp_sus anEmp=Emp[U,employee->incl(self),U]              | anEmp=Emp[U,employee->excl(self),U]      |
| U=Emp_sus-C                                              | U=Emp_sus-C                              |

| Sys_del isEmpty()                                        | isEmpty()                                |
| Sys_sus U=Sys_sus                                        | isEmpty()                                |
| Sys_add isEmpty()                                        | isEmpty()                                |

Table 3.5: PCDL Model with Gray-shaded Developer-defined Elements (B)
The two oddities shown in Fig. 3.6 are fixed in Fig. 3.7. The `newComp` operation adds the new `comp4` object and links it with the object `sys3`. Also, the attributes of the `newComp` operation call object are generated properly. These are shown with the gray-highlighted part in Fig. 3.7.

So, based on this demonstration, it can be stated that the PCDL approach presented in this chapter assists developers to systematically define and construct post- and frame conditions. Also, the shown transformation chain helps to ensure the correctness and adequateness of postconditions for a UML and OCL model.
3.4 Approach Validation

A new ContentSharingPlatform model is used to validate and test the PCDL approach. The model captures the functionalities of creating, sharing, unsharing, and removing files or folders. It has two classes, namely User and ContentItem. The class ContentItem has enum attribute contentType, which shows whether it is a file or a folder. The class User has four operations, the operation generate to create a new file or folder, the operation share to share a file or a folder with other users, the operation unshare which will unshare the shared item, and the operation remove to delete the file or folder from the system.

![Figure 3.8: Class Diagram of the ContentSharingPlatform Model](image)

The default PCDL model elements are modified according to the expected operation execution and are shown (gray-highlighted) in Table 3.6. The element ContentItem_add for the operation generate(aContentName:String, aContentType:ContentType) adds one new ContentItem object which has aContentName as attribute name and aContentType as attribute contentType, has empty member set (Set{}) and is linked with the self (User) object, and the element User_sus includes the newly generated ContentItem object in the item set of the self (User) object. For the operation share(aContentItem:ContentItem, aMember:User) of the class User, the elements User_sus and ContentItem_sus include the aContentItem in the shareItem set of the aMember object and aMember object in the member set of the aContentItem object, respectively. The unshare operation elements can be understood analogously to the share operation elements except that here the aContentItem and aMember objects exclude each other. For the operation remove(aContentItem:ContentItem), the element ContentItem_del deletes the already existing aContentItem object, and the element User_sus excludes the existing aContentItem object from the item set of the self (User) object.

The tabular PCDL model with the modified elements is transformed into the UML/OCL model containing combined frame- and postconditions according to the
Table 3.6: ContentSharingPlatform PCDL Model with Gray-shaded Developer-defined Elements

method described in Sect. 3.2. For the model validation, the newly generated model is transformed into the filmstrip model using the filmstripping. The configuration needed to define the search space for the model validator are as follows: 5 Snapshot and 10 User objects are specified, and the specification of the ContentItem object is in range 0..5. So, the initial system state (Snapshot) has 2 User objects. Also, the operations specification is in range 1..1. Here, the operations can be executed in any sequence, and based on that, other objects and links will be populated. The model validator using the configuration and the filmstrip model generates a filmstrip object diagram as a solution. From many possible solutions, one is shown in Fig. 3.9.

By analyzing the filmstrip object diagram, it can be concluded that all the operation are executed as expected and all the necessary links are generated, i.e., the PCDL elements are correctly formulated for all operations. Thus, developers can check the operation behavior in the generated filmstrip object diagram, and accordingly, if necessary, they can modify the PCDL elements. The outcome of the test case generation has lead to believe that the PCDL model is correctly transformed into the UML/OCL models and the generation of comprehensive postconditions is accurate.
3.5 Related Work

A substantial body of research has focused on generating and specifying frame conditions for validation and verification of software systems. The authors in [2, 10] have proposed a possible solution of the frame problem by specifying modifier sets and implemented in the tool KeY. In [51], the authors have shown the generation of frame conditions within the verification tool Boogie. However, these approaches are not directly applicable to UML and OCL. The approach in [18] allows to automatically derive frame conditions from postconditions using a paradigm classifiable as *nothing else changes*. However, the resulting frame conditions are often not what the developer intended and can be non-trivial if adjusted manually. The author in [23] uses a straightforward approach by explicitly specifying what is *not* in the frame by extending the postconditions with further constraints. The major drawback of this approach is that it is time-consuming, as the frame conditions have to be written manually and one has to maintain them later on in the case of design changes. The
idea of the approaches in [47, 14] is to specify the set of model elements that are allowed to be changed during an operation call together with the pre- and postconditions using so-called invariability clause (modifies only statements). However, the process of generating frame conditions for a model in terms of the invariability clause requires complete consideration of all model elements and their relationships. The graph transformation approach in [40] defines the transformation rules using deleted, sustained (preserved) and added elements, but does not use OCL or a textual formula language, while our approach is completely based on OCL. Also, the concept of a formal distinction between deleted, sustained and added objects for OCL post-conditions has not been previously used. In [54, 72, 4], operation pre- and postconditions are specified by visual contracts which is basically a diagrammatic notation, however, in our approach, the operation pre- and postconditions are specified using OCL. In [65], the OCL invariants are translated into graph constraints to be used in graph-based approaches, whereas our approach is a logic-oriented approach which directly uses OCL. In comparison to the graph transformation approaches, our approach considers invariants along with pre- and postconditions for state transitions during the operation calls. In contrast to all mentioned approaches, we realize comprehensive OCL postconditions (including frame conditions) that are systematically developed with PCDL and the new distinction between deleted, sustained and added objects. In addition, our method is accompanied by a transformation chain for checking model quality.
Chapter 4

Generating Filmstrip Configurations and Templates

This chapter describes the development and implementation of an approach for automatic generation of filmstrip configurations (which configures the model validator search space) and so-called filmstrip templates which together strengthen the model behavioral validation process in the tool USE [28]. The configuration guides the model validator by setting lower and upper bounds for the number of objects and links and attribute values, and also determines the success of the validation process. The filmstrip configuration consists of application and filmstrip objects, and there are some dependencies between them. Until now, the filmstrip configurations and all the dependencies are calculated manually. With the new implementation, developers specify the configuration for (application) model elements and accordingly, a filmstrip model configuration and a filmstrip template [29] are automatically generated. A filmstrip template reduces the work of our model validator by identifying recurring model parts.

4.1 Extension Overview

Figure 4.1 gives an overview on the filmstrip model validation process integrated with the proposed configuration option and the filmstrip template approach (highlighted with a dotted rectangle). In the process, the model validator takes the filmstrip model and partially auto-generated filmstrip model configuration as input and constructs different test cases in the form of filmstrip object diagrams. Also, based on the developer-specified configuration, the filmstrip template can be automatically constructed, which is basically a partial object diagram that only needs to be completed by the model validator. These implementations enhance and optimize the validation
process and make it more user-friendly. Therefore, a large number of developers can have the opportunity to validate different UML and OCL models. The integration of these options in the tool USE does not change the model validation process.

The CivilStatus model is chosen to understand the filmstrip model configuration and template generation process. The CivilStatus application model consists of the class Person with the association Marriage and three operations namely, birth, marry, and divorce. The application model is contained in the filmstrip model and is indicated in a gray-shaded style, Fig. 4.2.

The filmstrip model configuration has application model elements (e.g., application class objects and association links) and filmstrip elements (e.g., Snapshot and OperationCall objects and their association links), and they have some dependencies on each other. Up to now, the complete filmstrip model configuration with their dependencies is formulated manually. In the proposed configuration option, the developer only needs to specify the number of application objects (per system state) and the total number of operation calls, and based on that, the model elements dependencies are automatically calculated, and the complete filmstrip model configuration is generated. Also, using the developer-specified configuration, the filmstrip template can be constructed, which is then further instantiated by the model validator. In the example configuration, two Person objects per system state, one marry and one divorce operation calls, and one Marriage link are specified. The following section shows the transformation of this developer-specified example configuration into the filmstrip model configuration and the template construction.
4.1.1 Generation of Filmstrip Model Configuration

The automatically generated filmstrip model configuration from the developer-specified configuration is shown in Listing 4.1. The number of snapshot objects, total application class objects (all system states), and the filmstrip association links (Filmstrip, SnapshotOpC, Snapshot, PredSucc) are automatically calculated, and other elements (e.g., operations, association links, attributes) remain same in the filmstrip model configuration.

Listing 4.1: Complete CivilStatus Filmstrip Model Configuration

```
[default]
Integer_min = -10
Integer_max = 10
# ------------------------ Structural application model elements
# ------------------------ Person
Person_min = 6
Person_max = 6
# Marriage (person1:Person, person2:Person) 
Marriage_min = 1
Marriage_max = 1
# ------------------------ Behavioral application model elements
# ------------------------ marry_PersonOpC
marry_PersonOpC_min = 1
marry_PersonOpC_max = 1
```
In Listing 4.1, the number of Snapshot objects is calculated based on the total number of operation calls (OpCalls). The Snapshot represents the system state, and the operation can be executed on any initial Snapshot, and successful execution leads to a new Snapshot. Therefore, three Snapshot objects are needed for the execution of two operations marry and divorce (number of Snapshot objects = number of total OpCalls + 1). In the filmstrip model configuration, number of application objects is referred to the total number of class objects of all the system states (snapshots). For example, in the initial configuration, there exist two Person objects per snapshot, so for three snapshots, six Person objects are needed (total number of application objects = number of application objects per snapshot*number of Snapshot objects).

The Snapshot links (for example, SnapshotPerson) connect the snapshot and the class objects, so the number of Snapshot links is the same as the total number of class objects. The PredSucc links show the incarnations of the class objects, i.e., to connect the same class objects in a different snapshot. For example, for one class object and three snapshots, two incarnations are required for the object. So, for three objects, two PredSucc links are necessary to connect. Here, we have two Person objects and three snapshots, so four PredSuccPerson links are generated (number of PredSucc links of the class X = number of X objects per snapshot*(number of Snapshot objects - 1)). The Filmstrip links connect the incarnations of the Snapshot object (number
of Filmstrip links = number of Snapshot - 1), and the SnapshotOpC links connect
the Snapshot and OperationCall objects (number of SnapshotOpC links = number
of total OpCalls).

4.1.2 Construction of Filmstrip Template

The filmstrip templates can be automatically constructed from the developer-specified
configurations. The template consists of snapshot objects and application objects
without attribute values, but with already established connecting links between them.
For example, if \( n \) operation calls on \( m \) application model objects are known from the
configuration, one can pre-compute the needed \( n+1 \) snapshot objects, each one being
connected to \( m \) application objects and one can establish the needed links automati-

cally.

Figure 4.3 shows the abstract view on the template, which is constructed from the
developer-specified configuration. The model elements shown in black (Snapshot and
Person objects without attribute values and links) represent the filmstrip template,
and the part highlighted in gray has to be generated by the model validator. So,
the model validator has only to find the proper OperationCall objects, application
model links, and attribute values.

Figure 4.4 shows the filmstrip object diagram generated by the model validator
with the filmstrip template. Here, the model validator only decides the operationCall
objects (marry_personopc1 and divorce_personopc1), the Marriage and Snapshot-
OpC links, and the values of the attributes (highlighted with gray-shaded style). The
overhead for handling the Snapshot and application model objects are effectively
reduced.
4.2 Development

This section describes the implementation of the approaches presented in Sect. 4.1. The introduction of the approaches does not affect the existing filmstrip model validation process, however only the process of configuring the search space in the context of the application model elements (i.e., objects per system states, links, operations) are presented and can be edited on a graphic user interface (GUI), and it automatically translates into filmstrip model configuration. Moreover, the option for automatic filmstrip template construction can be accessed through the GUI. This new interface also integrates the USE model validator so that the complete validation process can be automated.

The implementation has been carried out in Java 8 using IDE eclipse (EE), and the programming interface (API) from Swing [62] is used for the construction of the user interface, as is also used in the tool USE. The new implementation also mostly follows the principles of the Model-View-Controller (MVC) design pattern [32].

4.2.1 Implementation of Filmstrip Model Configuration Option

This section mainly describes the implementation of the filmstrip model configuration option in the kodkod plugin [49]. Also, the complete structure and functioning of
the extended kodkod plugin within the tool USE are explained. Figure 4.5 shows the fragment of the kodkod plugin framework. The part highlighted in a white-on-darkgray style is the newly implemented Java classes, and part highlighted in a black-on-lightgray style is the extended existing Java classes for the implementation of the filmstrip model configuration option.

Figure 4.5: Fragment of the Kodkod Plugin Framework

The KodkodValidateConfigurationAction is the initial step for accessing the GUI. Here, the model is distinguished as an application model and filmstrip model, and accordingly gives access to the configuration window, i.e., the ModelValidatorConfigurationWindow is for the application model, and the ModelValidatorFilmstripConfigurationWindow is for the filmstrip model.
ModelValidatorFilmstripConfigurationWindow is responsible for the design and functionality of the filmstrip model configuration option window for configuring the search space. Here, all necessary data structures, tables, frames and the menu component are initialized, and then called in a specific order. Also, either existing property file (which can have many configurations) is loaded or a new property file with initial configuration is initialized. The configuration data are stored in different configuration settings (SettingConfiguration). Here, the model configurations such as the number of model elements (operations, attributes, association links, class objects) and the total number of operation calls are gathered, and complete filmstrip model configuration is automatically calculated for further processing in the validation process.

The tables and necessary components are constructed according to the Builder design pattern [32]. The tableBuilder instantiates the corresponding TableModel with the required configuration settings and initializes the JTable. Also, the renderers and editors relevant to the table are appended to it. The filmstrip model configuration needs three tables for basic types (String, Integer and Real) and five other for classes, operations, associations, attributes and total operation calls. The data model for the operations (TableModelOperation) and total operation call (TableModelTotalOpC) are newly implemented, and the data model for classes and associations is extended in such a way that the filmstrip model classes and associations are appropriately configured.

The data structures underlying the tables are part of a composition that includes all configuration settings. Thus, their existence is closely related to the existence of the parent class for configuration settings (SettingsConfiguration). The tables and their methods access the individual components of the configuration settings for retrieving data, for example, the data for String data type table (TableModelString) access StringSettings. The newly implemented data models for operations and total operation call access ClassSettings, as the application model operations are the operation call classes of the filmstrip model and the total operation call is linked to the filmstrip class Snapshot (total number of operation calls - 1) (the filmstrip transformation is explained in Sect. 2.3.4).

The ConfigurationFileManager manages multiple configurations, i.e., it can add and update the configurations and can also remove the specific configuration from the property file, and passes the configurations to the PropertyWriter. The propertyWriter writes the configurations in the specific readable format or up-
dates the existing property file. New methods have been implemented within the PropertyWriter class to manage and export the filmstrip model configurations.

### 4.2.2 Filmstrip Model Configuration and Validation through GUI

In the tool USE, if a model is loaded, a click on this icon \( \text{\includegraphics[width=1cm]{icon}} \) in the toolbar or by calling it from the plugin menu, the user interface for configuring the model validator search space can be called. The newly implemented configuration window (Fig. 4.7) will be opened once the filmstrip model is loaded through the model configuration option window (Fig. 4.6). The entire GUI for configuring the search space are divided into two tabs, namely Basic Types (Fig. 4.7) and Model Elements (Fig. 4.8).

![Figure 4.6: Model Configuration Option Window](image)

![Figure 4.7: Filmstrip Model Configuration Window - Basic Types](image)

The Basic Types tab contains tables for Integer, String and Real data types (left of Fig. 4.7), and a field with the legends (right of Fig. 4.7) for the column names,
where the functions of the columns are explained in detail and examples of value assignments are given. In the integer table, minimum and maximum are used to set the limits for the values, i.e., how high or low an attribute value of this type should be generated during validation. The minimum and maximum in the real table have a similar meaning, except that these numbers can be written as floating point numbers. The value in the Step range column in the real table sets the step size between the generated real values in the search space, and the increase or decrease occurs in step range depending on the given value. The column Min. Div. Values and Max. Div. Values of the string table are object-related limits. They do not represent values like the minimum and maximum of the real and integer tables, however, it describes how many different string values should be there in the search space. It contains concrete values that should occur in the search space.

The Model Elements tab contains five tables for configuring class objects, operations, attributes, association links and total operation call. The class table (upper right Fig. 4.8) is implemented for configuring the application model classes. The columns with the names Min. Object Quantity and Max. Object Quantity describes the minimum and maximum number of instances (objects) of the class. The upper left and lower left corner of the figure shows the attribute table and association table configured according to the selected class, respectively.
The bottom right of Fig. 4.8 shows two tables, one for the operation according to the selected class and another for total operation call. Here, one can configure the operations by specifying minimum and maximum operation quantity. Based on the specified minimum and maximum value of the total operation call, the minimum and maximum value of the Snapshot objects are retrieved, and based on this Snapshot and application class objects, other filmstrip elements are calculated (explained in Sect. 4.1).

The menu bar at the top of the GUI is always accessible. The File menu contains basic options like Open, Save, Save as, Closed and Export as filmstrip configuration. The Export as filmstrip configuration option allows the developer to export the complete calculated filmstrip model configurations. The Configuration menu lists the sub-items Rename, Delete, Clone, Reset, New, Sort table entries and Validate. This menu item deals exclusively with actions that combine configuration sets under different configuration names. The menu sub-item validate has the same function as the Validate button at the bottom left of the GUI, i.e., translating the given configuration from the table into the filmstrip model configuration, passing it to the validation process and closing the GUI.

4.2.3 Filmstrip Template Construction Option

The filmstrip template can be automatically constructed based on the total number of operation calls and application objects, as explained in Sect. 4.1. The filmstrip template construction option (the button Construct Filmstrip Template) has been implemented and can be found at the bottom left, Fig. 4.8. The implementation of the functioning of the filmstrip template is based on the shown abstract algorithm in Listing 4.2 written in SOIL. The abstract model shown in the following algorithm has two classes, namely classX and classY. In the configuration, x number of ClassX objects, y number of ClassY objects and n number of operation calls are specified.

Listing 4.2: Abstract Filmstrip Template in SOIL

```plaintext
numOpC:=n ; numClassX:=x ; numClassY:=y ;
predSnap:=new Snapshot ;
a0SQ:=oclEmpty(Sequence(ClassX )) ;
b0SQ:=oclEmpty(Sequence(ClassY )) ;
for i in Sequence{1..numClassX} do
    a:=new ClassX ; a0SQ:=a0SQ->including(a);
    insert (a , predSnap) into SnapshotClassX ;
end ;
for i in Sequence{1..numClassY} do
    b:=new ClassY ; b0SQ:=b0SQ->including(b);
```

47
insert (b, predSnap) into SnapshotClassY;
end;
for j in Sequence\{1..numOpC\} do
  s:=new Snapshot;
insert (predSnap, s) into Filmstrip;
predSnap:=s;
a1SQ:=oclEmpty(Sequence(ClassX));
for i in Sequence\{1..numClassX\} do
  a:=new ClassX; a1SQ:=a1SQ->including(a);
insert (a, s) into SnapshotClassX;
insert (a0SQ->at(i),a1SQ->at(i)) into PredSuccClassX;
end; a0SQ:=a1SQ;
b1SQ:=oclEmpty(Sequence(ClassY));
for i in Sequence\{1..numClassY\} do
  b:=new ClassY; b1SQ:=b1SQ->including(b);
insert (b, s) into SnapshotClassY;
insert (b0SQ->at(i),b1SQ->at(i)) into PredSuccClassY;
end; b0SQ:=b1SQ;
end;

In this algorithm, the SOIL statements until the start of the third for loop (numOpC) constructs the initial objects of the classes, one Snapshot object and the link between Snapshot and class objects for the first system state. The rest of the SOIL statements construct other system states with elements, i.e., Snapshot and class objects, Filmstrip links between Snapshot objects and predSucc links between the class objects of the different system states. Basically, the predicted object diagram elements from the total number of operation calls (min. value from the total operation call table) and the number of class objects (min. object quantity of the class table) are constructed in advance.

The action of the button Construct Filmstrip Template will generate the partial filmstrip object diagram according to the given configuration and once the diagram is successfully constructed, a message shown in Fig. 4.9 is displayed. The model validator then only needs to complete the partial object diagram, i.e., only a few elements have to be decided to achieve the complete filmstrip object diagram (test case), which reduces the overall execution time.

4.3 Quality Assurance

The purpose of this section is to validate and test the correctness of the implemented code for the filmstrip model configuration window and verify whether the code contains any faults. The main focus of the testing is to assure the process of the con-
configuration specification and model validation using the newly implemented GUI and also to verify the auto-generated filmstrip model configuration and the filmstrip template. Two test cases with the different models (TollCollect model and SocialNetwork model) are chosen for the testing.

4.3.1 Test Case of the TollCollect Model

The first model chosen to verify the filmstrip configuration and validation process using GUI is the TollCollect model. It contains two classes, namely Truck and Point, and two associations, namely Connect and Current. The class Point provides a point in the route network and class Truck shows the current level of debt and the sequence of trip points. Figure 4.10 shows the class diagram of the TollCollect filmstrip model. The application model is completely contained in the filmstrip model and indicated in a gray-shaded style. To configure the filmstrip model for the validation, one can initialize the newly developed configuration window in the tool USE. The initialization utilizes the KodkodValidateConfigurationAction and ModelValidatorFilmstripConfigurationWindow along with TableBuilder and SettingsConfiguration.

The scenario for the test case is “one truck and two points exist, and only all truck operations should be executed”. For this scenario, one Truck and two Point objects, all truck operation in range 1..1, all association links in the range 0..* (* means infinity) and total five operation calls are specified in the configuration. The filmstrip model configuration window of the TollCollect model with the specified configuration is shown in Fig. 4.11.

The filmstrip template can be automatically generated from the developer-specified configuration. Using the configuration (Fig. 4.11), the template should generate six Snapshot objects, each one being connected to two Point and one Truck application objects (total six Truck and twelve Point objects), as explained in Sect. 4.1. Also,
the needed links (PredSucc and Snapshot) are generated. The action of the button Construct Filmstrip Template successfully constructs the partial object diagram, Fig. 4.12.

The figure shows that the template is accurately constructed and the number
of generated objects and links is same as expected. A successful construction also means the links are connected to the objects correctly, i.e., PredSucc links show the incarnations of the objects, all application objects are properly linked to the Snapshot objects (the links SnapshotPoint and SnapshotTruck) and Snapshot objects are connected with the Filmstrip links.

The developer-specified configuration is further transformed into the filmstrip configuration. The transformed configuration should have six Truck and twelve Point objects, six SnapshotTruck and twelve SnapshotPoint links, six Snapshot objects, five SnapshotOpC and five Filmstrip links, and other elements same as developer-specified configuration.
specified (operations and links) as explained in Sect. 4.1. The action of the button Validate should automatically generate complete filmstrip model configuration and should initialize the model validator to generate test cases for the valid configuration.

The generated test case for the developer-specified configuration is shown in Fig. 4.13. The figure shows that all the operations of the class Truck are executed as expected. The number of object and links are populated according to the expected filmstrip model configuration. So, it can be concluded that the transformation of the developer-specified configuration to the complete filmstrip model configuration is successful, and the development is correct. Also, the model validator is successfully executed and has generated the expected test case. Thus, the Validate button functionalities are assumed to be correctly implemented.

4.3.2 Test Case of the SocialNetwork Model

The second chosen model is the SocialNetwork model, in which a user can invite, accept and decline a friendship request. Figure 4.14 shows the class diagram of the SocialNetwork filmstrip model. The SocialNetwork application model (indicated in a gray-shaded style) consists the classes Friendship and Profile with the associations Invite and Invitee. This model is different from the TollCollect model, in the sense that the SocialNetwork model has enumeration data type (Status attribute of the Friendship class), and has the operation whose execution adds new object (e.g., operation invite adds the Friendship object) rather than only adding links and changing attribute values. So, this model can give more perspective to the testing. Also, the selected test case of the SocialNetwork model is different. Here, to generate the expected scenario, the number of links and attributes values are restricted and the operation specification are given in the range.

![Figure 4.14: Class Diagram of the SocialNetwork Filmstrip Model](image)

52
The test scenario is “there exist three profiles, and three friendship requests should be first sent between them and then they should be accepted”. In the configuration for this scenario, three Profile objects and six total operation calls are specified, Friendship objects are in range 0..3 (as it is dependent on the execution of invite operation) and all operation are in range 0..6. Also, to generate the expected scenario, the links and attribute values are restricted, i.e., both Invite and Invitee association links are in range 12..12, and Status attribute value is specified as \{invite, accept\}. The filmstrip configuration window of the SocialNetwork model with the specified configuration is shown in Fig. 4.15.

The filmstrip template from the specified configuration should generate seven Snapshot objects, each one being connected to three Profile objects (total twenty-one Profile objects). Also, the needed links (PredSucc and Snapshot) are generated. Here, minimum quantity of the Friendship object is zero. Thus, the Friendship objects should not be constructed in the template. The action of the button Construct Filmstrip Template successfully constructs the partial object diagram, Fig. 4.16.

In Fig. 4.16, the objects and links are generated as expected. Also, the Friendship objects are not generated in the template, as the model validator will decide the operation calls and according to that, the Friendship objects will be generated.

The developer-specified configuration is further transformed into the filmstrip configuration. The transformed configuration should have twelve Profile objects,
twelve SnapshotProfile links, Friendship objects in range 0..21, seven Snapshot objects, six SnapshotOpC and six Filmstrip links, and other elements remain same as developer-specified (operations and links), as explained in Sect. 4.1. The action of the button Validate automatically constructs the complete filmstrip configuration, initializes the model validator and generates the test case as shown in Fig. 4.17.

The figure shows that the three friendship requests have been sent (invite operation calls), and then, they are accepted (accept operation calls) by profiles, as mentioned in the scenario. The number of objects and links are populated according
Figure 4.17: Generated Filmstrip Object Diagram (Test Case) - SocialNetwork Model
to the expected configuration. So, it can be concluded that the transformation of the configuration and their implementation is correct. Also, no decline operation is called as attribute Status value has only \{invite, accept\}. The given links specification guided the model validator to generate the expected operation call sequence (first three invite and then three accept operation calls). Thus, the model validator is successfully executed and the Validate button development are believed to be correct.
Chapter 5

Generating OCL Invariants

This chapter deals with the generation of additional OCL invariants, which along with configurations, are given to the model validator to construct the test cases for checking and validating model behavior in our filmstrip validation process. This contribution proposes a test case schema (TC schema) approach [24], in which the developer can diagrammatically express a scenario which then can automatically be transformed into an OCL invariant. The schema is a visual representation of a behavioral test scenario, and it is instantiated by the model validator to achieve different concrete test cases.

5.1 Test Case (TC) Schema Approach

The scenarios for model validation are generated using configurations and additional OCL invariants. The configuration gives the finite search space for object diagrams by setting lower and upper bounds for the number of objects and links and attribute values. External OCL invariants are specified to guide the object diagram generation into a particular direction, e.g., for attesting that objects or links with particular properties exist (for example, for the initial or final scenario state). These invariants may also be used to configure the (imaginary) sequence diagram from the application model, e.g., for requiring that a certain sequence of operations is called [36]. Thus, it is essential to specify OCL invariants for model behavior validation. Up to now, these additional OCL invariants had to be written manually.

However, due to various complexities in OCL, it is very difficult for developers with little or no knowledge of OCL to write OCL expressions [19]. In addition, manually writing OCL can lead to inconsistent and erroneous OCL expressions. Therefore, to formulate precise and valid OCL invariants, a lot of expertise and background knowledge about the OCL is needed.
In this chapter, to address the challenge of writing OCL invariants for model validation, we have proposed a transformation-based approach to make scenario generation independent from the OCL expertise or knowledge. In this approach, developers can express the scenario diagrammatically by constructing a so-called test case schema (TC schema), which then can automatically be transformed into an OCL invariant. Furthermore, the model validator is used to instantiate the abstract TC schema in order to generate multiple concrete test cases. In Fig. 5.1, the dotted rectangle shows the newly introduced OCL transformation approach, and its integration into the existing filmstrip transformation and validation process.

Figure 5.1: Overview on Filmstrip Validation (Gray Box) with TC Schema Approach (Dotted Box)

In order to provide an overview on our proposed approach, we have chosen a ContentSharingPlatform model in which a user can generate or remove the files and folders, and share and unshare them with other users as an example. Figure 5.2 shows the class diagram of the ContentSharingPlatform filmstrip model. The application model consists the classes User and ContentItem with the associations Ownership and Share, and is completely contained in the filmstrip model and indicated in a gray-shaded style.

Figure 5.2: Class Diagram of the ContentSharingPlatform Filmstrip Model
The TC schema can have different snapshots which represent system states, and it can contain application model objects and links. The objects of different snapshots can be connected through filmstrip model links. As a filmstrip object diagram has a sequence of operation calls, a user can imagine an operation call between every two snapshots in the TC schema to express a scenario. In the proposed OCL transformation, OCL expressions are generated separately for all the objects and links in the TC schema which are then combined into a single complex OCL invariant. The TC schema can be interpreted differently depending on the drawn links and other possible application links. So, there are different ways of handling the transformation, and thereby different OCL representations are possible.

![Figure 5.3: Scenario: a ContentItem Unshared by a User](image)

We have considered a small scenario of the `ContentSharingPlatform` model in which, initially a user owns one content item which is shared with another user, and then it is unshared by the owner. In Fig. 5.3, the scenario is expressed through the object diagram which is roughly a TC schema. Later, some links will be added to constrict this TC schema. In the example (Fig. 5.3), the user *Ada* owns the content item *Work* and shares it with the user *Bob*. Then the user *Ada* unshares the content item *Work* with the user *Bob*. So, to realize the scenario, the operation call `Ada.unshare(Work, Bob)` has been executed. `Snapshot1` and `Snapshot2` represent the system states before and after the operation *unshare* is called, respectively. The generated OCL invariant for the scenario depicted in the diagram is as follows:

```oclnotation
context 
inv AdaUnshareWorkWithBob:
  User.allInstances->exists(ada1, bob1, ada2, bob2 | Set{ada1, bob1, ada2, bob2}->size() = 4 and
  ada1.succ = ada2 and bob1.succ = bob2 and
  ContentItem.allInstances->exists(work1, work2 | Set{work1, work2}->size() = 2 and work1.succ = work2 and
```

58
The OCL invariant is generated based on the drawn and other possible links in the diagram. In the invariant, the variable names with suffix ‘1’ represent the Snapshot1 objects and with suffix ‘2’ represent the Snapshot2 objects. A detailed description about handling the transformation and OCL representation are discussed later in the chapter.

5.2 Example Scenarios for TC Schema

A TC schema is basically a partial filmstrip object diagram consisting of different snapshots that represent system states, and these snapshots can contain application model objects and links. The objects of different snapshots can be connected through filmstrip (pred,succ) links. We now show two example scenarios, one each of the SocialNetwork model and the ContentSharingPlatform model to understand the construction of the TC schema.

5.2.1 SocialNetwork Model - Test Scenario 1

In the SocialNetwork model, a user can invite, accept and reject a friendship request. Figure 5.4 shows the class diagram of the SocialNetwork filmstrip model. The appli-

Figure 5.4: Class Diagram of the SocialNetwork Filmstrip Model
cation model (indicated in a gray-shaded style) consists the classes Friendship and Profile with the associations Invite and Invitee.

The description of user defined test scenario 1 is as follows: there exist two user profiles. In the initial state, they are not linked with each other, and in the final state, they are linked with each other through a friendship request. Figure 5.5 shows the TC schema for this scenario.

![Figure 5.5: TC Schema of Test Scenario 1](image)

In Snapshot1, there exist two user profiles which are not linked with each other. In Snapshot3, the Profile3 is linked to Profile6 through Invite and Invitee links as well as a Friendship object. For each snapshot, developers have a choice between a so-called open snapshot and a closed snapshot, and the specification of snapshots will be according to the expected scenario generation. If a snapshot is classified as closed, only the stated links (if any) are allowed between the snapshot objects in a generated test case; if a snapshot is classified as open, other links are allowed between the snapshot objects in a generated test case. For example, in this scenario no links should exist in Snapshot1, and only drawn Friendship object and links should exist in Snapshot3. Thus in the transformation, these snapshots are considered as the closed snapshots and Snapshot2 is handled as the open snapshot. In the diagram, the filmstrip (pred, succ) aggregation links are used to connect different snapshot objects which basically represent the incarnations of the objects. For example, the objects Profile2 and Profile3 are respectively first and second incarnation of the object Profile1.

5.2.2 ContentSharingPlatform Model - Test Scenario 2

The description of test scenario 2 is as follows: initially, a user owns a content item and shares it with user A, and in the final state, the item is unshared with user A.
and shared with user B. For this scenario, three User and one ContentItem objects are needed for each snapshot. The TC schema of test scenario 2 is shown in Fig. 5.6.

In Snapshot1, there exist three users and one content item, and the user owns the content item, i.e., User1 is linked to ContentItem1 through a Ownership link, and shares it with user A, i.e., User4 is linked to ContentItem1 through a Share link. In Snapshot3, the content item is shared with user B, i.e., User9 is linked to ContentItem3 through a Share link. Thus, Snapshot1 and Snapshot3 are considered as the closed snapshots as there should not be any other additional links, and Snapshot2 is considered as the open snapshot.

Here, the TC schema is presented with two different example test scenarios to show the applicability and diversity of the approach. In the TC schemas, during the state transition, the object (Friendship) and links are added in test scenario 1, whereas in test scenario 2, only the links (Share) are added. Also, to understand the open and close snapshot transformation more precisely, different types of states are considered. For example, in test scenario 1, only the absent or present links are handled in the single closed snapshot, however, in test scenario 2, both absent and present links are handled together in the single closed snapshot. These TC schemas, according to their open and closed classification, are instantiated by the model validator to generate different test cases and are shown in Sect. 5.4. The detailed description of the classification of snapshots and their different OCL representations are discussed in Sect. 5.3.
5.3 Transformation of a TC Schema into an OCL Invariant

As previously stated, a TC schema is comprised of snapshots (application objects and links) and filmstrip links. So during the transformation from a TC schema to an OCL invariant, these elements must be transformed into OCL expressions. The OCL expressions for the filmstrip (pred, succ) links which connect different snapshot objects are directly generated using the \texttt{succ} role name. However, different OCL representations are possible for the snapshots, as a snapshot may have other possible application links, apart from shown links. So, the transformation can be varied based on the handling of these unspecified application links for each snapshot.

The developer can classify a snapshot as an \textit{open} snapshot or as a \textit{closed} snapshot. She can choose from these options in order to generate different OCL invariants during the transformation. By choosing open or closed, the developer determines a so-called connectedness of the snapshots. The notion connectedness refers to the way (open or closed) snapshots are considered for the OCL transformation. In the open snapshot case, exactly the mentioned objects and links are fixed by the OCL invariant, but it is possible that more links are present in the generated test case. However, in the closed snapshot case, apart from mentioned objects and links, other possible application links which are not explicitly mentioned, are excluded in the generated OCL invariant. To understand the snapshot classifications, the small abstract example is chosen and shown in Fig. 5.7.

![Figure 5.7: An Abstract Example: Open vs. Closed Snapshots](image)

In this example, a many-to-many relationship exists between the classes P and R. The link between the objects P1 and R1 is represented with the OCL expressions \texttt{P1.roleR->includes(R1)} and \texttt{R1.roleP->includes(P1)}. And to represent the absent link between the objects P1 and R2, the expressions \texttt{P1.roleR->excludes(R2)} and \texttt{R2.roleP->excludes(P1)} (gray font in Fig. 5.7) are used. With this example, it is certain that in an open snapshot, only the present links are considered in the
OCL expressions, while in a closed snapshot, both the present and absent links are considered in the OCL expressions.

The model validator will construct test cases depending on the generated OCL invariant. In the closed snapshot, only present links in the snapshot will be constructed in the test cases. In the open snapshot, present links will be constructed, and absent application links (not drawn in the snapshot) can be decided by the model validator, whether or not to be constructed in the test cases. Examples of scenario generation using the USE model validator are shown in Sect. 5.4.

5.3.1 Examplification through Examples

In a TC schema, if all snapshots are considered as closed snapshots then OCL expressions are generated for all absent links, and could generate a developer expected scenario. However, generated OCL invariant with only closed snapshots could be more restrictive and complex. And if all snapshots are considered as open snapshots then unexpected scenario generation could be possible. Therefore, in our approach, a developer can choose between an open and closed option for every single snapshot to efficiently generate expected scenarios. To illustrate the generation of an OCL invariant, we continue with the TC schemas shown in Sect. 5.2.

In an OCL invariant, each expression represents different elements such as application objects and links, and filmstrip links of a TC schema. To individualize each object, the size property is used for each set of objects in an OCL invariant. Filmstrip links of a TC schema are transformed using succ role name. Application links are transformed based on the connectedness of the snapshots. The excludes or != (not equal to), and includes or = (equal to) properties are used to respectively represent absent and present application links according to different relationships between classes. For example, the excludes and includes properties are used for links of a many-to-many association. The generated OCL invariant for test scenario 1 (Sect. 5.2.1) is as follows:

```oclass
context Snapshot inv initNoFriendshipFinFriendship:
Profile.allInstances->exists(p1,p2,p3,p4,p5,p6|
  Set{p1,p2,p3,p4,p5,p6}->size()=6 and
  p1.succ=p2 and p2.succ=p3 and p4.succ=p5 and p5.succ=p6 and
  Friendship.allInstances->exists(f1 |
  //Snapshot1
  p1.friendshipR.invitee->excludes(p4) and
  p1.friendshipE.inviter->excludes(p4) and
  p4.friendshipR.invitee->excludes(p1) and
  p4.friendshipE.inviter->excludes(p1) and
```
For the closed snapshots, OCL expressions are generated guaranteeing (a) the absence of links between Profile1 and Profile4 in Snapshot1 and (b) the presence of links between Profile3 and Profile6 in Snapshot3. In the open snapshot (Snapshot2), OCL expressions are not generated, as there are no links. In test scenario 2 (Fig. 5.6), Snapshot1 and Snapshot3 are considered as the closed snapshots, and Snapshot2 are considered as the open snapshot. The generated OCL invariant for test scenario 2 (Sect. 5.2.2) is as follows:

```ocl
class Snapshot
context Snapshot inv initShareWithUserAFinShareWithUserB:
User.allInstances->exists({u1, u2, u3, u4, u5, u6, u7, u8, u9}\nSet{u1, u2, u3, u4, u5, u6, u7, u8, u9}->size()==9 and u1.succ=u2 and u2.succ=u3 and u4.succ=u5 and u5.succ=u6 and u7.succ=u8 and u8.succ=u9 and ContentItem.allInstances->exists({c1, c2, c3}\nSet{c1, c2, c3}->size()==3 and c1.succ=c2 and c2.succ=c3 and //Snapshot1
u1.item->includes(c1) and u4.shareItem->includes(c1) and c1.owner=u1 and c1.member->includes(u4) and c1.member->excludes(u7) and u7.shareItem->excludes(c1) and //Snapshot2
u2.item->includes(c2) and c2.owner=u2 and //Snapshot3
u3.item->includes(c3) and u9.shareItem->includes(c3) and c3.owner=u3 and c3.member->includes(u9) and c3.member->excludes(u6) and u6.shareItem->excludes(c3) )
```

Here also based on the connectedness of the snapshot, the OCL invariants are generated. For example, in Snapshot1, the OCL expressions are generated for the link between User1 and ContentItem1 and User4 and ContentItem1, and also for the absent link between User7 and ContentItem1. The same as for Snapshot3. However, Snapshot2 is determined as an open snapshot. So, the OCL expressions are generated only for the present link between User2 and ContentItem2.
5.4 Applying the Model Validator for Scenario Generation

The model validator uses a given configuration and the generated OCL invariant for setting the sequence of operation calls by fixing (a) attribute values and (b) objects and links that have been left open in the TC schema in order to construct different test cases. We check the feasibility of our approach by transforming a TC schema into an OCL invariant and analyzing the generated test cases. The scenario generation using the model validator is demonstrated through the mentioned test scenarios (in Sect. 5.2) of the SocialNetwork model and the ContentSharingPlatform model.

5.4.1 Generated Test Cases of the SocialNetwork Model

To generate the test scenario 1 (Fig. 5.5), the generated OCL invariant for the scenario (Sect. 5.3.1) with the specified configuration are given to the model validator, and it constructs the concrete test cases in the form of filmstrip object diagrams. In the configuration, 3 Snapshot objects, and for each snapshot, 3 Profile objects are specified. Also, all the operations specification are in the range 0..2. So, the model validator will decide on the sequence of operation calls based on the given OCL invariant in valid filmstrip object diagrams.

Figure 5.8: Automatically Generated Test Case 1 of the Scenario 1
The generated test cases (filmstrip object diagrams) are shown in Figs. 5.8, 5.9 and 5.10: the first test case is an invite; invite; the second is an invite; accept;
and the third is an \textit{invite}; \textit{decline}. In the TC schema, one friendship is expected between two profiles. In order to satisfy the scenario at least one \textit{invite} operation call should exist, and another operation call could be an \textit{invite}, \textit{accept}, or \textit{decline} as the attribute \textit{status} of the \textit{friendship} object is not specified. In all the test cases, the test schema is satisfied (highlighted with dashed rectangle). The objects and links in the open snapshots have been decided by the model validator depending on the operation calls.

### 5.4.2 Generated Test Cases of the ContentSharingPlatform Model

For the test scenario 2 (Fig. 5.6), in the configuration, 3 \texttt{Snapshot} objects, and for each snapshot, 4 \texttt{User} and 1 \texttt{ContentItem} objects are specified. Also, all the operations specification are in the range 0..2. So, based on the configuration and the given OCL invariant, the model validator will decide on the sequence of operation calls in the test cases.

![Object diagram](image)

Figure 5.11: Automatically Generated Test Case 1 of the Scenario 2

Figures 5.11 and 5.12 show the filmstrip object diagrams which are generated by the model validator for test scenario 2 (Fig. 5.6). The part of the diagrams which is highlighted with a gray dashed rectangle shows the developer expected scenario. In
the TC schema, initially, a content item is shared with one user, and in final snapshot, the same content item is shared with another user. So, the content item can be either first unshared and then shared with another user (Fig. 5.11) or shared first and then unshared with the previous user (Fig. 5.12). In order to satisfy the scenario, one unshare and share operation call should exist in any sequence. Here, also the test schema in both test cases is satisfied. The objects and links in the open snapshots have been decided by the model validator depending on the operation calls.

The results of the scenario generation illustrate that a valid sequence of operation calls of the TC schema is present in the constructed test cases (filmstrip object diagrams). Also, in the shown test cases, the expected test scenarios are precisely generated. These show the successful transformation of an OCL invariant from a given TC schema, and validate that our concept of distinguishing between open and closed snapshots leads to the desired results.

5.5 Related Work

There are several contributions discussing different techniques and approaches for OCL constraint and test case generation. In [48], the tool MoMuT::UML is pr-
presented to generate fault based test cases for UML state machine models. In [20], the authors describe symbolic scenarios as operation sequences to generate functional test cases. In [13], the authors propose a method to generate test data on a higher-order representation of OCL models. In [17], the authors present an approach to analyze graph transformation rules based on an OCL representation. The semantics of transformation rules together with their properties are transformed into OCL expressions. In [30], a visualOCL language has been proposed for a graphical representation of the OCL, and the transformation of visualOCL to a textual OCL using graph transformation rules has been presented. In [1], the authors present a library of reusable OCL specification patterns. They utilize specific patterns for which OCL constraints can be automatically generated. In [7], a mechanism to connect design patterns with OCL constraint patterns is proposed which allows generating OCL constraints automatically. In contrast to the above mentioned approaches, we generate an OCL constraint from an SRD which consists of a specific number of objects and links based on a specified scenario. So, instead of using any kind of patterns and graph transformation rules, we directly generate an OCL expression for each element of an SRD. In [75], the authors propose an algorithm which automatically generates OCL constraint templates by extracting needed information from a UML class diagram. This OCL template guides software designers in generating effective OCL expressions. However, with our approach, the developer can directly generate OCL constraints without any intermediate step such as guiding OCL templates. In [69], the OCL constraints and class diagrams are generated from the use case specifications which are described by state machines using a proposed algorithm. In contrast, we are generating an OCL constraint from an SRD which is basically an object diagram with different snapshots. In [8], the authors are using Semantic Business Vocabulary and Rules (SBVR) transforming constraints and operation pre- and postconditions written in the natural language to OCL statements. However, we transform a scenario which is in form of a diagram. In [15], the authors propose a theorem proving interactive environment (HOL-OCL) for analyzing and validating UML models and OCL constraints that may need user assistance. However, our validation approach is automatic. In [18], the tool UMLtoCSP allows a user to perform verification and validation of a UML/OCL model based on Constraint Logic Programming. However, in that approach, additional OCL constraints are written manually to validate the model properties. In contrast to all these works, our approach is the only one generating OCL constraints automatically from a developer-specified scenario to generate concrete test cases for behavior model validation.
Chapter 6
Constructing Dynamic Scenarios

This chapter focuses on the scenario generation for the validation and verification of model properties about dynamics. A catalogue of scenario patterns has been proposed which guides the developer to construct different test cases [25] [27]. This chapter significantly describes the catalogue and shows detailed descriptions and manifestations of all patterns using different UML and OCL models. Also, the nature and purpose of all scenario patterns is explained to ease the process for a developer when to use them for particular models or model fragments.

6.1 Dynamic Scenario Pattern Catalogue

A developer can create dynamic scenarios to check and to test model behavioral properties. 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). In our context, 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 after each operation call.

Depending on the model under consideration, the developer needs to take into account many different dynamic scenarios for checking the model behavior properties. However, the choice of these scenarios can be difficult and cumbersome tasks. So, to relieve the burden of recognizing and covering all necessary scenarios, we have proposed a catalogue of different scenario patterns. The aim of this catalogue is to give advice and develop guidelines for constructing dynamic scenarios. 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 catalogue reduces the burden of identifying all necessary and crucial scenarios by giving
systematic guidelines to the developer for constructing operation sequence calls (i.e., dynamic scenarios) that check dynamic model properties.

All scenario patterns are manifested using our model behavior validation and verification technique called filmstripping (explained in Sect. 2.3.4). However, the scenario patterns are formulated independent of the filmstripping approach and can be also applied in other tools and approaches. Using our USE model validator [38], test cases in the form of (filmstrip) object diagrams are automatically constructed for the given dynamic scenarios, and by analyzing the state transitions in the diagram behavioral properties of a UML and OCL model can be checked. The dynamic scenarios are determined by configurations (specifications that determine how classes, associations and attributes are populated in the object diagram) and additional OCL invariants that are not part of the application but only serve to direct and help the process of finding test cases.

In this chapter, the notions ‘scenario patterns’, ‘scenario’, and ‘test case’ are frequently used and it is necessary to distinguish them in order to understand the development process. Therefore, we summarize these terminologies and describe the relationship between them as follows: one ‘scenario pattern’ is an abstraction of many ‘(dynamic) scenarios’ i.e., many different scenarios can be emerged from one scenario pattern, and from each single ‘(dynamic) scenario’ many different positive or negative ‘test cases’ can be generated. With ‘positive’, we refer to a test case in which all operations are executable and with ‘negative’ we refer to a test case in which at least one operation call cannot be executed.

An ordinary UML/OCL model is chosen to understand different patterns and generation of the scenarios. The model have $k$ number of classes (namely $A$, $B$, $C$, ...) with $n$ number of operations (namely $opX()$, $opY()$, $opZ()$, ...). We assume a descriptive behavioral model which has OCL class invariants and OCL operation contracts (pre- and postconditions). 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 ‘$c\cdot opZ(); b1\cdot opY(); a\cdot opX(); b2\cdot opY()$’ 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 or invariant and thus will not be executed completely.
Our catalogue currently contains eleven different patterns and is described in tabular form in Fig. 6.1. 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 are proposed based on our own work experience, and one can discover more such scenario patterns and extend the catalogue.

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

Figure 6.1: Catalogue of Scenario Patterns

All scenario patterns proposed in Fig. 6.1 have their own importance and usefulness. But not all patterns may be applicable for all models. The usefulness and applicability of the patterns 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 relation with each other. The patterns ALLOPS and SML can also be applied to all type of models, but it would be more practical to use them for the models which have a large number of operations. The pattern REPEAT is useful for those models in which a single operation can be called repeatedly. Sometimes, an operation which is followed by a specific sequence of operation calls is called again, in the sense 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 executions of operations add and remove model elements (e.g., objects or links). The pattern DEADEND applies only to those models which have an end object model in which no operation can be executed. The pattern ABSENCE can be applied to any model, and it helps the developer to analyze whether some operation in a model are more important than the other operations. In the patterns INIT2FIN and REGULAR, developers can create initial and final conditions or regular expressions over operation calls, respectively, for checking particular behavior of the model. Thus, they can also be applied to any model.

The intention of the catalogue, as said, is to give methodological support, advice and inspiration, in the sense that, after a set of dynamic scenarios has 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 in a fruitful way for smaller and larger models.

6.2 Scenario Pattern Manifestations

This section shows the complete manifestation of all eleven scenario patterns with different UML and OCL models. Along with the manifestation, we have described the nature and purpose of each pattern which can help the developer in choosing the patterns for a particular model. Model validation and verification are performed using our filmstripping approach, and the proposed catalogue is applied for constructing different test cases (filmstrip object diagrams), and by analyzing the test cases one can check model behaviour properties.

6.2.1 Each Operation Once (ONCE)

Nature and purpose. In the pattern ONCE, each single operation of the model should be executed once in a scenario, i.e., if the model has total n operations then n unique scenarios will be generated. The pattern checks the behavior of each single operation separately, before checking behavior in the context of other operations with other patterns. With this pattern, a necessary condition for the consistency of the model can be verified by checking whether each single operation is executable at all.
Example. The Scheduler model [73] is chosen to manifest this pattern. The model (Fig. 6.2) has classes Process and Scheduler. The class Scheduler has four operations, the operation Init to initiate a new scheduler, the operation New to add a new process to the waiting queue, the operation Ready to put a process that is currently in the waiting queue either in ready queue (if other process is active) or directly activate the process, and the operation Swap to swap the currently active process by putting it in the waiting. In this pattern, these four operations are separately executed and the scenario is generated in the form of filmstrip object diagrams (test cases) using the model validator. In the configuration, number of objects (each system state - 1 processor and 2 scheduler) and operation calls (single operation per execution) are specified to construct the scenario. The filmstrip object diagrams for the operations Init, New, Ready and Swap are shown in Fig. 6.3, Fig. 6.4, Fig. 6.5 and Fig. 6.6, respectively.
The test cases are generated as expected, each has one single operation call, and in it each system state (snapshot) has 1 Processor and 2 Scheduler objects. The operation Init (Fig. 6.3) execution has initialized the scheduler (the object scheduler1), the operation New (Fig. 6.4) has added the object process1 into waiting queue and is depicted with the link waiting between the object scheduler1 and process1, the operation Ready (Fig. 6.5) execution has put the object process3 into ready queue (the link ready between scheduler2 and process3), the operation Swap (Fig. 6.6) has put the object process3 (active process) in the waiting queue (the link waiting between process3 and scheduler1) and has activated the process which was in the ready queue (the link Active between process2 and scheduler1).

6.2.2 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, according to the expected operation behavior. The intention of this pattern is to detect relationships between operations.

Example. The CivilStatus model (Fig. 6.7) is chosen to demonstrate this pattern. The model has three operations, the operation birth to set the specific gender of the person (initially a gender of the person is female and civil status is single), the operation marry and divorce to add and remove the link Marriage between two Person objects with the different gender, respectively. So, there are total nine possible
pairs. In the configuration, 2 Person objects are planned for each snapshot. The model validator tries to execute all pairs in order to construct the scenarios in the form of filmstrip object diagrams (test cases) using the generated filmstrip model and the given configuration.

<table>
<thead>
<tr>
<th>Operations</th>
<th>birth(aGender:Gender)</th>
<th>marry(aSpouse:Person)</th>
<th>divorce()</th>
</tr>
</thead>
<tbody>
<tr>
<td>birth(aGender:Gender)</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>marry(aSpouse:Person)</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>divorce()</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 6.1: Executable and Non-Executable Operation Pairs

In Table 6.1, the pairs that could be executed [yes] and that could not be executed [no] are shown. As expected, the birth-divorce operation pair cannot be executed because, the operation divorce cannot be executed without marrying. In the same way, other non-executable pairs can be understood. The generated filmstrip object diagrams for the executable pairs are shown in Fig. 6.8 to Fig. 6.11.
In the Fig. 6.8, the objects person5 and person3 are born with a specific gender and single civil status, in the birth-marry operation execution (Fig. 6.9), the object person4 is born and is married to the object person2 (dipicted with the Marriage link), in Fig. 6.10 (divorce-marriage operation pair), the objects person9 and person8 are divorced (removes the link Marriage) and are married again (adds the Marriage link), and in Fig. 6.11 (marry-divorce operation pair), the objects person1 and person2 are first married (adds the Marriage link) and then are divorced (removes the Marriage link). Thus, with these executable and non-executable pairs, the behavior of the operations is validated.

6.2.3 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 hold in at least one scenario. It tries to show that the contracts and invariants are not contradicting each other.

**Example.** This pattern is demonstrated with the *TollCollect* model (Fig. 6.12). It has two classes *Point* and *Truck* and total eight operations. The class *Point* has three operations, the operation *init* to initialize the point, the operation *northConnection* to get the north point, and the operation *southConnection* to get the south point. The class *Truck* has five operations, the operation *init* to initialize the truck, the
operation **enter** to enter a point, the operation **move** to go to another point, the operation **pay** to pay the debt, and the operation **bye** to exit the system. In this pattern, all these operations are executed in any sequence in a single scenario. In the configuration, the operations specification is given in range 1..1, which makes sure that all operations exist once in the scenario. One generated filmstrip object diagram for this scenario is shown in Fig. 6.13.

In the generated test case, all operations are called once as expected and the execution of each single operation holds all invariants and their specific pre- and postconditions. Thus, this generated 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. Furthermore, the intention is to show that the model pre- and postconditions (contracts) and invariants are not contradicting each other, because they can be satisfied. Here, this scenario can generate many different test cases (filmstrip object diagrams) by executing these operations in different orders.

### 6.2.4 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 should be generated. The goal of this pattern is to approach system behavior incrementally, i.e., a developer starts with the scenario which has a short sequence, then a middle-sized one and in the end a long sequence. Step by step, the level of scenario complexity is increased which can help for better analysis of model behavior. Ideally, the involved operations in the small, medium-sized and long scenarios are disjoint to each other in order to gain a broader spectrum of operation call sequences.

**Example.** For this pattern, a *HealthSystem* model is chosen, being an adaptation of the model developed in [12]. The model has been modified according to our requirement, as it originally did not have any operations or contracts. Apart from newly proposing the catalogue, we have newly introduced an approach for automatically generating operations for classes: for each attribute, an operation `set'AttributeName'` is generated, which sets the attribute; for each role name, an operation `link'RoleName'` is generated, which adds the specific link. For example, in Fig. 6.14, the class **Patient** has the attribute **name**, so we have the operation **setName**, which will set the name of
a Patient object. Patient is associated with the classes Healthrecord and Doctor with the role names healthRecord and treatedBy, respectively. Therefore, we have the operations linkHealthRecord and linkTreatedBy, which will construct links for BelongsTo and Treats, respectively. The post- and frame conditions of the operations are specified using our developed PCDL approach (explained in Sect. 3.1). It is a non-trivial model and has many operations making it suitable for this pattern.
The model has seventeen operations, and to fulfill the pattern criteria, three scenarios with four, eight and twelve operation calls are generated using the model validator. To assure that each called operation is unique in the scenario, the operations specification is given in range 0..1 in the configuration. Also, the below shown...
additional invariant (initAttributesAndLinks) is specified which make sure that initially there exist no links between objects, all string-valued attributes are empty ('"'), Boolean attribute values are false and Integer attribute values are 0. Figure 6.15, Fig. 6.16 and Fig. 6.17 are the filmstrip object diagrams with the short (4 calls), middle-sized (8 calls) and long (12 calls) operation call sequence, respectively.

As expected, in Fig. 6.15, the operation setdrug has set the value of attribute drug of the object prescription1, the operation linkheakthrecord has added the
link BelongsTo between the objects patient1 and healthrecord3, the operation linktreats has added the link Treats between the objects patient2 and doctor2, and the operation setdescription has set the attribute description value of the object medicalinfo2. Similarly, the other two generated object diagrams can be analyzed and the behavior of the operations can be checked.

**context** Snapshot inv initAttributesAndLinks:
\[
\text{self}.\text{pred}() = \text{null} \implies \\
\text{self}.\text{healthRecord}.\text{medicalInfo} \rightarrow \text{size}() = 0 \text{ and} \\
\text{self}.\text{healthRecord}.\text{prescription} \rightarrow \text{size}() = 0 \text{ and} \\
\text{self}.\text{patient}.\text{treatedBy} \rightarrow \text{size}() = 0 \text{ and} \\
\text{self}.\text{medicalInfo}.\text{prescription} \rightarrow \text{size}() = 0 \text{ and} \\
\text{self}.\text{healthRecord} \rightarrow \text{forAll}(x | x.\text{owner} \rightarrow \text{size}() = 0) \text{ and} \\
\text{self}.\text{prescription} \rightarrow \text{forAll}(x | x.\text{drug} = '') \text{ and} \\
\text{self}.\text{prescription} \rightarrow \text{forAll}(x | x.\text{amount} = 0) \text{ and} \\
\text{self}.\text{prescription} \rightarrow \text{forAll}(x | x.\text{creator} = '') \text{ and} \\
\text{self}.\text{patient} \rightarrow \text{forAll}(x | x.\text{name} = '') \text{ and} \\
\text{self}.\text{medicalInfo} \rightarrow \text{forAll}(x | x.\text{description} = '') \text{ and} \\
\text{self}.\text{medicalInfo} \rightarrow \text{forAll}(x | x.\text{creator} = '') \text{ and} \\
\text{self}.\text{medicalInfo} \rightarrow \text{forAll}(x | x.\text{open} = \text{false})
\]

### 6.2.5 Repeatable Operation (REPEAT)

**Nature and purpose.** In the scenario pattern REPEAT, the same operation should be executed repeatedly in a single scenario. This pattern analyzes the operation properties by checking if each single operation can be executed repeatedly or not, according to the specific operation behavior.

**Example.** To demonstrate the pattern, the PersonCompany model (Fig. 6.18) is chosen. In the model, a company can hire and fire many people, and to check these behaviors, two scenarios are generated. In the first scenario (Fig. 6.19), the operation fire is repeatedly executed for five times. During each fire operation execution, the link WorkFor is removed between Person and Company objects. In the second scenario (Fig. 6.20), the operation hire is also called repeatedly for five times. Here, as expected, the link WorkFor is added between the Person and Company objects.
Figure 6.19: Filmstrip Object Diagram - Operation fire

Figure 6.20: Filmstrip Object Diagram - Operation hire
6.2.6 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 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. 6.21) in which users can borrow and return book copies, is used to demonstrate this pattern. To construct the cyclic scenario, the first and last operation calls in the scenario are specified by the 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 generated. The scenario in the form of filmstrip object diagrams (test cases) are shown in Fig. 6.22 and Fig. 6.23, respectively and as expected, the first and the last operation call in the generated test cases are same. Also, the execution of borrow operations are added and return operations are removed the link borrow between the User and Copy objects. The additional invariants (firstOpC and lastOpC) for the operation borrow are shown below. For the operation return, the class borrow_UserOpC is replaced by the class return_UserOpC in the invariants.

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

context Snapshot inv lastOpC:
  self.succ()=null implies
    self.prd.opc.oclIsKindOf(borrow_UserOpC)
Figure 6.22: Filmstrip Object Diagram - Cyclic borrow Operation Calls

Figure 6.23: Filmstrip Object Diagram - Cyclic return Operation Calls
6.2.7 Adding and Successively Removing Objects or Links (MOUNTAIN)

**Nature and purpose.** Here, in the pattern MOUNTAIN, an operation call sequence should be generated that starts with no objects or links, then incrementally adds objects or links until a peak number is reached and 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 manifest this pattern, the Library model (Fig. 6.21) is used. The User class of the Library model has the operations borrow and return, which respectively add and remove the link Borrow between the objects User and Copy. The below shown three invariants (initNoLink, increments and decrements) ensure that the generated 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, and the integers in expressions self.pred().num<3 and self.pred().num>=4 ensure the size of the mountain, i.e., until the fourth snapshot (third operation) a link will be added, and from the fifth snapshot (fourth operation) the 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 6.24 shows the constructed filmstrip object diagram in which initially three increments (operation borrow) and then three decrements (operation return) of the Borrow links are generated. The filmstrip object diagram also contains the Book objects not shown in Fig. 6.24.
6.2.8 Dead-end Call Sequences (DEADEND)

Nature and 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 restricted. Not all models can be 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. 6.25), in which a user can invite, accept and decline a friendship request. To check the model behavior, in the configuration, two Profile objects are specified, and the operation specification is in range 2..3. In the generated scenarios (filmstrip object diagrams), only two operation calls exist. The first operation call is invite, and another operation call is accept (Fig. 6.26) or decline (Fig. 6.27). As there are
only two Profile objects, no further operation call is possible. If the operation specification is given in range 3..3, then the filmstrip object diagram generation for this scenario is unsatisfiable. This implies that in this scenario, after two operation calls, there is a dead end.

Figure 6.26: Filmstrip Object Diagram - invite-accept Operation Call

Figure 6.27: Filmstrip Object Diagram - invite-decline Operation Call

6.2.9 Absence of an Operation in Operation Call Sequence (ABSENCE)

Nature and purpose. The scenario 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 taste on the pragmatics of the operation importance.

Example. The SocialNetwork model (Fig. 6.25) is again used to demonstrate this pattern. It has three operations (invite, accept, decline). So, according to the pattern, three scenarios are possible by keeping each 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 generated scenario with absence of accept, has the operation call sequence invite-decline (Fig. 6.27), and absence of decline has the operation call sequence invite-accept (Fig. 6.26). The scenario generation for absence of invite is unsatisfiable because without friendship
invite the person cannot accept or decline the friendship request, i.e., the operation accept or decline cannot be called without the operation invite. So, it can be concluded that the operation invite plays an important role in the model.

6.2.10 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 properties of operations by accordingly specifying initial and final requirements.

Example. The pattern is demonstrated using the StudentReport model (Fig. 6.28), in which a student can register and deliver a report, a teacher can give grades, and both can view the marks. To generate the scenario using this pattern, the initial and final state conditions (shown below) are given to the model validator 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 range 9..9. The scenarios in the form of filmstrip object diagrams are constructed, and one is shown in Fig. 6.29. The figure shows that initially their exists 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 diagram also contains the Student and Teacher objects not shown in Fig. 6.29.

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)
6.2.11 Regular Expressions over Operation Calls (REGULAR)

Nature and purpose. In the scenario pattern REGULAR, the operation call sequence in the scenario should be generated according to a given regular expression over oper-
ations 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 generated or not.

Figure 6.30: Class Diagram of the Account Application Model

Example. To manifest this pattern, the Account model (Fig. 6.30) is used which has one class Acc. The class Acc has four operations, namely open to specify the account number and to unlock the account, deposit to deposit the specific amount in the account which increases balance accordingly, payout to payout the amount from the account which decreases the balance accordingly, and lock to lock the opened account. To fulfill the criteria for this pattern, the regular expression open();[deposit();payout();]+lock(); is specified, and the corresponding OCL invariants are carefully developed and are shown below.

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

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 range 6..6. Based on the given invariants and the configuration, the scenario in the form of filmstrip object diagram is generated and shown in Fig. 6.31. As seen in the figure, the first and last operation call are open and lock, respectively, and in between two times deposit-payout operation call sequence is generated. This complete operation call sequence satisfies the given regular expression.
6.3 Combination of Patterns

Depending on the model and its behavior, the combination of the above proposed patterns can provide further suggestions to the developer to construct scenarios. However, some pattern combinations make sense and some do not. To demonstrate and show how the combination of patterns can be helpful to generate significant scenarios, we use the `ContentSharingPlatform` model as an example. The patterns `PAIR` and `REPEAT` are chosen as a combination of patterns to generate different positive and negative test case scenarios. The nature of this pattern (`PAIR-REPEAT`) is to check whether the combination of all asymmetric pairs can be repeatedly executed or not, according to the expected operation behavior.

**Example.** The `ContentSharingPlatform` model (Fig. 6.32) captures the functionalities of creating, sharing, unsharing, and removing files or folders. The model has four operations, the operation `generate` to create a new file or folder, the operation `share` to share a file or a folder with other users, the operation `unshare` which will unshare the shared item, and the operation `remove` to delete the file or folder from the system. So, there are total twelve asymmetric possible pairs of operations. In the configuration, 2 `User` objects and 0 or 1 `ContentItem` object are planned for each snapshot. The
model validator tries to execute all pairs in order to construct the scenarios in the form of filmstrip object diagrams (test cases) using the generated filmstrip model and the given configuration.

<table>
<thead>
<tr>
<th>generate - share</th>
<th>PAIR</th>
<th>PAIR-REPEAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate - unshare</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>generate - remove</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>share - generate</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>share - unshare</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>share - remove</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>unshare - generate</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>unshare - share</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>unshare - remove</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>remove - generate</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>remove - share</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>remove - unshare</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 6.2: Executable and Non-Executable Operation Pairs and Repetition of Pairs

In Table 6.2, based on the specified configuration, all the executable [yes] and non-executable [no] pairs and their repetition are shown. As obvious, the pairs that are not executable, their repetition is also not possible. The pairs that could be executed but could not be repeatedly executed are highlighted as black-on-gray. As expected, the **generate-share** operation pair cannot be executed repeatedly, as in the configuration between two users maximum one content item should exist. Also, the execution of **share-remove** and **unshare-remove** operation pairs are possible, but their repetitions are not possible because after the operation **remove**, the content

![Figure 6.33: Filmstrip Object Diagram - generate-remove Operation Pair Repitition](image)
item cannot be shared or unshared without generating it. Apart from these, there are four pairs whose repetition is possible, and their generated filmstrip object diagrams are shown in Fig. 6.33 to Fig. 6.36.

In all the generated filmstrip diagrams, as specified in the configurations, each snapshot has two User objects and zero or one ContentItem objects. Also, according to the described operation behaviors, the Ownership and Share links and contentItem objects are generated and removed. So here, the generated positive as well as negative scenarios using PAIR-REPEAT pattern helps to explore and validate the different aspects of the model behavior.
6.4 Evaluation

This section shows the summarization of all the patterns by showing the number of test cases, objects in single system state and operation calls. Also, it shows the complexity of the UML/OCL models, which have been used in this study, by showing the number of elements of the application and filmstrip models and the coverage of OCL expressions (invariants and pre- and postconditions).

The coverage, as described in Sect. 2.3.2, indicates the complexity of OCL invariants and contracts in the sense that it indicates the number of class model elements (attribute, association end) that are accessed in the particular OCL formulae. The number of elements and the coverage by OCL invariants and pre- and postconditions of the example model are shown to understand the complexity of the models. The table in Fig. 6.37 shows the overview on all application and filmstrip model elements, their coverage by invariants and pre- and postconditions, and overall details of the generated scenarios for all the proposed patterns.

The first line in the table expresses the following: the behavior pattern ONCE is manifested with a UML and OCL example application model Scheduler that has 2 classes, 3 invariants, 4 operations with 15 pre- and postconditions and 3 associations; the invariants, preconditions and postconditions complexity, i.e., the coverage, is measured as 15, 8, 157 (e.g., the value 15 for invariant coverage means that in the invariants, 15 accesses to attributes or association ends are made); the automatically transformed filmstrip model has 9 classes, 30 invariants, 9 operations, 9 associations; the filmstripping invariants coverage is 300; 4 test cases are generated having each
Figure 6.37: Evaluation: Patterns together with Evaluated Models, their Complexity and Constructed Scenarios

3 objects in each filmstrip snapshot; in each test case there is 1 operation call. Similarly, the other example model and their complexities shown in the table can be understood.

### 6.5 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 [1] presented a library of reusable OCL specification patterns to simplify the constraint definition in a UML/OCL behavioral model. The authors [31] introduced a model and a scenario-based pattern catalog to ensure the quality of specifications for the Modal Sequence Diagrams (MSD) requirement language. The paper [39] introduced 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 [1], [31] and [39], we propose the scenario patterns to guide developers for constructing dynamic scenarios to generate test cases. In [9], the authors proposed a catalog of anti-patterns that analyze a typical constraint interaction that cause correctness or a quality problem in UML class diagrams and suggest possible repairs. Our catalogue does not contain
test cases can be generated. In [22], the authors proposed 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 [20], the authors proposed a scenario expression language to describe scenarios as operation sequences to generate functional test cases. [55] presented an algorithm which is based on the formal operational semantics for deriving tests from sequence diagram specifications. In [57], 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 [53], the authors described the use of a genetic algorithm to generate test cases from finite state machines for class behavioral testing. In contrast to [20], [55], [57] and [53], we do not propose or use any explicit algorithms or languages for generating test cases. In [13], the authors proposed 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 [74] [80] described an approach to generate test cases automatically from state charts. In [63], the authors presented a tool called TCGen to automatically generate test cases using different testing criteria starting from UML models. In contrast to [74], [80] and [63], our work provides methodological suggestion and advice for developers to generate the test cases. [3] proposed 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 generation. [76] presented a prototype generator for generating traces to test the model behavior. However, our catalogue can be employed for developing operation call sequences to test model behavior properties.

In contrast to other mentioned works, our approach proposes a catalogue of patterns for developing dynamic scenarios to generate test cases independent of testing techniques and shows how behavioral patterns are manifested.
Chapter 7
Exploring Usability of ASP for Checking SBVR

The work presented in this thesis so far represents the main research contributions by
the author. This section illustrates the additional work done during the eight weeks of
industrial internship by the author at Tata Research Development and Design Center
(TRDDC), Pune, India. The main research work of the author so far concentrates
on the development and checking of UML/OCL models, whereas the internship work
focuses on a new language (SBVR) and validation approach (ASP).

The work is part of one of their ongoing project regarding a Validation and Ver-
ification (V&V) approach for domain models built in Semantics of Business Vocab-
ulary and Business Rules (SBVR) using the Answer Set Programming (ASP) logic
paradigm. The main task was to explore further the usability of ASP approach
for checking the SBVR model. The basics about SBVR and ASP are explained in
Sect. 2.4 and Sect. 2.5, respectively.

In [70] [45], the authors proposed and developed (a) an (semi-) automated ap-
proach for transformation of the natural language text to the SBVR Model via au-
thoring of Structured English (SE) rules, (b) the model-based generator to generate
model traversal functions to traverse the model and translate it to ASP rules and
constraints (ASP program), and (c) a V&V approach in which the DLV system [52]
is used as the solver for the generated ASP programs to construct the set of test cases
and test data for checking SBVR models.

The work shown in this chapter is the extension of the approach proposed in [45].
Here, the author have explored the usability of the ASP to validate and verify the
SBVR model by constructing the scenarios for checking different model properties,
and implemented the approach for automated generation of ground facts (test data)
for the V&V approach. Also, the guidelines are developed for negating the ASP
program to generate negative scenarios that help to further analyze the model for testing. Moreover, different test cases are generated to ensure the quality of the negative ASP programs and validate the SBVR model. Here, the test cases are formulated by specifying different facts, and the solver generates different answer sets (scenarios) based on the given facts.

The rest of the section is structured as follows. Section 7.1 explains formalized SBVR rules from natural language (NL). Section 7.2 describes the usability of ASP with SBVR by generating the scenarios for checking different model properties. In Sect. 7.3, the guidelines for generating the negative ASP rules are shown, and their quality assurance and the model validation are presented in Sect. 7.4.

7.1 Formalizing SBVR Structured English (SE) Rules

The SE rules are semi-automatically transformed from NL (English) text using available machine learning/natural-language processing (ML/NLP) based front-end engine that extracts the domain model and dictionary (i.e., various terms and concepts) from the regulatory text and provides suggestions to domain experts in authoring their regulatory rules in structured english [70]. The modules Customer Acceptance Policy (CAP) and Customer Identification Procedures (CIP) of the regulation Know Your Customer (KYC) published by Reserve Bank of India (RBI) [61] are chosen as an example NL text. From the KYC example text, few SE rules (Listing 7.1) are formulated to construct the small concrete example SBVR model for exploring usability of ASP. The basics about SBVR rules formulations is explained in Sect. 2.4.

Listing 7.1: Formulated SBVR Structured English (SE) Rules

\begin{verbatim}
rule r1 It is necessary that customer @is highRiskCust ||
  customer @is lowRiskCust
rule r2 It is necessary that customer @is salaried
  if customer @is lowRiskCust
rule r3 It is necessary that customer @is trusts ||
  customer @is pepOfForeignOrigin if customer @is highRiskCust
rule r4 It is necessary that bank @opensAccountFor customer
  if customer @approaches bank && customer not @is bannedCustomer
  bank @verifiesIdentityOf customer
rule r5 It is necessary that customer @is bannedCustomer if
  bank $computeIsBannedCust_value_true
rule r6 It is necessary that bank not @harassTo customer
rule r7 It is necessary that bank @harassTo customer
  if bank @closed account &&
  account $isProvidedReasonForClosed_value_false
\end{verbatim}
rule r8 It is necessary that bank verifiesIdentityOf customer
    if customer submits document
rule r9 It is necessary that customer submits document
    if customer is salaried & salaried hasRequired document
rule r10 It is necessary that salaried hasRequired document
    if document $type_value_PAN || document $type_value_aadhaar
rule r11 It is necessary that customer submits document
    if customer is trusts & trusts hasRequired document
rule r12 It is necessary that trusts hasRequired document
    if document $type_value_registrationCertificate ||
      document $type_value_telephoneBill
rule r13 It is necessary that customer submits document
    if customer is pepOfForeignOrigin & pepOfForeignOrigin hasRequired document
rule r14 It is necessary that pepOfForeignOrigin hasRequired document
    if document $type_value_infoFromPublicDomain
rule r15 It is necessary that customer owns account
    if bank opensAccountFor customer
rule r16 It is necessary that bank periodicUpdatecheckOf customer
    if bank UpdatedDetailsOf customer
rule r17 It is necessary that bank UpdatedDetailsOf customer
    if customer is lowRiskCust & customer owns account &
      account $UpdatePeriod is less than 5
rule r18 It is necessary that bank UpdatedDetailsOf customer
    if customer is highRiskCust & customer owns account &
      account $UpdatePeriod is less than 2

There are eighteen rules and one could understand those as follows: rule r1 categorizes a customer as high risk or low risk. Rules r2 and r3 define the low and high risk customers, i.e., a salaried person can be categorized as the low risk customer, and a trust or a politically exposed person (PEP) of foreign origin can be categorized as the high risk customer. Rule r4 defines requirements for a bank to open an account for a customer, i.e., the customer should approach the bank, verify their identities and is not a banned customer, and the banned customer is defined in rule r5. Rule r6 is the constraint that the bank should not harass the customer, and the harassment is defined in rule r7. Rule r8 to r14 specify the necessary documents required for customers to verify their identities. Rules r16 to r18 specify that bank periodically updates details of the customer, and the periodicity of the updation should not be less than once in five years in case of the low risk customer and not less than once in two years in case of the high risk customer. The periodically updates only possible on the existing accounts, i.e., a customer should have or owns the account, and it is defined in rule r15.
7.2 Generating Scenarios using ASP

The SBVR rules are automatically translated into the ASP rules and constraints (ASP program) using the already available model-based generator [45]. Apart from the ASP program, the solver also needs ground facts (test data) to perform automated analysis and generate scenarios or answer sets for checking model properties. Until now, the ground facts are written manually. During my time at TRDDC, I developed and implemented an approach for automatically generating the basic ground facts. These facts can be further modified by the developers to construct different test cases.

Listing 7.2: Fragment of the ASP Program

%rule r1
customerIsHighRiskCust(Customerid, HighRiskCustid) v
customerIsLowRiskCust(Customerid, LowRiskCustid); - customer(Customerid),
highriskcust1(HighRiskCustid), lowriskcust(LowRiskCustid).

%rule r2
customerIsSalaried(Customerid, Salariedid); -
customerIsLowRiskCust(Customerid, LowRiskCustid), salaried(Salariedid).

...
%rule r4
bankOpensAccountForCustomer(Bankid, Customerid) :-
customerApproachesBank(Customerid, Bankid),
bankVerifiesIdentityOfCustomer(Bankid, Customerid),
not customerIsBannedCustomer(Customerid, BannedCustomerid),
customer(Customerid), bannedcustomer(BannedCustomerid).

...
%rule r6
:- bankHarassToCustomer(Bankid, Customerid).

...

Rules in SBVR are translated as rules in ASP of the form Head :- Body, where Body is the antecedent that implies the consequent Head. The fragment of the ASP program is shown in Listing 7.2. Relations in SBVR become predicates in ASP, with related concepts as arguments referred by their ids e.g. bank @opensAccountFor customer becomes bankOpensAccountForCustomer(Bankid, Customerid). Attributes in SBVR become also predicates in ASP, with related concept and their value as arguments e.g. document $type_value_PAN becomes documentType(documentid, PAN). Statement-type rules without antecedents are translated as constraints, i.e., rules with an empty head, as :- Body. For example, rule r6 does not have antecedent thus it is translated into the ASP constraint :- bankHarassToCustomer(Bankid, Customerid). Disjunction in rule heads is used to build mutually exclusive paths, e.g., high and low risk categorization in rule r1.
Listing 7.3: Ground Facts

highriskcust (100).
lowriskcust (101).
salaried (102).
trusts (103).
pepofforeignorigin (104).
bank (105).
customer (106).
document (107).
bannedcustomer (108).
account (109).

% Attribute: compute IsBannedCustomer
bank(105, 106, 109, true). %modified from: bank(105, 106, 109, false).
%Attribute: isProvidedReasonForClosed
account(109, false ,1). %Modified from: account(109, true ,1).
account(109, false ,4). %Modified from: account(109, true ,4).
document(107, telephoneBill).
document(107, infoFromPublicDomain).
document(107, aadhaar).
document(107, registrationCertificate).
document(107, PAN).

%Additional facts
customerApproachesBank (106, 105).
bankClosedAccount (105, 109).

Listing 7.3 shows the facts or test data automatically generated (with some manual modification shown with the comment Modified from) for the ASP rules (Listing 7.2) using the newly implemented approach. Within this approach, the arguments of the ASP rules (concepts in SE rules) are transformed into facts with a specific identification number (start id from 100). Also, the Boolean attribute values in the facts are according to their specification in the rules, and not according to ideal scenario generation. For example, rule r5 has attribute computeIsBannedCustomer with the value true ($computeIsBannedCust_value_true), so in the auto generated ground facts, it is also true. However, with this specification, not all rules are satisfied, e.g., if rule r5 is true, then rule r4 will be false (not executed), as they are mutually exclusive rules. So, to generate the ideal scenarios (in which all the rules are executed and the constraints are satisfied), the developer should modify the facts accordingly. In Listing 7.3, three facts, one for the concept bank and two for the concept account, are modified for generating ideal scenarios. Also, it includes some atomic predicates which are not explained further in the model that need to be generated as facts (e.g., customerApproachesBank(106, 105)).

In general, the number of scenarios depends on the conditions and ground instances. To limit the number of scenarios, only essential conditions (e.g., customer
types - salaried, trusts and pepOfForeignOrigin) are modeled as disjunctions, while non-critical conditions such as documents submitted by a customer e.g. PAN, aadhaar, telephoneBill, etc. are modeled as normal clauses.

Based on the generated rules (Listing 7.2) and ground facts (Listing 7.3), three scenarios corresponding to two risk categories (lowRiskCust and highRiskCust) and three customer types (salaried, trusts and pepOfForeignOrigin) are constructed and are shown in Listing 7.4. As expected, three scenarios based on three customer types (highlighted in bold) are generated. Furthermore, the predicate bankOpensAccountForCustomer (highlighted in italic font) is present in all the scenarios which make sure that all the rules have been executed and constraints are satisfied.

Listing 7.4: Ideal Generated Scenarios

{documentType(107,PAN),documentType(107,aadhaar),
salariedHasRequiredDocument(102,108),
documentType(107,registrationCertificate),
documentType(107,telephoneBill),trustsHasRequiredDocument(103,107),
documentType(107,infoFromPublicDomain),
pepOfForeignOriginHasRequiredDocument(104,107),
bankComputeIsBannedCust(105,false),
accountIsProvidedReasonForClosed(109,true),accountUpdatePeriod(109,1),
customerIsLowRiskCust(106,101),
customerIsSalaried(106,102),customerSubmitsDocument(106,107),
bankVerifiesIdentityOfCustomer(105,106),
bankOpensAccountForCustomer(105,106),customerOwnsAccount(106,109),
bankUpdateDetailsOfCustomer(105,106),
bankPeriodicUpdationCheckOfCustomer(105,106)}

{...,customerIsHighRiskCust(106,100),
customerIspepOfForeignOrigin(106,104),
...,bankOpensAccountForCustomer(105,106),...}

{...,customerIsHighRiskCust(106,100),customerIsTrusts(106,103),
...,bankOpensAccountForCustomer(105,106),...}

7.2.1 Checking Model properties

As stated in [45], the authors have already checked the inconsistency for the SBVR model using the ASP. In this section, ASP is further explored to check two other crucial verification properties, i.e., cyclic dependency and tautology. The verification tasks are essential to determine the correctness of the rules and ultimately improve the quality of the model.
Cyclic Dependency

A cyclic dependency is formed when two or more SE rules have direct or indirect dependencies on each other. Cyclic dependencies are usually difficult to detect in complex and large models and are a common source of subtle bugs.

Listing 7.5: Additional Rule - Cyclic Dependency

rule rCD It is necessary that customer @approaches bank
    if customer not @owns account

To check the usability of ASP for verifying the cyclic dependency, an additional rule \( r_{CD} \) (shown in Listing 7.5) is formed. This rule along with rule \( r_4 \) and rule \( r_{15} \) (in Listing 7.1), will create cyclic dependency. The cyclic dependency in these rules can be understood in textual term as follows: rule \( r_4 \) (the bank opens account for the customer if the customer approaches the bank) depends on rule \( r_{CD} \) (the customer approaches bank if he/she does not own account), rule \( r_{CD} \) rely on rule \( r_{15} \) (customer owns account if bank opens account for the customer), and rule \( r_{15} \) further depends on rule \( r_4 \). Thus, a cycle is created with these rules.

The solver based on the ASP program (Fig. 7.1), rule \( r_{CD} \) and ideal ground facts gives an empty solution (i.e., scenarios are not generated) without any particular message. By further testing and analyzing, it can be concluded that cyclic dependency in the rules can cause either partial scenarios if the rules which create the cycle have multiple concepts in antecedent and one of them is false or will generate an empty solution. In the above shown case, rule \( r_4 \) has multiple antecedents, but with ideal facts, they all become true. Thus, the tool gives an empty solution.

It can be difficult to manually detect the cyclic dependency for a bigger model with a huge rule file. As a solution, the methodological suggestion has been proposed in which the consequent-antecedent table can be created from ASP rules, and this table can help the developer to find the closure for the root rule. If the closure of the root rule includes itself, then there must be the cyclic dependency in that rule. So, in this way, the cyclic dependency can be indicated, and the properties of the model can be verified.

Tautology

In the model, sometimes the constraints or rules are formulated accidentally in the way that these are always true with any given fact data. So, in this context, these constraints or rules are tautologies. Tautologies do not add any valuable information. Thus, they should be either removed or modified.
Listing 7.6: Additional Rule - Tautology

\begin{verbatim}
rule rT It is necessary that bank @closed account
   if account $isProvidedReasonForClosed_value_true
\end{verbatim}

To explore ASP for checking the tautology, the additional rule rT (shown in Listing 7.6) is created. This rule will make the constraint rule r6 (Listing 7.1) tautology, i.e., rule r6 will never be false. In textual terms, the constraint (r6) specifies that bank should not harass the customer, and rule r7 defines that bank harasses the customer if it closes the account (bank @closed account) and does not provide the reason for it (attribute account $isProvidedReasonForClosed_value_false). The attribute isProvidedReasonForClosed is given true in the facts which makes the antecedent bank @closed account true (rule rT) and account $isProvidedReasonForClosed_value_false false in rule r7. And if the attribute value isProvidedReasonForClosed is given false, then in rule r7, the antecedent bank @closed account is false and account $isProvidedReasonForClosed_value_false is true. So, due to the addition of rule rT, both the antecedents in the rule r7 cannot be true at the same time. Thus, rule r7 is false in any case, and constraint r6 is always true.

The solver based on the rules (Listing 7.1) along with rule rT constructs three ideal scenarios (Listing 7.4) and does not detect any tautologies, as the solver expects the given constraints to be true. However, it might be possible that the bank can harass the customer, i.e., the constraint r6 can be false. So, it is essential to detect and fix the tautology problem. To avoid tautology in the context of SBVR and ASP, the methodological suggestion is proposed in which the developer should make sure that the rules where the same attribute is mentioned must be independent of each other or not contradicting each other.

7.3 Negative Scenario Generation

The scenario generation using ASP usually gives four types of output, i.e., complete scenario (executes all rules), partial scenario, empty solution or error. However, it might be difficult for the developer to detect whether the partial scenario, empty solution or error is due to the incorrect rules or missing facts.

The negative scenarios could help to further test the model with another viewpoint on the SE rules, and one could understand it better with an example test case. In the test case, the facts for document PAN and aadhaar are excluded from the specified facts (Listing 7.3), so the tool generates two complete scenarios for trusts and
pepOfForeignOrigin customers (Listing 7.4) and one partial scenario for salaried customer (shown in Listing 7.7).

Listing 7.7: Partial Scenario for Salaried Customer
\[
\{ \text{documentType}(107, \text{registrationCertificate}), \\
\text{documentType}(107, \text{telephoneBill}), \text{trustsHasRequiredDocument}(103, 107), \\
\text{documentType}(107, \text{infoFromPublicDomain}), \\
\text{pepOfForeignOriginHasRequiredDocument}(104, 107), \\
\text{bankComputeIsBannedCust}(105, \text{false}), \\
\text{accountIsProvidedReasonForClosed}(109, \text{true}), \text{accountUpdatePeriod}(109, 1), \\
\text{accountUpdatePeriod}(109, 4), \text{customerIsLowRiskCust}(106, 101), \\
\text{customerIsSalaried}(106, 102) \}
\]

In the partial scenario, it is to be noted that not all rules are executed (e.g., the predicate \text{BankOpenAccountForCustomer} is absent). The partial scenario generation either could be because of faulty rules (e.g., cyclic dependency) or incorrect/missing facts. So, to further analyze the rules, a negative scenario generation approach is proposed in which all rules are negated. Thus, it gives negative traces of the rules which could help to check the model further. Listing 7.8 shows the negative scenario of salaried customer for the above described test case.

Listing 7.8: Negative Scenario for Salaried Customer
\[
\{ \text{notSalariedHasRequiredDocument}(102, 108), \\
\text{bankComputeIsBannedCust}(105, \text{false}), \\
\text{accountIsProvidedReasonForClosed}(109, \text{true}), \text{accountUpdatePeriod}(109, 1), \\
\text{accountUpdatePeriod}(109, 4), \text{customerIsLowRiskCust}(106, 101), \\
\text{customerIsSalaried}(106, 102), \text{notCustomerSubmitsDocument}(106, 107), \\
\text{notBankVerifiesIdentityOfCustomer}(105, 106), \\
\text{notBankOpensAccountForCustomer}(105, 106), \text{notCustomerOwnsAccount}(106, 109), \\
\text{notBankUpdateDetailsOfCustomer}(105, 106), \\
\text{notBankPeriodicUpdateCheckOfCustomer}(105, 106) \}
\]

By analyzing the scenario (Listing 7.8), one can conclude that salaried customer does not have the required documents to open an account (the predicate \text{notSalariedHasRequiredDocument}). Therefore, the partial scenario (Listing 7.7) is because of the missing facts and not because of the incorrect rules. Thus, the negative scenarios give more perspective and broader view on the rules for verification and validation of the SVBR model.

To construct the negative scenarios, all the ASP rules are negated using the ASP negation approach (explain in Sect. 2.5). As mentioned, two types of negation are possible in ASP: (a) \text{negation by failure} - fail to get facts from the knowledge base but might/might not exist in the universe, and (b) \text{true negation} - it is certain that the facts don’t exist in the universe. So, two types of negative ASP programs are possible to generate different negative scenarios. In negation by failure, the prefix “not”
in the names of the predicates is added (not “predicateName”), and the predicates of attributes (the predicates which are translated from the SE rule attributes) and relations which are not further defined (e.g., customerApproachesBank) should be negated using “not” keyword (not “predicateName”). In true negation, predicates are negated with “-” (- “predicateName”). The proposed guidelines for the negation of the ASP program for negative scenario generation is as follows:

- In both the cases, the negation starts from predicates of attributes and will negate further according to the linked predicates in the ASP program. The rules with the disjunction and their definition should not be negated (e.g., rule r1, r2 in Listing 7.2).

- The negative predicates of attributes and relations in the ASP program are negated differently. So, the abstract ASP rules are formulated for both types of predicates, and their negation with negation by failure and true negation are also shown in Table 7.1.

<table>
<thead>
<tr>
<th>Predicates (Y, Z) of Relations</th>
<th>Predicates (A, B) of Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP rules</td>
<td>Predicates (A, B) of Attributes</td>
</tr>
<tr>
<td>X :- not Y</td>
<td>X :- not A</td>
</tr>
<tr>
<td>X :- not Z</td>
<td>X :- -B*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Negation by failure</th>
<th>Predicates (A, B) of Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>notX :- not not Y</td>
<td>notX :- not Z</td>
</tr>
<tr>
<td>notX :- -notZ</td>
<td>notX :- -A</td>
</tr>
<tr>
<td>notX :- not -B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True negation</th>
<th>Predicates (A, B) of Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-X :- -not -Y</td>
<td>-X :- Z</td>
</tr>
<tr>
<td>-X :- -not -A</td>
<td>-X :- -B</td>
</tr>
<tr>
<td>Table 7.1: Negation of Negative Predicates of Relations and Attributes</td>
<td></td>
</tr>
</tbody>
</table>

- Predicates linked through “AND” should be transformed into “OR” and vice versa. The abstract example ASP rules in which predicates are linked with AND (X = A AND B) and OR (Y = C OR D), and their negation are shown in Table 7.2.

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP rules</td>
<td></td>
</tr>
<tr>
<td>X :- A, B</td>
<td>Y :- C</td>
</tr>
<tr>
<td></td>
<td>Y :- D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Negation by failure</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>notX :- not A</td>
<td></td>
</tr>
<tr>
<td>notX :- not B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>notY :- not C, not D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True negation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-X :- -A</td>
<td></td>
</tr>
<tr>
<td>-X :- -B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-Y :- -C, -D</td>
</tr>
<tr>
<td>Table 7.2: Negation of Predicates linked with AND and OR</td>
<td></td>
</tr>
</tbody>
</table>

Using these guidelines, the ASP program (Listing 7.2) is negated. The fragments of negative ASP programs with negation by failure and true negation are shown in Listing 7.9 and 7.10, respectively. Here, the constraints are negated in such a way that the meaning of constraint should not be changed, i.e., they are negated
two times (e.g., rule r6). To construct the ideal negative scenarios, the negation of the facts is also essential. In the case of negation by failure, the given facts negated either by specifying the unexpected/opposite attribute values in the facts (e.g., bank(105,106,112,false) becomes bank(105,106,112,true)) or by removing the facts which has only the attribute values (e.g., document(107,PAN)). In the case of true negation, the facts are negated with “-” (e.g., -document(107,PAN)).

Listing 7.9: Negative ASP Program with Negation by Failure

\%
rule r1
\% customerIsHighRiskCust(Customerid, HighRiskCustid) v
\% customerIsLowRiskCust(Customerid, LowRiskCustid): − customer(Customerid),
\% highriskcust(HighRiskCustid), lowriskcust(LowRiskCustid).
\%

\%
rule r4
notBankOpensAccountForCustomer(Bankid, Customerid) :-
not customerApproachesBank(Customerid, Bankid),
notBankVerifiesIdentityOfCustomer(Bankid, Customerid),
not -customerIsBannedCustomer(Customerid, BannedCustomerid),
customer(Customerid), bannedcustomer(BannedCustomerid).

\%
rule r6:
:- not notBankHarassToCustomer(Bankid, Customerid).
\%

Listing 7.10: Negative ASP Program with True Negation

\%
rule r1
\% customerIsHighRiskCust(Customerid, HighRiskCustid) v
\% customerIsLowRiskCust(Customerid, LowRiskCustid): − customer(Customerid),
\% highriskcust(HighRiskCustid), lowriskcust(LowRiskCustid).
\%

\%
rule r4
− bankOpensAccountForCustomer(Bankid, Customerid) :-
- customerApproachesBank(Customerid, Bankid),
- bankVerifiesIdentityOfCustomer(Bankid, Customerid),
not −customerIsBannedCustomer(Customerid, BannedCustomerid),
customer(Customerid), bannedcustomer(BannedCustomerid).

\%
rule r6:
:- not -bankHarassToCustomer(Bankid, Customerid).
\%

The solver based on the generated negative ASP program and ideal negative ground facts generates three scenarios for each negation type corresponding to two risk categories (\textit{lowriskCust} and \textit{highRiskCust}) and three customer types (\textit{salaried}, \textit{trusts} and \textit{pepOfForeignOrigin}). The negative scenarios generated using negation by failure are shown in Listing 7.11, and using true negation are shown in Listing 7.12. As expected, three scenarios (for each negation type) based on three customer types (highlighted in bold) are generated. Furthermore, the presence of the
predicates not\text{bankOpensAccountForCustomer} or \text{−bankOpensAccountForCustomer} (highlighted in italic font) makes sure that the complete negative scenarios are generated.

Listing 7.11: Negative Scenario Generation with Negation by Failure
\[
\{ \text{notSalariedHasRequiredDocument} (102,108), \\
\text{notTrustsHasRequiredDocument} (103,107), \\
\text{notPepOfForeignOriginHasRequiredDocument} (104,107), \\
\text{bankComptesIsBannedCust} (105,\text{false}), \\
\text{accountIsProvidedReasonForClosed} (109,\text{true}), \\
\text{accountUpdatePeriod} (109,1), \\
\text{customerIsSalaried} (106,102), \\
\text{customerIsLowRiskCust} (106,101), \\
\text{customerIsSalaried} (106,102), \\
\text{notCustomerSubmitsDocument} (106,107), \\
\text{notBankVerifiesIdentityOfCustomer} (105,106), \\
\text{notBankOpensAccountForCustomer} (105,106), \\
\text{notCustomerOwnsAccount} (106,109), \\
\text{notBankUpdateDetailsOfCustomer} (105,106), \\
\text{notBankPeriodicUpdateCheckOfCustomer} (105,106) \}
\]

{\ldots, \text{customerIsHighRiskCust} (106,100), \\
\text{customerIsPepOfForeignOrigin} (106,104), \\
\ldots, \text{notBankOpensAccountForCustomer} (105,106), \ldots}\}

{\ldots, \text{customerIsHighRiskCust} (106,100), \text{customerIsTrusts} (106,103), \\
\ldots, \text{notBankOpensAccountForCustomer} (105,106), \ldots}\}

Listing 7.12: Negative Scenario Generation with True Negation
\[
\{ \text{−documentType} (107,\text{PAN}), \text{−documentType} (107,\text{aadhaar}), \\
\text{−salariedHasRequiredDocument} (102,108), \\
\text{−documentType} (107,\text{registrationCertificate}), \\
\text{−documentType} (107,\text{telephoneBill}), \\
\text{−trustsHasRequiredDocument} (103,107), \\
\text{−documentType} (107,\text{infoFromPublicDomain}), \\
\text{−pepOfForeignOriginHasRequiredDocument} (104,107), \\
\text{−bankComptesIsBannedCust} (105,\text{true}), \\
\text{−accountIsProvidedReasonForClosed} (109,\text{true}), \text{−accountUpdatePeriod} (109,1), \\
\text{−accountUpdatePeriod} (109,4), \text{customerIsLowRiskCust} (106,101), \\
\text{customerIsSalaried} (106,102), \text{−customerSubmitsDocument} (106,107), \\
\text{−bankVerifiesIdentityOfCustomer} (105,106), \\
\text{−bankOpensAccountForCustomer} (105,106), \text{−customerOwnsAccount} (106,109), \\
\text{−bankUpdateDetailsOfCustomer} (105,106), \\
\text{−bankPeriodicUpdateCheckOfCustomer} (105,106) \}
\]

{\ldots, \text{−accountUpdatePeriod} (109,4), \\
\text{customerIsHighRiskCust} (106,100), \text{customerIsPepOfForeignOrigin} (106,104), \\
\text{−bankOpensAccountForCustomer} (105,106), \ldots}\}

{\ldots, \text{−accountUpdatePeriod} (109,4), \\
\text{customerIsHighRiskCust} (106,100), \text{customerIsTrusts} (106,103), \\
\text{−bankOpensAccountForCustomer} (105,106), \ldots}\}
7.4 Quality Assurance

Model verification and validation are performed using a set of created test cases with test data for ASP program provided as ground facts and checking generated scenarios against expected results. To ensure the quality of the negative ASP programs and to validate the SBVR model, the test cases are formulated to construct different scenarios.

Test Case 1: Modification in Document Facts

Ideally, all documents (PAN or aadhar, registrationCertificate or telephoneBill, infoFromPublicDomain) are given as facts which satisfy the requirement for the identity verification of the different type of customers (salaried, trusts, pepOfForeignOrigin). In this test case, document infoFromPublicDomain is negated with true negation, document PAN exists and all the other documents are absent in the ground facts (Listing 7.3).

According to this test case, the facts include the document (PAN) to verify salaried customer, the documents to verify trusts customer are not mentioned i.e., it might exist but is not submitted to the bank (negation by failure) or does not exist at all (true negation), and document (infoFromPublicDomain) to verify pepOfForeignOrigin customer is negated i.e., the document does not exist at all (true negation). The results of the scenario generation with positive and negative ASP programs are described in Table 7.3.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Salaried</th>
<th>Trusts</th>
<th>pepOfForeignOrigin</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Positive) ASP program</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Negative ASP program (negation by failure)</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Negative ASP program (true negation)</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 7.3: Test case 1: Generated Scenarios with Positive and Negative ASP Programs

The table can be understood as follows: (a) with the positive ASP program, one complete scenario is generated for salaried customer as document PAN is given (b) with the negative ASP program (negation by failure), two negative scenarios are generated for customer type trusts and pepOfForeignOrigin, as the document (infoFromPublicDomain) for pepOfForeignOrigin customer is negated and no document is provided for trusts customer (c) with negative ASP program (true negation), one negative scenario is generated for pepOfForeignOrigin customer, as only for this customer the required document is negated.
Test Case 2: Combination of Two Boolean Attribute Values

Here, the two Boolean attribute IsBannedCust and IsProvidedReasonForClosed values (true or false) and their combination are considered in the facts. The ground facts (Listing 7.3) are modified according to the test data shown in Listing 7.13.

Listing 7.13: Modified Ground Facts

\[
\begin{align*}
\text{bank}(105,106,112, X). \\
\text{account}(112,Y,1).
\end{align*}
\]

\(X = \text{values of attribute IsBannedCust}\)

\(Y = \text{value of attribute IsProvidedReasonForClosed}\)

The two Boolean attributes give four combinations of values, i.e., four different test data are generated. The scenario generation with these different test data and positive and negative ASP programs are shown in Table 7.4.

<table>
<thead>
<tr>
<th>(X: \text{computeIsBannedCust})</th>
<th>(Y: \text{IsProvidedReasonForClosed})</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{(Positive) ASP program})</td>
<td>error</td>
<td>partial scenarios</td>
<td>error</td>
<td>complete scenarios</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Negative ASP program})</td>
<td>error</td>
<td>complete scenarios (negative)</td>
<td>error</td>
<td>complete scenarios (negative)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{Negative ASP program})</td>
<td>error</td>
<td>complete scenarios (negative)</td>
<td>error</td>
<td>complete scenarios (negative)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.4: Test case 2: Generated Scenarios using Different Boolean Attribute Values with Positive and Negative ASP Programs

Attribute IsProvidedReasonForClosed is associated with constraint \(r6\). So, in the case of the positive and negative ASP program, the false value of attribute IsProvidedReasonForClosed violates constraint \(r6\). For this constraint violation, the ASP solver gives an error (expressed with error in Table 7.4).

The true value of attribute IsProvidedReasonForClosed combined with the false value of attribute computeIsBannedCust is the ideal case for the positive ASP program and therefore generates three complete scenarios. But, if both the attribute values are true, then in the case of the positive ASP program, rule \(r4\) is not satisfied, and the three partial scenarios are generated.

The true value of attribute IsProvidedReasonForClosed along with the true or false value of attribute computeIsBannedCust generates three negative (complete) scenarios for the negative ASP programs. The negation of rule \(r4\) in the textual term is: the bank does not open an account if the customer does not approach the bank or the customer does not verify the identity or the customer is a banned
customer. Thus, the customer is a banned customer is not the only entity not to open the account. So, in both the situation, i.e., for the true or false value of attribute computeIsBannedCust, all rules are satisfied and complete negative scenarios are constructed.

The execution of all the test cases illustrate that the desired results are generated. The positive and negative scenarios are precisely constructed as expected. Thus, the developed negative ASP programs are assumed to be correct. Furthermore, it also ensures the quality of all ASP programs and validate the SBVR model.
Chapter 8

Conclusion and Future Work

In model-driven engineering (MDE), models play a central role in software development processes, and model quality assurance techniques like validation and verification become indispensable in order to avoid flaws in the early stages of the software design process. This thesis summarizes the contributions of the author in the field of validation and verification of UML/OCL behavioral models. The approaches proposed in this thesis strengthen our filmstrip model validation technique and make it more developer-friendly.

8.1 Conclusion

The first approach presented in this thesis focuses on the development of postconditions together with frame conditions for a UML and OCL model, which are essential for the model validation process. A tabular PCDL approach is proposed that semi-automatically generates comprehensive OCL postconditions. The approach introduces PCDL elements that are based on a new formal distinction between deleted, sustained and added objects to cover all aspects of operation execution, and that simplifies the postcondition specification. The method for the automatic transformation of the PCDL elements into OCL postconditions is also presented. This approach relieves the burden of writing OCL post- and frame conditions manually from the developer. Also, a transformation chain is introduced starting from a user-defined model yielding a test case model, using our filmstripping approach and model validator for validation of comprehensive postconditions. Improvement steps in an iterative process are available that guide the developer in generating precise comprehensive postconditions.

The second contribution extends the tool USE by implementing valuable functionalities, i.e., filmstrip model configurations and filmstrip templates for our filmstrip
model validation approach. With the new implementation, developers can specify the configuration for application model elements and operations, and accordingly, the configuration for filmstrip model elements is automatically calculated and the template can be generated. The filmstrip template reduces the work of our model validator by identifying recurring model parts, and thereby, the model validation time is reduced.

The third contribution focuses on the generation of OCL invariants, which are essential for constructing a scenario for model validation. We proposed a transformation approach for automatically generating the invariant from a test case (TC) Schema, which is a diagrammatic representation of a scenario constructed by the developer. The open and closed options are offered for each snapshot in a TC schema to help the developer efficiently generate expected scenarios. We showed scenario generation using the model validator which constructed valid filmstrip object diagrams based on the generated OCL invariants and showed the successful transformation of an OCL invariant from a TC schema.

The fourth contribution proposes a catalogue of different scenario patterns which guides and advises the developer for constructing a behavioral scenario. The catalogue can be applied in a UML and OCL model for developing operation call sequences that check dynamic model properties. The nature and purpose of all scenario patterns are described in order to ease the process of applying the patterns for the considered model. We have explained and demonstrated the complete catalogue by different UML and OCL models through scenario generation.

The final contribution summarizes the additional work of the author during her industrial internship. The author explored the usability of the Answer Set Programming (ASP) approach for checking the Semantics of Business Vocabulary and Business Rules (SBVR) model.

8.2 Future Work

Future work can be explored in various directions. The first would be the implementation of the proposed PCDL approach in the tool USE. The development of the user interface could support more analysis options and could ease the handling of unaffected classes for which an operation does not make changes. The applicability and usability of the approach would increase by the new automatic generation using USE, as the modelers would no longer need to formulate OCL post- and frame conditions manually. The authors have used small examples to explain and show the concept.
So, the future work would also focus on using the larger case studies with complex models and scenarios to determine the feasibility of the approach.

Systematically constructing operations of the classes of a UML/OCL model could be another direction of the future work. In this approach, for each attribute and role name, operations should be constructed. For attributes, the operation set ‘AttributeName’ should be constructed, which will manipulate the attribute value. And for each role name, an operation link ‘RoleName’ and an operation unlink ‘RoleName’ should be constructed, which will add and remove a specific link, respectively. As the behavior of those operation are known, this approach can be combined with the PCDL approach to systematically generate the pre- and postconditions of the operations. This approach would help the novel users to adequately develop the UML/OCL behavioral model.

The future work would also concentrate in extending the tool USE by implementing the proposed automatic OCL invariant transformation approach from a TC schema. It would enhance and make our approach more user-friendly. Also, the object attribute specification in a TC schema should help a developer to express the scenario effectively and this could also be considered in the transformation as an future work. Also, larger case studies would help in checking the applicability.

A possible future work would be also to apply the proposed catalogue to other approaches for checking dynamic model properties and to include more patterns. Furthermore, support could be provided 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. Also, the future work would try to provide support for determining the patterns that can be applied in the model under consideration.
Bibliography


[75] Li Tan, Zongyuan Yang, and Jinkui Xie. OCL constraints automatic generation for UML class diagram. In IEEE Int. Conf. on Software Engineering and Service Sciences (ICSESS’2010), pages 392–395, 2010.


