DISSERTATION

A SCENARIO-BASED TECHNIQUE TO ANALYZE UML DESIGN CLASS MODELS

Submitted by

Lijun Yu

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2014

Doctoral Committee:

Advisor: Robert B. France

Indrakshi Ray
Sudipto Ghosh
Yashwant Malaiya
Dan Turk

Copyright by Lijun Yu 2014

All Rights Reserved

ABSTRACT

A SCENARIO-BASED TECHNIQUE TO ANALYZE UML DESIGN CLASS MODELS

Identifying and resolving design problems in the early design phases can help reduce the number of design errors in implementations. In this dissertation a tool-supported lightweight static analysis technique is proposed to rigorously analyze UML design class models that include operations specified using the Object Constraint Language (OCL). A UML design class model is analyzed against a given set of scenarios that describe desired or undesired behaviors. The technique can leverage existing class model analysis tools such as USE and OCLE. The analysis technique is lightweight in that it analyzes functionality specified in a UML design class model within the scope of a given set of scenarios. It is static because it does not require that the UML design class model be executable.

The technique is used to (1) transform a UML design class model to a snapshot transition model that captures valid state transitions, (2) transform given scenarios to snapshot transitions and (3) determine if the snapshot transitions conform or not to the snapshot transition model. A design inconsistency exists if snapshot transitions that represent desired behaviors do not conform to the snapshot transition model, or if snapshot transitions representing undesired behaviors conform to the snapshot transition model.

A Scenario-based UML Design Analysis tool was developed using Kermeta and the Eclipse Modeling Framework. The tool can be used to transform an Ecore design class model to a snapshot transition model and transform scenarios to snapshot transitions. The tool is integrated with the USE analysis tool.

We used the Scenario-based UML Design Analysis technique to analyze two design class models: a Train Management System model and a Generalized Spatio-Temporal RBAC model. The two demonstration case studies show how the technique can be used to analyze the inconsistencies between UML design class models and scenarios.

We performed a pilot study to evaluate the effectiveness of the Scenario-based UML Design Analysis technique. In the pilot study the technique uncovered at least as many design inconsistencies as manual inspection techniques uncovered, and the technique did not uncover false inconsistencies. The pilot study provides some evidence that the Scenario-based UML Design Analysis technique is effective.

The dissertation also proposes two scenario generation techniques. These techniques can be used to ease the manual effort needed to produce scenarios. The scenario generation techniques can be used to automatically generate a family of scenarios that conform to specified scenario generation criteria.

ACKNOWLEDGEMENTS

I am thankful to a lot of people who are helpful to me in my life and Ph.D. study. Without their patient help and support, I would not have been able to complete my dissertation.

First of all, I owe special thank to my advisor, Dr. Robert B. France for his guidance, inspiration and patience. I am very grateful to his dedication and flexibility when I have to work part-time and later remotely on my research. I would like to thank Dr. Indrakshi Ray for always supporting and advising me. I'd like to thank my research committee members Dr. Sudipto Ghosh, Dr. Yashwant Malaiya and Dr. Dan Turk for their advice. I'd like to thank Dr. Phillip Chapman from the Department of Statistics and Dr. Tao Yue from Simula Research Laboratory for advising me in the pilot study.

I would like to thank the National Science Foundation for sponsoring research project "SHF: Small: Scenario-Based Validation of Design Models" under grant #1018711. I'd like to thank Wuliang Sun and Kayle Hoehn for their excellent work in implementing the tool and resolving major issues in implementation. I'd like to thank Mustafa Al-Lail, Mohammed Al-refai and Sai Mandalaparty for their help in the pilot study.

I would like to thank the graduate committee and Dr. Dale Grit who accepted me to the Ph.D. program and granted me assistantship. I'd like to thank Carol Calliham for helping me apply to the program. I would like to thank Sharon Van Gorder and all other people in the department who have helped me in my study.

Finally, I am grateful to my wife, Li Huang, for her love and support. I thank her for accompanying and encouraging me during my hard time. I'd like to thank our daughter Annie and Emily for giving us a lot of happiness. I'd like to thank my parents for their unselfish love, and my brother and sister for their love and support.

DEDICATION

This dissertation is dedicated to my wife.

TABLE OF CONTENTS

CHAPTER 1 1
INTRODUCTION 1
1.1 Problem Statement *1*
1.2 Outline of Solution *4*
1.3 Scope of Research *6*
1.4 Dissertation Organization *6*
CHAPTER 2 8
BACKGROUND 8
2.1 The Unified Modeling Language *8*
2.2 The Meta-Object Facility *13*
2.3 The Eclipse Modeling Framework and Ecore *14*
2.4 The Kermeta Metamodeling Language *15*
CHAPTER 3 17
RELATED WORK 17
3.1 Formal analysis techniques *17*
3.2 UML animation and testing *20*
3.3 USE and OCLE *22*
3.4 UML test input and scenario generation *22*
CHAPTER 4 25
SCENARIO-BASED UML DESIGN ANALYSIS TECHNIQUE 25
4.1 A simple Role-Based Access Control example *26*
4.2 Generating the Snapshot Transition Model *29*
4.3 Generating Scenario Snapshot Transitions *40*
4.4 Checking consistency in USE *41*
4.5 Algorithm complexity analysis *42*
CHAPTER 5 45
IMPLEMENTATION 45
5.1 Tool architecture *45*
5.2 The STM Generator and STM Invariant Generator *48*
5.3 The Scenario Generator *56*
5.4 USE consistency check *61*
CHAPTER 6 63
DEMONSTRATION CASE STUDIES 63
6.1 The Train Management System model *63*
6.2 The Generalized Spatio-Temporal RBAC model *74*
6.3 Conclusion *91*
CHAPTER 7 92
PILOT STUDY 92
7.1. Experiment planning *92*
7.2. Experiment results and analysis *94*
7.3. Conclusion and discussions *97*
CHAPTER 8 98
GENERATING SCENARIOS USING JAL OPERATION DEFINITIONS 98

8.1 The scenario generation technique *98*
8.2 An hierarchical RBAC example *103*
8.3. Analyze HRBAC constraints *108*
CHAPTER 9 112
GENERATING SCENARIOS USING OCL OPERATION DEFINITIONS 112
9.1 The Location-aware Role-Based Access Control model *112*
9.2 The scenario generation technique *115*
CHAPTER 10 128
CONCLUSIONS AND FUTURE WORK 128
10.1 Contributions *128*
10.2 Discussions of research questions *129*
10.3 Future work *133*
REFERENCES 134

LIST OF TABLES

Table 7.1. Formulation of the experiment objective 93
Table 7.2. TMS experiment results 95
Table 7.3. GSTRBAC experiment results 95
Table 10.1. Time analysis of model transformation 131

LIST OF FIGURES

Figure 1.1: Scenario-based UML design analysis technique 4
Figure 2.1. A car inventory application design class diagram.... 9
Figure 2.2. UML sequence diagram 12
Figure 2.3. UML four-layer metamodeling architecture 13
Figure 2.4. Partial Ecore metamodel 14
Figure 4.1: Scenario-based UML design analysis technique 25
Figure. 4.2. Partial RBAC design class model 27
Figure 4.3. Partial RBAC class model and its snapshot transition model 31
Figure 4.4. Assign Accountant role snapshot transition.... 41
Figure 5.1. Overview of the Scenario-based UML Design Analysis tool 46
Figure 5.2. RBAC Ecore design class diagram (diagram view).... 48
Figure 5.3. RBAC Ecore design class diagram (tree view) 49
Figure 5.4. RBAC Ecore snapshot transition diagram 50
Figure 5.5. Snapshot transition model generation algorithm 51
Figure 5.6. OCL operation specification transformation main algorithm 54
Figure 5.7. USE snapshot transition model 55
Figure 5.8. Explicit specification of an RBAC scenario 57
Figure 5.9. Metalmodel of the action specification language 58
Figure 5.10. Action specification of an RBAC scenario 59
Figure 5.11. USE snapshot transitions 61
Figure 5.12. USE consistency checking 62
Figure 6.1. TMS design class diagram 65
Figure 6.2. TMS snapshot 1.1 68
Figure 6.3. TMS snapshot 1.2 69
Figure 6.4. TMS snapshot 1.3 70
Figure 6.5. TMS snapshot 2.1 71
Figure 6.6. TMS snapshot 2.2 72
Figure 6.7. TMS snapshot 2.3 73
Figure 6.8. GSTRBAC design class diagram – main view 75
Figure 6.9. GSTRBAC design class diagram – SOD view 76
Figure 6.10. GSTRBAC snapshot 1.1 80
Figure 6.11. GSTRBAC snapshot 1.2 81
Figure 6.12. GSTRBAC snapshot 1.3 82
Figure 6.13. GSTRBAC snapshot 2.1 83
Figure 6.14. GSTRBAC snapshot 2.2 84
Figure 6.15. GSTRBAC snapshot 2.3 85
Figure 6.16. GSTRBAC snapshot 2.4 86
Figure 6.17. GSTRBAC snapshot 2.5 87
Figure 6.18. GSTRBAC inconsistencies in snapshot 2.5 88
Figure 6.19. GSTRBAC snapshot 3.1 89
Figure 6.20. GSTRBAC snapshot 3.2 90
Figure 6.21. GSTRBAC snapshot 3.3 91
Figure 7.1. Histogram of experiment results 96

Figure 8.1. Scenario generation algorithm *102*
Figure 8.2. Hierarchical RBAC design class model *103*
Figure 9.1. The LRBAC UML design class diagram *113*
Figure 9.2. Generating transition sequences *116*
Figure 9.3. The analysis operation sequence *118*
Figure 9.4. Snapshots before User_UpdateLocation_Transition *126*
Figure 9.5. Snapshots after User_UpdateLocation_Transition *127*

Chapter 1

Introduction

1.1 Problem Statement

The Unified Modeling Language (UML) is the de-facto standard object-oriented modeling language [UML]. UML class models are often used by developers to describe object-oriented software designs [Whittle03]. Software design is a creative and labor-intensive process and thus there are opportunities for introducing errors into UML design class models. Design errors should be identified and resolved as early as possible because these errors may be much more costly to fix in later software development phases [Blum92] [Boehm81]. There is a need for analysis tools and techniques that uncover errors in UML design class models.

A UML design class model can be used to describe two aspects of a software design: structure and functionality. The structural aspect of a software design is described in terms of classes, relationships between classes, and class invariants. Functionality is described using operation specifications. Operation specifications and class invariants can be expressed in the Object Constraint Language (OCL) [OCL].

Existing UML structural analysis tools such as OCLE [OCLE] and USE [USE] can be used to check if an object configuration, called a *snapshot*, conforms to a class model. However, OCLE does not provide any support for analyzing operation specifications and USE can only be used to analyze operation specifications by simulating behavior of operations in an interactive mode.

Formal analysis tools such as the Alloy [Alloy] can be used to find counter-examples that violate certain constraints within the scope of a limited number of objects. Formal model checking tools [Clark99] do exhaustive search in a constrained state space to check whether a given property is satisfied or not. However, to analyze UML design class models using Alloy or model checking tools, we need to transform UML design models to the Alloy or the notation of model checking tools. The verifier should be familiar with the Alloy language and the notation of a model checking tool. Furthermore, one has to prove the correctness of the transformations in order to trust the analysis results. This can be difficult in practice.

Formal theorem proving tools such as Isabelle can be used to formally reason about modeled properties in an interactive manner [Brucker08]. However, to analyze UML design class models using theorem proving tools the verifier must be familiar with formal notations and proof strategies used by the tools.

Analyzing all behaviors specified in UML design class models can be expensive. Sometimes all that is desired is to check a subset of behaviors. This dissertation proposes a lightweight analysis technique called Scenario-based UML Design Analysis that allows modelers to analyze UML design class models against functionality described by a set of scenarios, where a scenario is an execution trace that consists of an initial system state and a sequence of operation calls and system states after each operation call.

The research aims at answering the following questions:

1. How can a scenario be checked against a UML design class model? Some scenarios describe desired functionalities, others describe undesired functionalities. The technique should be able to check that the UML design class model is consistent with the former and inconsistent with the latter.

2. Can existing structural analysis tools such as USE be leveraged to support scenario-based analysis of UML design class models? Existing structural analysis tools can be used to check the consistency between a UML class model and a snapshot. It will save a lot of effort if we can leverage existing UML analysis tools in building support for scenario-based analysis.
3. How effective is the Scenario-based UML Design Analysis technique in terms of the number of design inconsistencies that can be uncovered? The technique should be at least as effective as human beings in identifying design inconsistencies.

This research work starts to explore answers to the following two questions, but more work is needed:

4. Can useful scenarios be automatically generated? Scenario creation is time consuming. An automatic scenario generation technique can ease the scenario creation task.
5. Can the Scenario-based UML Design Analysis technique be scaled to analyze large industrial models?

The technique is lightweight because it aims to uncover design errors within the scope of a set of scenarios. It does not attempt to explore all possible scenarios covered by a UML design. The technique is static because it does not require that the UML design class model be executable. The technique provides a less expensive and less exhaustive alternative to more heavyweight analysis techniques. It is less expensive in that modelers need only be familiar with UML and do not have to put effort into learning and using sophisticated formal languages and proof techniques.

1.2 Outline of Solution

An overview of the Scenario-based UML Design Analysis technique is shown in Fig. 1.1. There are two roles involved in the Scenario-based UML Design Analysis process: designer and verifier. The designer creates a UML design class model that includes OCL operation specifications, and *operation scope specifications* which specify the set of classes, attributes and links that are changed by each operation. The verifier creates a set of scenarios that will be used to analyze the UML design.

In this dissertation a *scenario* describes an execution trace. It consists of an initial system state (*snapshot*) and a sequence of operation calls. Formally, a *scenario* is a sequence $\langle S_0, OP_1, S_1, \ldots, OP_n, S_n \rangle$ where $OP_1, OP_2, \ldots, OP_n$ is a sequence of operation calls with actual parameters, S_i is the system state before operation OP_i is executed and S_{i+1} is system state after the operation OP_i is executed.

A scenario can be legal or illegal: a *legal scenario* describes functionality that is desired, while an *illegal scenario* describes functionality that is not desired.

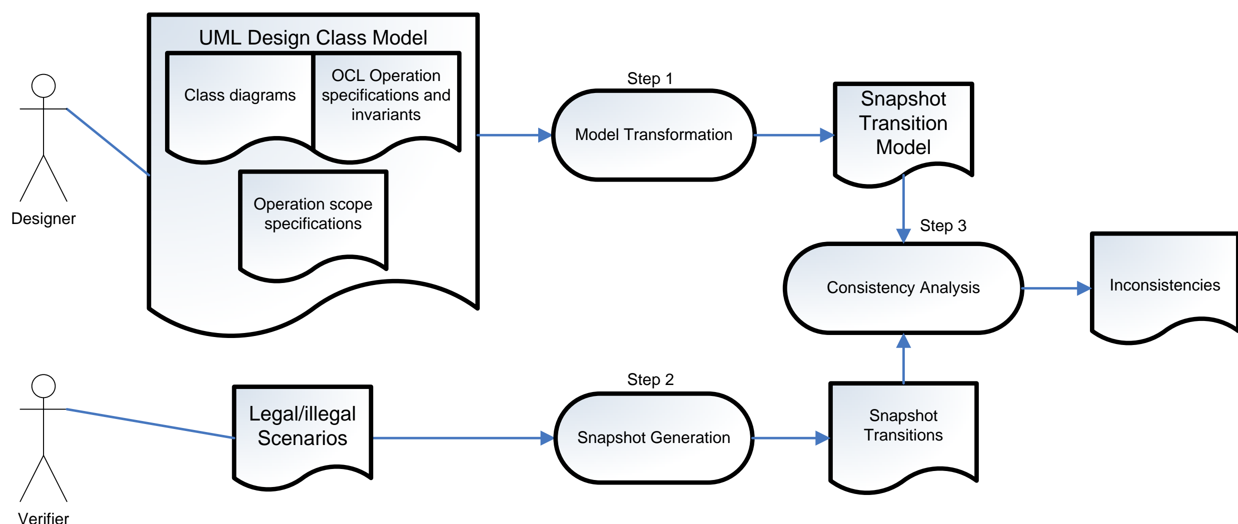


Figure 1.1: Scenario-based UML design analysis technique

The technique consists of three major automated steps.

- In the first step, the UML design class model is automatically transformed to a s*napshot transition model.* A *snapshot transition model* is a UML class model that specifies valid *snapshot transitions*, that is, all valid changes to object configurations (snapshots) triggered by the execution of operations. A *snapshot transition* describes the effects of an operation invocation on a system state. A *snapshot transition* consists of (1) the name and parameter values of the operation that triggers the transition, (2) a *before-snapshot* describing the state of the system before the operation is executed, and (3) an *after-snapshot* describing the state of the system after the operation has been executed.
- In the second step, scenarios created by a verifier are used to generate a sequence of snapshot transitions that describe desired or undesired functionality from the perspective of the verifier. The verifier marks scenarios as legal or illegal.
- In the third step, the snapshot transitions produced in the second step are checked against the snapshot transition model to determine whether the snapshot transitions are consistent with the snapshot transition model. This check can be done by the UML structural analysis tool, USE. The output of the scenario-based UML design analysis technique is a set of inconsistencies. These inconsistencies are reported in the form of class invariant violations. An inconsistency between the UML design class model and scenarios implies defects in the UML design class model or defects in the scenarios.

In addition to the Scenario-based UML Design Analysis technique, this dissertation discusses two scenario generation techniques that automatically generate scenarios from operation invocation patterns and operation definitions. Each pattern describes all possible operation sequences and operation definitions describe effects of operations in the scenarios.

1.3 Scope of Research

The Scenario-based UML Design Analysis technique assumes that operations are invoked sequentially and thus it cannot be used to analyze parallel and concurrent behaviors. The analysis of such behaviors is not in the scope of this research.

The analysis technique is a consistency checking technique. It is up to the verifier and developer to determine the source of inconsistencies. It is also up to the verifier and developer to change the UML design or scenarios based on the inconsistencies found by the technique.

The Scenario-based UML Design Analysis technique cannot be used to determine whether a scenario is effective or not to identify defects in the UML design. The verifier is responsible for creating scenarios of interest to analyze the design.

The scenario generation techniques cannot be used to determine whether enough scenarios have been generated. It is a hard problem to generate enough scenarios for scenario-based analysis, because the technique has to select a small number of scenarios that can effectively uncover design defects from a huge state space of all possible scenarios of the UML design class model. The scenario generation techniques discussed in this dissertation are an initial attempt to automating the generation of scenarios.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows:

- Chapter 2 presents the background needed to understand the analysis technique.
- Chapter 3 surveys related work in analysis and testing of UML design models.
- Chapter 4 describes the Scenario-based UML Design Analysis technique.

- Chapter 5 describes the implementation of the Scenario-based UML Design Analysis tool.
- Chapter 6 describes two demonstration case studies of the Scenario-based UML Design Analysis technique.
- Chapter 7 discusses pilot study for evaluating the technique.
- Chapter 8 and 9 discuss two scenario generation techniques.
- Chapter 10 concludes the dissertation and discusses future work.

Chapter 2

Background

This chapter provides background information needed to understand the research presented in this dissertation. Section 2.1 gives an overview of UML design class diagrams, the Object Constraints Language and UML sequence diagrams. Section 2.2 describes the Meta-Object Facility. Section 2.3 describes the Eclipse Modeling Framework and the Ecore metamodel. Section 2.4 discusses the Kermeta model transformation language.

2.1 The Unified Modeling Language

The Unified Modeling Language (UML) is the de-facto standard object-oriented modeling language for modeling software systems [UML]. UML specifications are developed and maintained by the Object Management Group (OMG). UML is a set of modeling notations for describing static structures and behaviors of software systems. This dissertation uses UML v2.4.

The UML 2.4 specification defines seven structural diagrams: class, object, composite structure, profile, package, component and deployment diagrams. It also defines four kinds of behavioral diagrams: user case, activity, state machine and interaction diagrams.

In this research, we use UML class models, UML object diagrams and UML sequence diagrams. A UML design class model consists of (1) a UML design class diagram that describes the structure of software systems, and (2) class constraints including class invariants and operation constraints specified using the Object Constraint Language. An object diagram describes a configuration of objects. It is also called instance diagram because it is often intended

to be an instance of a class diagram. In this research, object diagrams are used to represent system states and are called *snapshots*.

2.1.1 UML Design Class Diagram

A UML design class diagram describes classes of the object-oriented software systems, class properties and class operations, and relationships between classes. Fig. 2.1 shows an example of a UML design class diagram of a car inventory application.

A class describes a family of objects that have common attributes, operations and constraints. An attribute has a name and a type. Properties relate an instance of the class to a value or collection of values. An operation defines a service that can be executed on each instance of a class. An operation has a name, return type and a list of parameters.

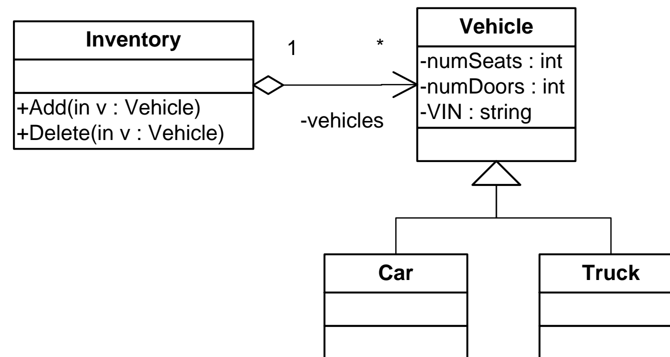


Figure 2.1. A car inventory application design class diagram

The generalization relationship indicates that a subclass is a specialization of another general super-class. For instance, the *car* class is specialization of a *vehicle* class. A subclass inherits properties of its super-class. The objects of a subclass are a subset of objects of its super-class. In Fig. 2.1, *Car* and *Truck* are subclasses of the *Vehicle* class.

An association specifies links between objects. An association can have a name. The ends of an association, called association-ends, have optional properties such as a name and multiplicity. A binary association connects two classes. An association can be bi-directional or uni-directional.

A bi-directional association allows us to navigate from any one of the two classes to another. For instance, the association between *student* and *course* is bi-directional, we can navigate from a *student* object to *courses* or navigate from a *course* object to *students*. A uni-directional association only allows us to navigate from only one class to another. For instance, the association between an *employee* and *address* class is uni-directional, we can only navigate from an *employee* object to an *address* object.

An aggregation is a special type of association. It represents part-whole relationship between two classes. For example, in Fig. 2.1 the *Inventory* class aggregates a *Vehicle* class. An *inventory* object contains a number of *vehicle* objects.

A composition is a special type of aggregation. In a composition relationship the lifecycle of the part class objects is dependent on the whole class objects. For example, the relationship between a *car* class and an *engine* class is composition. A *car* object owns an *engine* object and it will destroy the *engine* object when its lifecycle ends.

2.1.2 The Object Constraint Language

The Object Constraint Language (OCL) is a declarative formal constraint language for UML [OCL]. An OCL expression queries objects. OCL describes the effects of an operation in terms of conditions on the states before and after execution of the operation instead of how an operation is executed to produce the effects. OCL is a typed language. OCL has basic built-in types such as Boolean, Integer, String and Real, and it supports collection types such as Set, Bag, Sequence and OrderedSet. OCL has operations to query collections. For example, there are two boolean operations on collections: forAll and exists. The forAll operation return true if the boolean expression specified by the operation is satisfied by all objects in the collection. The

exists operation returns true if the boolean expression specified by the operation is satisfied by at least one object in the collection.

In this research OCL is used to specify class invariants and operations. OCL class invariants are predicates that constrain all the objects of the class. The class invariants must be satisfied after an object is constructed and after any public operation is executed.

For example, in the vehicle inventory model shown in Fig. 2.1, the invariants for the Inventory class can be stated as below:

```
context Inventory
inv: self.vehicles->forAll( numSeats >= 2 and numSeats <= 5 )
inv: self.vehicles->forAll( numDoors >= 2 and numDoors <= 4 )
```

The invariants states that any vehicle object added to the inventory object should have at least two seats and two doors, and at most five seats and four doors.

OCL operation specifications define the behavior of an operation by specifying the conditions that must be satisfied before an operation is executed (pre-condition) and after the operation is executed (post-condition).

For example, the *Inventory::Add* operation specification is given below:

```
context Inventory::Add (v: Vehicle)
pre: not self.vehicles@pre->includes(v)
post: self.vehicles->includes(v)
```

The operation specification states that before the *vehicle* is added to the *inventory*, it should not exist at the start of the operation, and after the operation is called the *vehicle* must be included in the *inventory*.

2.1.3 UML Sequence Diagram

A UML sequence diagram is used to describe a sequence of interactions between roles and objects. An object is a specific instance of a class. For example, *Bob* is an object of the *Student*

class. A role is a kind of object. For example, *freshman* is a role of the *Student* class. Each role or object in the UML sequence diagram is represented as a lifeline. A role or an object interacts with other objects by sending messages. A message sent to a receiving object represents an invocation of an operation in the receiving object. A message can be synchronous or asynchronous. The sender of a synchronous message is blocked from sending out another message before it receives the response while the sender of an asynchronous message does not have such a limitation. In this research we use synchronous messages only because the messages that are covered by the analysis technique are public operation calls in primarily synchronous programming languages such as Java. Combined fragments such as alternatives, options and loops also give and describe an example of a sequence diagram that describes a scenario will not appear in the analysis sequence diagrams.

Fig. 2.2 shows an example of a UML sequence diagram. In the diagram the *AssignRole* operation is called to assign *cashier* Role to *Bob*. *Bob* activates the *cashier* role and finally calls *CheckAccess* on the Session object and gets Denied response.

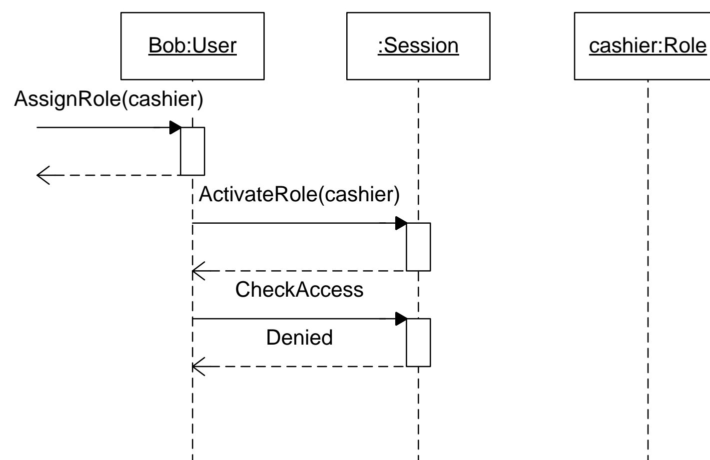


Figure 2.2. UML sequence diagram

2.2 The Meta-Object Facility

UML is a language that is used by developers to describe models of a system or software. A user model is an abstraction of real-world objects (e.g., objects in the running software) and the real-world objects are realizations of the model. The metamodel of UML describes the UML syntax and well-formedness rules. The language used to describe the metamodel is a subset of the UML called the Meta-Object Facility (MOF). The four layers described above form the four-layer metamodeling architecture as shown in Fig. 2.3. The MOF layer is at the M3 layer, the UML is at the M2 layer, the UML user model is at the M1 layer and the real-world objects are at the M0 layer.

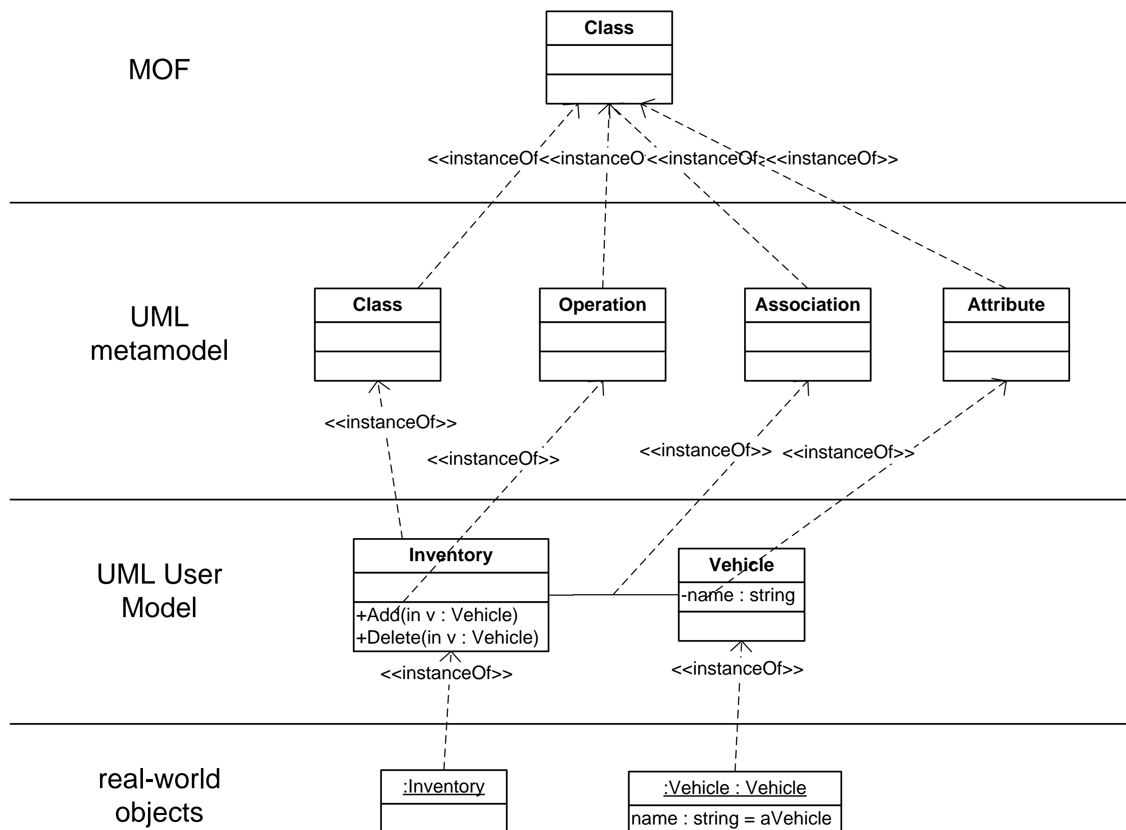


Figure 2.3. UML four-layer metamodeling architecture

The MOF consists of two main packages: the Essential MOF (EMOF) package and the Complete MOF (CMOF) package. The EMOF is a subset of MOF that models classes with attributes and operations.

2.3 The Eclipse Modeling Framework and Ecore

The Eclipse Modeling Framework (EMF) is a modeling framework for the Eclipse platform [Steinberg09][EMF]. EMF is used to create, manipulate and validate models and to generate source code from models.

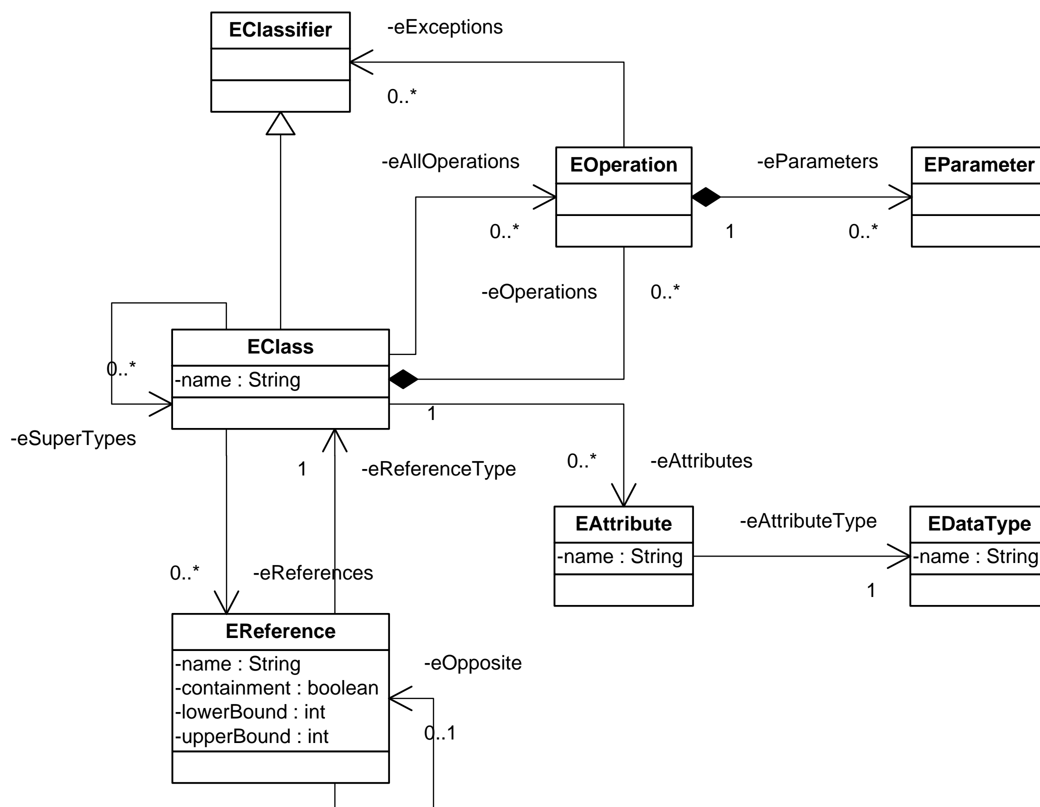


Figure 2.4. Partial Ecore metamodel

The metamodel of EMF is called Ecore. Fig. 2.4 shows part of the *Ecore* metamodel. There are four major entities in Ecore: *EClass*, *EReference*, *EOperation* and *EAttribute*:

- *EClass* models an EMF class. An *EClass* can inherit from multiple super classes.

- *EAttribute* models an attribute of an *Ecore* class. An *Ecore* attribute has a name and data type. An *EClass* object may have a number of attributes.
- *EOperation* models an operation of an *Ecore* class. An *Ecore* operation has a optional list of parameters and exceptions. An *EClass* object may have an *eOperations* reference representing operations of the class, and *eAllOperations* reference representing all operations of the class and its super classes.
- *EReference* models an association end of an *Ecore* class. The containment attribute of an *EReference* indicates whether the reference is a whole-part containment relationship or not. A containment reference in *Ecore* is comparable to a composition relationship in UML.

Ecore is self-describing: The metamodel of *Ecore* is *Ecore*, *Ecore* is meta-model and a meta-metamodel. Ecore is comparable to EMOF package of MOF.

This research uses the Ecore metamodel to implement the Scenario-based UML Design Analysis tool on EMF platform.

2.4 The Kermeta Metamodeling Language

Kermeta is a metamodeling language that can be used to describe both structure and behavior of metamodels [Muller05] [Kermeta]. It is designed to be compliant with EMOF and Ecore. EMOF only defines structures. Kermeta adds an action meta-language to EMOF that can be used to define behavior of operations in metamodels. By weaving the executability into the metamodels, Kermeta can be used to implement domain-specific meta-languages, constraint languages and transformation languages. In this research, the language is used to implement a

transformation algorithm used to produce a snapshot transition model from a design class model, and the snapshot generation algorithm.

Chapter 3

Related Work

This chapter describes related work in the areas of analysis and testing of UML design models, and UML test scenario generation. Section 3.1 describes related work in analyzing UML design models using formal analysis techniques. Section 3.2 describes related work in UML animation and testing. Section 3.3 describes related work in UML static analysis tools such as USE and OCLE. Section 3.4 describes related work in UML test input generation and scenario generation.

3.1 Formal analysis techniques

This section surveys related work on formal analysis of UML models. Section 3.1.1 describes formal analysis of UML models using Alloy. Section 3.1.2 describes analysis of UML models using model checking techniques. Section 3.1.3 describes analysis of UML models using formal theorem proving techniques.

3.1.1 Alloy

Alloy is a formal notation based on set theory and first-order relational logic [Alloy]. Alloy models structures of software systems using signatures. A signature can have fields and it can inherit from a parent signature. A fact is a logical constraint that must be satisfied by the system. An assertion is a constraint that is not necessarily true. Operations of the model are defined using functions and predicates in a declarative manner. A function is an expression that maps a list of parameters to output. A predicate is a parameterized constraint. A predicate can be used to define an operation as a relation between before and after states.

Alloy can be used to automatically find a model that satisfies specified properties within a bounded search space. To check a property, Alloy either generates a model to show that the property is satisfiable, or finds a counter-example that violates the property.

Analyzing UML models using Alloy requires the transformation of UML models to Alloy models. Existing work on UML2Alloy tool can be used to transform a UML class model to Alloy [UML2Alloy]. However, the analysis of UML models in Alloy requires that the analyzer be familiar with Alloy notation because the analysis results are shown in Alloy. Shah et. al. extended the UML2Alloy work to transform analysis results back to UML [Shah09].

It is a challenging problem to prove the correctness of transformation from UML design class model to Alloy and the transformations that exist do not cover all UML class modeling concepts. As Shah et al. admitted, UML and Alloy have different approaches to object-oriented concepts including inheritance, overriding and pre-defined types. Some UML and OCL concepts such as redefinition, multiple inheritance and OCL bags cannot be represented directly in Alloy. OCL nested collections cannot be transformed to Alloy because it is impossible to express higher-order relations in Alloy.

3.1.2 Model checking techniques

Model checking is used to verify the design of a hardware or software system against a set of temporal properties [Clark99]. Given a system model, a model checking technique decides whether a desired property, expressed as a temporal logic formula, is satisfied or not in the model. Propositional temporal logic is a branch of symbolic logic used to express propositions whose values are dependent on time. There are two basic temporal operators in temporal logic: always and eventually. There are two major types of properties that can be expressed using temporal

logic: a safety property is a property that is always true during any execution of the system, and a liveness property is a property that is eventually true during execution.

To check a desired safety or liveness property, the model checker exhaustively searches the state space of the structure. If the desired property is satisfied, it returns true, otherwise, it returns a counter-example that shows how the desired property is violated.

Model checking has been applied to automate the verification of the safety and correctness of finite state-based systems [Clark99]. There is work that aims to support model-checking of UML behavioral models. vUML [Lilius99] is a tool that is used to automatically convert UML statecharts to PROMELA specifications and then invoke SPIN to verify the desired properties and check inconsistencies. Eshuis [Eshuis06] applied symbolic model checking to UML activity models. The activity models are formalized and transformed to the input language of NuSMV [NuSMV99]. The translations are used to check the data integrity constraints expressed in the workflow described by the activity models. A transformation process is needed to convert the UML specifications into the input language of the model checker.

The limitation of model checking techniques is that they suffer from state explosion problem [Valmari98] [Clarke01]. Since model checkers exhaustively search the state space of a model to verify temporal property, the state explosion can occur when the model contains many components that make parallel state transitions [Clarke01]. There is ongoing work in the model checking research community to alleviate the state explosion problem but it remains a major problem in analyzing large industrial software systems.

In order to use model checking techniques to analyze UML class models, the models must be transformed to the input languages of the model checkers. Thus the verifier must be familiar with these notations to do formal analysis. Compared with these formal analysis tools, the

Scenario-based UML Design Analysis technique does not require that the verifier be familiar with notations other than UML and OCL. Instead of doing heavyweight exhaustive analysis the technique is lightweight because it analyzes UML design class models in the scope of a set of scenarios. On the other hand, the technique presented in this dissertation cannot be used to analyze temporal properties of UML design class model. There is ongoing research ay Al-lail et al. on using snapshot transition models to support analysis of temporal properties [Al-Lail13].

3.1.3 Formal theorem proving technique

Formal theorem provers such as Isabelle can be used to reason about properties described in UML models [Brucker08]. In the work of Brucker, et, al., an interactive proof environment for UML/OCL models called HOL-OCL is developed on top of Isabelle, an interactive theorem prover for Higher-Order Logic (HOL) [Isabelle02]. HOL-OCL can be used to formally analyze UML models, for example, it can be used to check consistency between UML models, prove temporal properties of UML models and prove a UML class model is refinement of another class model.

In order to use formal theorem proving techniques to analyze UML class models, the verifier must be familiar with the formal notations. Compared with the formal theorem proving techniques, the Scenario-based UML Design Analysis technique does not require that the verifier be familiar with the notations such as Isabelle in the analysis process.

3.2 UML animation and testing

The UML animation and testing approach (UMLAnT) is used to systematically test executable design UML design class models, that is, class models with executable method descriptions. [Trung05]. In UMLAnT a UML design under test (DUT) is a detailed platform

independent model (PIM) described by UML design class diagrams, UML sequence diagrams and method descriptions expressed in an action language called the Java Action Language (JAL). The UML design also contains OCL specifications of operation behaviors. Test inputs are exercised by the executable UML design model. A USE tool plugin is used to maintain object configurations during the test and to check OCL constraints against the object configurations generated during model execution. In UMLAnT a sequence model must be provided in the design to describe a test scenario. The sequence diagram is also used to define test criteria; i.e., to determine when enough test cases have been generated. Each scenario is triggered by a single operation call referred to as a system operation.

The Scenario-based UML Design Analysis technique and UMLAnT are both UML consistency checking techniques. The Scenario-based UML Design Analysis checks consistency between UML design class model and scenarios, while UMLAnT checks consistency between UML operation specifications described in a design class model and UML operations described using JAL. The Scenario-based UML Design Analysis is a static technique because it does not execute the UML design model, while UMLAnT executes test input sequences and operation actions and check operation pre and post conditions before and after an operation is executed.

The Scenario-based UML Design Analysis technique complements UMLAnT in analyzing UML design class models. Before detailed operation actions are specified for a UML design, the verifier can create scenarios to analyze the UML design. After detailed operation actions are specified, UMLAnT can be used to test the UML design against a sequence model.

3.3 USE and OCLE

Existing UML modeling tools like OCLE [OCLE] and USE [USE] provide support for validating syntactic and structural properties. OCLE for example can detect syntactic errors in models and syntax errors in OCL specifications. USE and OCLE can be used to check the consistency between a UML design class model and an object model.

Neither of these tools can be directly used to analyze functionality in scenarios. OCLE does not support analysis of operation specifications in class models against snapshots. The USE tool can be used to validate pre and post-conditions of operations against snapshots in interactive command mode, however, the verifier has to manually enter USE commands to build all snapshots of a scenario. The process to manually build snapshots in USE is time-consuming and error-prone.

3.4 UML test input and scenario generation

This section discusses related work in UML test input and scenario generation. Section 3.4.1 discusses UML animation techniques. Section 3.4.2 discusses UML test input generation techniques.

3.4.1 UML animation techniques

Scenarios can be obtained by executing models to produce traces or by using constraint solving techniques.

Oliver and Kent propose a technique to animate a UML design [Oliver99]. In their work UML design class diagrams are animated by performing a sequence of actions on an initial snapshot. An action is an operation call on an object with arguments. All possible execution paths of the OCL post-condition of the operation is calculated. Each execution path is mapped to

operations on snapshots. After applying all execution paths on the snapshot a set of possible after-states are generated. The after-states that violate the multiplicity constraints are discarded. The final after-states are possible snapshots of the animation of the action.

In another piece of work Krieger and Knapp use a SAT solver to find new system state that satisfies operation post-conditions [Krieger08]. In their work OCL post-conditions and class invariants are translated to arithmetic formulas. A SAT solver Kodkod [Torlak07] is used to find models that satisfy the arithmetic formulas.

In this research work scenarios are generated by either executing the verifier's operational definitions of operations using JAL or solving constraints based on the verifier's OCL operation definitions. None of the research work above can be used to execute JAL or UML actions to generate scenarios. The work in [Krieger08] can be used to generate a next system state that satisfies the verifier's OCL object definitions but it does not generate complete scenarios.

3.4.2 UML test input generation techniques

There are a few research works that generate test sequences or scenarios from UML requirements use cases.

Briand and Labiche proposed an approach to generate test data and test oracles from UML analysis model for system testing [Briand02]. In their work, system test requirements are automatically derived from UML analysis artifacts. Valid use case sequences are generated based on use case sequential constraints described using activity diagrams. Use case sequences can be interleaved and each use case may have use case variances which are described using a decision table. The method depends on the verifier's knowledge to select test cases from a large number of interleaved use case sequences and use case variances. Also not all the use case and use case variance sequences are feasible. In this work, a constraint solving technique is used to

find initial system state and system operation parameters for all feasible paths in the activity model.

Nebut et. al. proposed a use-case driven approach to generate system test inputs [Nebut06]. In their work use cases are fully specified with pre and post conditions. Use cases are built into a Use Case simulation and Transition System (UCTS). Valid instantiated use case sequences are generated by exhaustively simulating the system. The limitation of the approach is that the space of UCTS may be huge when many use cases can be applied at each step of simulation.

Kundu and Samanta use UML activity diagram that describes activity sequences inside one use case to generate system test cases [Kundu09]. In their work the activity diagram is converted to an activity graph and test sequences are generated from the graph based on different coverage criteria.

The approaches describe above generate test sequences for testing code that implements UML models. The approaches cannot be used to generate scenarios because the scenarios in this research work include not only test operation sequences, but also states after each operation is called.

Chapter 4

Scenario-based UML Design Analysis Technique

This chapter gives a description of the Scenario-based UML Design Analysis technique. An overview of the Scenario-based UML Design Analysis technique is shown in Fig. 4.1. There are two roles involved in the Scenario-based UML Design Analysis process: designer and verifier. The designer creates a UML design class model that includes OCL operation specifications, and *operation scope specifications* which specify the set of classes, attributes and links that are changed by each operation. The verifier creates a set of scenarios that will be used to analyze the UML design. A *scenario* describes functionality from the perspective of the verifier. It consists of an initial system state, a sequence of operation calls, and system states after each operation call. A *legal scenario* describes functionality that is desired and an *illegal scenario* describes functionality that is not desired. The technique consists of three major steps as shown in Fig. 4.1.

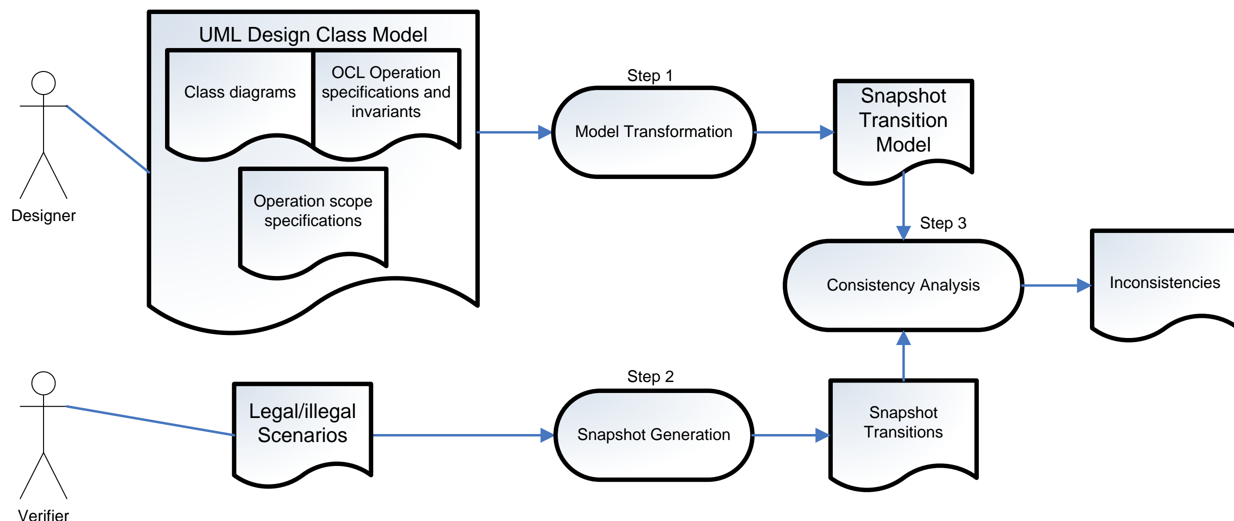


Figure 4.1: Scenario-based UML design analysis technique

In the first step, the UML design class model is automatically transformed to a s*napshot transition model,* a UML class model that specifies valid *snapshot transitions.* A *snapshot transition* describes system state changes triggered by an operation call, it consists of (1) parameter values of the operation that triggers the transition, (2) a *before-snapshot* describing the system state before the operation is executed, and (3) an *after-snapshot* describing the system state after the operation has been executed.

In the second step, scenarios created by a verifier are used to generate a sequence of snapshot transitions.

In the third step, the snapshot transitions produced in the second step are checked against the snapshot transition model using the UML structural analysis tool, USE. USE reports a set of inconsistencies in the form of class invariant violations. An inconsistency between the UML design class model and scenarios implies defects in the UML design class model, or defects in the scenarios, or defects in both the design class model and scenarios.

The rest of the chapter is organized as follows: Section 4.1 describes a partial RBAC design class model that is used to illustrate the technique. Section 4.2 presents an algorithm for transforming a UML design class model to a *snapshot transition model.* Section 4.3 describes how scenarios are transformed to *snapshot transitions*. Section 4.4 describes how the USE tool is used to check the consistency between the UML design model and the scenarios. Section 4.5 analyzes the complexity of the transformation algorithms.

4.1 A simple Role-Based Access Control example

Role-based access control (RBAC) is the de facto access control model used in commercial organizations [Ferraiolo01]. In RBAC, users are assigned to roles, and roles are associated with

permissions that determine what operations and data a user playing the role can access. The users initiate sessions in which they activate a subset of roles assigned to them. The operations that a user can perform in a session depend on the permissions associated with the activated roles. Constraints can be specified on the RBAC model to prevent conflict of interest situations in an organization. Specifically, there are two types of constraints: *Static Separation of Duties* (SSD) and *Dynamic Separation of Duties* (DSD). These are defined as relationships between roles. SSD requires that conflicting roles not be assigned to the same user. DSD imposes a more relaxed requirement: It allows conflicting roles to be assigned to the same user, but does not allow conflicting roles to be activated in the same session.

The part of the RBAC model used to illustrate the approach is shown in Fig. 4.2. This partial RBAC model shows only the relationships between roles, users and sessions. The figure shows only the elements used to illustrate the approach in this chapter.

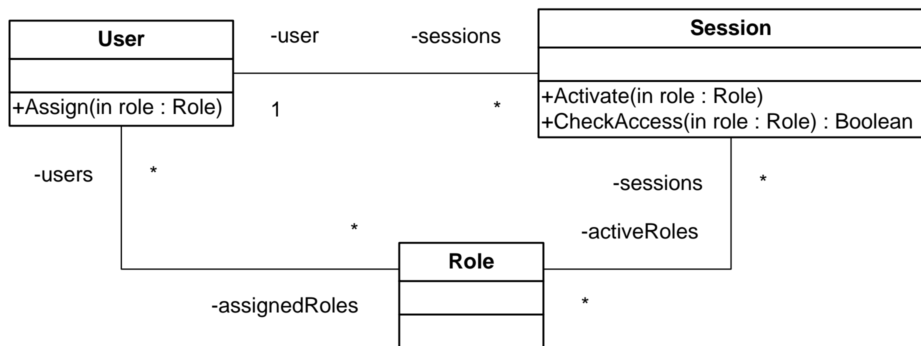


Figure. 4.2. Partial RBAC design class model

The operation specifications of the RBAC model are given below:

```
context User::Assign(role:Role)
// pre-condition: role is not included in assigned roles of the user
pre: self.assignedRoles->forAll(r | r <> role)
// post-condition:
// role is included in assigned roles of the user
```

```
// and all other assigned roles remain assigned to the user
post: self.assignedRoles->exists(r | r = role)
      and self.assignedRoles@pre->forAll(r1 |
          self.assignedRoles->exists(r2 | r1 = r2))
      and (self.assignedRoles->size() = self.
          assignedRoles@pre->size() + 1)

context Session::Activate(role:Role)
// pre-condition: role is not activated in active roles of the user
pre: self.activeRoles->forAll(r | r <> role)
// post-condition:
// role is included in active roles of the user
// and all other activated roles remain active in the session
post: self.activeRoles->exists(r | r = role)
      and self.activeRoles@pre->forAll(r1 |
          self.activeRoles->exists(r2 | r1 = r2))
      and (self.activeRoles ->size() =
        self.activeRoles@pre->size()+ 1)

context Session::CheckAccess(role:Role)
// pre-condition: true
// post-condition: return true if role is includes in active roles
// of current session
post: result = self.activeRoles()->exists (r| r = role)
```

The static separation of duty (SSD) property of RBAC restricts the assignment of conflicting roles to one user. This property is expressed as an invariant on the *User* class. The SSD property is one of the properties that we verify an example scenario against. The example scenario involves two users, *Alice* and *Bob*, and the following roles: *Cashier*, *Accountant* and *Teller*. The SSD property in this example is the following: The role *Accountant* and *Cashier* cannot be assigned to the same user. The specification of this SSD property is given below:

```
context User
//Static separation of duty constraint
inv SSD: not (self.assignedRoles->exists(r | r = Accountant)
     and self.assignedRoles->exists(r | r = Cashier))
```

The example scenario that will be analyzed describes an illegal situation in which a user is assigned to two roles that violate the above SSD property. The scenario starts in an initial state consisting of a User object *Bob*, an *Accountant* role and a *Cashier* role. The scenario consists of the following steps:

(1) *Bob* is assigned *Accountant* role through a call to the *Assign()* operation. After the operation is called, the *Accountant* role is included in *assignedRoles* collection of *Bob*.

(2) *Bob* is assigned the *Cashier* role through a call to the *Assign()* operation. After the operation is called, the *Cashier* and *Accountant* roles are included in *assignedRoles* collection of *Bob*.

This scenario is classified as an illegal scenario, because the last system state in the scenario violates the SSD constraint associated with associated with the User class.

4.2 Generating the Snapshot Transition Model

In order to use tools such as USE and OCLE to support scenario-based analysis, a class model that characterizes valid snapshot transitions is generated from a UML design class model. The generated class model is called a *snapshot transition model* (STM). A snapshot transition model consists of (1) a *Snapshot* class representing states of the system before and after execution of operations, (2) a hierarchy of *Transition* classes representing specified operations, and (3) invariants defined in the *Transition* classes that constrain the before and after states (snapshots) associated with transitions caused by the execution of operations.

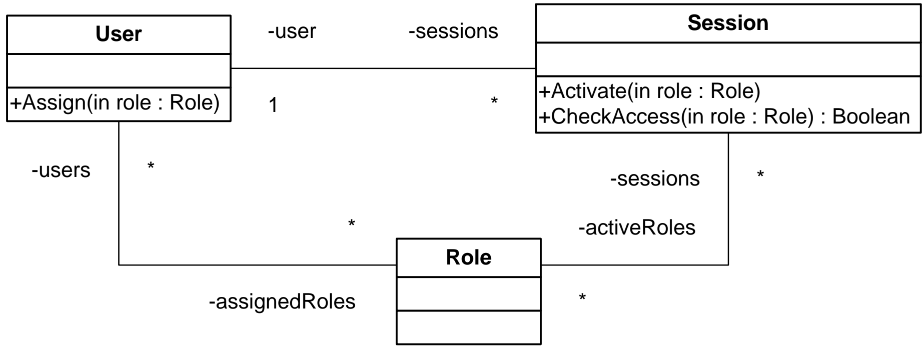
In the following we present the steps for creating an STM. An overview of the steps is given below, and the following subsections elaborate on each step.

- Step 1: Create a Snapshot class that represents valid states (object configurations).
- Step 2: Create a Transition subclass for each operation in the design class model.
- Step 3: Generate initial Transition invariants for operation specifications.
- Step 4: Add frame constraints to the Transition invariants. Frame constraints specify that objects and links that are not affected by the operation are the same in the before and after snapshots.

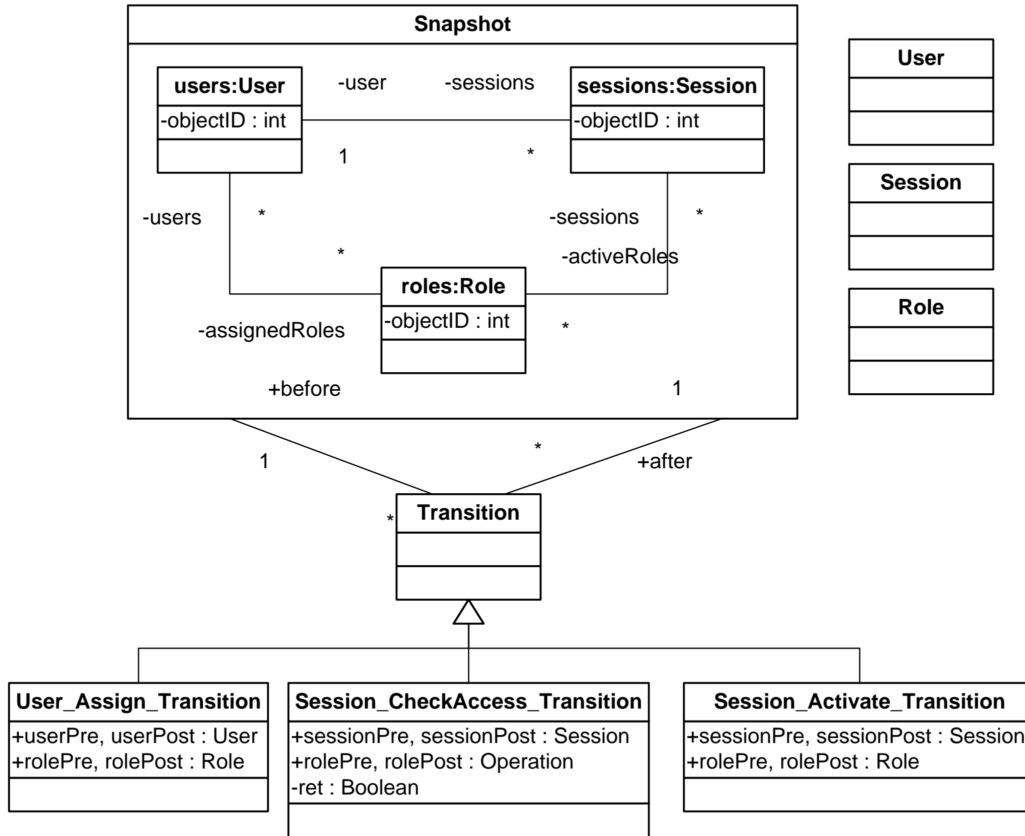
The RBAC application class model is used to illustrate the steps described in the following sub-sections.

4.2.1 Create a Snapshot class

The *Snapshot* class represents a set of system states (snapshots), where a state consists of a configuration of object states. An object state is an assignment of values to the attributes of the object's class. A Snapshot class is thus modeled as a structured class that consists of a configuration of UML parts representing object states. Each part represents a set of object states and is thus associated with a class in the design class model. For example, the Snapshot class in Fig. 4.3 for the partial RBAC design model in Fig. 4.3 consists of a configuration of parts, where the *users* part represents states of *User* objects, the *roles* part represents states of *Role* objects, and the *session* part represents states of *Session* objects. The states are defined by classes in the *snapshot transition model* that have the same name as the corresponding classes in the design class model.



(a) RBAC design class model



(b) RBAC snapshot transition model

Figure 4.3. Partial RBAC class model and its snapshot transition model

Each part has an *objectID* attribute that relates object states across different snapshots. For example, the bob User object in different snapshots have the same *objectID*. Note that instances of these classes represent immutable object states, not mutable objects; for example, instances of the User class in the STM represent immutable object states, while User class in the originating

design model represent mutable User objects. This subtle difference is important in understanding how transition systems are characterized by a STM: the snapshots in a transition system are (immutable) values that are related by transitions (execution instances of operations), thus an STM characterizes instances of behaviors (i.e., scenarios).

4.2.2 Create a Transition class hierarchy

A superclass called *Transition* that has before and after associations to the *Snapshot* class is created (see the Transition class in Fig. 4.3). A transition object is a representation of the effect of an operation's execution, where the effect is defined by a before-state and after-state pair. The *Transition* superclass is specialized by Transition subclasses that each describes the effects of an operation specified in the design class model. A *Transition* subclass is created for each operation in the design class model. Given an operation *ClassName::operationName*, a *Transition* subclass is created as follows:

- Create an empty subclass of *Transition* with the name *ClassName_operationName_Transition*. For example, *User_Assign_Transition* class is created for operation *User::Assign*.
- Create a class property that references the before state of the operation's target object and another that references the after state of the target object. The property referencing the before state is named *classNamePre*, and the other property is named *classNamePost*. For example, in Fig. 4.3, the *User_Assign_Transition* class has attributes *userPre* and *userPost*, which are references to before and after states for the target *User* object of the *Assign()* operation.
- For each value (i.e., non-object) parameter in the operation, create a class property (attribute) with the same name and type in the *Transition* class.

- For each parameter that is an object reference, create two class properties with the same type as the object reference. One of the properties represents the before state of the object and is thus named *parameterNamePre*, and the other represents the after state of the object and is named *parameterNamePost*. For example, the operation *CheckAccess* in the *Session* class has a reference to parameter *role*, and this parameter is transformed to the attributes *rolePre* and *rolePost* in the *Session_CheckAccess_Transition* class shown in Fig. 4.3. The parameters that represent before and after object states are collectively referred to as *preState* and *postState* attributes.
- If there is a return parameter, create a property *ret* with the same type as the return parameter. For example, the *CheckAccess()* operator in the *Session* class returns a boolean value, and this return value is represented by the attribute *ret* in *Session_CheckAccess_Transition* class shown in Fig. 4.3.

The Transition class hierarchy shown in Fig. 4.3 (b) was produced using the above steps.

4.2.3 Generate Transition invariants from operation specifications

We use the *Assign()* operator defined in the User class to illustrate how invariants that relate before and after states are generated from operation specifications. The definition of the *User::Assign* operation in is repeated below:

```
//pre- and post- conditions of the Assign method
context User::Assign(role:Role)
pre: self.assignedRoles->forAll(r | r <> role)
post: self.assignedRoles->exists(r | r = role)
     and self.assignedRoles@pre->forAll(r1 |
          self.assignedRoles->exists(r2 | r1 = r2))
     and (self.assignedRoles->size() =
          self.assignedRoles@pre->size() + 1)
```

For each operation specification in the design class model, an invariant is produced as follows:

- Replace all references to self in the pre-condition by the name of the *Transition* class attribute representing the target object before state (all references to self must be explicit in the operation specification for this to work). Also, replace all references to an object parameter in the pre-condition by the name of the attribute representing the before state of the object, and replace all references to the object in the post-condition by the name of the attribute representing the object's after state.
  For example, the precondition of the *Assign()* operation,
  ```
  self.assignedRoles->forAll(r|r <> role)
  ```
  is transformed to (changes are in bold print)
  ```
  userPre.assignedRoles->forAll(r | r <> rolePre)
  ```
- Replace all references to self in an expression involving @pre by the name of the attribute representing the before state of the object.
  For example, the *Assign()* post-condition clause
  ```
  self.assignedRoles@pre->
  ```
  is transformed to
  ```
  userPre.assignedRoles->
  ```
- Replace all references to self in the post-condition that are not part of a @pre expression by the name of the attribute representing the after state of the target object.
  For example, the *Assign()* post-condition clause
  ```
  self.assignedRoles->exists
  ```
  is transformed to
  ```
  userPost.assignedRoles->exists
  ```
- Replace all references to objects by references to objectID attributes.
  For example, the clause
  ```
  userPre.assignedRoles->forAll(r| r <> rolePre)
  ```
  is further transformed to
  ```
  userPre.assignedRoles->forAll(r| r.objectID <> rolePre.objectID)
  ```

4.2.4 Add frame constraints to the Transition subclass

One challenging aspect of transforming OCL operation specifications is to generate *frame constraints* for an operation. The frame constraints ensure that objects and links that are not affected by the operation remain unchanged in the before and after snapshots.

In order to simplify the generation of *frame constraints*, we require that the designer creates *operation scope specifications* which specifies the set of classes, attributes and links that are changed by each operation. For example: the scope specification of operation User::AssignRole is specified below:

```
Operation: User::AssignRole
Modifier_Class: User, Role
Modifier_Attribute:
Modifier_Link: User.UserAssign, Role.UserAssign
```

The scope specification states that only the *UserAssign* association between *User* and *Role* classes are changed after the operation is invoked.

Based on the operation scope specification, the frame constraints are generated for objects and associations that are not changed by the operation as follows:

- Add constraints that assert the existence of the object states referenced by preState attributes in the before states. The constraint has the form before.*partName* -> includes(*namePre*). Similarly, add constraints that assert the existence of the object states referenced by postState attributes.

  For example, the following clauses assert the existence of the target user states in the before and after states of the snapshot respectively:

  ```
  before.users:User.objectID->includes(userPre.objectID)
  after.users:User.objectID->includes(userPost.objectID)
  ```

- Add frame constraints that state that objects and relationships that have not had their state changed in an operation have the same before and after state. These constraints can take two forms as illustrated in the examples given below:

  For example, the constraint stating that the set of session objects is unchanged by the

Assign() operation is stated below:

```
after.sessions:Session.objectID=
before.sessions:Session.objectID
```

The constraint stating that the user objects not affected by the operation have the same before and after states is stated below:

```
after.users:User.objectID->excluding(userPost.objectID)=
before.users:User.objectID->excluding(userPre.objectID)
```

The full invariants for the *User_Assign_Transition*, *Session_Activate_Transition* and *Session_CheckAccess_Transition* classes are given below:

```
context User_Assign_Transition
//From Assign() pre-condition
userPre.assignedRoles->forAll(r | r.objectID <> rolePre.objectID)
and
//From Assign() post-condition
userPost.assignedRoles->exists(r | r.objectID = rolePost.objectID)
and
userPre.assignedRoles->forAll(r1 | userPost.assignedRoles->
   exists(r2 | r1.objectID = r2.objectID)) and
  userPost.assignedRoles->size() =
     userPre.assignedRoles->size() + 1 and
//Frame constraints
//userPre is included in before snapshot
before.users:User.objectID->includes(userPre.objectID) and
//userPost is included in after snapshot
after.users:User.objectID->includes(userPost.objectID) and
//The rest of users in before and after snapshots are the same
after.users:User.objectID->excluding(userPost.objectID) =
     before.users:User.objectID->excluding(userPre.objectID) and
//rolePre is included in before snapshot
before.roles:Role.objectID->includes(rolePre.objectID) and
```

```
//rolePost is included in after snapshot
after.roles:Role.objectID->includes(rolePost.objectID) and
//The rest of roles in before and after snapshots are the same
after.roles:Role.objectID->excluding(rolePost.objectID) =
     before.roles:Role.objectID->excluding(rolePre.objectID) and
//All sessions in before and after snapshots are the same
after.sessions:Session.objectID = before.sessions:Session.objectID
//All associations between the user and session classes in before
//and after snapshots are the same
and before.users:User->forAll(u1 | after.users:User->exists(u2 |
u1.objectID = u2.objectID and u1.sessions:Session.objectID =
u2.sessions:Session.objectID))
//All associations between the role and session classes in before
//and after snapshots are the same
and before.roles:Role->forAll(r1 | after.roles:Role->exists(r2 |
r1.objectID = r2.objectID and r1.sessions:Session.objectID =
r2.sessions:Session.objectID))

context Session_Activate_Transition
//From Activate() pre-condition
sessionPre.activeRoles->forAll(r| r.objectID <> rolePre.objectID)
and
//From Activate() post-condition
sessionPost.activeRoles->exists(r| r.objectID = rolePost.objectID)
and
sessionPre.activeRoles->forAll(r1 | sessionPost.activeRoles->
   exists(r2 | r1.objectID = r2.objectID)) and
  sessionPost.activeRoles->size() =
     sessionPre.activeRoles->size() + 1 and
//Frame constraints
//sessionPre is included in before snapshot
before.sessions:Session.objectID->includes(sessionPre.objectID)
```

```
//sessionPost is included in after snapshot
after.sessions:Session.objectID->includes(sessionPost.objectID)
//The rest of sessions are the same in before and after snapshots
after.sessions:Session.objectID->excluding(sessionPost.objectID)
=
before.sessions:Session.objectID->excluding(sessionPre.objectID)
//rolePre is included in before snapshot
and before.roles:Role.objectID->includes(rolePre.objectID) and
//rolePost is included in after snapshot
and after.roles.objectID->includes(rolePost.objectID) and
//The rest of roles are the same in before and after snapshots
after.roles:Role.objectID->excluding(rolePost.objectID) =
before.roles:Role.objectID->excluding(rolePre.objectID) and
//All users are the same in before and after snapshots
and after.users:User.objectID = before.users:User.objectID
//All associations between the user and session classes in before
//and after snapshots are the same
and before.users:User->forAll(u1 | after.users:User->exists(u2 |
u1.objectID = u2.objectID and u1.sessions:Session.objectID =
u2.sessions:Session.objectID))
//All associations between the user and role classes in before
//and after snapshots are the same
and before.users:User->forAll(u1 | after.users:User->exists(u2 |
u1.objectID = u2.objectID and u1.assignedRoles:Role.objectID =
u2.assignedRoles:Role.objectID))

context Session_CheckAccess_Transition
//From CheckAccess() pre-condition
true and
//From CheckAccess() post-condition
ret = sessionPost.activeRoles->exists(r| r.objectID =
rolePost.objectID) and
```

```
//Frame constraints
//sessionPre is included in before snapshot
before.sessions:Session.objectID->includes(sessionPre.objectID)
//sessionPost is included in after snapshot
after.sessions:Session.objectID->includes(sessionPost.objectID)
//The rest of sessions are the same in before and after snapshots
after.sessions:Session.objectID->excluding(sessionPost.objectID)
=
before.sessions:Session.objectID->excluding(sessionPre.objectID)
//rolePre is included in before snapshot
and before.roles:Role.objectID->includes(rolePre.objectID) and
//rolePost is included in after snapshot
and after.roles.objectID->includes(rolePost.objectID) and
//The rest of roles are the same in before and after snapshots
after.roles:Role.objectID->excluding(rolePost.objectID) =
before.roles:Role.objectID->excluding(rolePre.objectID) and
//All users are the same in before and after snapshots
and after.users:User.objectID = before.users:User.objectID
//All associations between the user and session classes in before
//and after snapshots are the same
and before.users:User->forAll(u1 | after.users:User->exists(u2 |
u1.objectID = u2.objectID and u1.sessions:Session.objectID =
u2.sessions:Session.objectID))
//All associations between the user and role classes in before
//and after snapshots are the same
and before.users:User->forAll(u1 | after.users:User->exists(u2 |
u1.objectID = u2.objectID and u1.assignedRoles:Role.objectID =
u2.assignedRoles:Role.objectID))
//All associations between the role and session classes in before
//and after snapshots are the same
```

```
and before.roles:Role->forAll(r1 | after.roles:Role->exists(r2 |
r1.objectID = r2.objectID and r1.sessions:Session.objectID =
r2.sessions:Session.objectID))
```

4.3 Generating Scenario Snapshot Transitions

Scenario snapshot transitions can be automatically generated from the sequence diagrams describing *scenarios* created by the verifier. The generation process is described below:

For each operation *op* invoked on object *obj* in the scenario:

- Find the corresponding *Transition* subclass for operation *op*.
  (1) Get the class *c* of object obj.
  (2) If *c* has an operation that overrides operation *op*, the return transition subclass is *c_op_Transition*. The UML operation overriding rule is described in [Büttner04] and [UML]. According to UML 2.0, an operation of a subclass overrides the operation of its parent class if the name of the two operations match and the type of every formal parameter (and result value) of the operation matches or specializes a corresponding parameter (result value) of the parent operation. It is assumed that the overriding operation in the subclass redefines all pre and post conditions.
  (3) Repeat step 2 on the parent class of *c* until no operation is found.
- Create an instance of the corresponding *Transition* subclass for operation *op*.
- Create an instance of *snapshot* class for the *snapshot* before the operation call if the operation is the first one in the scenario.
- Create an instance of *snapshot* class for the snapshot after the operation class.
- Link the two *snapshot* instances to before and after snapshot of the *Transition* subclass.
- Fill attributes of the *Transition* subclass with parameters of the operation call.

Fig. 4.4 shows a *snapshot transition* in which user *Bob* is assigned *Accountant* role. In this transition, the before and after snapshots both connect to three objects: the *Cashier* role, the *Accountant* role and the *Bob* user. After the *User::Assign* operation on *Bob* user *Account* role is invoked, the *Accountant* role is added to *assignedRole* link of *Bob* user.

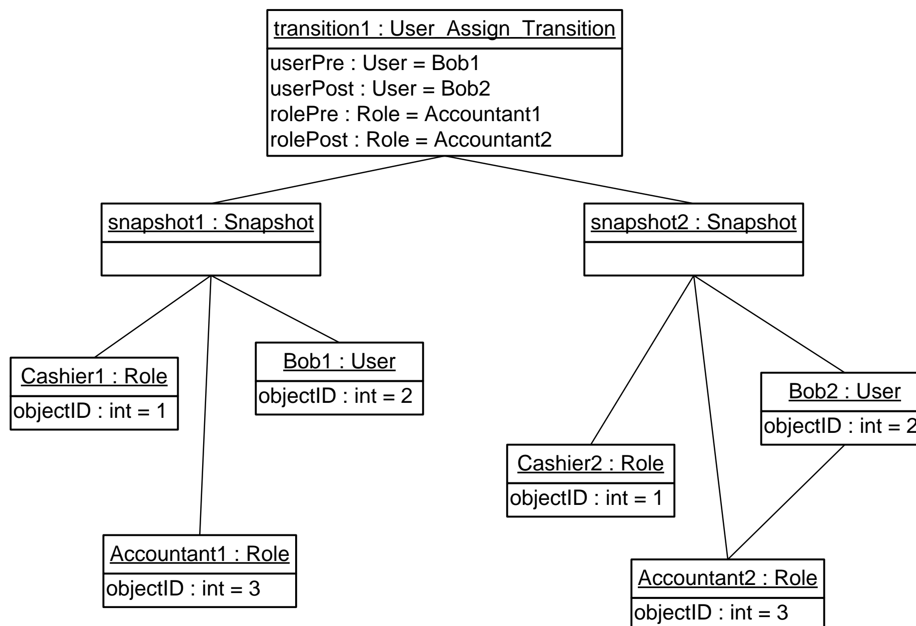


Figure 4.4. Assign Accountant role snapshot transition

4.4 Checking consistency in USE

Design errors are uncovered by checking for inconsistencies between the s*napshot transition model* and the *scenario snapshot transitions*. This is done using the USE tool which checks whether *scenario snapshot transitions* conform to the *snapshot transition model*. *Scenario snapshot transitions* are instances of the *snapshot transition model* so that we can feed the *snapshot transition model* and *scenario snapshot transitions* to USE and check whether they are consistent. Inconsistencies imply errors in either the UML design class model or the scenarios. It is up to the modeler and verifier to analyze the inconsistencies, find the root cause of the inconsistencies and resolve the inconsistencies.

For the RBAC example we analyzed a scenario against the *snapshot transition model* shown in Fig. 4.3. The scenario consists of two operations: *Bob.assign(Accountant)* and *Bob.assign(Cashier).* We used the USE tool. The USE tool reported an error arising after the assignment of two conflicting roles *Accountant* and *Cashier* to the same user *Bob*.

We also performed a second analysis involving a dynamic separation of duty (DSD) property which prohibits some roles assigned to a user to be activated at the same time in a session. As we expected USE reported an error arising from the activation of two conflicting roles in the same session created by *Bob*.

4.5 Algorithm complexity analysis

This section analyzes complexity of the major procedure used in the technique. Section 4.5.1 analyzes complexity of the snapshot transition model generation procedure. Section 4.5.2 analyzes complexity of the snapshot transitions generation procedure. Section 4.5.3 analyzes complexity of USE consistency check. Section 4.5.4 summarizes the analysis.

4.5.1 Snapshot transition model generation algorithm analysis

Let the total number of class in the UML design class model be *C*, the average number of properties including attributes and associations of each class be *A*, the total number of operations be *P*, the average number of reference parameters of each operation be *Q1* and the average number of value parameters of each operation be *Q2*. The average size of the syntax tree of pre and post-conditions of an operation is denoted by *T*.

The time complexity of generating the Snapshot class is *O(C)* because the Snapshot class is associated to every class in the UML design.

The time complexity of generating the Transition subclasses for each operation is $O(P) * (2 * O(Q1) + O(Q2) + 1)$. Every reference parameter of the operation is added as two attributes of the Transition subclass, one prefixed with pre and another prefixed with post. Every value parameter of the operation and an optional return value are added as attributes of the Transition subclass.

The time complexity of transforming the OCL operation specifications depends on:

(1) The time complexity of parsing all OCL expressions of each operation and transforming them to invariants, which is $O(P) * O(T)$, and

(2) The time complexity of adding frame constraints for each operation, in the worse case we need to add frame constraints for all classes and properties, the time complexity is $O(P) * O(C) * O(A)$.

Total time complexity of generating the snapshot transition model is

$O(C) + O(P) * (2 * O(Q1) + O(Q2) + 1) + O(P) * O(T) + O(P) * O(C) * O(A)$

$= O(C) + O(P) * (2 * O(Q1) + O(Q2) + 1 + O(T) + O(C) * O(A))$

4.5.2 Snapshot transitions generation algorithm analysis

The time complexity of generating snapshot transitions depends on the number of objects in the snapshot transitions. Let the number of snapshots in the scenario be S and the average number of objects in a snapshot be B. The time complexity of generating the snapshot transitions is $O(S) * O(B)$. The time complexity is proportional to the number of instances in the scenario.

4.5.3 USE consistency check complexity analysis

USE is used to check consistency between each snapshot transition instance and invariants of the corresponding snapshot transition subclass. The complexity of checking consistency

between a class model and a snapshot depends on the complexity of the class invariants and the number of instances in the snapshot.

The invariants in the snapshot transition model are checked against each snapshot transition in the snapshot transitions, so the complexity of USE consistency checking depends on three factors: the number of snapshot transitions (operations) in the scenario, the average number of objects in the before and after snapshots, and the complexity of invariants in the snapshot transition model.

4.5.4 Summary

To sum up the algorithm complexity analysis, the complexity of snapshot transition model generation algorithm depends on the complexity of the UML design class model, including number of classes, class properties, operations, number of parameters of operations and the complexity of operation constraints. The complexity of snapshot transitions generation is proportional to the number of instances of the scenarios. The complexity to check consistency in USE depends on the number of snapshot transitions in the scenario, the number of objects in the before and after snapshots, and the complexity of invariants in the snapshot transition model.

Chapter 5

Implementation

This chapter describes the tool we developed to support the Scenario-based UML Design Analysis technique. The tool was developed using the Eclipse Modeling Framework and Kermeta, a metamodeling programming language.

Section 5.1 describes the tool's architecture. Section 5.2 describes the components that implement the snapshot transition model (STM) generation procedure. Section 5.3 describes the component that implements the scenario generation procedure. Section 5.4 describes component that implements how the USE tool is used to check consistency between the snapshot transition model and snapshot transitions.

5.1 Tool architecture

The architecture of the tool is shown in Fig. 5.1. The architecture consists of three layers:

- The EMF layer: This layer includes Ecore Metamodel and Ecore Model Editor that allows editing Ecore models in Eclipse.
- The Kermeta layer: This layer includes Kermeta package and an OCL Metamodel called OCLCST [Garcia07]. The OCL Metamodel is used to load and transform OCL operation specifications.
- The tool layer: This layer includes all components we implemented in the Scenario-based UML Design Analysis tool package.

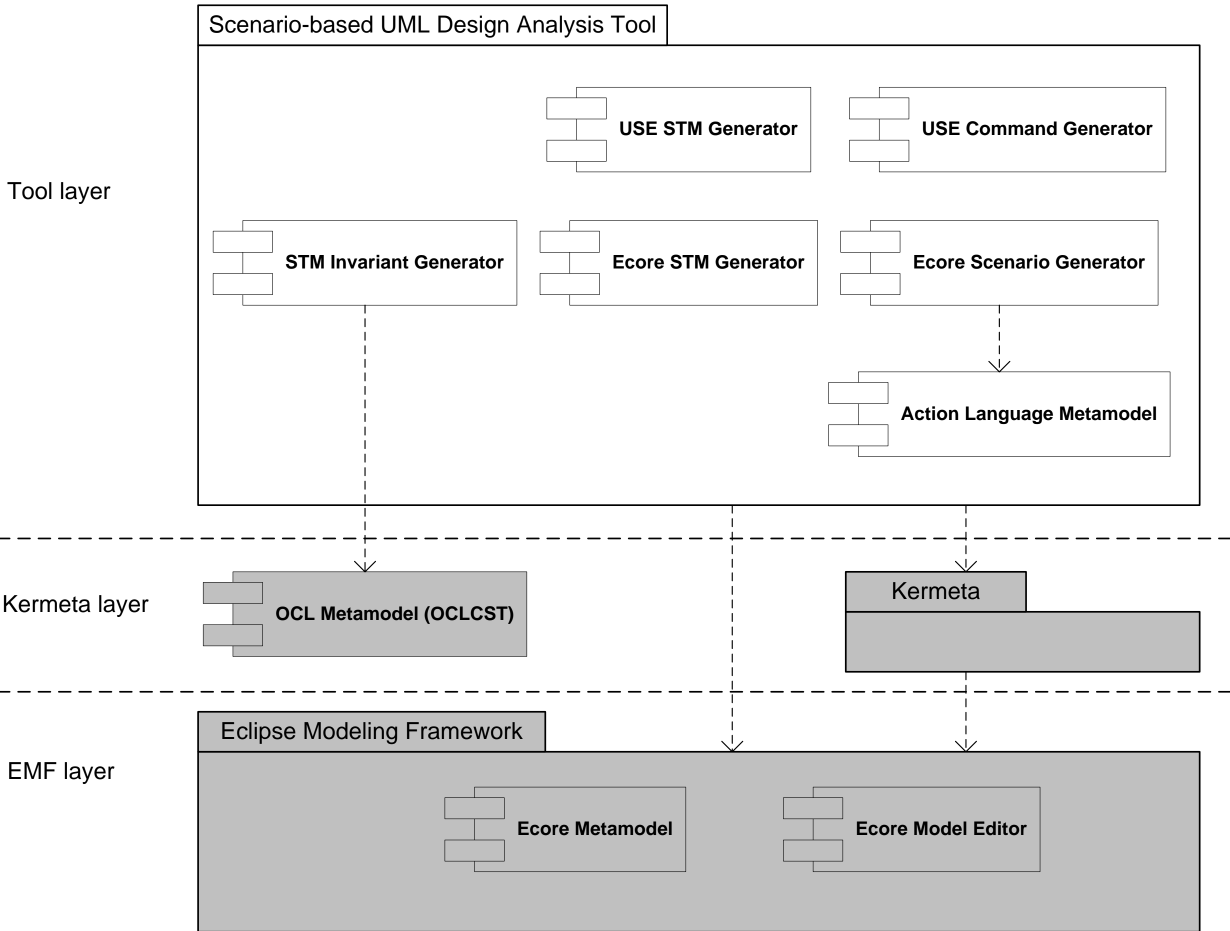


Figure 5.1. Overview of the Scenario-based UML Design Analysis tool

The Scenario-based UML Design Analysis tool package uses existing platforms and metamodels including Kermeta, EMF and an OCL metamodel, OCLCST, which are all grayed out in Fig. 5.1. The tool contains the following components:

- Ecore STM Generator: This is an Eclipse plugin that generates snapshot transition model in Ecore.
- STM Invariant Generator: This is an Eclipse plugin that transforms OCL operation specifications to invariants of the snapshot transition model.

- Ecore Scenario Generator: This is an Eclipse plugin that generates scenario snapshot transitions in Ecore.
- Action language Metamodel: This defines a language used to specify scenarios using actions. The metamodel is used by the Ecore Scenario Generator.
- USE STM Generator: This is an Eclipse plugin that transforms Ecore snapshot transition model to USE.
- USE Command Generator: This is an Eclipse plugin that transforms Ecore scenario snapshot transitions to USE commands.

To use the tool, the designer creates Ecore design class diagram using Ecore Model Editor and OCL operation specifications using a text editor. The verifier either creates scenario specifications using an action language defined by the Action Language Metamodel, or explicitly specifies a scenario. The explicit scenario specification includes a sequence of snapshots created using Ecore model editor, and a sequence of operations using a text editor.

The Ecore design class diagram is transformed to an Ecore snapshot transition model using the Ecore STM Generator, and OCL operation specifications are transformed to invariants of the Ecore snapshot transition model using the Ecore STM invariants Generator. The USE STM Generator is then used to transform the Ecore snapshot transition model and invariants to USE.

The Ecore Scenario Generator is used to generate Ecore snapshot transitions from the verifier’s scenario specifications. And the USE Command Generator is used to generate USE commands to construct USE snapshot transitions from the Ecore snapshot transitions.

Finally the USE tool is used to load the USE STM, run USE commands to construct USE snapshot transitions, and check consistency between the USE STM and snapshot transitions.

5.2 The STM Generator and STM Invariant Generator

The input to the Ecore STM generator is an Ecore design class diagram created by the designer. Fig. 5.2 shows an example RBAC Ecore design class diagram. And Fig. 5.3 shows the design class diagram in the Ecore Model Editor.

All OCL operation specifications are created using a text editor. For example, the following is OCL specification of operation *User::AssignRole*.

```
context User::AssignRole(r : Role) : Void
pre: self.UserAssign->excludes(r)
post: self.UserAssign = self.UserAssign@pre->including(r) and
    self.UUID = self@pre.UUID
```

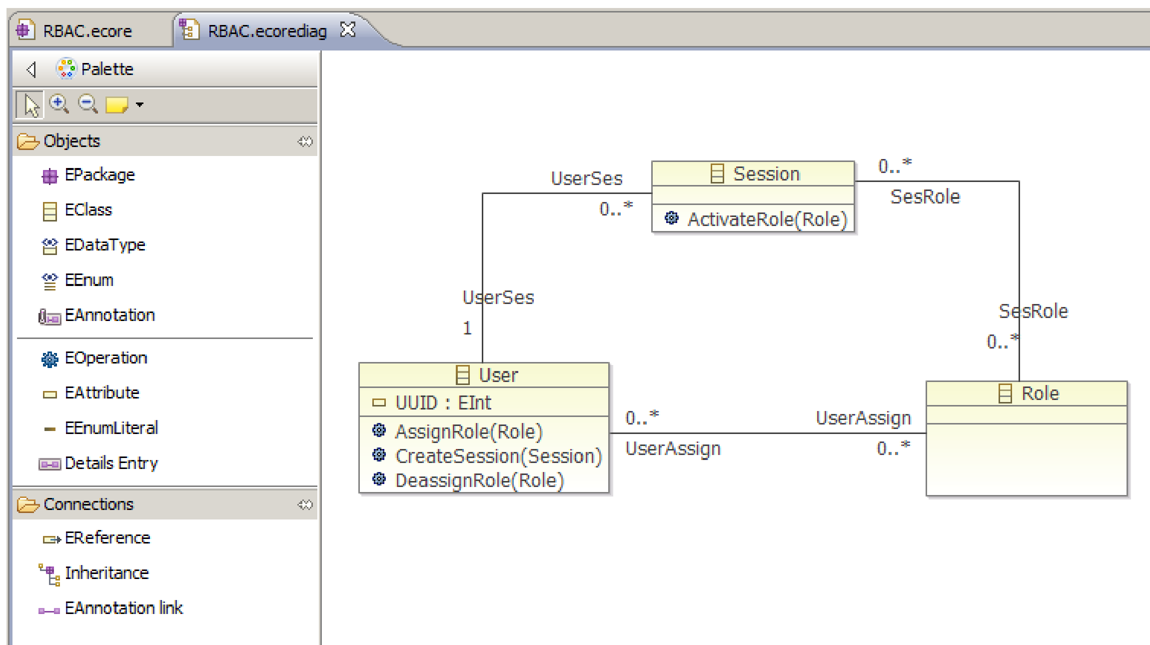


Figure 5.2. RBAC Ecore design class diagram (diagram view)

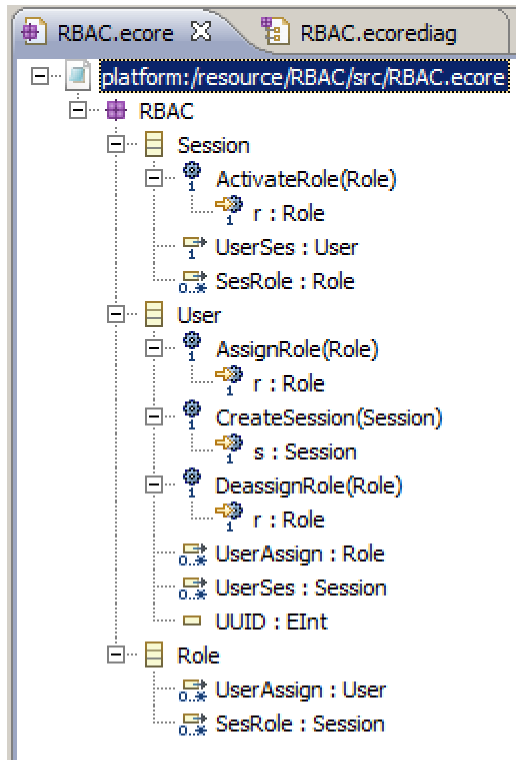


Figure 5.3. RBAC Ecore design class diagram (tree view)

5.2.1 Generating Ecore snapshot transition diagram

The Ecore STM Generator first generates an Ecore snapshot diagram from the input Ecore design class diagram. Based on the Ecore metamdel, the Ecore STM Generator loads the Ecore design class diagram, applies model transformation rules and generates an Ecore snapshot transition diagram. Fig. 5.4 shows the snapshot transition diagram of the RBAC model. In the Ecore snapshot transition diagram the Snapshot class is linked to all classes in the input Ecore design class diagram using composition. The Snapshot class is not a composite structure as described in Fig. 4.3 because composite structure is not supported in Ecore metamodel, we use Ecore composition reference to simulate UML composite structure.

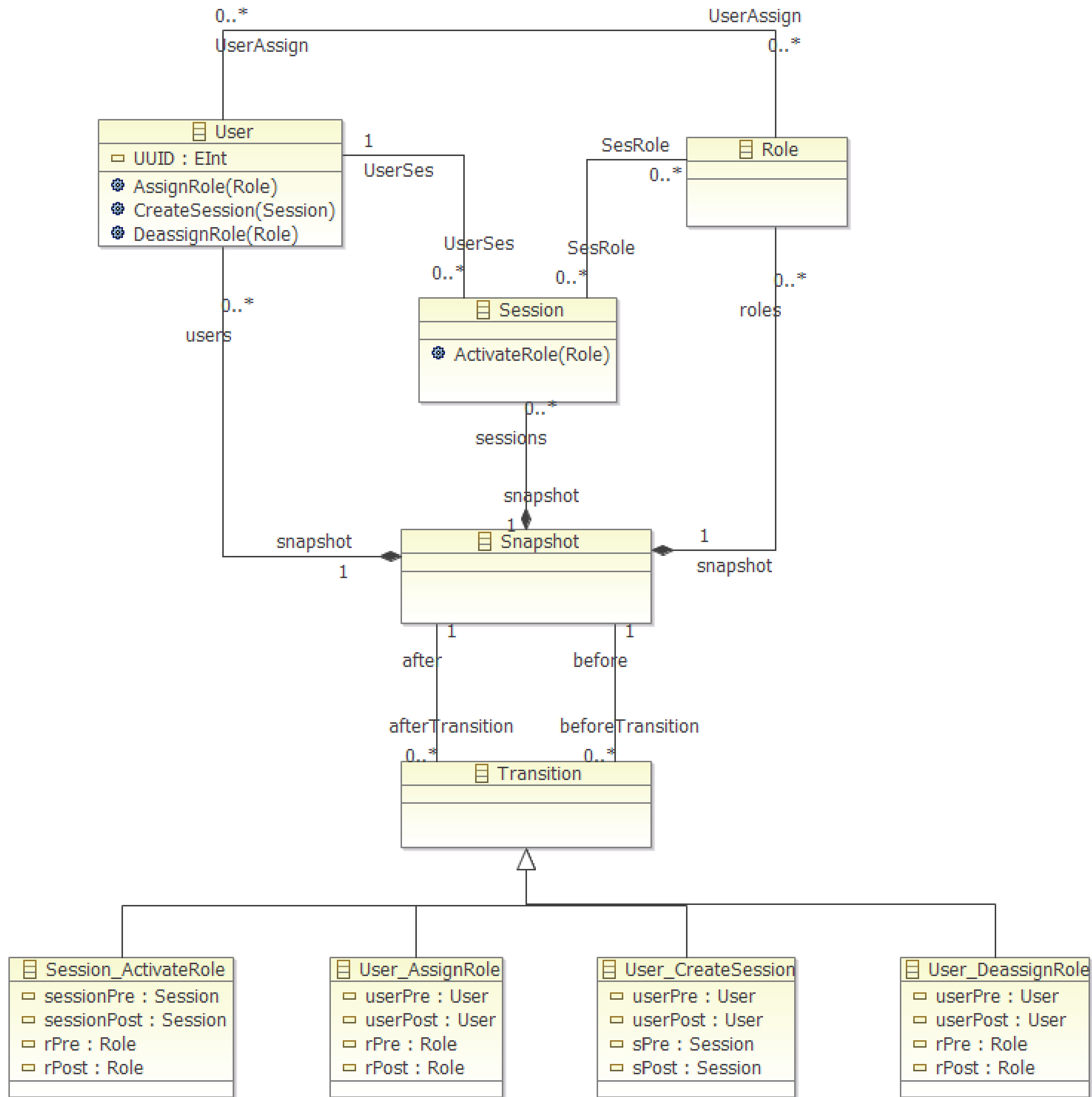


Figure 5.4. RBAC Ecore snapshot transition diagram

Fig. 5.5 shows the Kermeta-based algorithm used to generate the snapshot transition diagram (STM). The algorithm creates a snapshot class and links the snapshot class to all other classes in the design class model, creates a transition class that refers to a before and after snapshot, creates transition classes for each operation and finally returns the transformed model.

```
Snapshot Transition Model generation algorithm
Input: inputModel : EPackage
Output: EPackage
Steps:
1. Create a snapshot class
snapshot = EClass.new
2. Add composition reference between each class and the snapshot class
foreach EClass cls in inputModel do
a) Create a composition reference that points to the cls class
   composition = EReference.new
   composition.eType = cls
b) Create a clsref reference that points to the snapshot class
   clsref = EReference.new
   clsref.eType = snapshot
c) Set eOpposite attribute for the clsref and composition classes
   clsref.eOpposite = composition
   composition.eOpposite = clsref
3. Create a transition class
   transition = EClass.new
4. Setup the before reference between the transition and snapshot classes
   before = EReference.new
   before.eType = snapshot
   beforeTrans = EReference.new
   beforeTrans.eType = transition
   before.eOpposite = beforeTrans; beforeTrans.eOpposite = before
5. Setup the after reference between the transition and snapshot classes
   after = EReference.new
   after.eType = snapshot
   afterTrans = EReference.new
   afterTrans.eType = transition
   after.eOpposite = afterTrans; afterTrans.eOpposite = after
6. Create snapshot transition classes for each operation
foreach EClass cls in inputModel do
   foreach EOperation op in cls do
   // Create a transition class for the operation
   a) opcls = EClass.new
      opcls.name = cls.name + "_" + op.name
   // Add paramaters of the op operation as attributes of the snapshot transition class
   b) For each EParameter param do
         attrPre = EAttribute.new
         attrPre.name = param.name + "Pre"
         attrPre.eType = param.eType
         attrPost = EAttribute.new
         attrPost.name = param.name + "Post"
         attrPost.eType = param.eType
7. return inputModel
```

Figure 5.5. Snapshot transition model generation algorithm

5.2.2 Transforming OCL operation specifications

The STM Invariant Generator uses an OCL metamodel to parse the OCL operation specifications and transform them to invariants of snapshot transition subclasses.

The STM Invariant Generator transforms the original OCL operation specifications to invariants. For example, the following invariants are transformed from *User::AssignRole* operation:

```
context User_AssignRole
inv frompre: (userPre.UserAssign.ID->excludes(rPre.ID))
inv frompost: ((userPost.UserAssign.ID =
(userPre.UserAssign.ID->including(rPost.ID))) and (userPost.UUID =
userPre.UUID))
```

The ID is an internal attribute that is added by the tool to each class in the Ecore class diagram. It is used to identify the same object across multiple snapshots in a scenario. For example, in an RBAC role assignment and activation snapshot transitions, each snapshot contains a copy cashier role instance with the same ID.

In order to generate *frame constraints*, the scope specification of operation User::AssignRole is specified below:

```
Operation: User::AssignRole
Modifier_Class: User, Role
Modifier_Attribute:
Modifier_Link: User.UserAssign, Role.UserAssign
```

The scope specification restricts that only the UserAssign references between User and Role classes are changed after the operation is invoked.

Based on the operation scope specification, the STM Invariant Generator generates the following frame constraints for *User::AssignRole* operations. The frame constraints are part of the invariants of *User_AssignRole* class:

```
before.sessions->forAll(o1 | after.sessions->exists(o2 | o1.ID =
o2.ID))
and before.sessions->forAll(o1 | after.sessions->exists(o2 | o1.ID
= o2.ID and o1.UserSes.ID = o2.UserSes.ID and o1.SesRole.ID =
o2.SesRole.ID and o1.snapshot.ID = o2.snapshot.ID))
and before.users->forAll(o1 | after.users->exists(o2 | o1.ID = o2.ID
and o1.UUID = o2.UUID))
and before.users->forAll(o1 | after.users->exists(o2 | o1.ID = o2.ID
and o1.UserSes.ID = o2.UserSes.ID and o1.snapshot.ID =
o2.snapshot.ID))
and before.roles->forAll(o1 | after.roles->exists(o2 | o1.ID =
o2.ID))
and before.roles->forAll(o1 | after.roles->exists(o2 | o1.ID = o2.ID
and o1.SesRole.ID = o2.SesRole.ID and o1.snapshot.ID =
o2.snapshot.ID))
```

Fig. 5.6 shows the main Kermeta-based algorithm for transforming OCL specifications. The algorithm takes an Ecore design class model and parsed OCL operation specifications as input. The top-level package declaration of the parsed OCL specifications includes a set of class invariants and operation specifications. The algorithm recursively visits each OCL class invariant, operation pre and post condition specification body, transforms the OCL specifications to invariants and writes the transformed OCL to an output file.

```
OCL operation specification transformation algorithm
Input:    ecoreModel : EPackage        // Ecore class model
          parsedOcl: Resource          // Parsed OCL
Output: outputfile: File               // Transformed OCL file
Steps:
1. Create a PrePost2InvVisitor instance
PrePost2InvVisitor visitor = PrePost2InvVisitor.new
2. Visit top-level package declaration of the parsed OCL instances
PackageDeclarationCS pkg =  getPackageDeclaration(parsedOcl.instances)
foreach ContextDeclCS contextDecl in pkg.contextDecls do
a) Visit class invariants
if contextDecl isInstanceOf ClassifierContextDeclCS
   contextDecl.accept(visitor)
endif
b) Visit operation specifications
if contextDecl isInstanceOf OperationContextDeclCS
   i) Visit operationCS, including operation name, parameters and return type
   OperationCS opCS = contextDecl.operationCS
   opCS.accept(visitor)
   ii) Visit each operation pre and post condition specification body
   foreach PrePostOrBodyDeclCS ppbd in contextDecl.prePostOrBodyDecls do
      // Transform the ppbd to invariant
      write("inv from" + ppbd.kind.name.toString + ": ")
      // Visit OperationCallExpCS
      OperationCallExpCS opCallExpCS = ppbd. expressionCS
      opCallExpCS.accept(visitor)
      // Write new line
      writeln("")
endif
3. Write the visited OCL to output file
WriteOutputFile(outputfile, visitor)
```

Figure 5.6. OCL operation specification transformation main algorithm

5.2.3 Generating USE snapshot transition model

After the Ecore snapshot transition model and OCL operation specifications are generated, the USE STM Generator visits the Ecore snapshot transition model and mechanically transforms it to a USE snapshot transition model based on USE grammar.

The USE grammar is close to UML and Ecore. For example, below is USE specifications of the *User* class and the *UserAssign* association between the *User* class and the *Role* class:

```
class User
attributes
 UUID : Integer
 ID : Integer
operations
 AssignRole(r : Role)
 CreateSession(s : Session)
 DeassignRole(r : Role)
end

association UserAssignUserAssign4 between
 Role[0..*] role UserAssign
 User[0..*] role UserAssign
end
```

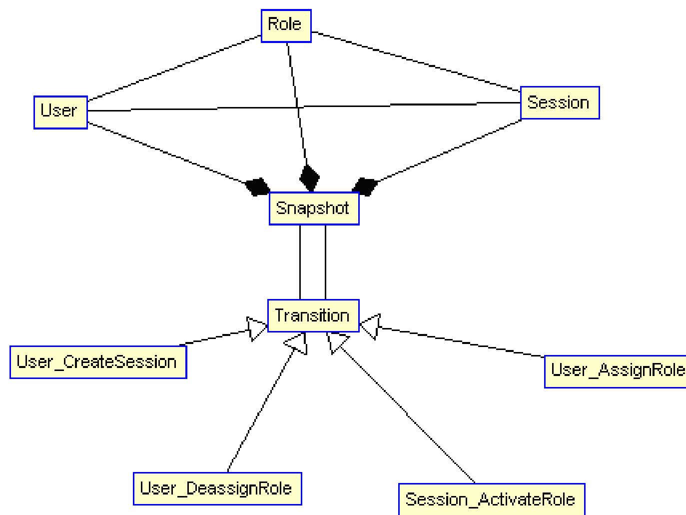


Figure 5.7. USE snapshot transition model

Fig. 5.7 shows the USE snapshot transition model of the RBAC application. It contains four transitions for the operations defined in the original RBAC design class model.

5.3 The Scenario Generator

The input of the Ecore Scenario Generator is the verifier's scenario specifications. The generator allows the verifier to create two types of scenario specifications: explicit scenario specification which includes a sequence of snapshots and operation calls; action language specification which specifies a sequence of actions to construct each snapshot in the scenario.

5.3.1 Explicit specification of scenarios

The verifier can explicitly specify a scenario as a sequence of snapshots and operation calls. A snapshot is an instance of the design class diagram. An operation call is defined by the operation name and parameter values.

Fig. 5.8 shows an example of explicit specification of an RBAC scenario. The scenario starts with an initial snapshot which contains a user instance *bob* and a role instance *cashier*. Operation 1 assigns *cashier* role to *bob*. After operation 1 is called in snapshot 1 the *cashier* role and *bob* is associated. Operation 2 creates a session instance *s1* from *bob*. After operation 2 is called in snapshot 2 session *s1* is linked to *bob*. Operation 3 activates the *cashier* role in session *s1*. After operation 3 is called in snapshot 3 the *cashier* role is linked to session *s1*. Operation 4 deactivates the *cashier* role. After operation 4 is called in snapshot 4 the link between *cashier* and *s1* is removed.

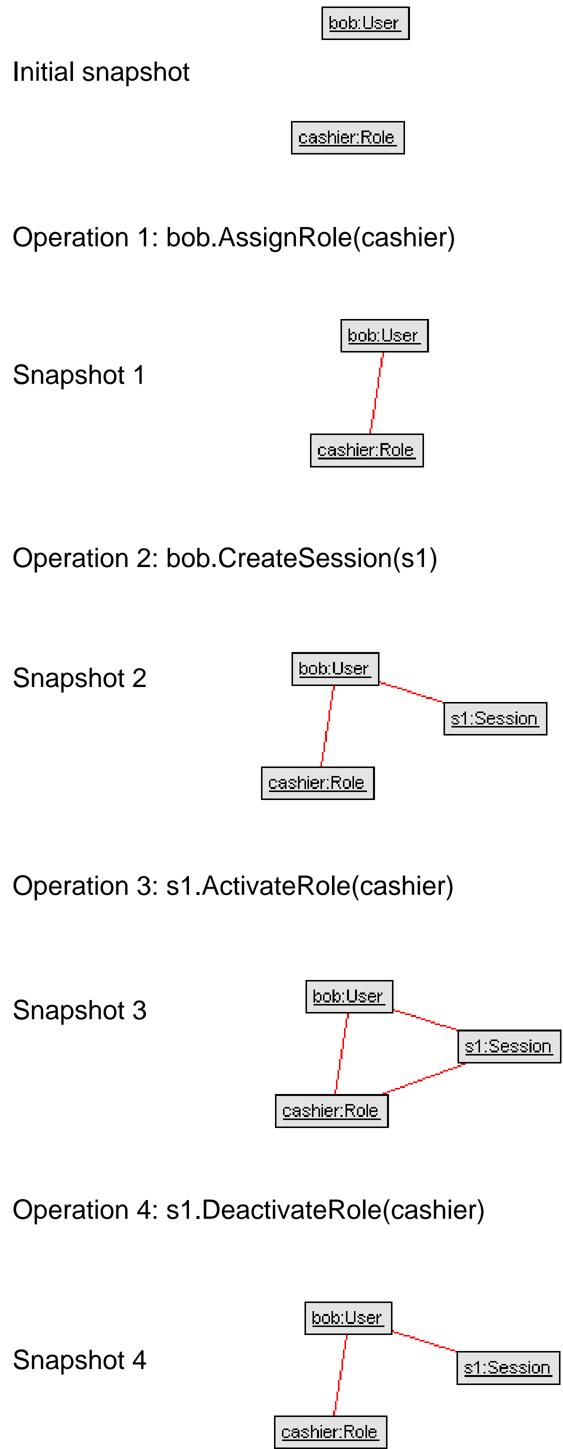


Figure 5.8. Explicit specification of an RBAC scenario

5.3.2 Action language specification of scenarios

The *action specification language* is used to specify a scenario as a sequence of actions. The language is defined using a metamodel as described in Fig. 5.9.

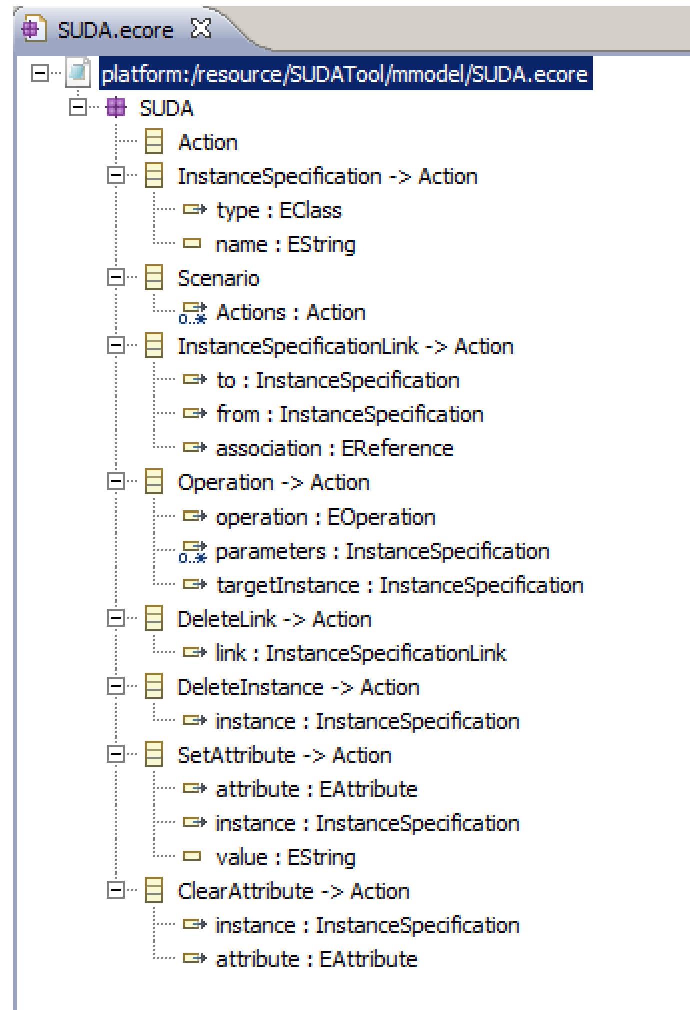


Figure 5.9. Metalmodel of the action specification language

In the metamodel a scenario is defined as a sequence of actions. An action contains the following subclasses:

- instance specification action: creates an object of a class
- delete instance action: destroys an object
- set attribute action: sets an attribute for an object

- clear attribute action: clears an attribute for an object
- instance specification link action: creates a links between two objects
- delete link action: removes an link
- operation action: specifies an operation call, including the operation name, target object and parameters

Fig. 5.10 shows an example of an RBAC scenario specified using the action specification language. The RBAC scenario is described in natural English as below:

1) Snapshot: User *bob*, Role *cashier* and Session *s1*
2) Operation: assign *cashier* role to *bob*
3) Snapshot: *bob* and *cashier* are linked
4) Operation: *bob* creates session *s1*
5) Snapshot: *bob* and *s1* are linked
6) Operation: *cashier* is activated in session *s1*
7) Snapshot: *s1* and *cashier* are linked
8) Operation: *bob* de-assigns role *cashier*
9) Snapshot: *bob* and *cashier* are de-linked, *s1* and *cashier* are de-linked

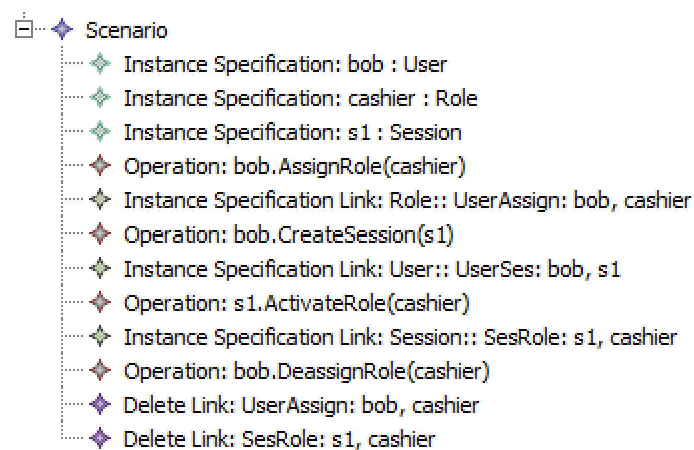


Figure 5.10. Action specification of an RBAC scenario

5.3.3 Generating snapshot transitions

For each operation call in the scenario, the Ecore Scenario Generator finds the operation that matches it in the Ecore design class model, creates a corresponding snapshot transition object and links the snapshot transition object to two snapshots. The output is a sequence of snapshot transition instances.

Objects and links in each snapshot are generated from the scenarios. If the scenario is specified directly, the Ecore Scenario Generator clones objects and links from the snapshots in the scenario. Objects with the same name are assigned the same ID. If the scenario is specified using the action language, the tool first interprets the actions to generate the initial snapshot. A unique ID is assigned for each object created by the action language. To generate a snapshot after an operation call, the tool clones all the objects and links from the before snapshot, and then applies the actions to the after-snapshot.

5.3.4 Generating USE commands

The USE Command Generator processes the snapshot transitions and generates a sequence of *USE commands*. USE commands are actions that are used to create and manipulate an instance model in USE. Basic actions include creating an object, setting an attribute of an object and linking two objects. The generated USE commands are used to create snapshot transitions in USE.

For example, the following USE commands create a snapshot object *snapshot_1*, create a user object *bob_1,* set object ID of *bob_1* as 0, and finally create a role object *cashier_1* and set object ID as 1.

```
!create snapshot_1 : Snapshot
!create bob_1 : User
!set bob_1.ID := 0
```

```
!create cashier_1 : Role
!set cashier_1.ID := 1
```

5.4 USE consistency check

In this step, the USE snapshot transition model and OCL specifications and the USE commands are input to USE. The USE commands are used to generate USE snapshot transitions.

Fig. 5.11 shows the USE snapshot transitions generated by the USE commands. There are four operations and five snapshots in the USE snapshot transitions model. The initial snapshot contains role cashier, user *bob* and session *s1*. The first operation *User_AssignRole* adds a link between *bob* and *cashier*. The second operation *User_CreateSession* adds a link between *s1* and *bob*. The third operation *Session_ActivateRole* adds a link between *s1* and *cashier*. The last operation *User_DeassignRole* deletes the link between *cashier* and *bob* and the link between *cashier* and *s1*.

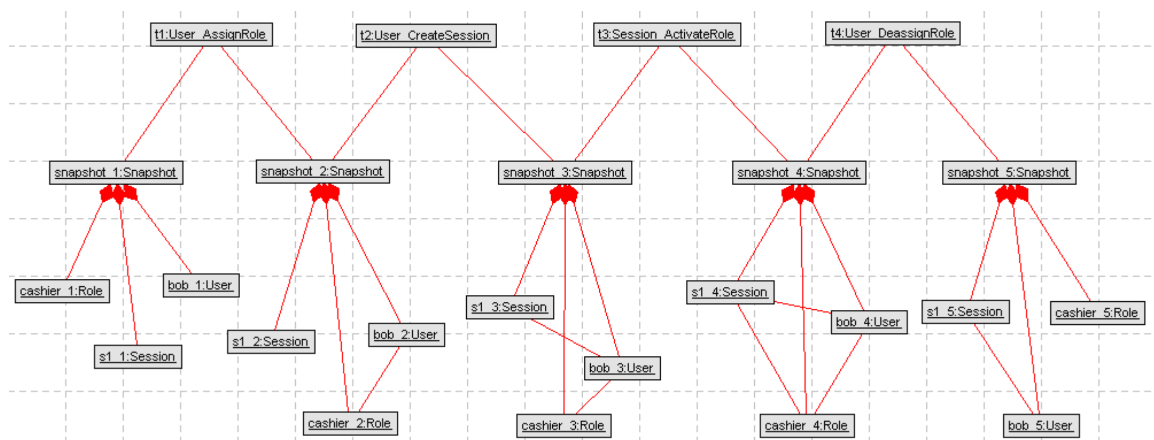


Figure 5.11. USE snapshot transitions

After checking consistency between the USE snapshot transition model in Fig. 5.7 and USE snapshot transitions in Fig. 5.11, USE shows the pre and post conditions of *User_DeassignRole* are violated in the snapshot transitions (Fig. 5.12). Take the pre-condition for example, the

pre-condition contains two *sub-expressions:self.UserAssign->includes(r)* and *self.UserSes.SesRole->excludes(r).* The first sub-expression requires that the role is assigned to the user before de-assigned which is evaluated as true. The second sub-expression requires that the role is not activated in any user sessions before de-assigned which is false.

Figure 5.12. USE consistency checking

Chapter 6

Demonstration Case Studies

This chapter presents exemplar applications of the Scenario-based UML Design Analysis technique on design class models for two systems: a Train Management System model and a Generalized Spatio-Temporal RBAC model. The two demonstration case studies will illustrate how design inconsistencies can be uncovered using the Scenario-based UML Design Analysis technique.

6.1 The Train Management System model

The Train Management System (TMS) is used to monitor train traffic in a train network. The train network consists of trains and stations. There can be zero or more one-way routes between any two stations. Each route is divided into segments. Each segment has two sensors: an entry sensor which detects trains as they enter the segment and an exit sensor which detects trains as they leave the segment. Each segment has a traffic light at the end. The train can only enter the next segment when this traffic light of current segment is green. A train may have a journey. A journey consists of a sequence of routes and stations at which the train will stop. A journey is valid if it does not traverse any closed routes or closed stations and it does not stop at stations that are not on the routes of the journey.

The length and speed of trains is ignored in this system. It is also important to note that the segments are not contiguous; between any two segments in a route there is an non-monitored section, that is a section that does not have input and output sensors. Thus one cannot assume

that a train leaving a segment (a monitored section) in one time instance is in the next segment (next monitored section) in the next time instance.

In the following subsections we describe design class model of the TMS, scenarios and inconsistencies between the design class model and scenarios identified using the Scenario-based UML Design Analysis technique.

6.1.1 The TMS design class model

Fig. 6.1 shows the design class diagram of the Train Management System. The *Train* class has two association ends with the *Segment* class: *currentSeg* refers to the current segment of the train and *lastExitedSegment* refers to the last segment that the train has exited. The Segment class has two association ends with the *SensorHandler* class: the *entrySensor* refers to the *SensorHandler* at the entry of the segment and the *exitSensor* refer to the SensorHandler at the exiti of the segment. Each *Segment* can have a previous *Segment* and a next *Segment*. The *TrainManager* class links to all *SensorHandler* and *Train* objects. The *Train* has multiple *Journeys* and each *Journey* has multiple *Routes*. The Journey has multiple *stopStations*. Each *Route* has a *beginStation* and an *endStation*. Each *Station* has multiple *segments*.

Below are OCL specifications of four major operations: *Train::OnSegmentEnter, Train::OnSegmentExit,* *Segment::OnTrainExit* and *Segment::OnTrainExit*. The *Train::OnSegmentEnter* operation requires, as a postcondition that *currentSeg* of the train equals the segment it is entering. The *Train::OnSegmentExit* operation requires that if the traffic light of the current segment is not green then the train becomes runaway train and that the route of the current segment becomes closed.

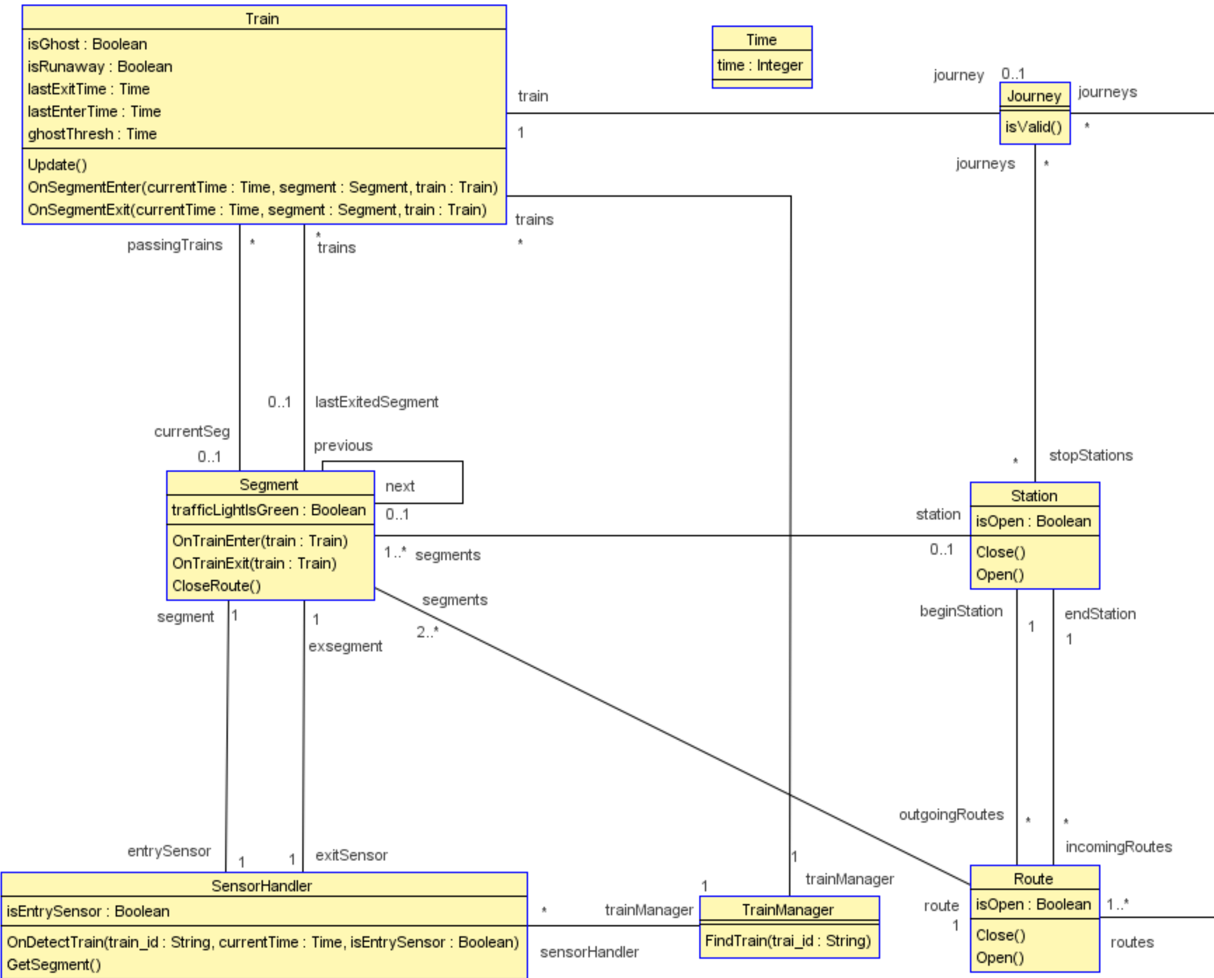


Figure 6.1. TMS design class diagram

```
context Train::OnSegmentEnter(currentTime:Time, segment:Segment,
train:Train)
pre pre1: train = self
// After the train enters the segment
// (1) The segment becomes current segment of the train.
post post1: self.currentSeg = segment
// (2) Last enter time of the train is equal to currentTime
post post2: self.lastEnterTime = currentTime

context Train::OnSegmentExit(currentTime:Time, segment:Segment,
train:Train)
pre pre1: self = train
```

```
// After the train exits the segment
// (1) If the traffic light of current segment is not green before the train
// exits the segment, the train becomes a runaway train and the route that
// contains the train will be closed.
post post1:(not self.currentSeg.trafficLightIsGreen@pre)
implies (self.isRunaway and  not self.currentSeg.route.isOpen)
// (2) Last enter time of the train is equal to currentTime.
post post2: self.lastExitTime = currentTime
// (3) Previous segment becomes last exited segment of the train.
post post3: self.lastExitedSegment = self.currentSeg@pre

context Segment::OnTrainEnter(train:Train)
// Before the train enters the segment, the segment must be current segment
// of the train.
pre pre1: train.currentSeg = self
// After the train enters the segment, the train will be one of the passing
// trains of the segment.
post post1: self.passingTrains->includes(train)

context Segment::OnTrainExit(train:Train)
// Before the train exits the segment, the train must be one of the passing
// trains of the segment.
pre pre1: self.passingTrains->includes(train)
// After the train exits the segment
// (1) The train should not be one of the passing trains of the segment.
post post1: self.passingTrains->excludes(train)
// (2) The traffic light of the segment should not be green.
post post2: not self.trafficLightIsGreen
// (3) If the previous segment exists, also the route of the segment is
// open, and there are no passing trains on the segment, then the traffic
// light of previous segment must be green, otherwise it is not green.
post post3: if (self.previous->notEmpty() and self.route.isOpen and
self.passingTrains->isEmpty() ) then (self.previous.trafficLightIsGreen)
else (not self.previous.trafficLightIsGreen) endif
```

```
// (4) If the previous and next segment exists, also the route of the segment
// is open, and there are no passing trains on the segment, then the traffic
// light of previous segment must be green, otherwise it is not green.
post post4: if (self.previous->notEmpty() and self.next->notEmpty() and
self.route.isOpen and self.passingTrains->isEmpty()) then
(self.previous.trafficLightIsGreen) else
(not self.previous.trafficLightIsGreen endif
// (5) If the previous segment exists but the next segment does not exist,
// then each segment at the station of the segment should not have next
// segment or have green traffic light
post post5: (self.previous->notEmpty() and self.next->isEmpty())
implies self.station.segments->forAll(st: Segment | st.next->isEmpty()
and not st.trafficLightIsGreen)
// (6) If the previous segment does not exist but the next segment exists,
// then for each segment at the station of the segment, if it has no previous
// segment or passing trains, then for each segment at the station of this
// segment, if the route of the segment is open and the segment has no next
// segment, then tranffic light of the segment is green
post post6: (self.previous->isEmpty() and self.next->notEmpty())
implies (self.station.segments->forAll(st: Segment|
st.previous->isEmpty() and st.passingTrains->isEmpty()) implies
(self.station.segments->forAll(st1: Segment | st1.route.isOpen and
st1.next->isEmpty() implies st1.trafficLightIsGreen)))
```

6.1.2 TMS Scenario one

In this scenario a train *t1* is on segment *seg1* initially, firstly it exits *seg1* so that *seg1* becomes last exited segment of the train, then it enters next segment *seg2* so that *seg2* becomes current segment of the train and the traffic light of *seg1* is no longer green. The scenario contains three snapshots: snapshot 1.1 shown in Fig. 6.2, snapshot 1.2 shown in Fig. 6.3 and snapshot 1.3 shown in Fig. 6.4. Operation *t1:onSegmentExit(time,seg1,t1)* is called between snapshot 1.1 and

snapshot 1.2, operation *t1:onSegmentEnter(time,seg2,t1)* is called between snapshot 1.2 and snapshot 1.3.

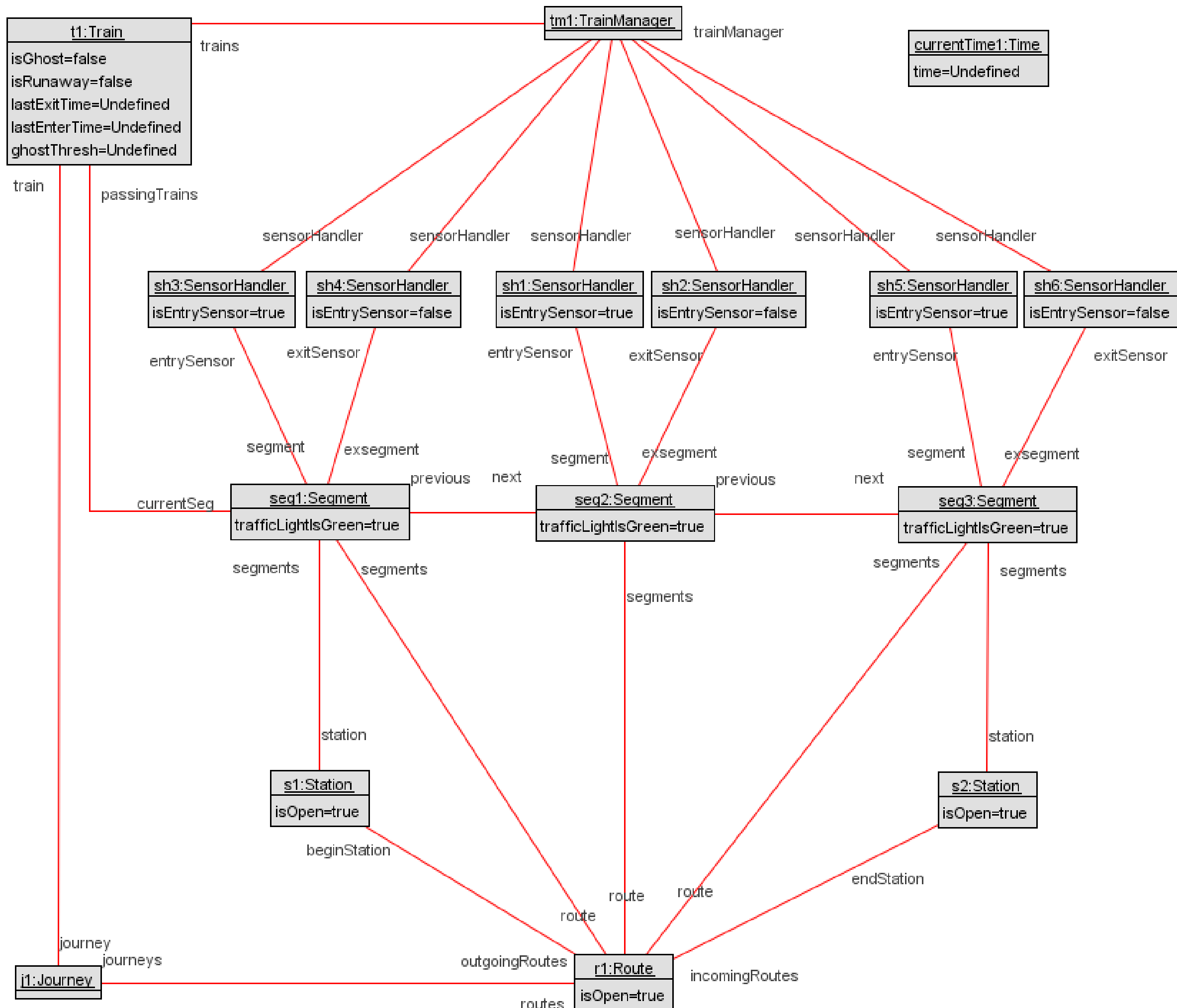


Figure 6.2. TMS snapshot 1.1

Snapshot 1.1 in Fig. 6.2 contains one *Train* object *t1*. *t1* has a journey *j1* and *j1* has a route *r1*. *r1* has three segments *seg1*, *seg2*, *seg3*. *currentSeg* of *t1* is *seg1*. Traffic lights of three segments are all green.

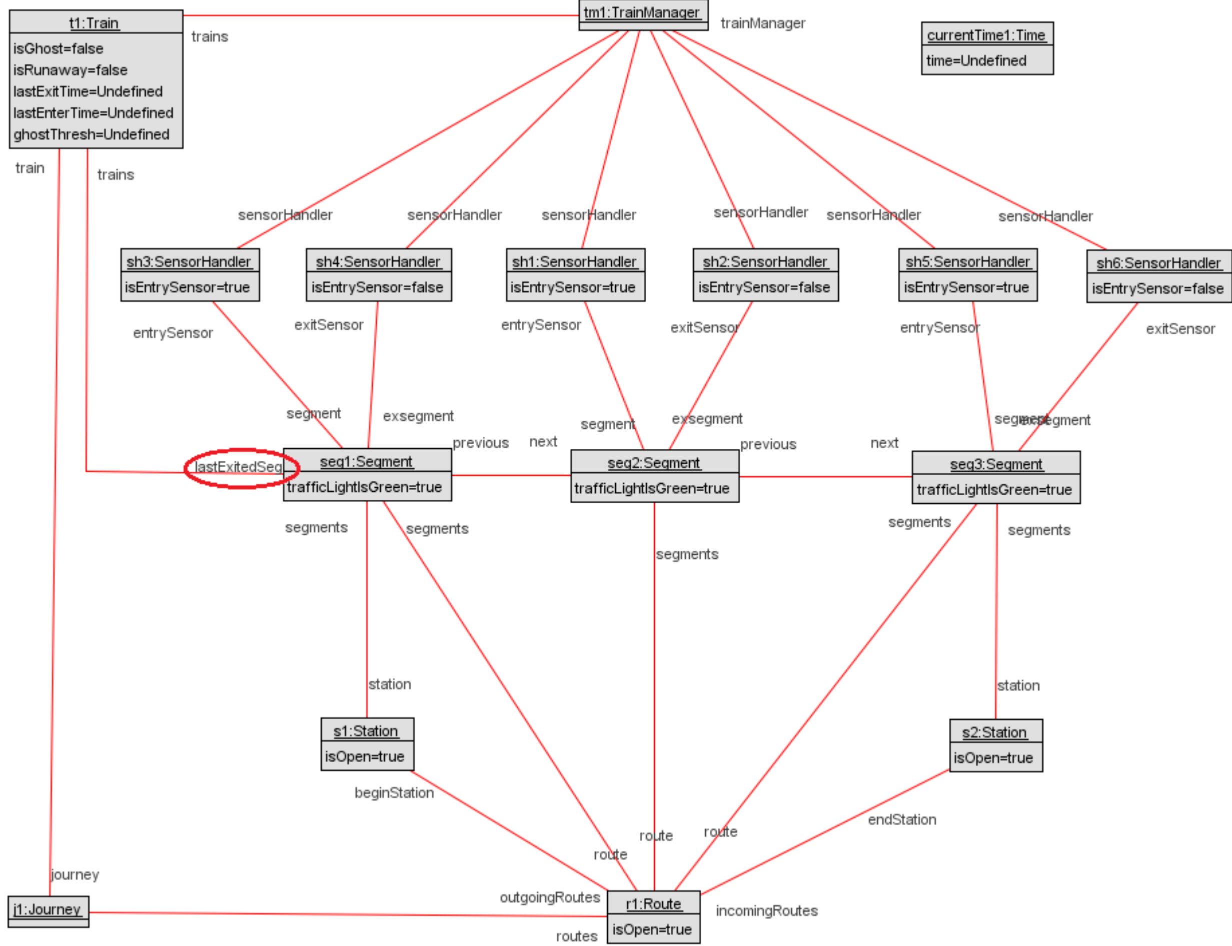


Figure 6.3. TMS snapshot 1.2

After operation *t1:onSegmentExit(time,seg1,t1)* is called, in snapshot 1.2 (Fig. 6.3) the association end from train *t1* to segment *seg1* becomes *lastExitedSeg*. The change in association end lastExitedSeg is circled in Fig 6.3. Before the operation is called, the association end is *currentSeg*.

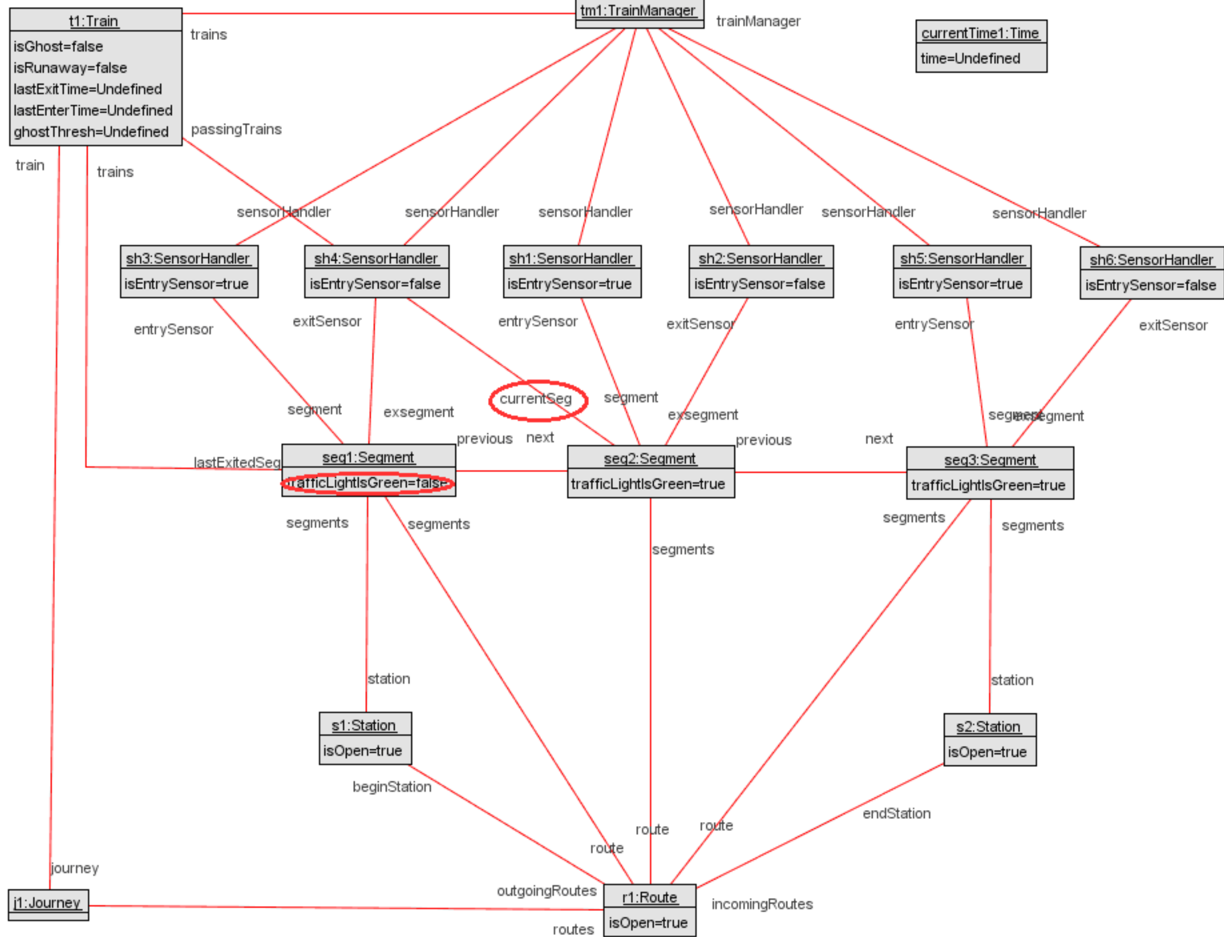


Figure 6.4. TMS snapshot 1.3

After operation *t1:onSegmentEnter(time,seg2,t1)* is called, in snapshot 1.3 (Fig. 6.4) a new association between train *t1* and segment *seg2* is added, *seg2* is now currentSeg *of t1*. Also *trafficLightIsGreen* attribute of *seg1* becomes false. The changes are circled in Fig 6.4.

We analyze the scenario using the Scenario-based UML Design Analysis tool. The analysis result shows an inconsistency that the *trafficLightIsGreen* attribute should not be *false* in snapshot 1.3, operation *Train::OnSegmentExit* does not explicitly specify that the *trafficLightIsGreen* should be changed or not, so we added frame constraints to ensure the attribute should not be changed after the operation is called.

6.1.3 TMS Scenario two

In this scenario a train *t1* is on segment *seg1* initially and the traffic light of *seg1* is not green, firstly it exits *seg1* so that *seg1* becomes last exited segment of the train and the train becomes a runaway train, then it enters next segment *seg2* so that *seg2* becomes current segment of the train. The scenario contains three snapshots, snapshot 2.1 shown in Fig. 6.5, snapshot 2.2 shown in Fig. 6.6 and snapshot 2.3 shown in Fig. 6.7. Operation *t1:onSegmentExit(time,seg1,t1)* is called between snapshot 2.1 and snapshot 2, operation *t1:onSegmentEnter(time,seg2,t1)* is called between snapshot 2.2 and snapshot 3.

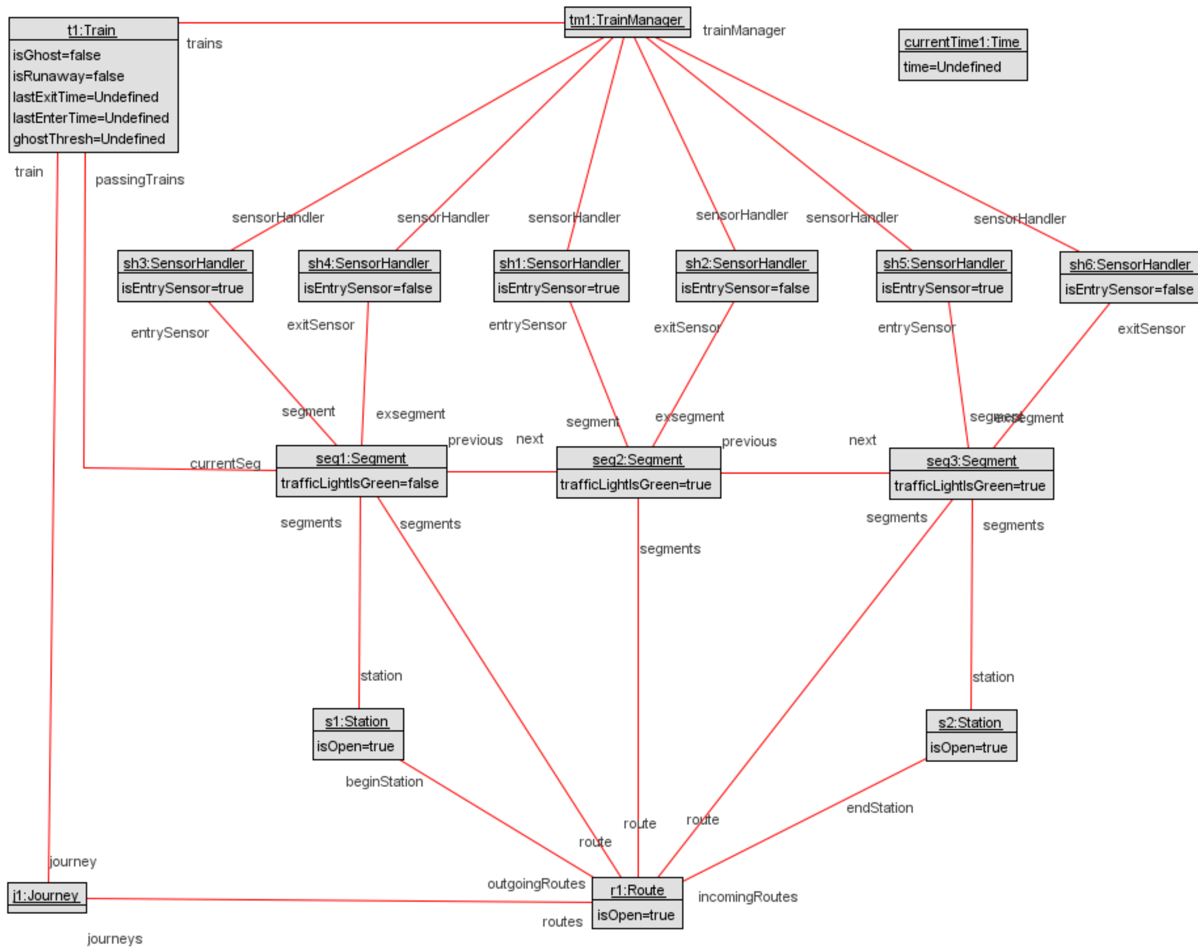


Figure 6.5. TMS snapshot 2.1

Snapshot 2.1 in Fig. 6.5 contains one *Train* object *t1*. *t1* has a journey *j1* and *j1* has a route *r1*. *r1* has three segments *seg1*, *seg2*, *seg3*. *currentSeg* of *t1* is *seg1*. Traffic lights of segments are all green except for segment *seg1*.

After operation *t1:onSegmentExit(time,seg1,t1)* is called, in snapshot 2.2 (Fig. 6.6) the association end from train *t1* to segment *s1* becomes *lastExitedSeg*. Before the operation is called, the association end is *currentSeg.* Also *isRunaway* attribute of *t1* becomes true. The two changes are circled in Fig. 6.6.

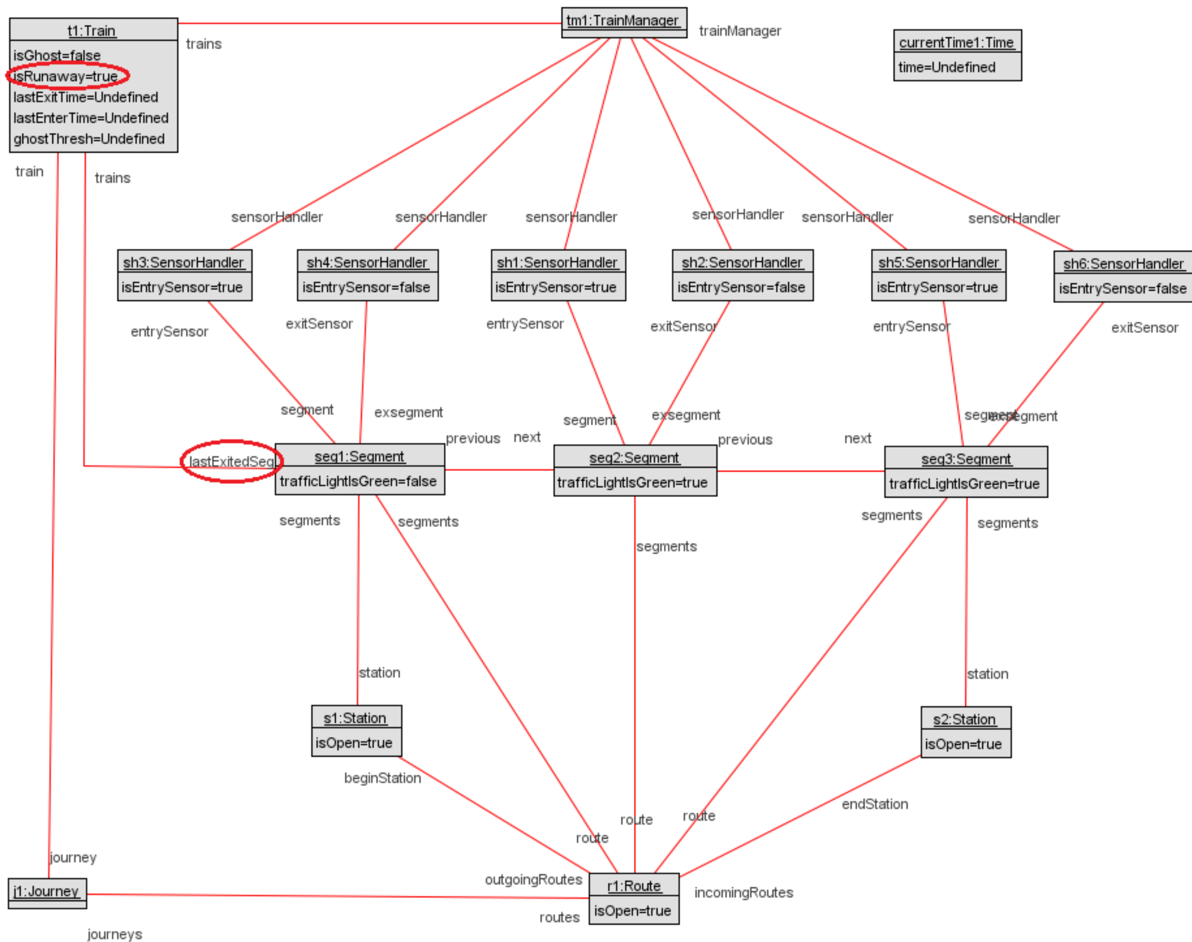


Figure 6.6. TMS snapshot 2.2

After operation *t1:onSegmentEnter(time,seg2,t1)* is called, in snapshot 2.3 (Fig. 6.7) a new association between train *t1* and segment *seg2* is added, *seg2* is now currentSeg *of t1* and *t1* is in *passingTrains* collection of *seg2*, the traffic light of *seg1* remains green. The newly added association between *t1* and *seg2* is circled in Fig. 6.7.

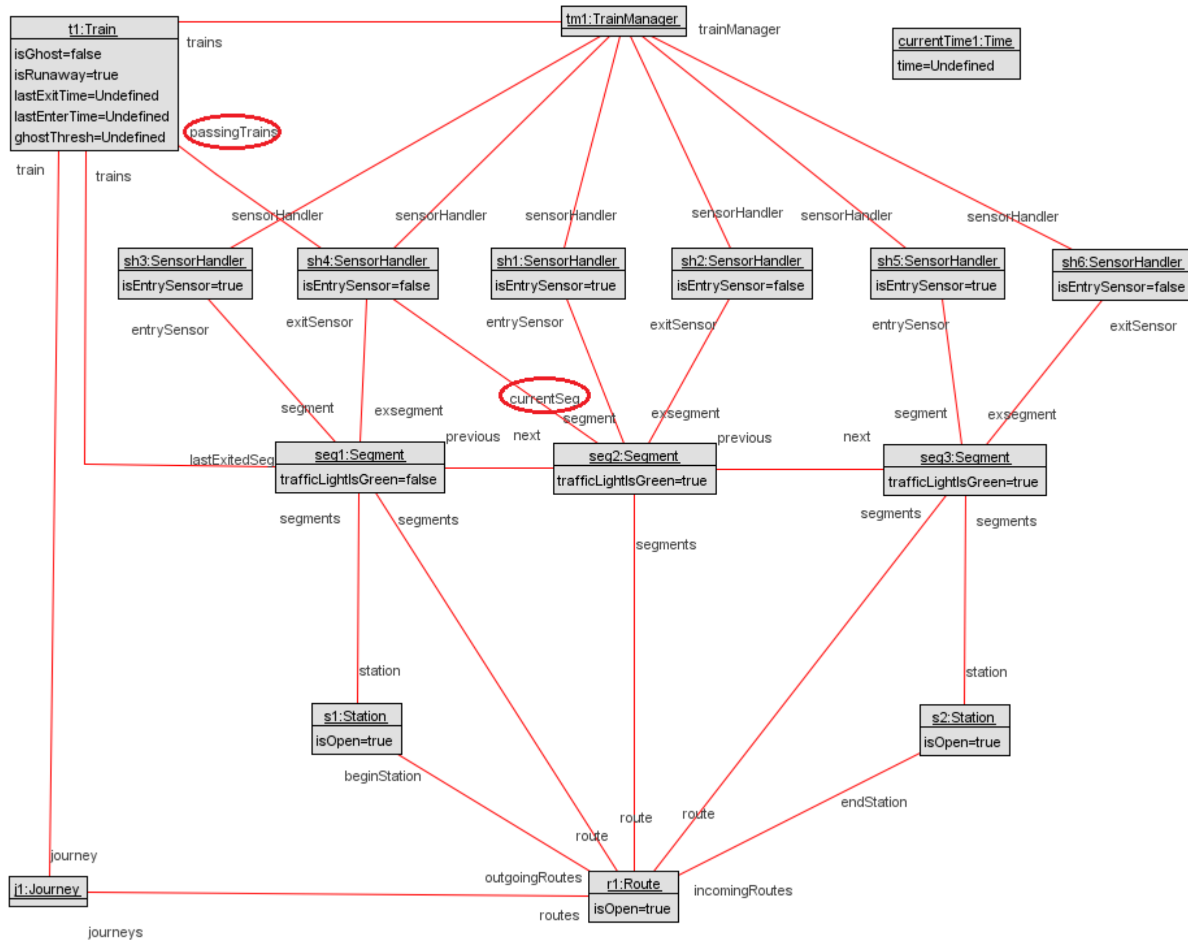


Figure 6.7. TMS snapshot 2.3

Analysis of this scenario shows that the *isOpen* attribute of *Route* object *r1* in snapshot 2.2 is *true* which is not consistent with operation *Train::OnSegmentExit*. In snapshot 2.1 *trafficLightIsGreen* attribute of *t1.currentSeg* is false, according to operation constraint *post1* in operation *Train::OnSegmentExit*, the train *t1* should be a runaway train and the route associated with segment *s1* should be closed, i.e., *isOpen* attribute of *r1* should be false.

6.2 The Generalized Spatio-Temporal RBAC model

The Generalized Spatio-Temporal RBAC model (GSTRBAC) is an extension to Role-Based Access Control model [Ray07]. It allows specifying location-aware and time-based access control constraints. In GTSRBAC location and time are associated with various entities in standard RBAC model, including user, role, permission, user assignment, role assignment, permission assignment and separation of duty. For example, location and time associated with role can be used to specify that the role can only be activated at the certain location and time. Location and time associated with permission can be used to specify that the permission can only be operated at the certain location and time.

A UML GSTRBAC model was analyzed in [Abdunabi13] using the USE. This section analyzes an adapted UML GSTRBAC model using the Scenario-based UML Design Analysis technique.

6.2.1 The GSTRBAC design class model

In the main view of GSTRBAC UML design class model (Fig. 6.8) time and location are encapsulated in a generalized *STZone* class. RBAC entities User, Role and Permission are modeled as *User*, *Role* and *Permission* class. *Object* and *Activity* classes model the object and operation entities in RBAC. Classes *UserRoleAssignment*, *UserRoleActivation* and *PermissionAssignment* describe user role assignment relation, user role activation relation and role permission assignment relation in RBAC. The *STZone* class is associated with *User*, *Role*, *Permission*, *Object*, *UserRoleAssignment*, *UserRoleActivation* and *PermissionAssignment.*

Figure 6.8. GSTRBAC design class diagram – main view

Fig. 6.9 shows the separation of duty classes of the GSTRBAC UML design class model. *RSSOD* class models static separation of duty constraint: conflicting roles under RSSOD cannot be assigned to the same user in certain STZone. *DSSOD* class models dynamic separation of duty constraint: conflicting roles under DSSOD cannot be activated in certain STZone by the same user. PSSOD class models permission-based separation of duty constraint: conflicting permissions cannot be assigned to the same role in certain STZone.

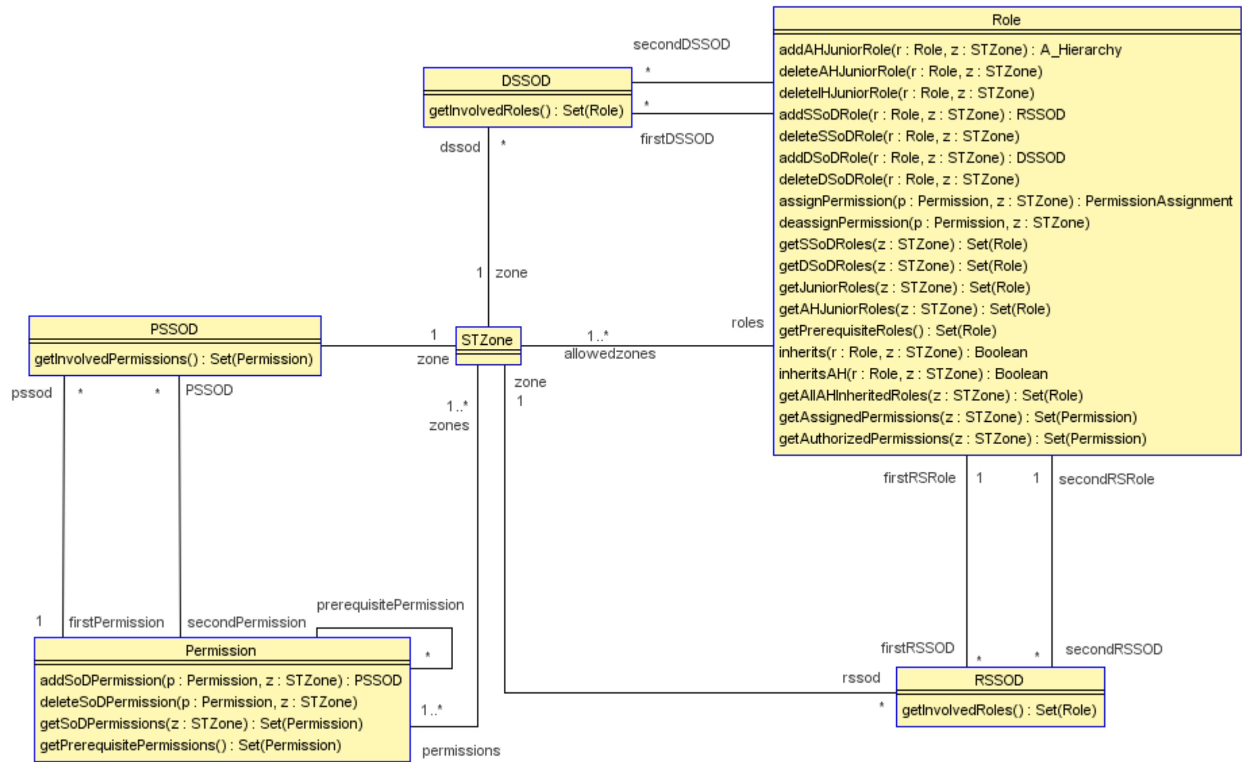


Figure 6.9. GSTRBAC design class diagram – SOD view

Below are OCL operation specifications of major operations *User::UpdateZone, User::assignRole, User::deassignRole, User::activateRole* and *User::deactivateRole*:

```
context
User::updateZone(z:STZone)
pre: true
// After the user updates zone, the zone is included in current zones
// of the user.
post: (self.currentzones->includes(z))

context
User::assignRole(r:Role,z:STZone):UserRoleAssignment
// Before the user is assigned role r at STZone z,
// (1) role r and STZone z must be defined.
pre assignRolePreCond1_definedObjects:
```

```
r.isDefined and z.isDefined
// (2) STZone z must be included in current zones of the user and
allowed zones of the role.
pre assignRolePreCond2_ZoneIncluded: self.currentzones->includes(z)
and r.allowedzones->includes(z)
// (3) Role r should not be assigned to the user.
pre assignRolePreCond3_RoleNotAssigned:
self.getAssignedRoles(z)->excludes(r)
// (4) Role r should not belong to any static separation of duty roles
// of any roles assigned to the user.
pre assignRolePreCond4_RoleNotSSoD:
self.getAssignedRoles(z)->collect(r |
   r.getSSoDRoles(z))->excludes(r)
// After the user is assigned role r at STZone z,
// (1) The number of assignments of the user is one greater than
// previous assignments.
post AssignSTRolePostCond1_NewUserRoleRelation:
(self.assignments - self.assignments@pre)->size()=1
// (2) The new assignment should include role r at zone z
post AssignSTRolePostCond2_NewRoleAssignment: (self.assignments -
self.assignments@pre)->forAll( rl |
   rl.oclIsNew() and rl.zone=z and rl.role->includes(r))
// (3) The assigned roles of the user should include role r
post AssignSTRolePostCond3_RoleIsAssigned:
self.getAssignedRoles(z)->includes(r)

context
User::deassignRole(r:Role,z:STZone)
// The pre and post conditions are close to User::assignRole
pre deassignRolePreCond1_RoleIsAssigned:
self.getAssignedRoles(z)->includes(r)
```

```
post deassignRolePostCond1_RoleDeassigned:
self.getAssignedRoles(z)->excludes(r)
post deassignRolePostCond2_RoleAssignmentObjectDeleted:
(self.assignments@pre - self.assignments)->size()=1 and
(UserRoleAssignment.allInstances@pre -
UserRoleAssignment.allInstances)->size()=1

context User::activateRole(r:Role,z:STZone):UserRoleActivation
// Before the role r is activated at STZone z,
// (1) role r and STZone z must be defined.
pre activateRolePreCond1_denfinedObject:
r.isDefined and z.isDefined
// (2) STZone z must be included in current zones of the user and
allowed zones of the role.
pre activateRolePreCond2_ZoneIncluded:
self.currentzones->includes(z) and r.allowedzones->includes(z)
// (3) Role r should not be activated by the user.
pre activateRolePreCond3_RoleNot:
self.getActivatedRoles(z)->excludes(r)
// (4) Role r is assigned to the user.
pre activateRolePreCond4_RoleIsAssigned:
getAssignedRoles(z)->includes(r)
// After the user activates role r at STZone z,
// (1) The number of activations of the user is one greater than
// previous activations.
post activateRolePostCond1_NewUserRoleRelation: (self.activations
- self.activations@pre)->size()=1
// (2) The new activation should include role r at zone z
post activateRolePostCond2_NewRoleActivation: (self.activations -
self.activations@pre)->forAll( rl | rl.oclIsNew() and rl.zone=z and
rl.role->includes(r))
// (3) The activated roles of the user should include role r
```

```
post activateRolePostCond3_RoleIsAssigned:
self.getActivatedRoles(z)->includes(r)

context
User::deactivateRole(r:Role,z:STZone)
// The pre and post conditions are close to User::activateRole
pre deactivateRolePreCond1_RoleIsActivated:
self.getActivatedRoles(z)->includes(r)
post deactivateRolePostCond1_RoleDeactivated:
self.getActivatedRoles(z)->excludes(r)
post deactivateRolePostCond2_RoleActivationDeleted:
(self.activations@pre - self.activations)->size()=1 and
(UserRoleActivation.allInstances@pre -
UserRoleActivation.allInstances)->size()=1
```

6.2.2 GSTRBAC scenario one

In this scenario user *Ben* and two roles *SP* and *TE* are located in the same STZone *z0*. *Ben* is assigned *SP* role first, then *Ben* activates *TE* role (note *TE* role is not assigned to *Ben*).

The initial snapshot of the scenario (Fig. 6.10) contains user *Ben* and two roles: *SP* and *TE*. *SP* is assigned permission *p0* and *TE* is assigned permission *p1*. The two roles and two permission assignments are all at STZone *z0*.

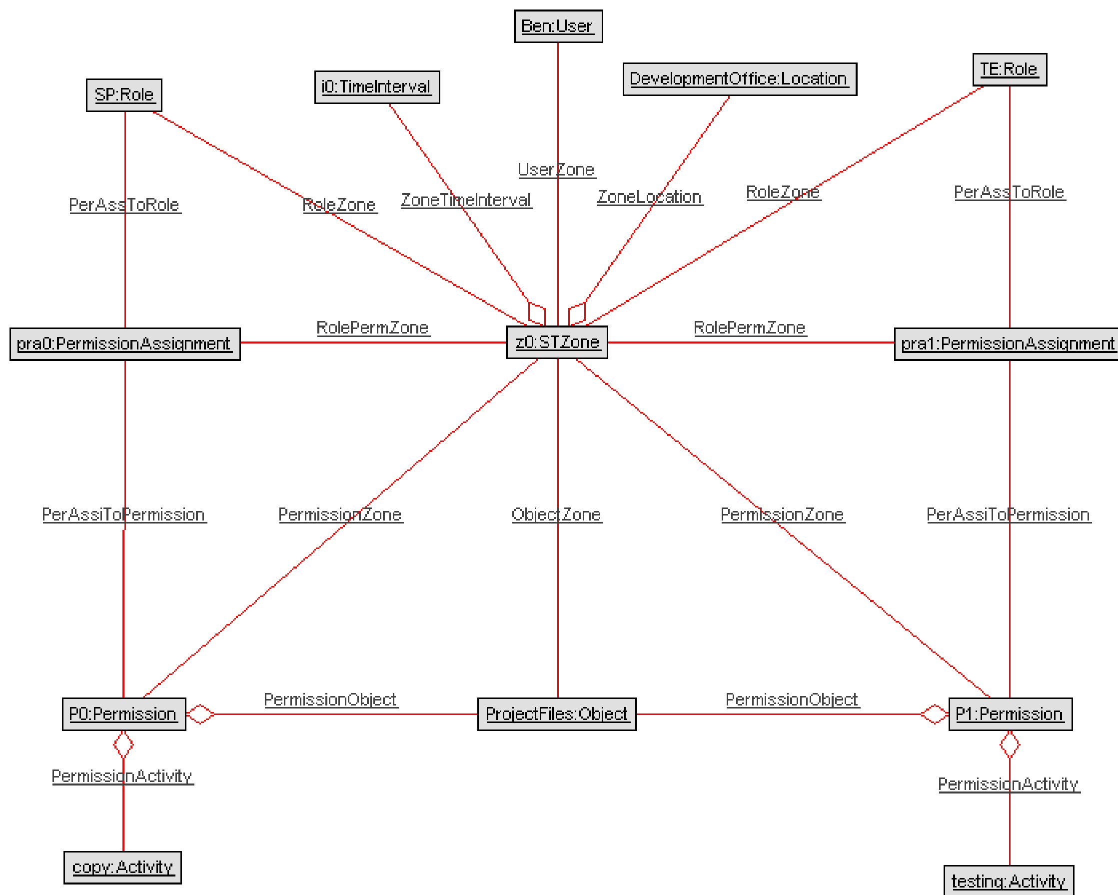


Figure 6.10. GSTRBAC snapshot 1.1

Operation *Ben.assignRole(SP, z0)* is called to assign *Ben SP* role at STZone *z0*. In the next snapshot (Fig. 6.11), UserRoleAssignment instance *uras0* is created between *Ben*, *SP* and *z0*. Transition from snapshot 1.1 to snapshot 1.2 is consistent with the design. The new *uras0* instance and three associations are circled in Fig. 6.11.

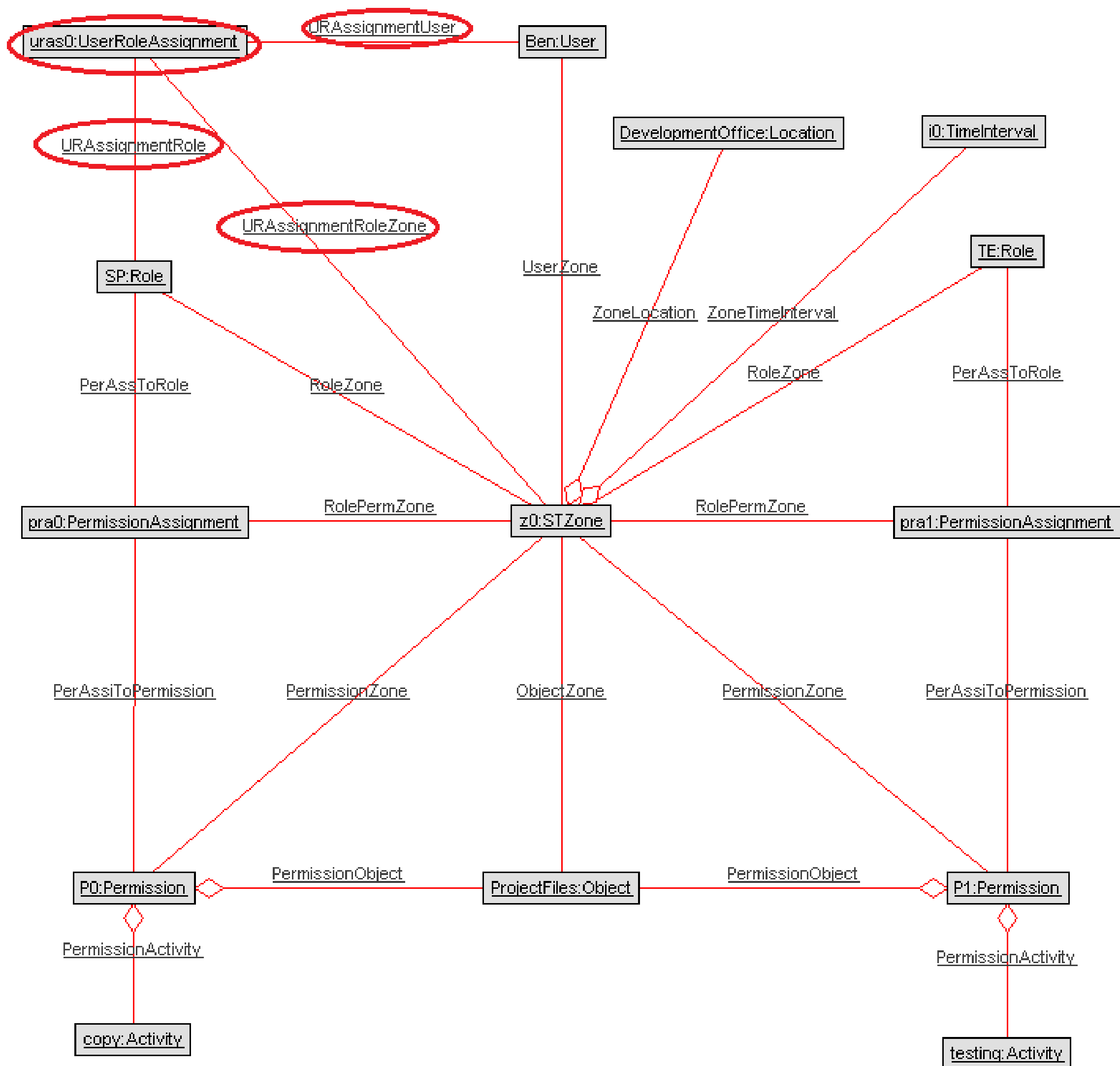


Figure 6.11. GSTRBAC snapshot 1.2

Operation *Ben.activateRole(TE, z0)* is called to activate *TE* role at STZone *z0*. In snapshot 1.3 (Fig. 6.12), UserRoleActivation instance *urac0* is created between *Ben*, *TE* and *z0*. The new *urac0* instance and three new associations are circled in Fig. 6.12.

Transition from snapshot 1.2 to snapshot 1.3 is not consistent with the design. One precondition of *User::activateRole* requires that the role must be assigned to the user before it is

activated. This pre-condition is not satisfied because *TE* role is not assigned to *Ben* before activation.

```
pre activateRolePreCond4_RoleIsAssigned:
getAssignedRoles(z)->includes(r)
```

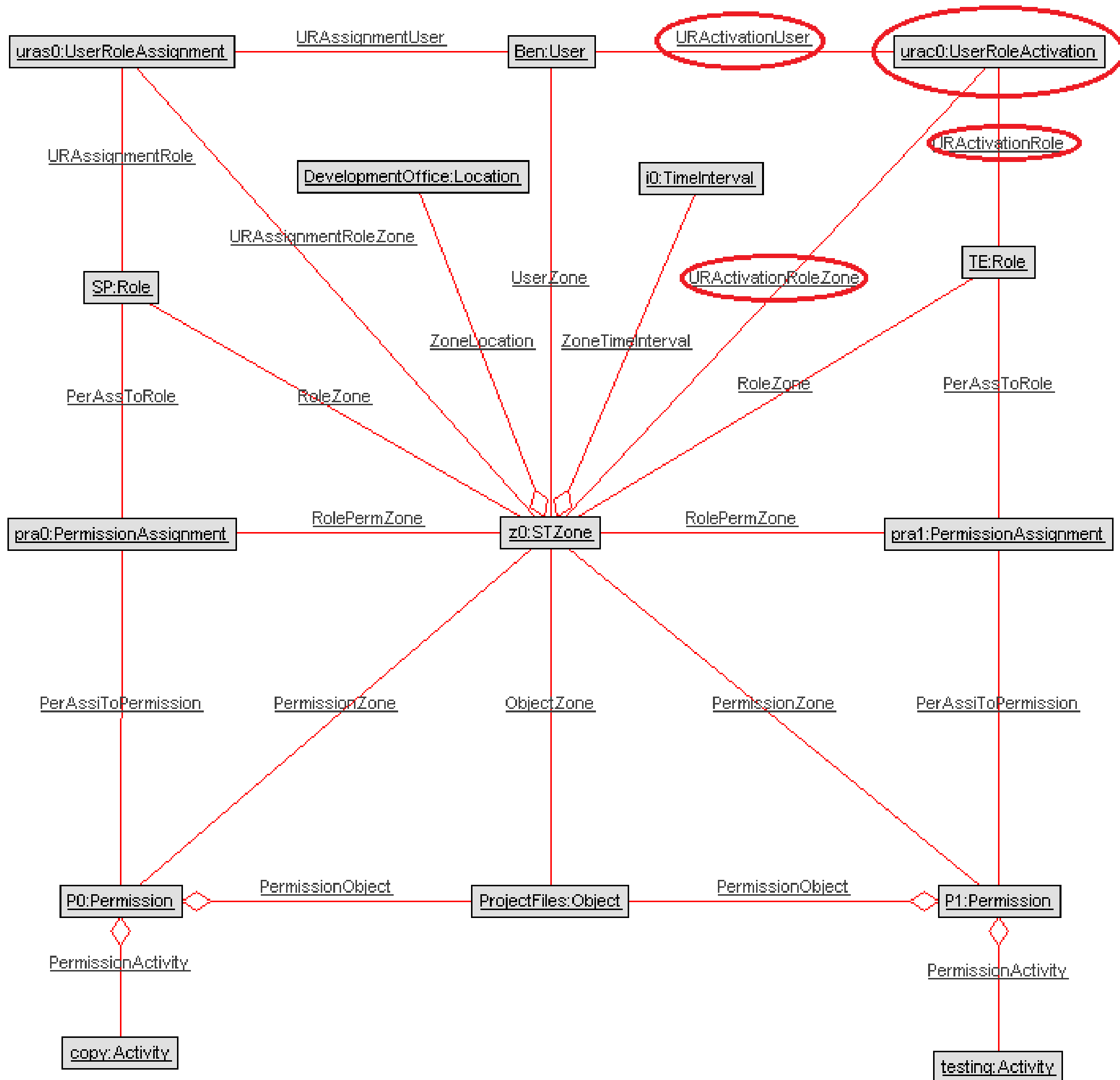


Figure 6.12. GSTRBAC snapshot 1.3

6.2.3 GSTRBAC scenario two

In this scenario user *Ben* and two roles *SP* and *TE* are initially located in the same STZone *z0*. *Ben* is assigned *SP* role at *z0* first, then *Ben* moves to STZone *z1, Ben* is assigned *TE* role,

and finally *Ben* activates *TE* role at STZone *z1* (note allowed zone of *TE* role is *z0*). The initial snapshot of the scenario (Fig. 6.13) contains user *Ben* and two roles: *SP* and *TE*. *SP* is assigned permission *p0* and *TE* is assigned permission *p1*. The two roles and two permission assignments are all at STZone *z0*. STZone *z0* is at location *DevelopmentOffice*, another STZone *z0* is at location *TestingOffice*.

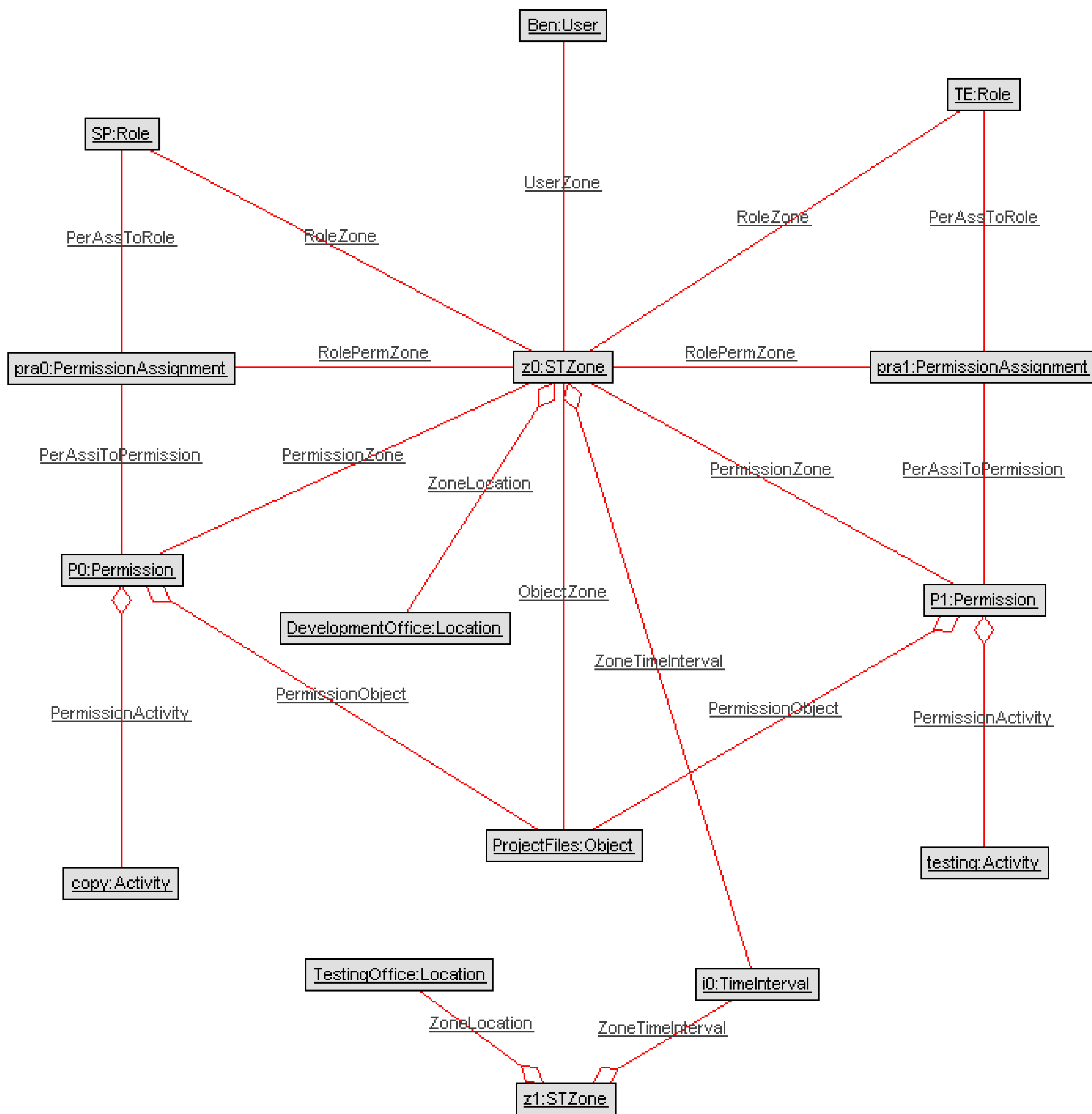


Figure 6.13. GSTRBAC snapshot 2.1

Operation *Ben.assignRole(SP, z0)* is called to assign *Ben SP* role at STZone *z0*. In snapshot 2.2 (Fig. 6.11), UserRoleAssignment instance *uras0* is created between *Ben*, *SP* and *z0*. The new

uras0 instance and its associations are circled in Fig. 6.11. Transition from snapshot 2.1 to snapshot 2.2 is consistent with the design.

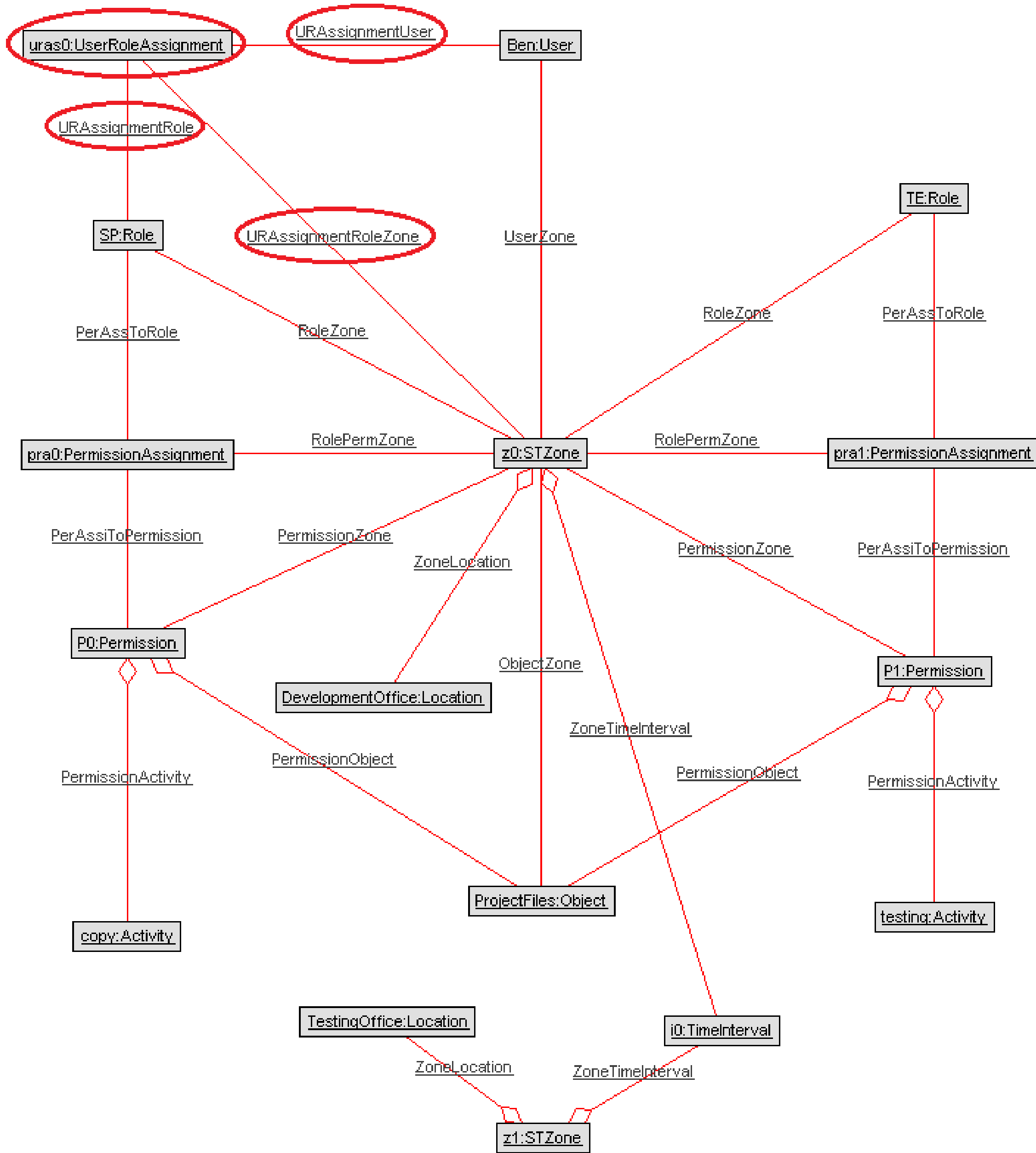


Figure 6.14. GSTRBAC snapshot 2.2

Operation *Ben.updateZone(z1)* is called to update STZone of *Ben* to *z1*. In snapshot 2.3 (Fig. 6.15) *Ben* is associated to STZone *z1* (as circled in Fig. 6.15), and UserRoleAssignment instance

uras0 is removed. From the verifier's perspective, the user role assignment becomes invalid after *Ben* moves to STZone *z1* since UserRoleAssignment *uras0* is associated with STZone *z0*.

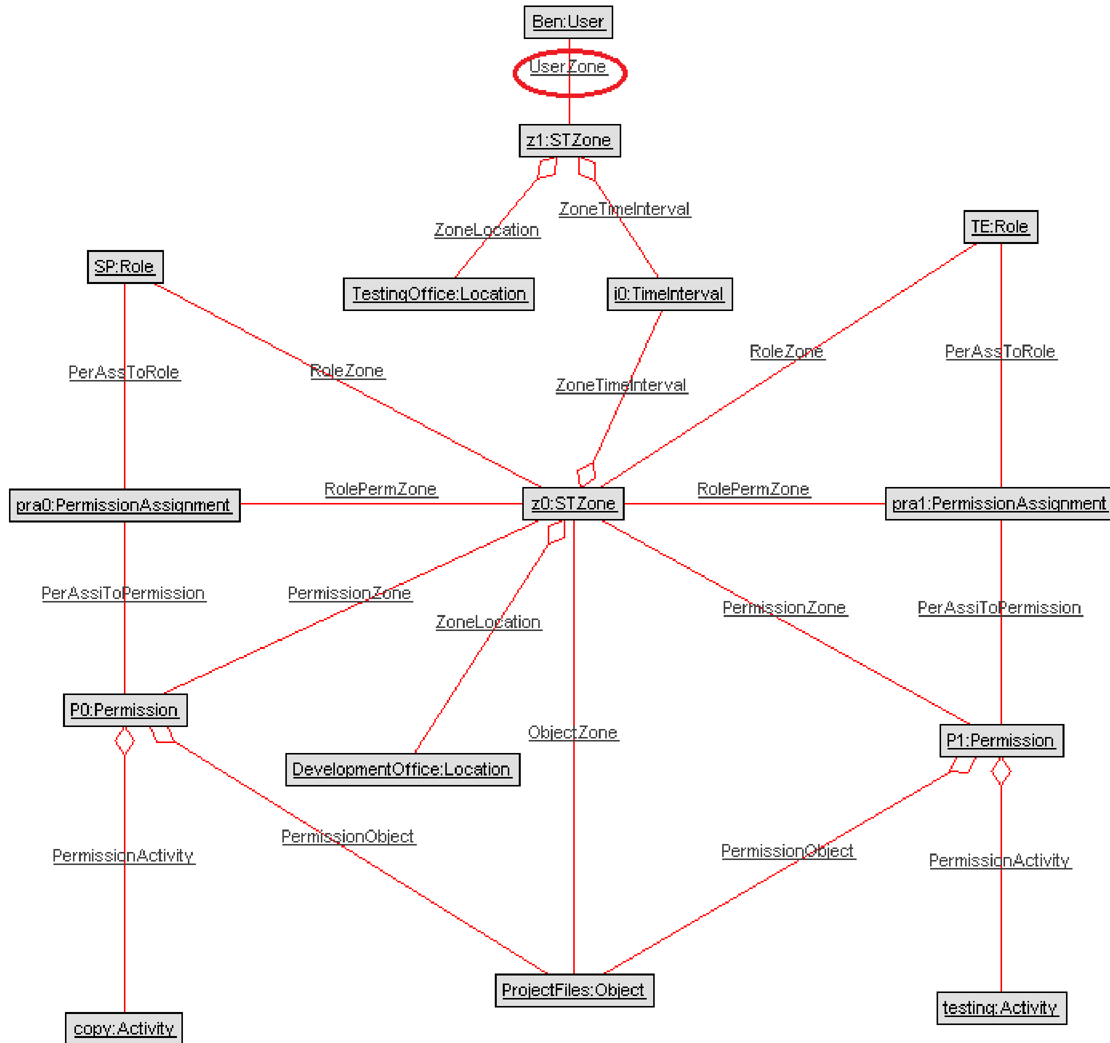


Figure 6.15. GSTRBAC snapshot 2.3

The transition from snapshot 2.2 to 2.3 is not consistent with operation *User::updateZone.* The frame constraints of the operation do not allow UserRoleAssignment instances be changed after the operation is called.

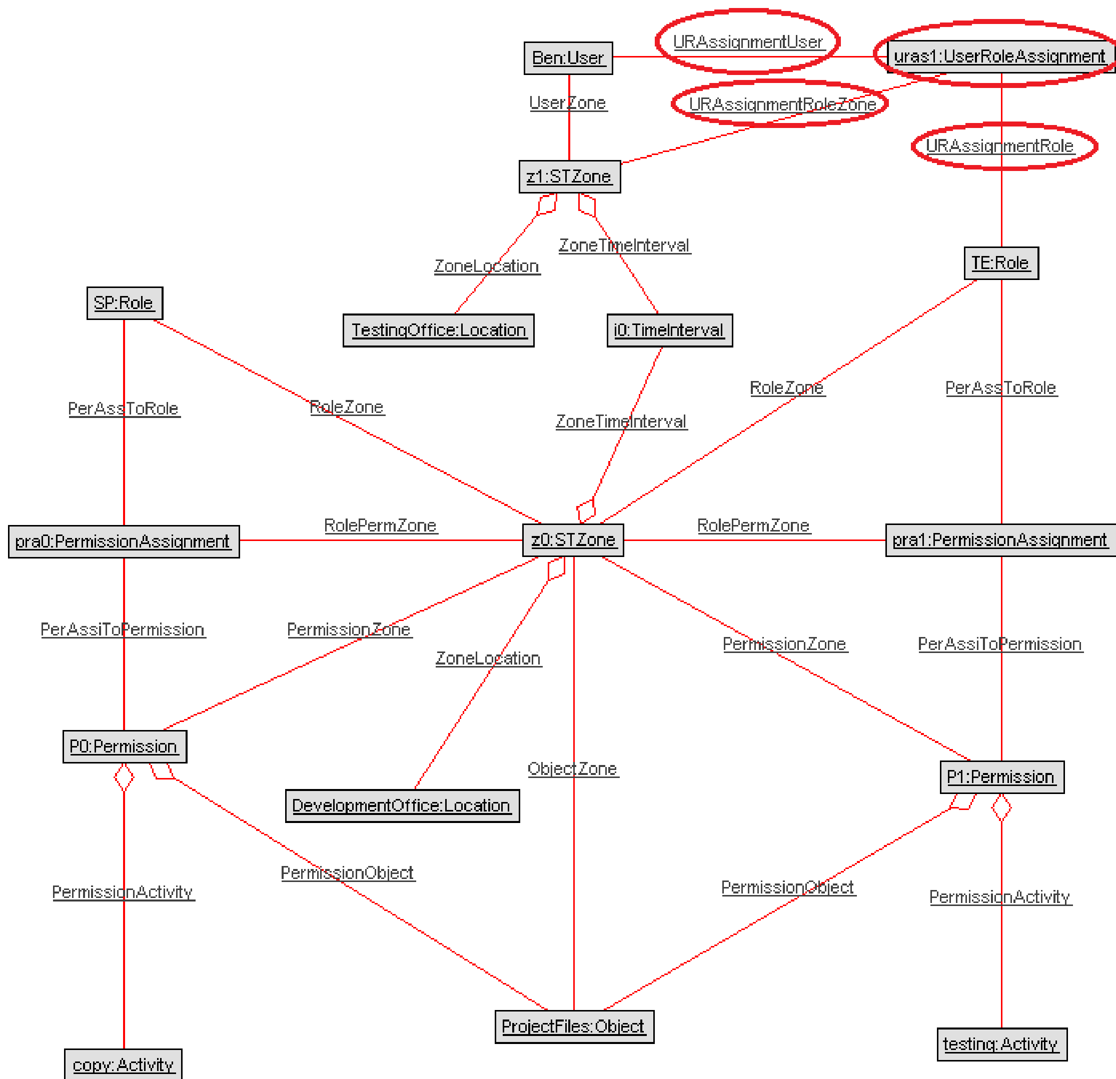


Figure 6.16. GSTRBAC snapshot 2.4

Operation *Ben.assignRole(TE, z1)* is called to assign *TE* role to *Ben* at STZone *z1*. In snapshot 2.4 (Fig. 6.16) UserRoleAssignment instance *uras1* is created between *Ben*, *TE* and *z1*. The new *uras1* instance and its associations are circled in Fig. 6.16.

The transition from snapshot 2.3 to 2.4 is not consistent with operation *User::assignRole*. Precondition *assignRolePreCond2_ZoneIncluded* is not satisfied because *allowedzones* of role *TE* is *z0* which does not include STZone *z1*:

```
pre assignRolePreCond2_ZoneIncluded: self.currentzones->includes(z)
and r.allowedzones->includes(z)
```

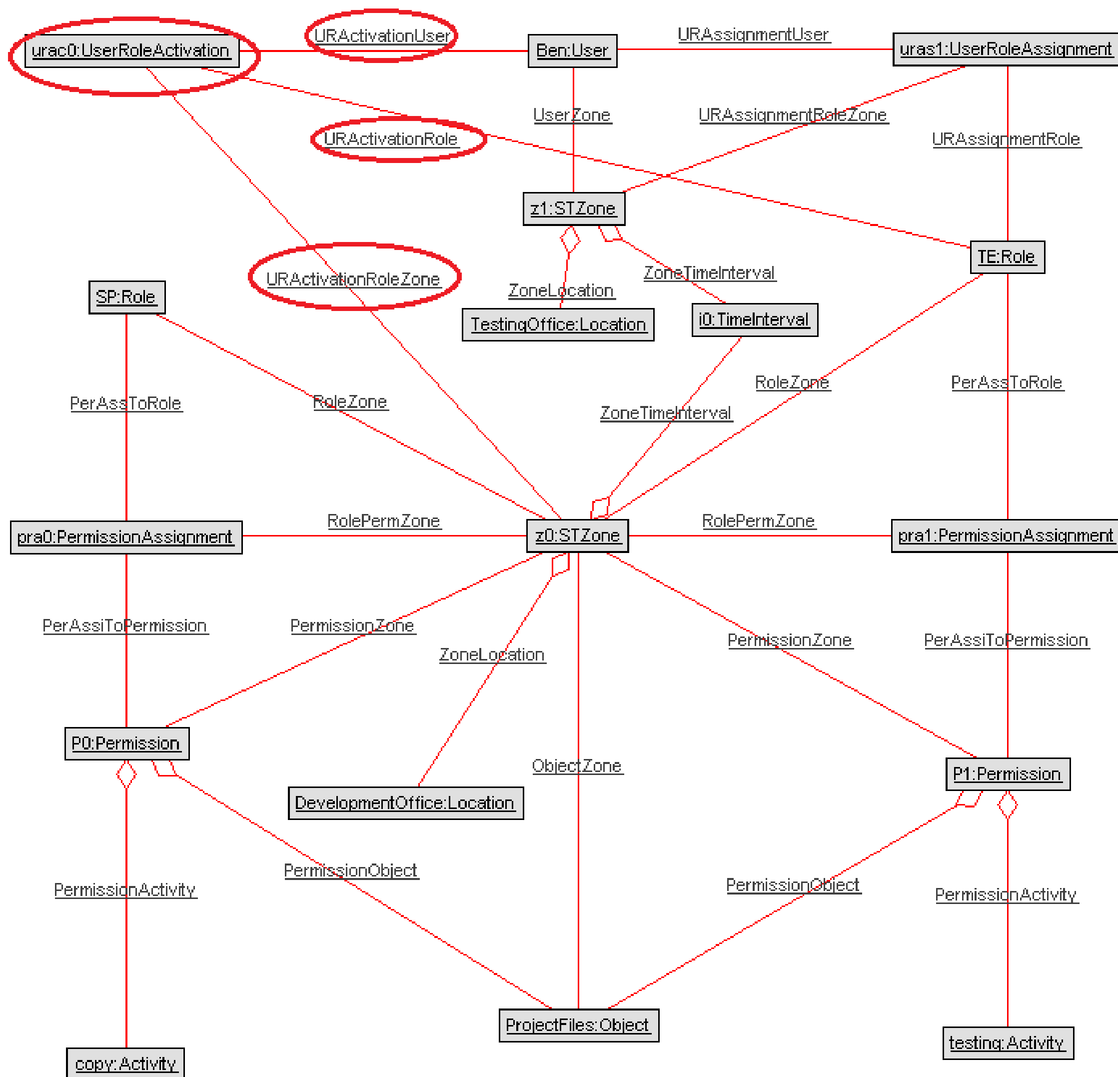


Figure 6.17. GSTRBAC snapshot 2.5

The last operation *Ben.activateRole(TE, z0)* is called to activate *TE* at *z0.* In Fig 6.17 the new *urac0* instance and its associations are circled.

The transition from snapshot 2.4 to 2.5 is not consistent with operation *User::activateRole*. Precondition *activateRolePreCond2_ZoneIncluded* is not satisfied because *currentzones* of *Ben* is *z1* which does not include STZone *z0*:

```
pre activateRolePreCond2_ZoneIncluded:
self.currentzones->includes(z) and r.allowedzones->includes(z)
```

Precondition *activateRolePreCond4_RoleIsAssigned* is not satisfied. The only assigned role *TE* is at STZone *z1* and *Ben*'s assigned roles at STZone *z0* is empty:

```
pre activateRolePreCond4_RoleIsAssigned:
getAssignedRoles(z)->includes(r)


class User
operations
getAssignedRoles(z:STZone):Set(Role)=self.assignments->
select(r|r.zone=z)->collect( r| r.role)->asSet()
```

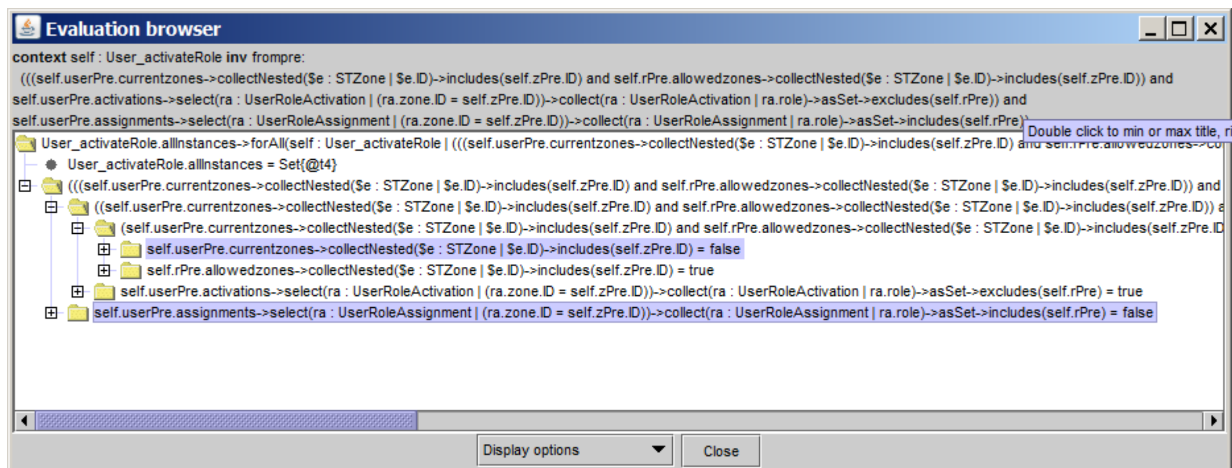


Figure 6.18. GSTRBAC inconsistencies in snapshot 2.5

Fig. 6.18 shows violations of precondition *activateRolePreCond2_ZoneIncluded* and *activateRolePreCond4_RoleIsAssigned*.

6.2.4 GSTRBAC scenario three

In this scenario user *Ben* is initially assigned two roles *SP* and *TE* at STZone *z0*. *SP* and *TE* are dynamic separation of duty roles at STZone *z0. Ben* activates *TE* role first then activates *SP* role at STZone *z0*.

The initial snapshot of the scenario (Fig. 6.19) contains user *Ben* and two roles: *SP* and *TE*. *SP* and *TE* are both assigned to *Ben* at STZone *z0*. DSSOD (dynamic separation of duty) instance *dssod1* is created to forbid role *SP* and *TE* be activated simultaneously at STZone *z0*.

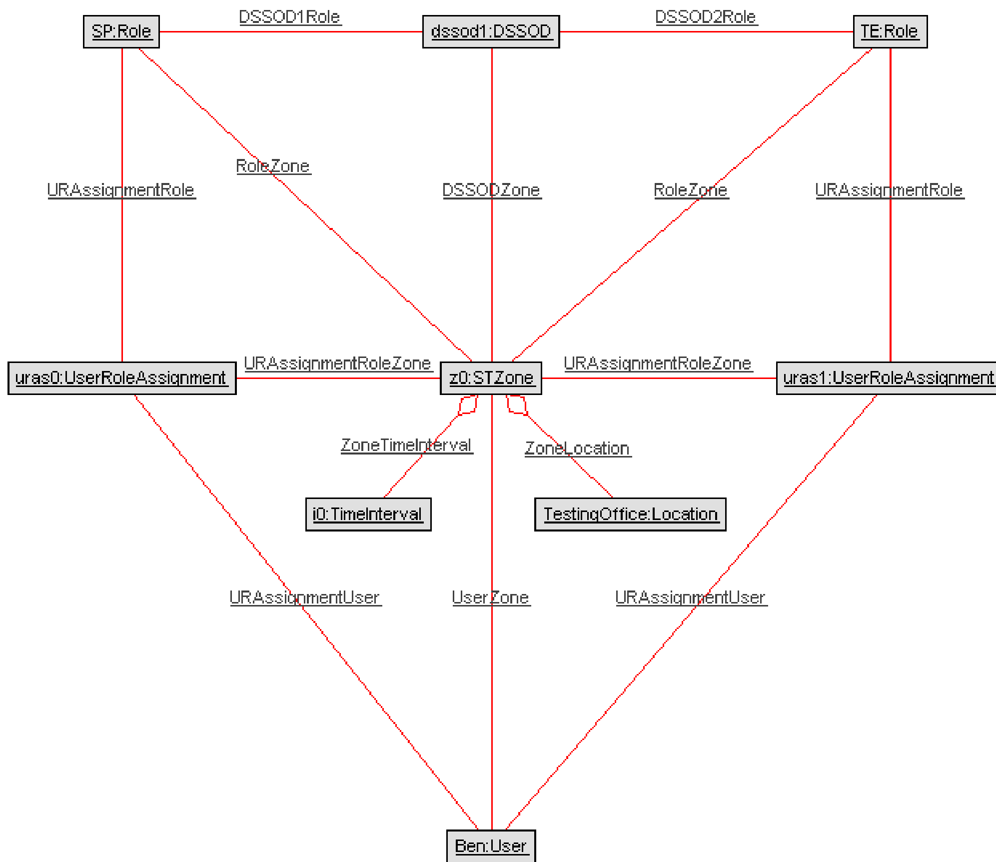


Figure 6.19. GSTRBAC snapshot 3.1

Operation *Ben.activateRole(TE, z0)* and *Ben.activateRole(SP, z0)* are called to activate *TE* and *SP* roles at STZone *z0*. In snapshot 3.2 (Fig. 6.20) the *TE* role is activated. In Fig 6.20 the

new *urac0* instance and its associations are circled. In snapshot 3.3 (Fig. 6.21) the *SP* role is activated. In Fig 6.21 the new *urac1* instance and its associations are circled.

The transition from snapshot 3.1 to 3.2 and 3.2 to 3.3 are both consistent with operation *User::activateRole*.

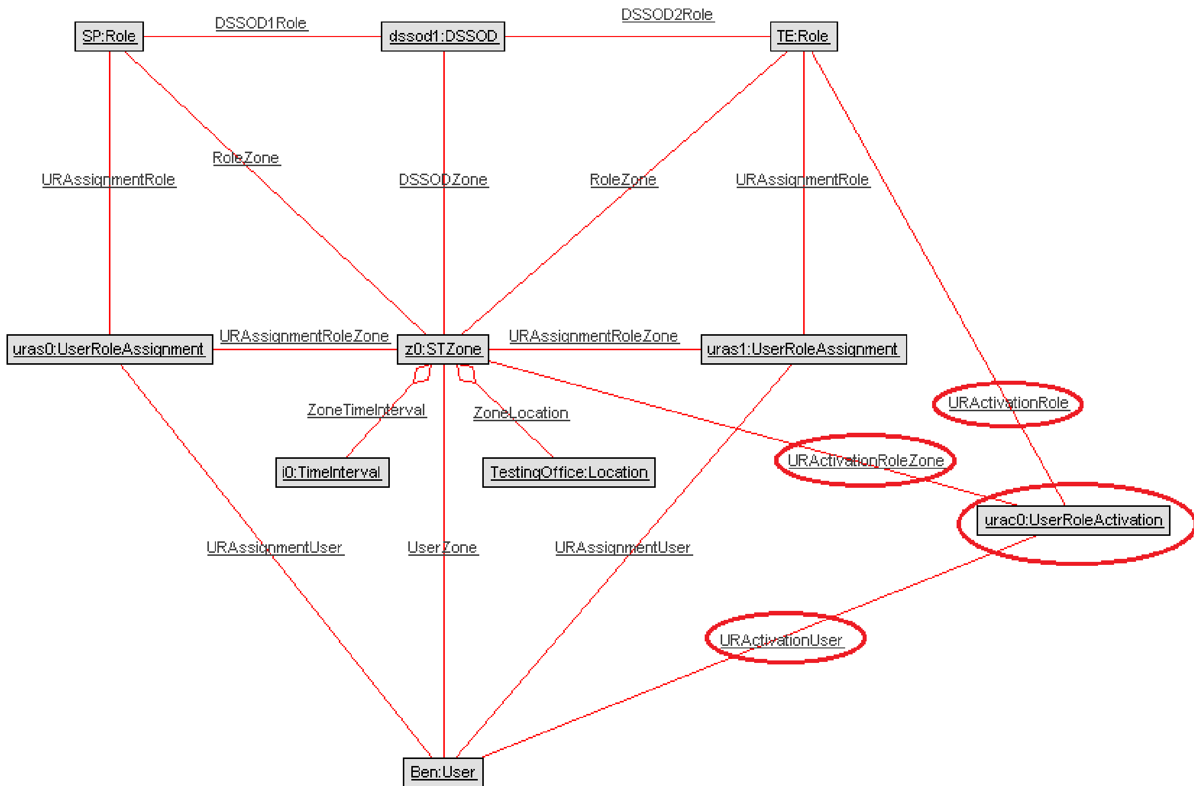


Figure 6.20. GSTRBAC snapshot 3.2

However, the scenario is invalid from the verifier's perspective. The two conflicting roles are not supposed to be activated by the same user at the same STZone. It implies design error in precondition of operation *User::activateRole*. It should check whether a conflicting role is being activated.

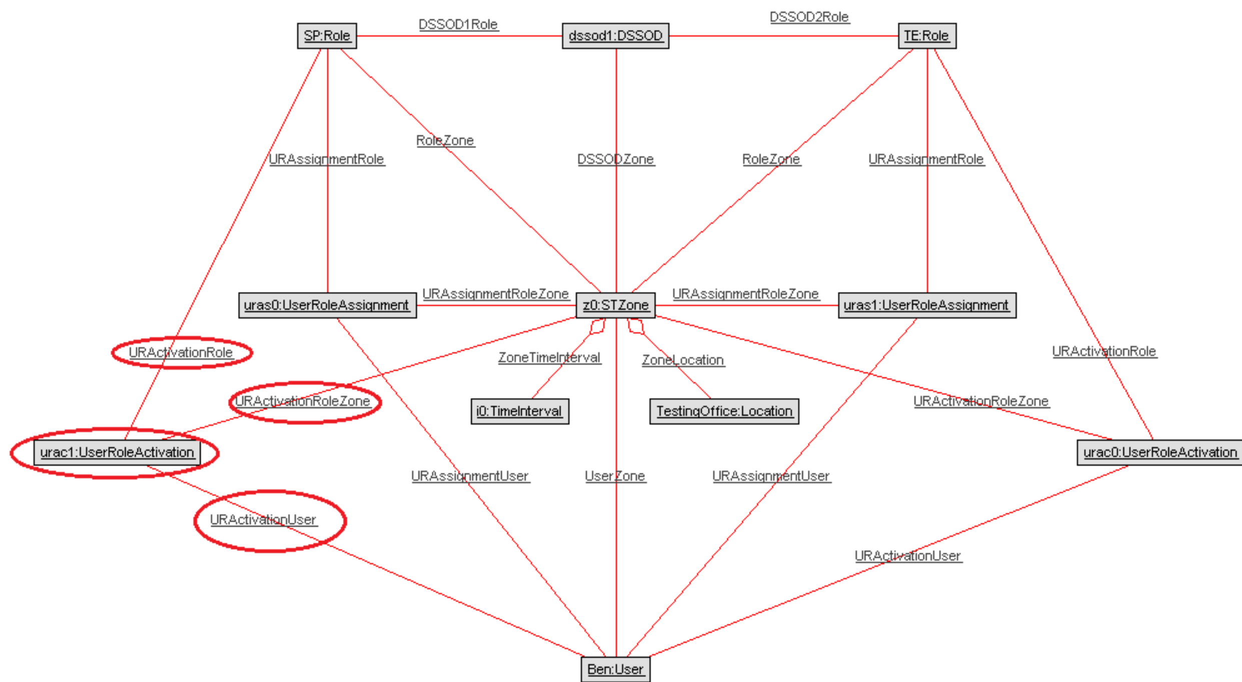


Figure 6.21. GSTRBAC snapshot 3.3

6.3 Conclusion

This chapter discusses case studies of two application designs. The case studies show how the Scenario-based UML Design Analysis technique can be used to find inconsistencies between UML designs and scenarios.

During the analysis process, the verifier reads the UML designs and manually creates scenarios from his/her perspective. The scenarios and UML designs are input to the Scenario-based UML Design Analysis tool and then the consistencies between the transformed snapshot transition models and snapshot transitions are checked in USE. Without using the technique the verifier has to manually check OCL specifications against the scenarios which is time-consuming and error-prone.

Chapter 7

Pilot Study

The Scenario-based UML Design Analysis technique is further evaluated by a pilot study. In the pilot study a group of graduate students are invited to manually inspect the UML design class model and scenarios to find inconsistencies. At the same time the Scenario-based UML Design Analysis tool is used to find inconsistencies between the UML design class model and scenarios. We compare inconsistencies found by the group of students and the tool and decide whether the Scenario-based UML Design Analysis technique is effective or not.

The rest of this chapter is organized as follows: Section 7.1 discusses experiment planning, including the experiment definition, context selection, hypotheses formulation, experiment design and measurements and data collection. Section 7.2 discusses experiment results and analysis. Section 7.3 discusses threats to validity. Section 7.4 concludes the pilot study.

7.1. Experiment planning

7.1.1 Experiment goal, research question and hypothesis

The experiment objective is formulated in the form of Goal-Question-Metric (GQM) goals in table 7.1.

The experiment of the goal has one independent variable *design verification method* and two dependent variables *number of inconsistencies detected* (NID) which is the number of inconsistencies detected between the UML design class model and scenarios, and *number of false inconsistencies detected* (NFID) which is the number of falsely detected inconsistencies.

There are two treatments: the Scenario-based UML Design Analysis technique and manual design inspection technique.

Table 7.1. Formulation of the experiment objective

Analyze	the Scenario-based UML Design Analysis technique
for the purpose of	Evaluating
with respect to	the effectiveness of identifying inconsistencies between UML class model and scenarios when comparing with a traditional manual inspection process
from the point of view of	the design verification engineer
In the context of	graduate computer science students

The Scenario-based UML Design Analysis technique is considered effective if it leads to equal or higher number of inconsistencies and equal or lower number of false inconsistencies than a manual design inspection technique will uncover.

7.1.2 Context selection and subjects

The context of the experiment is Software Specification & Design, a graduate level software engineering course at Colorado State University. The subjects are a number of graduate or senior Computer Science students that are enrolled in the course. The students enrolled in the course are familiar with UML and OCL notations, and they were trained on how to manually find design inconsistencies between a UML design class model and scenarios.

7.1.3 Experiment design

The UML design class models used in the experiment were produced by students at the Software Specification & Design course or created in our previous research projects.

A group of students who are familiar with the Scenario-based UML Design Analysis technique create scenarios. These students are given the UML design class diagram only. OCL

operation specifications are not given to these students. The students who create scenarios do not participate in finding design inconsistencies using manual inspection techniques.

Another group of students who are familiar with UML, OCL and design inspection technique individually review the UML design model and scenarios to find inconsistencies. This group of students is trained on how to manually inspect design inconsistencies between the UML design model and scenarios.

The Scenario-based UML Design Analysis tool is used to find inconsistencies between the scenarios and the UML design.

We record inconsistencies found by each student and the tool. We repeat the experiment on a number of UML design models. The results of experiments are consistent if they both show that the Scenario-based UML Design Analysis technique is more effective than design review, or vice versa. If the results are not consistent, we will analyze the data and find the reason of the inconsistency, modify the process and study more applications if necessary.

7.1.4 Measurements and data collection

During the experiment, we will measure total number of inconsistencies uncovered by SDA and MDI, total number of false inconsistencies uncovered by SDA and MDI, and total time spent by each student in manual inspection. The students in the manual inspection group can also give feedback on the manual inspection process.

7.2. Experiment results and analysis

We performed pilot study on two design models, the TMS design class model (Fig. 6.1) and GSTRBAC design class model (Fig. 6.8 and Fig. 6.9). Two graduate students were asked to

manually inspect inconsistencies between the UML design class models and scenarios. The Scenario-based UML Design Analysis tool was used to check inconsistencies.

Table 7.2. TMS experiment results

TMS scenarios	NID (Tool)	NFID (Tool)	NID (HumanA)	NFID (HumanA)	NID (HumanB)	NFID (HumanB)
Scenario 1	6	0	2	1	6	1
Total	6	0	2	1	6	1

Table 7.3. GSTRBAC experiment results

GSTRBAC scenarios	NID (Tool)	NFID (Tool)	NID (HumanA)	NFID (HumanA)	NID (HumanB)	NFID (HumanB)
Scenario 1	0	0	0	0	0	0
Scenario 2	1	0	1	0	1	0
Scenario 3	1	0	1	0	1	0
Scenario 4	1	0	1	0	1	0
Scenario 5	0	0	0	1	0	0
Scenario 6	3	0	2	0	2	0
Scenario 7	2	0	1	0	0	0
Scenario 8	2	0	2	0	2	0
Scenario 9	2	0	2	2	2	0
Scenario 10	0	0	0	0	0	0
Scenario 11	3	0	1	0	1	0
Scenario 12	3	0	0	2	2	0
Total	17	0	11	6	12	0

Table 7.2 and table 7.3 shows number of inconsistencies found by the tool and two graduate students of the TMS and GSTRBAC design class model. The first column shows the scenario ID. The second column NID shows the number of inconsistencies identified by the tool. The third column NFID shows the number of inconsistencies falsely identified by the tool. NID (HumanA) and NFID (HumanA) columns show the number of inconsistencies and number of false

inconsistencies found by the first graduate student. NID (HumanB) and NFID (HumanB) columns show the number of inconsistencies and number of false inconsistencies found by the second graduate student.

The tool did not identify false inconsistencies in the pilot study of the two design class models. Fig. 7.1 shows a histogram of the number of inconsistencies identified by the tool and two graduate students in the thirteen scenarios. In scenario 1, 7, 8, 12 and 13, the tool identified more design inconsistencies than the two graduate students. In scenario 3, 4, 5, 9 and 10, the tool identified the same number of design inconsistencies as the two graduate students. In scenario 2, 6 and 11, neither the tool nor the two graduate students found any design inconsistencies, but the graduate students may find false design inconsistencies. To sum up, the tool identified equal or more number of inconsistencies in all the scenarios than the two students.

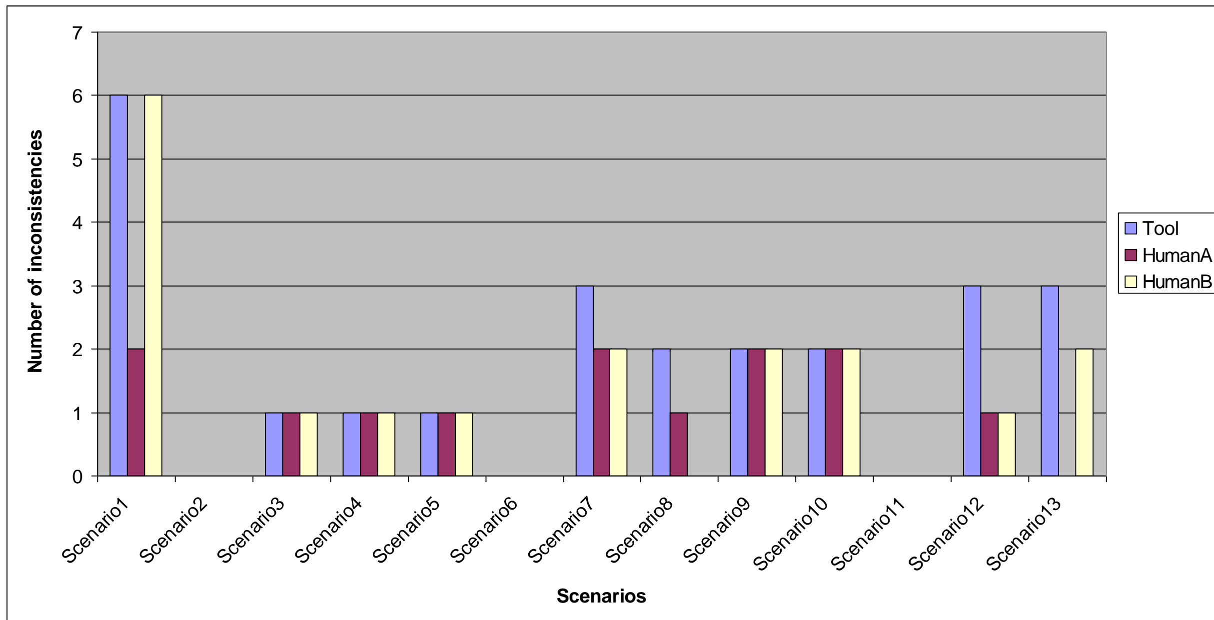


Figure 7.1. Histogram of experiment results

7.3. Conclusion and discussions

The pilot study shows that the Scenario-based UML Design Analysis tool seems to be effective because it uncovered at least as many design inconsistencies as manual inspection techniques uncovered and the technique did not uncover false inconsistencies. Actually the inconsistencies uncovered by the two graduate students are subset of the inconsistencies uncovered by the tool.

As to the cost of finding inconsistency, the tool can be used to analyze a design class model and scenarios automatically which takes a few minutes to generate necessary models and scenarios and check them in USE. To manually inspect inconsistencies, the human beings must be trained with domain knowledge and the manual inspection techniques. It is also time-consuming and error-prone for the human being to manually inspect each scenario. According to the feedback of the two graduate students, it took them about two hours reading the GSTRBAC design class diagram and constraints, it took them about 30 to 45 minutes on average to check one GSTRBAC scenario and 15-20 minutes to review the TMS scenario.

The main threat to validity of the pilot study is statistical conclusion validity [Wohlin12]. Due to the unavailability of graduate students, we were not able to control the number of students and the number of design class models (scenarios). In the formal experiment, we should invite more students to do manual inspection and study more design class models and scenarios. It is also desirable to seed more inconsistencies into the UML designs and scenarios. Both of the measures are helpful to mitigate the threat to statistical conclusion validity of the experiment.

Chapter 8

Generating Scenarios using JAL Operation Definitions

The Scenario-based UML Design Analysis technique requires that the verifier creates scenarios to analyze UML design class models. This chapter presents a scenario generation technique that automates the generation of scenarios using operational operation definitions [Yu09]. The operation definitions are specified using Jave-like Action Language (JAL). Java-like Action Language (JAL) is an imperative action language developed in our research group that is used to describe effects of operations [Trung05A]. Scenario snapshot transitions are generated by executing the JAL operation definitions using the UMLAnT (UML Animation and Testing) tool [Trung05A].

The rest of this section is organized as follows: section 8.1 discusses the scenario generation technique, section 8.2 discusses UML design class model of an RBAC example and RBAC constraints modeled as OCL invariants, section 8.3 discuss analysis of RBAC constraints using the scenario generation technique.

8.1 The scenario generation technique

This scenario generation technique automatically generates a set of scenarios based on the verifier's JAL operation definitions and *operation invocation patterns* [Yu09]. The *operation invocation pattern* describes sequence of operations as regular expressions. The scenario generation technique allows a verifier to produce a set of scenarios describing legal and illegal functionality. The scenario generation technique takes into consideration domain-specific knowledge about sequences of operation calls that reflect typical usages and sequences. This

knowledge is encoded in operation call sequence patterns that are used by the verifier to generate scenarios.

The scenario generation technique is based upon a naïve scenario generation algorithm. The naïve algorithm generates too many scenarios, and thus we extend it by allowing the verifier to target specific families of scenarios by specifying patterns.

The naïve scenario generation algorithm does the following:

- Builds an operation invocation tree from a set of operations and parameter values.
- Traverses the operation invocation tree to produce all possible sequences of operation invocations, and
- Animates each sequence of operation invocations to produce a sequence of snapshot transitions. The verifier must then label each of the generated snapshot transition sequences as legal or illegal.

Each node in an operation invocation tree represents a particular invocation of a system operation on an object. The invocation is referred to as an operation instance. Each node contains an object identifier (the receiver of the operation call), an operation name and a value for each operation parameter. The root of the tree represents the system initialization point and it contains information about the start state. Child nodes represent operation invocations that can occur after the invocation represented by the parent node. A scenario is a path that starts at the root and ends at any node in the tree.

To reduce the number of scenarios produced by the above algorithm, the technique allows a verifier to

- limit the depth of the tree.
- limit the number of objects of a class that can be in a start state, and

- explicitly define a small domain for each input parameter of the operation.

For example, given an operation *User::AssignRole(r:Role),* the verifier can restrict the User domain size to 2 users objects, and define a small domain for the Role parameter as follows: *Domain(Role) = {clerk, seniorClerk}.*

The extended generation technique allows a verifier to specify patterns of operation sequences that restrict

- the operation calls that are used to build the operation invocation tree.
- the order in which operations can be invoked.

These patterns are called operation invocation patterns. An operation invocation pattern is a characterization of particular sequences of operation invocations that the verifier feels typifies good and problematic usages of the system. The patterns are manually created using the best available domain expertise and experience related to the sequences of operations that are likely to uncover policy violations. The patterns are described in terms of constraints on initial states and on the sequencing of operation calls. The use of these patterns allows the verifier to focus the analysis on particular sequences of invocation calls.

For example, a verifier can create the following pattern of operation calls for analyzing role activation functionality:

Initial State Constraint
u ***in Domain****(User) // There is at least one user*
#Role>3 //At least 4 roles are in the start state
Call Pattern
*u.CreateSession(.)****return(****s:Session){1,2} u.AssignRole(.){2,4}*
u->s.ActivateRole(.){1..4}

The first part of the pattern description constrains the initial state. In this case the initial state must consist of a *User* object, *u*, and at least four roles.

The second part is the pattern of operation calls. The expression *caller->callee.Op()* (e.g., see last line of the above Call Pattern) identifies the sender (*caller*) and receiver (*callee*) of an operation call message. If the caller is omitted then it is assumed that the message is coming from an external actor. The analysis we perform using the Snapshot transition model does not require that the sender of an operation call be known; this information is currently used only to visualize the operation sequence as a sequence diagram that shows both senders and receivers of messages.

The pattern describes the following sequences of operation calls:

- Start with 1 or 2 calls to the *CreateSession*() operation for a user, *u* using any parameters (as indicated by the "." in the parameter list of the operation), and each successfully returning a new session, *s*, (indicated by *return(s:Session)*),
- followed by 2 to 4 operation calls to the *AssignRole*() for user *u*, and
- end with 1 to 4 calls made by the user *u* to activate roles in the sessions previously created by calls to *CreateSession*().

In order to generate snapshot transitions, a verifier must provide descriptions of operation functionality to the snapshot generation algorithm using JAL. The verifier can use the technique to generate legal *scenarios* by using correct JAL operation definitions (or more precisely, correct JAL operation definitions in his/her perspective), and generate illegal *scenarios* by injecting errors in the JAL so that it produces illegal *snapshot transitions*.

For example, a verifier can define the legal effects of the operation User::AssignRole as follows:

```
JAL_User_AssignRole
if (!this.userRoles._exists(role)) {
    this.userRoles._add(role);
}
```

Scenario generation algorithm

Inputs. UML design class model, maximum number of operations *Max*, parameter domain definitions, operation JAL definitions, tree node r. Operation invocation patterns.

Outputs. Set of scenarios.

Algorithm steps

For each operation call do:

If operations from root to current tree node r and op match an operation invocation pattern:

1. Create one tree node n and add it as child of r.
2. Store information about the operation call (e.g., operation name, parameters, receiving object identifier) in tree node n.
3. Execute desired JAL description associated with the operation using the start state stored in r to get the next system state. Store the next system state in tree node n.
4. Print the sequence of operation calls from the tree root to tree node n as an output scenario.
5. If *Max > 1*
   a) Call the scenario generation algorithm recursively with tree node n and *Max* - 1 as maximum number of operations.

Figure 8.1. Scenario generation algorithm

The scenario generation algorithm is described in Fig. 8.1. Snapshot transitions are generated by traversing the operation invocation tree and interpreting the associated JAL descriptions of behavior using UMLAnT. The verifier then needs to determine whether the generated scenarios describe legal or illegal behaviors.

8.2 An hierarchical RBAC example

In this section we present a hierarchical RBAC (HRBAC) policy model in two parts: in the first part we give a UML design class model that describes HRBAC classes and operations, in the second part we describe HRBAC constraints using OCL invariants.

8.2.1. HRBAC design class model

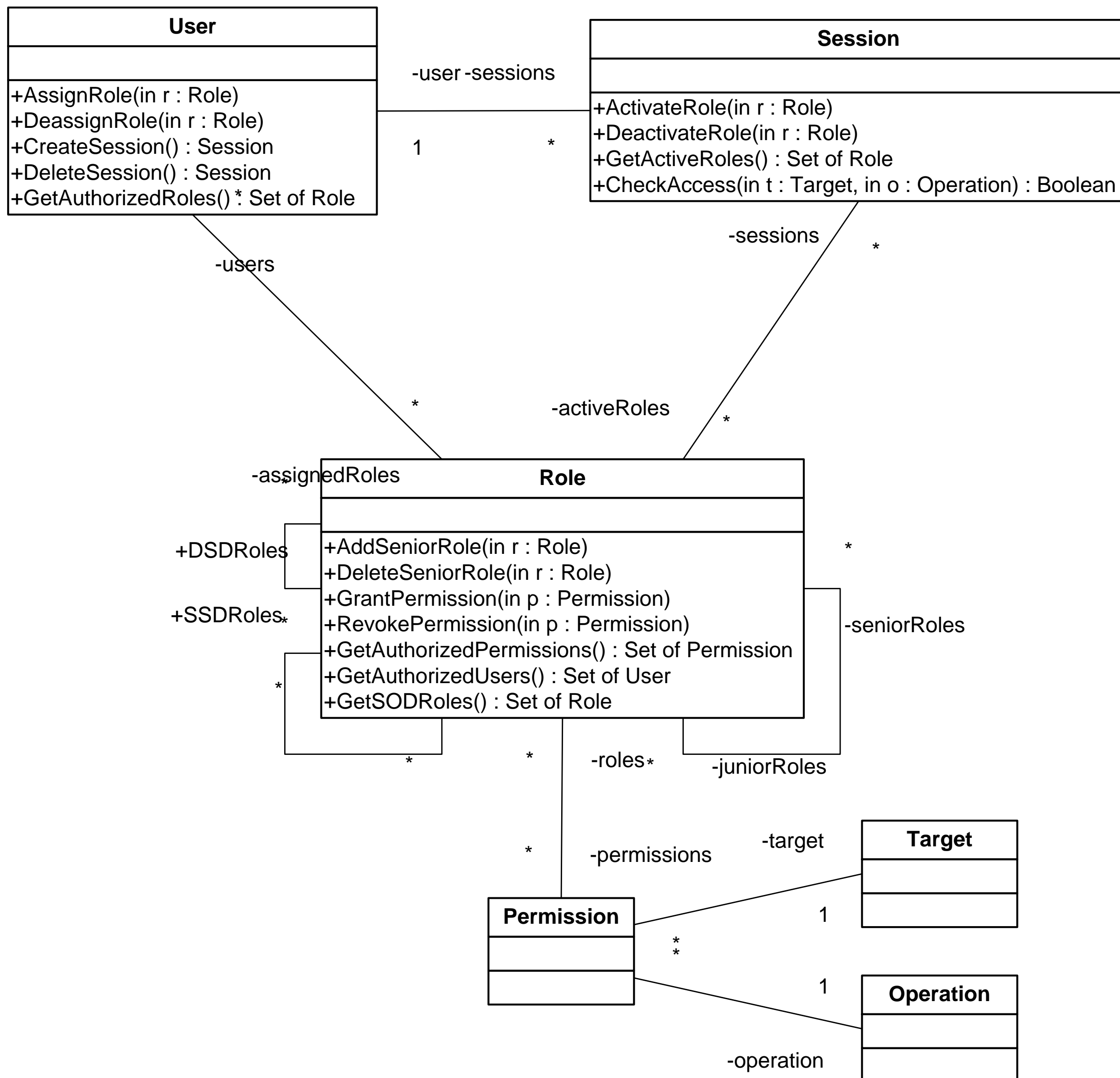


Figure 8.2. Hierarchical RBAC design class model

In the hierarchical RBAC design class model shown in Fig. 8.2, the *User*, *Role* and *Session* classes model users, roles and sessions entities in RBAC. The *Permission* class describes RBAC permissions in terms of operations that can be performed on targets. The *assignedRoles* association end determines the set of roles directly assigned to a user. The operation *GetAuthorizedRoles*() returns all roles directly and indirectly assigned to a user. The *activeRoles* association end determines the set of roles directly activated in a session, and the operation *GetActiveRoles*() returns all roles directly activated in a session. The association end p*ermissions* is the set of all permissions directly associated with a role, and the operation *GetAuthorizedPermissions*() returns all permissions directly and indirectly associated with a role. The *seniorRoles* and *juniorRoles* association ends define the role hierarchy relationships. The *SSDRoles* association end defines the set of role pairs that are constrained by SSD. The *DSDRoles* association end defines the set of role pairs that are constrained by DSD.

Operations are specified using the OCL. For example, the operation *GetAuthorizedRoles*() in *User* is defined using a query operation *GetDominatedRoles*() as follows:

```
// Get set of authorized roles to the user.
context User::GetAuthorizedRoles():Set(Role)
post: result = self.assignedRoles.GetDominatedRoles()->asSet()

// Get set of dominated roles to the role.
context Role::GetDominatedRoles():Set(Role)
body:
let oneStep:Set(Role)= Set{self} in
result = if oneStep.juniorRoles->isEmpty() then
  oneStep
else
oneStep->union(oneStep.juniorRoles.GetDominatedRoles())->asSet()
endif
```

The operations that are involved in the analysis are given below:

```
context User::AssignRole(r:Role)
// Assign a role to the user.
pre: not self.GetAuthorizedRoles()->includes(r)
post: self.GetAuthorizedRoles()->includes(r)

context Session::ActivateRole(r:Role)
// Activate a role in the session.
pre: not self.GetActiveRoles()->includes(r)
post: self.GetActiveRoles()->includes(r)

context Session::GetActivateRoles:Set(Role)
// Return activated roles in the session.
pre: true
post: result = self.activeRole

context Role::AddSeniorRole(r:Role)
// Add a senior role to current role.
pre: true
post: self.seniorRoles->includes(r) and
r.juniorRoles->includes(self)

context Role::CheckAccess(t:Target, o:Operation):Boolean
// Query operation that checks permissions
// of all active roles to see whether there
// is a match for the target and operation.
pre true
post: result =
self.GetActiveRoles().GetAuthorizedPermissions()->exists (p |
p.target = t and p.operation = o)
```

8.2.2. HRBAC constraints

8.2.2.1. Role activation constraint

A fundamental constraint in role activation is that a role can be activated by a user only if it has been assigned to the user. We express this constraint as an OCL invariant named RBAC_Policy_1:

RBAC_Policy_1: *A user can only activate roles that are assigned to him.*

```
context Session
inv RBAC_Policy_1:
self.user.authorizedRoles->
includesAll(self.activeRoles)
```

8.2.2.2. Role hierarchy constraints

According to the definition of role hierarchy in the NIST RBAC standard [Ferraiolo01], a senior role dominating its junior roles implies that the senior role inherits all the permissions of its junior roles, and a junior role inherits all the assigned users of the senior role. RBAC_Policy_2 expresses this constraint:

RBAC_Policy_2: *A senior role inherits all permissions from junior roles, and a junior role inherits all the users of its senior roles.*

```
context Role
inv RBAC_Policy_2:
seniorRoles->forAll(s | s.authorizedPermissions->
intersection(self.authorizedPermissions) =
self.authorizedPermissions) and
self.seniorRoles->forAll(s | s.authorizedUsers->
intersection(self.authorizedUsers) = s.authorizedUsers)
```

The role hierarchy is a partial order on roles and there should not be any cycles in the role hierarchy. We use an OCL query operation on roles called *Dominates* in the policy statement. The expression *r1.Dominates(r2)*, where *r1* and *r2* are roles, returns true if *r2* is a descendant of *r1* in a senior-junior role structure. The constraint is expressed by RBAC_Policy_3:

```
context Role::Dominates(r:Role):Boolean
pre true
post:
if (self.juniorRoles->includes(r)) then
result = true
else
result = self.juniorRoles->exists(j | j.Dominates(r))
endif
```

```
RBAC_Policy_3: There must be no cycles in senior-junior role
relationships.
context Role
inv RBAC_Policy_3:
not self.Dominates(self)
```

8.2.2.3. Separation of duty constraints.

RBAC_Policy_4 expresses the static separation of duty constraint, and RBAC_Policy_5 expresses the dynamic separation of duty constraint:

```
RBAC_Policy_4: Conflict of interest roles cannot be assigned to the
same user (SSD).
context User
inv RBAC_Policy_4:
not self.GetAuthorizedRoles()->exists(r1, r2 |
r1.SSDRoles->includes(r2))
```

RBAC_Policy_5: *Conflict of interest roles can not be activated by the same user simultaneously (DSD).*

```
context User
inv RBAC_Policy_5:
not self.sessions.GetActiveRoles()->exists(r1, r2 | r1.DSDRoles->
includes(r2))
```

8.3. Analyze HRBAC constraints

In this sub-section we show how some of the HRBAC constraints given in Section 8.2 can be analyzed using the method.

8.3.1. Analyze role activation constraint.

To analyze the role activation constraint (*RBAC_Policy_1*), we use the following operation invocation pattern:

Initial State Constraint

Domain*(User) = {Bob}*

Domain*(Role)={clerk, seniorClerk}*

Call Pattern

[no Bob.AssignRole(r)]{0..2}

*Bob.CreateSession(.)****return****(s:Session) Bob->s.ActivateRole(r){1..2}.*

The expression *[no Bob.AssignRole(r)]* is used to match all operation calls except calls of the form *Bob.AssignRole(r)*.

The above pattern describes sequences of operations which end with 1 or 2 invocations of the *ActivateRole()* operation, and start with 0 to 2 operation invocations that do not include operation calls that assign the activated roles to the user *Bob*.

The verifier describes the effect of the *ActivateRole* operation using JAL – The JAL description simply activates the role. Scenarios generated from this pattern would allow roles to be activated even though they are not assigned to the user. For this reason, the verifier knows that the pattern would produce illegal scenarios.

An example of an illegal scenario generated from the above pattern is described as below:

- The scenario starts from an initial system state with one user instance *Bob* and one Role instance *clerk*.
- The user *Bob* creates one session. After the operation is called, a new Session instance *session* is created.
- The user *Bob* activates the *clerk* role. After the Session::ActivateRole operation is called, the activation succeeds and *clerk* is added to the *activeRoles* association of the session.

The HRBAC design model should reject the illegal behavior described by the scenario. Analysis with USE revealed that the HRBAC design model is consistent with the scenario. The defect in the design class model is that the operation *Session::ActivateRole* activates any role that is not activated. The pre-condition should check whether the role is assigned or not.

8.3.2. Analyze separation of duty constraints.

We use the following operation invocation pattern to check enforcement of the SOD constraints:

Initial State Constraint

Domain*(User) = Bob*

cashier ***in Domain****(Role)*

accountant ***in Domain****(Role)*

cashier ***in*** *accountant.SSDRoles // the roles conflict*

Call Pattern

[

*[.]**

Bob.AssignRole(cashier)

Bob.AssignRole(accountant)

]{1}

[

*Bob.CreateSession(.)***return***(s:Session)*

s.ActivateRole(r){2..4} ***where(*** *r = accountant* ***and*** *r = cashier)*

]{0..1}

The expression *[.]* matches any operation call and "*" represents the multiplicity "0 or more". The where clause stipulates that at least one of the *Activate*() calls must activate the *accountant* role, and at least one of the *Activate*() calls must activate the *cashier* role.

The illegal scenario below is generated from the pattern:

- The scenario starts in a state consisting of two conflict of interest roles, *cashier* and *accountant*, and a user *Bob*.
- *User::AssignRole* operation is called to assign the *cashier* role to user *Bob*. After the operation is called, *Bob* is assigned the *cashier* role.
- *User::AssignRole* operation is called to assign the *accountant* role to user *Bob*. After the operation is called, *Bob* is assigned the *accountant* role.

The scenario violates the static separation of duty constraint defined as RBAC_Policy_4 and thus it should be rejected by the HRBAC design. In the design model, the *User::AssignRole* operation specified in Section 2 only checks whether the role is assigned to the user or not before it assigns the role, so that the illegal scenario is consistent with the HRBAC design. To enforce the static separation of duty constraint in an HRBAC design, the operation should also check whether the role to be assigned is in conflict of interest with roles that have been assigned to the user.

The illegal scenario below was used to analyze the dynamic separation of duty constraint:

- The scenario starts in a state consisting of two conflict of interest roles, *cashier* and *accountant*, and a user *Bob*.
- *Bob* creates a new Session instance *session*.
- *User::AssignRole* operation is called to assign the *cashier* role to user *Bob*. After the operation is called, *Bob* is assigned the *cashier* role.
- *User::AssignRole* operation is called to assign the *accountant* role to user *Bob*. After the operation is called, *Bob* is assigned the *accountant* role.
- *Session::ActivateRole* operation is called to activate the *cashier* role in the Session instance. After the operation is called, the *cashier* role is activated.
- *Session::ActivateRole* operation is called to activate the *accountant* role in the Session instance. After the operation is called, the *accountant* role is activated.

In the illegal scenario the user *Bob* is assigned two conflict of interest roles cashier and accountant, and *Bob* activates both roles in one session. Again, the *Session::ActivateRole()* operation does not check that the role to be activated is in a conflict of interest with a role in a session created by the user.

Chapter 9

Generating Scenarios using OCL Operation Definitions

This chapter presents another scenario generation technique using OCL operation definitions [Yu12]. To generate snapshot transitions, the UML class diagram and OCL operation definitions are transformed to Alloy to generate scenarios.

The rest of this section is organized as follows: section 9.1 discusses the Location-aware Role-Based Access Control example UML model. Section 9.2 discusses the scenario generation technique and applies the technique to analyze the example model.

9.1 The Location-aware Role-Based Access Control model

The Location-aware Role-Based Access Control (LRBAC) is an extension to the standard RBAC model [Ray05] [Ray06] [Ray07]. LRBAC uses spatial information of the user and object to enhance the security of location-sensitive applications. In LRBAC, user and object are both associated with locations. The location information of the user and object is taken into consideration in determining whether the user can access the object. The role is associated with assign location and activation location. A role can only be assigned (activated) by a user when the user location is in the assign (activation) location of the role. The permission is also associated with role location and object location. A user acquires certain permission to operate an object only if the user activates the role that is granted the permission and the user location is in role location of the permission and the object location is in object location of the permission.

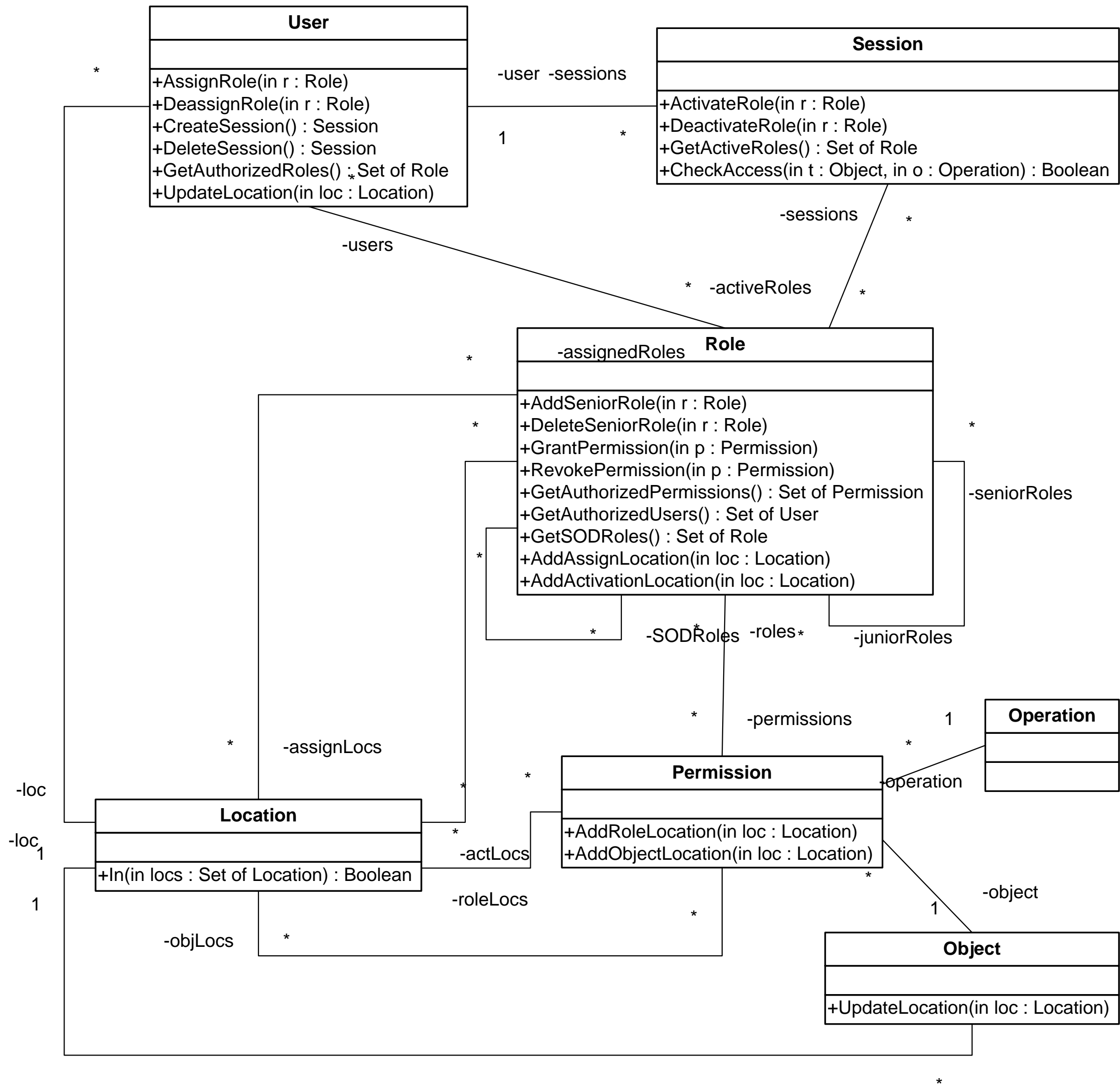


Figure 9.1. The LRBAC UML design class diagram

In the LRBAC design class model (Fig. 9.1), the *User*, *Role, Session* and *Permission* classes model users, roles, sessions and permissions entities in standard RBAC. The *Permission* class describes RBAC permissions in terms of operations that can be performed on objects. The *assignedRoles* association end of the *Role* class determines the set of roles directly assigned to a user. The operation *GetAuthorizedRoles*() returns all assigned roles and their dominated roles indirectly assigned to the user. The *activeRoles* association end determines the set of roles

activated in a session, and the operation *GetActiveRoles*() returns all roles activated in a session. The association end p*ermissions* is the set of all permissions directly associated with a role, and the operation *GetAuthorizedPermissions*() returns all permissions associated with a role and its dominated roles. The *seniorRoles* and *juniorRoles* association ends define the role hierarchy relationships. The *SODRoles* association end defines the set of separation of duty role pairs.

The *Location* class describes the new location entity in LRBAC. In location-aware applications the location of the users and objects can be updated and queried. The *UpdateLocation* operation sets the new locations of the user or object. The *loc* association ends in User-Location and Object-Location associations return the updated location of the user or object. The method *Location::In* checks whether the location is contained by a set of locations. The *assignLocs* and *actLocs* describes the set of assign locations and activation locations of the role. The *roleLocs* and *objLocs* association ends describe the set of role locations and object locations of the permission.

Operations are specified using the OCL. The operations that are involved in the analysis are given below:

```
context User::AssignRole(r:Role)
// Assign a role to the user.
pre: not self.GetAuthorizedRoles()->includes(r) and
self.loc.In(r.assignLocs)
post: self.GetAuthorizedRoles()->includes(r)

context Session::ActivateRole(r:Role)
// Activate a role in the session.
pre: not self.GetActiveRoles()->includes(r) and
self.user.loc.In(r.assignLocs) and
self.user.loc.In(r.actLocs)
```

```
post: self.GetActiveRoles()->includes(r)

context Session::CheckAccess(t:Object, o:Operation):Boolean
pre: true
post: result =
self.GetActiveRoles().GetAuthorizedPermissions()->exists (p |
p.object = t and p.operation = o and self.user.loc.In(p.roleLocs)
and o.loc.In(p.objLocs)))
```

9.2 The scenario generation technique

The scenario generation technique (see Fig. 9.2) requires the verifier to create *scenario generation criteria* and OCL *operation definitions* for operations that will be used in generated scenarios. The technique uses the static aspects of the UML design class model (i.e., the classes and associations, but not the operation specifications), and the verifier's OCL operation definitions to generate an Alloy model. The *scenario generation criteria* are used to produce Alloy predicates that are included in the Alloy model. These predicates are run using the Alloy Analyzer to generate snapshot transition sequences expressed as Alloy instance models. The Alloy instance models are then transformed to snapshot transition sequences that can be input to USE for analysis.

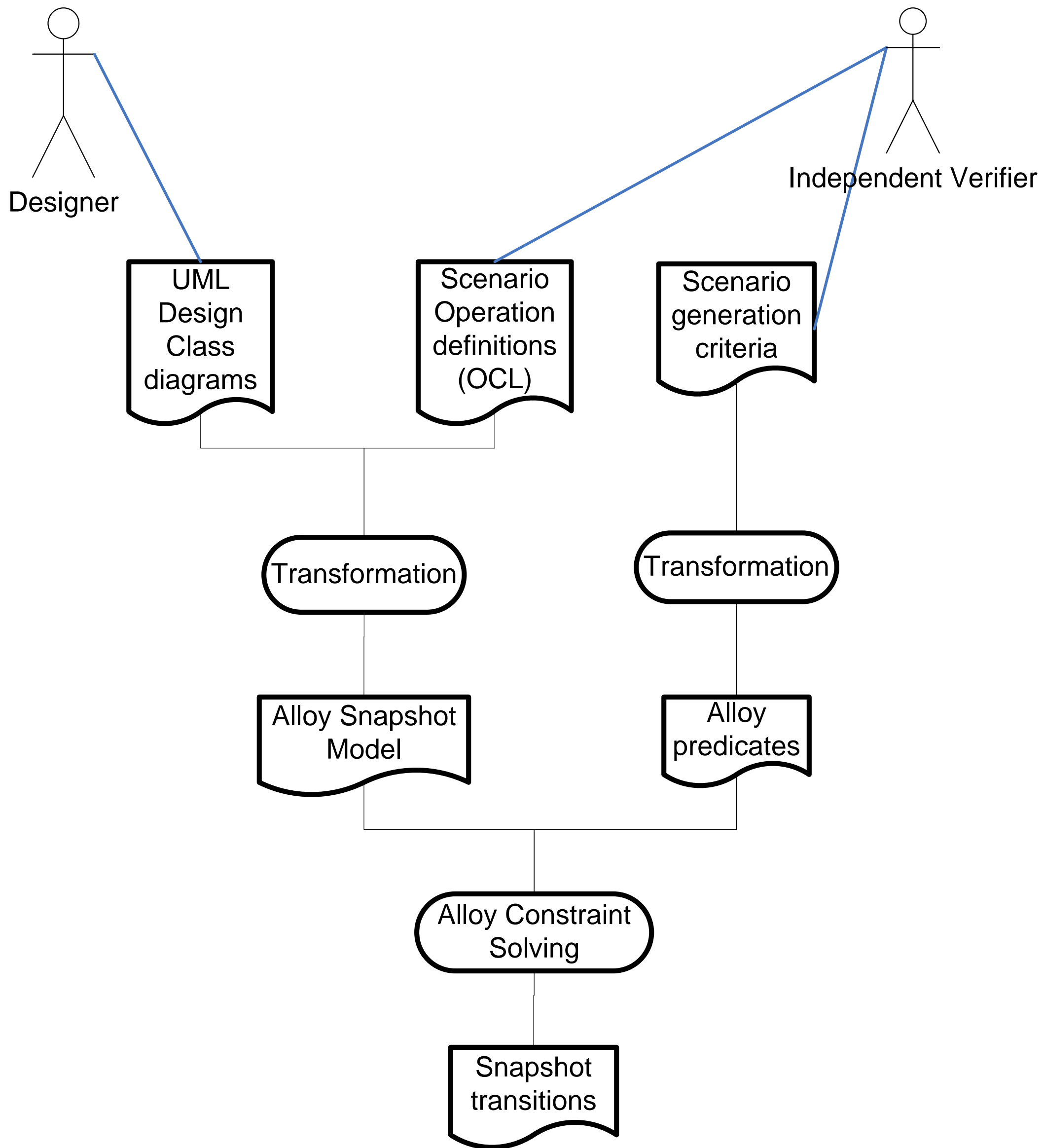


Figure 9.2. Generating transition sequences

The rest of the sections are organized as follows: In section 9.2.1 we describe the types of scenario generation criteria that verifiers can define. In section 9.2.2 we give examples of scenario operation definitions and in section 9.2.3 we describe how scenarios are generated.

9.2.1. Defining scenario generation criteria

In the extended Scenario-based UML Design Analysis technique, a verifier can define the following types of scenario generation criteria:

- **Operation sequence criteria**: an operation sequence criterion characterizes a family of operation sequences. Scenarios that satisfy this type of criteria must include operation calls that abide by the relative ordering of calls defined by the criterion.
- **Structural coverage criteria**: a structural coverage criterion specifies properties of objects and associations that must hold in snapshots before and after each operation. These properties are expressed as OCL constraints.
- **Operation coverage criteria**: an operation coverage criterion specifies operation behaviors that must be covered in the generated scenarios. These criteria are specified using OCL constraints.

A scenario generation criterion consists of an initial state constraint part in which the verifier specifies structural constraints, a call pattern part in which the verifier specifies an operation sequence criterion, an optional structural coverage criterion, and an operation constraint part in which the verifier specifies optional operation coverage criteria. This form builds upon our early work on *operation invocation patterns* [Yu09].

The following describes the criteria that will be used to generate scenarios for analyzing the LRBAC model. The criteria we use characterize scenarios that will be used to analyze check access behaviors involving users updating their locations after activating assigned roles. The intent is to check that the design model properly handles access control when a user changes location.

Operation sequence criteria. The verifier defines an operation sequence criterion in the form of an operation invocation pattern. In the pattern, a user creates a session, and some time after the user is assigned a role that is later activated; after, the user updates its location and then a request is made to access a resource which triggers an invocation of the *CheckAccess()*

operation. The operation sequence criterion is expressed as a pattern as shown below (the numbers in brackets restrict the number of occurrences of the operation calls that can be made):

```
User::CreateSession(){1}
User::AssignRole(){1}
Session::ActivateRole(){1}
User::UpdateLocation(){1}
Session::CheckAccess(){1}
```

An operation sequence that satisfies this criterion is shown in Fig. 9.3.

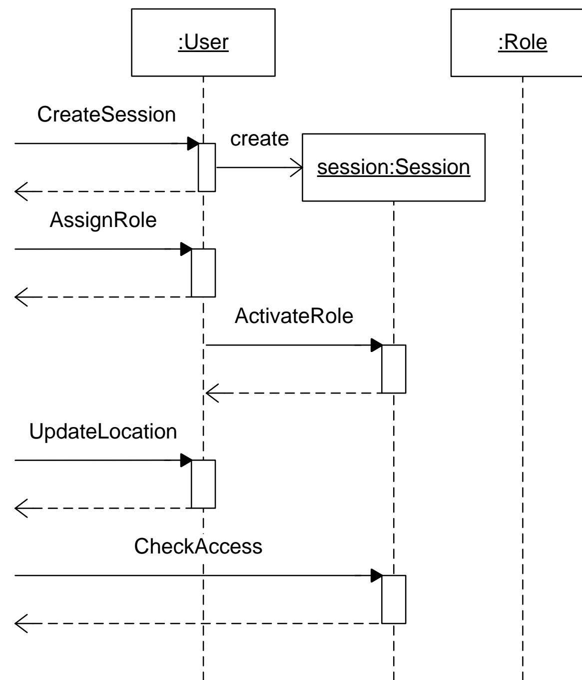


Figure 9.3. The analysis operation sequence

Structural coverage criteria. The verifier defines a criterion stating that the snapshot before the *CheckAccess()* operation in Fig. 9.3 must satisfy the following property (# is the set cardinality operator):

```
#User = 1 and #Location = 2 and #Role = 1 and #Role.permissions =
1 and User.Loc  <> Permission.roleLocs
```

The property states that the snapshot should contain one user, two locations, one role with a granted permission, and that the set of user locations is not equal to the role set of permission role locations.

Operation coverage criteria. The verifier is interested in generating scenarios in which the user location is included in role assignment locations. Thus the following operation coverage criterion is defined for *User::AssignRole* and *Session::ActivateRole* operations. The criterion ensures that the user location is included in role assignment locations before the two operations are called.

```
behavior context:  User::AssignRole(r:Role)
precondition includes: self.loc.In(r.assignLocs)

behavior context: Session::ActivateRole(r:Role)
precondition includes: self.user.loc.In(r.assignLocs) and self.user.loc.In(r.actLocs)
```

All of the above criteria are bundled into the single scenario generation criterion shown below:

```
Initial State Constraint
{}
Call Pattern
[
User::CreateSession(){1}
User::AssignRole(){1}
Session::ActivateRole(){1}
User::UpdateLocation(){1} where (#User = 1 and #Location = 2 and #Role
= 1 and #Role.permissions = 1 and User.Loc  <> Permission.roleLocs )
Session::CheckAccess(){1}
]
Operation Constraint
```

```
{
behavior context:  User::AssignRole(r:Role)
precondition includes: self.loc.In(r.assignLocs)

behavior context: Session::ActivateRole(r:Role)
precondition includes: self.user.loc.In(r.assignLocs) and
self.user.loc.In(r.actLocs)
}
```

9.2.2. Defining scenario operations

An OCL operation specification in a design class model should be complete in the sense that it defines effects for all scenarios involving calls to the operations. A verifier's scenario operation definition does not need to be as encompassing; it should define only the effects produced in the scenarios defined by the verifier.

For example, consider a case in which a verifier analyzes an LRBAC design model using the following scenario:

- A user is in a location in which he cannot activate any roles, and
- The user attempts to retrieve information that he is not allowed to access.

In this scenario the *CheckAccess()* operation should return false, indicating that the user is denied access. The verifier thus defines the *Session::CheckAccess()* operation as follows:

```
context Session::CheckAccess(t:Object, o:Operation):Boolean
pre: not self.user.loc.In(self.activeRoles.assignLocs)
post: result = false
```

Similarly, the verifier defines *User::AssignRole* and *User::UpdateLocation* operations as:

```
context User::AssignRole(r:Role)
pre: not self.assignedRoles->includes(r)
post: self.assignedRoles ()->includes(r)
```

```
context User::UpdateLocation(loc:Location)
pre: true
post: not loc = loc@pre
```

9.2.3. Generating scenarios

This section discusses four major steps in the scenario generation process: the first step generates the Alloy snapshot transition model, the second step generates the snapshot sequence constraint, the third step generates Alloy predicates for criteria, and the last step generates Alloy snapshot transitions.

9.2.3.1 Generating the Alloy snapshot transition model.

The verifier's scenario operation definitions and the designer's design class models are transformed to a snapshot transition model, which is then transformed to an Alloy model. In this step we use the design class diagram created by the designer and the OCL operation definitions created by the verifier to generate a snapshot transition model. Details of the snapshot transition model transformation algorithm are described in [Yu08]. The Alloy snapshot transition model includes the following elements:

1. *A signature for each class in the UML class diagram:* All attributes in the UML class are transformed to fields of the signature, and class invariants are expressed as predicates in the Alloy. Rules on how to transform a UML class model to Alloy are discussed in [Anastasakis10]. For example, in the LRBAC example, the following signatures are generated:

   ```
   sig User{}
   sig Role{}
   sig Session{}
   ```

2. *A snapshot signature that includes:*
   - Set of objects for each signature generated in the above step.

- All associations in the design class diagram are specified as fields, and additional constraints that force the associations to link objects in the snapshot only are added to the Alloy model.

Part of the *Snapshot* signature for the LRBAC example is shown below:

```
sig Snapshot {
    // LRBAC Objects
    users:some User,
    roles:some Role,
    sessions:some Session,
    permissions:some Permission,
    operations: some Operation,
    objects: some Object,
    locations: some Location,
    // LRBAC associations
    userrole: User set ->set Role,
    sessionrole:Session set->set Role
    …
}
```

3. *A transition signature that includes a before and after snapshot:* An example is given below.

```
abstract sig Transition
{
    before: one Snapshot,
    after: one Snapshot
}
```

4. *A specialized signature (sub-signature) of the Transition signature for each operation in the design class model:* The sub-signature contains fields representing pre- and post-forms of parameters as defined in the snapshot transition model. The OCL specification of the

operation is transformed to constraints of the sub-signature. We finally add frame constraints to the sub-signature to make that objects and associations not affected by the operation remain the same in before and after snapshots. For example, we generate the following *User_UpdateLocation_Transition* signature for *User::UpdateLocation()* operation:

```
sig User_UpdateLocation_Transition
extends Transition
{
   uPre:User,
   uPost:User,
   locPre:Location,
   locPost:Location,
}{
   // Postcondition
   uPre.(before.userlocation) = locPre
   uPost.(after.userlocation) = locPost
   locPre != locPost

   // Frame conditions
   uPre = uPost
   uPre in before.users
   locPre in before.locations
   uPost in after.users
   locPost in after.locations
   …
}
```

9.2.3.2 Generating the snapshot sequence constraint.

In this step, a snapshot sequence constraint is generated in order to associate two consecutive snapshots with a transition. First, an Alloy *ordering* type is used to cast a set of *states* into a

sequence of *states* (e.g., open util/ordering[Snapshot] as SO). Second, an Alloy fact, *traces*, is defined to relate a snapshot to its next snapshot through a transition as shown below:

```
open util/ordering[Snapshot] as SO
fact traces {
all s: Snapshot - SO/last |
let s' = s.next | one t : Transition |
t.before = s and t.after = s'}
```

9.2.3.3 Generating Alloy predicates for criteria.

In this step, the scenario coverage criteria are translated to Alloy predicates. Each operation sequence criterion is translated to an Alloy predicate. In the example, the scenario operation sequence pattern contains five operations: *User::CreateSession(), User::AssignRole(), Session::ActivateRole(), User::UpdateLocation()* and *Session::CheckAccess().*

The pattern is transformed to an Alloy predicate as below:

```
pred operation_pattern1 {
one s: Snapshot - SO/last | let s0 = s | let s1 = SO/next[s0] |
   let s2 = SO/next[s1] | let s3 = SO/next[s2] |
    let s4 = SO/next[s3] | let s5 = SO/next[s4] |
one t1: User_CreateSession_Transition,
    t2 : User_AssignRole_Transition,
   t3 : Session_ActivateRole_Transition,
   t4 : User_UpdateLocation_Transition,
   t5: Session_CheckAccess_Transition |
 t1.before = s0 and t1.after = s1 and
 t2.before = s1 and t2.after = s2 and
 t3.before = s2 and t3.after = s3 and
 t4.before = s3 and t4.after = s4 and
 t5.before = s4 and t5.after = s5}
```

Each structural coverage criterion is translated to a predicate in the operation pattern generated above. For example the following structural coverage criterion:

```
#User = 1 and #Location = 2 and #Role = 1 and
#Role.permissions = 1 and
User.Loc  <> Permission.roleLocs
```

is translated to predicates on *s4* in *operation_pattern1*:

```
#s4.users = 1 and #s4.locations = 2 and #s4.role = 1 and
#s4.rolepermission = 1 and
(s4.users).(s4.userlocation)
   != (s4.permissions).(s4.permrolelocation)
```

Each operation coverage criterion is translated to a predicate in its corresponding Transition signature. For example, the following operation coverage criterion:

```
behavior context:  User::AssignRole(r:Role)
precondition includes: self.loc.In(r.assignLocs)
```

is translated to the following predicate in *User_AssignRole_Transition:*

```
uPre.(before.userlocation) in rPre.(before.roleassignlocation)
```

9.2.3.4 Generating Alloy snapshot transitions.

By running the alloy predicates, we will get a set of snapshot transitions. For example, one possible snapshot before and after the transition specified by *User_UpdateLocation_Transition* is shown in Fig. 9.4 and Fig. 9.5. In the before snapshot, the user is at *Location0*, and the user location is included in role assign location and role activation locations of the role, thus the user has permission of operation on the object. In the after snapshot, the user location is updated to *Location1*, and *Location1* is not included in role assign and activation locations, so that *Session::CheckAccess()* should return false after this user location update.

If we check the *Session_CheckAccess_Transition* snapshot transition against the original snapshot transition model, we will find that it is not consistent with the snapshot transition model. The reason is that the *Session::CheckAccess()* operation specification in the design model does not check whether the role is still enabled after the user changes location. If we add the conditions below (shown in bold text) to the specification, it will resolve the inconsistency:

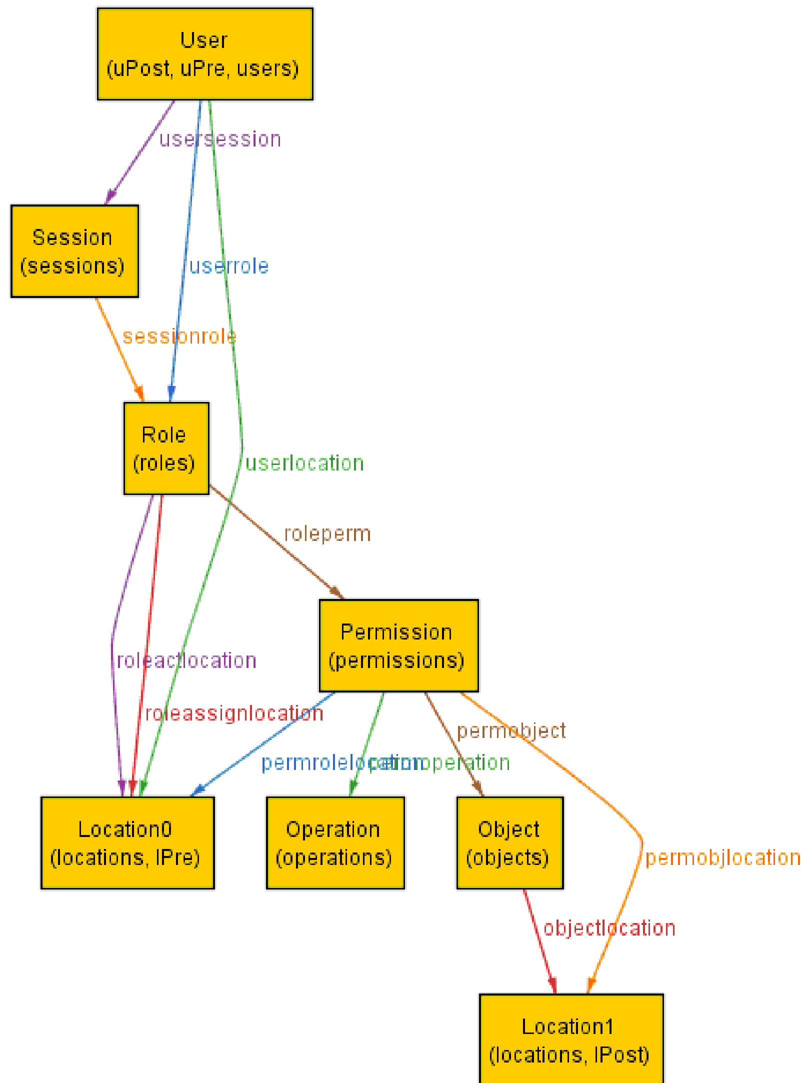


Figure 9.4. Snapshots before User_UpdateLocation_Transition

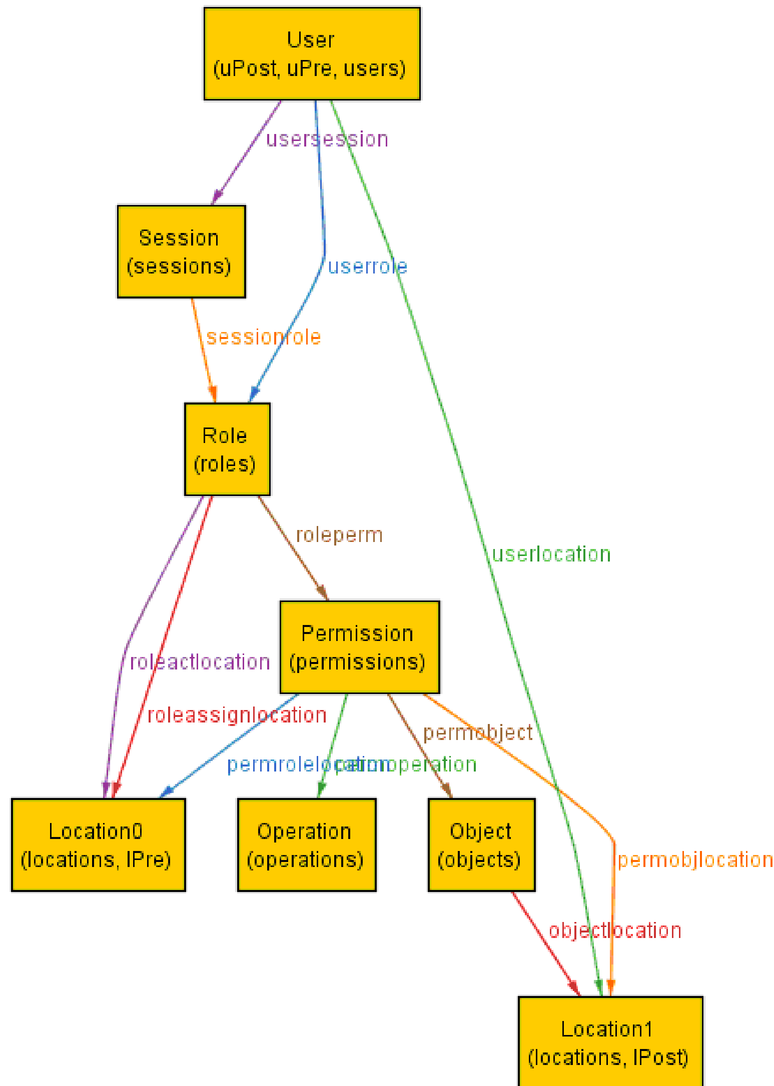


Figure 9.5. Snapshots after User_UpdateLocation_Transition

```
context Session::CheckAccess(t:Object, o:Operation):Boolean
pre: true
post: result = self.GetActiveRoles()
->exists( r | self.user.loc.In(r.assignLocs) and
self.user.loc.In(r.actLocs) and
r.GetAuthorizedPermissions()->exists (p | p.object = t and
p.operation = o and
self.user.loc.In(p.roleLocs) and o.loc.In(p.objLocs)))
```

Chapter 10

Conclusions and Future Work

This chapter summarizes the dissertation. Section 10.1 describes contributions of the dissertation. Section 10.2 reviews and answers research questions. Section 10.3 discusses future work.

10.1 Contributions

This main contribution of the dissertation is a lightweight and static technique for analyzing UML design class models. A UML design class model is analyzed against a set of scenarios that describe desired or undesired behaviors created from the verifier's perspective. The analysis technique is lightweight because it analyzes functionality specified in a UML design class model within the scope of a set of scenarios. It is static because it does not require that the UML design class model be executable. The technique does not transform UML design models to other formal notations such as Alloy, the analysis is done by leveraging existing UML structural analysis tool USE.

The technique is a consistency checking technique. Inconsistencies imply errors in the UML design class model, errors in the scenarios or errors in both the UML design and scenarios. It is up to the modeler and the verifier to analyze the inconsistencies, find the cause of the inconsistencies and resolve the inconsistencies. After the design error is identified and fixed, the technique can be used to check whether the inconsistencies have been resolved in the updated UML design and scenarios.

The dissertation presents a Scenario-based UML Design Analysis tool developed using Kermeta and Eclipse Modeling Framework. The tool can be used to transform Ecore design class model to a USE snapshot transition model, and transform scenarios to snapshot transitions that can be input to USE.

We used the Scenario-based UML Design Analysis technique to analyze two UML design class models: a Train Management System model and a Generalized Spatio-Temporal RBAC model. The case studies show how the technique can be used to check inconsistencies between the UML design class models and scenarios.

We performed a pilot study of two design class models to evaluate the effectiveness of the Scenario-based UML Design Analysis technique. In the pilot study of two UML designs, the technique uncovered at least as many design inconsistencies as manual inspection techniques uncovered, and the technique did not uncover false inconsistencies. The pilot study shows the technique seems to be effective.

The dissertation presents two scenario generation techniques. These techniques can be used to ease the manual effort needed to produce scenarios. Based on the verifier's operation definitions the scenario generation techniques can be used to automatically generate a family of scenarios that conform to patterns of operation sequences.

10.2 Discussions of research questions

This section reviews and answers five research questions and discusses open issues of the research.

Research question 1: How can a scenario be checked against a UML design class model?

The technique is used to (1) transform a UML design class model to a snapshot transition model that captures valid state transitions, (2) transform scenarios to snapshot transitions and (3) check whether the snapshot transitions are instances of the snapshot transition model using USE.

Research question 2: Can existing structural analysis tools such as USE be leveraged to support scenario-based analysis of class models?

Existing UML analysis tools such as USE can be used to check whether a snapshot is an instance of a UML design class model. The technique leverages existing USE tool to check whether the snapshot transitions transformed from scenarios are instances of the snapshot transition model.

Research question 3: How effective is the Scenario-based UML Design Analysis technique in terms of the number of design inconsistencies that can be uncovered?

The pilot study of two design class models shows that the Scenario-based UML Design Analysis technique seems to be effective, as it uncovered at least as many design inconsistencies as manual inspection techniques uncovered and it did not uncover any false inconsistencies. Due to the lack of graduate students to create scenarios and manually inspect design inconsistencies, we cannot control the number of students and number of UML designs in the pilot study. A formal controlled experiment is desired to further evaluate the technique.

Research question 4: Can scenarios be automatically generated?

Scenarios can be automatically generated. Chapter 8 and 9 presents two scenario generation techniques. The verifier needs to define operation definitions that specify effects of operations and operation sequence patterns. The scenario generation techniques can be used to automatically generate a family of scenarios that conform to the scenario coverage criteria. The criteria are defined by the verifier based on his or her domain knowledge and experience.

However, it is a challenging problem to generate just enough number of scenarios that cover a UML design. The scenario generation techniques discussed in this dissertation are an initial attempt to solve this issue.

Research question 5: Can the technique be scaled to analyze large industrial design models?

Based on the algorithm complexity analysis in section 4.5, the complexity of snapshot transition model generation algorithm depends on the size of the UML design class model and the complexity of OCL operation constraints, and the complexity of snapshot transitions generation is proportional to the number of instances in a scenario. The complexity to check consistency between snapshot transitions and the snapshot transition model in USE depends on the number of operations in a scenario, number of instances in the before and after snapshot and complexity of invariants in the snapshot transition model.

Table 10.1. Time analysis of model transformation

Scenario	**Time (seconds)**
Scenario 1 (TMS)	71
Scenario 1 (GSTRBAC)	39
Scenario 2 (GSTRBAC)	50
Scenario 3 (GSTRBAC)	78
Scenario 4 (GSTRBAC)	78
Scenario 5 (GSTRBAC)	54
Scenario 6 (GSTRBAC)	84
Scenario 7 (GSTRBAC)	64
Scenario 8 (GSTRBAC)	53
Scenario 9 (GSTRBAC)	70
Scenario 10 (GSTRBAC)	69
Scenario 11 (GSTRBAC)	119
Scenario 12 (GSTRBAC)	72

Table 10.1 shows time taken to transform the UML design class model to snapshot transition model and to transform each scenario in the pilot study to snapshot transitions. The time taken to run USE commands to build the snapshot transitions in USE of these scenarios is about 2-3 seconds. The time was measured on a laptop with Intel ® Core ™ 2 Duo CPU T6600 2.20GHz processor and 4GB physical memory. The laptop ran Windows 7 Home Premium operating system, Eclipse SDK Version 3.5.0, KerMeta Version: 1.3.2 and USE 3.0.1.

The complexity analysis and time analysis shows that the technique can possibly be used to analyze larger UML designs and scenarios. Future work is required to analyze and optimize the analysis of large industrial models.

Open issue 1: What kinds of design errors can be uncovered using legal/illegal scenarios?

The technique is a consistency checking technique. A legal scenario is supposed to be consistent with the UML design, and an illegal scenario is supposed to be inconsistent with the UML design.

Based on our study, illegal scenarios are typically used to identify weak pre-condition error and weak post-condition error. These two types of design errors can not be identified by legal scenarios, because the weak pre/post conditions are still consistent with the legal scenarios. For other types of design errors such as strong pre-condition (i.e., the pre-condition is too strong so that some valid inputs are treated as invalid) and unsatisfiable post-condition (i.e., the post-condition is too strong so that it can not be satified), the verifier can create illegal scenarios to identify such errors, but it seems more straightforward to create legal scenarios to identify such design errors.

Future work is required to study how different types of design errors can be uncovered by inconsistencies identified using legal and illegal scenarios.

Open issue 2: Should the verifier mark which part of an illegal scenario is illegal?

It is not required for the verifier to mark which part of an illegal scenario is not legal for consistency checking purpose. However, to help identifying design errors from inconsistencies between an illegal scenario and a UML design, it is recommended that the verifier specifies which part of an illegal scenario is not legal.

10.3 Future work

We studied two demonstration UML designs. Future work should study more complicated industrial UML designs. We need to optimize the snapshot transition model in case the generation of snapshot transition model or USE consistency check becomes a bottleneck in analyzing large industrial models.

We need to further study how different types of design errors are identified from inconsistencies uncovered using legal and illegal scenarios.

In the pilot study two graduate students manually reviewed the UML designs and scenarios. And we studied 13 scenarios of two UML designs. In the future work of formal controlled experiment, larger number of graduate students should be invited and trained to do manual inspection. The controlled experiment should study more scenarios and UML designs, and more design inconsistencies should be seeded.

An effective scenario generation strategy is still open for future research. One future direction is to study how to produce legal and illegal scenarios to cover every branch of OCL operation constraints.

References

[Abdunabi13] Ramadan Abdunabi, Mustafa Al-Lail, Indrakshi Ray, Robert B. France: Specification, Validation, and Enforcement of a Generalized Spatio-Temporal Role-Based Access Control Model. *IEEE Systems Journal* 7(3): 501-515 (2013).

[Al-Lail13] Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, Indrakshi Ray: "An Approach to Analyzing Temporal Properties in UML Class Models", MoDeVVa@MoDELS 2013: 77-86.

[Alloy] D. Jackson, "Alloy: a lightweight object modeling notation", *ACM Transactions on Software Engineering and Methodology*, Volume 11, Issue 2, April 2002, pages 256-290.

[Blum92] Blum, B. I. 1992 Software Engineering: a Holistic View. Oxford University Press, Inc.

[Boehm81] B. W. Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Briand02] Lionel Briand, Yvan Labiche, "A UML-Based Approach to System Testing", *Software and Systems Modeling*, vol. 1 (1), pp. 10-42, 2002.

[Brucker08] Achim D. Brucker and Burkhart Wolff. HOL-OCL - A Formal Proof Environment for UML/OCL. In *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science (4961), pages 97-100.

[Büttner04] Fabian Büttner and Martin Gogolla. On Generalization and Overriding in UML 2.0. In Jean Bezivin, Thomas Baar, Tracy Gardner, Martin Gogolla, Reiner H?hnle, Heinrich Hu?mann, Octavian Patrascoiu, Peter H. Schmitt, and Jos Warmer, editors, Proc. *UML'2004 Workshop OCL and Model Driven Engineering*, pages 69-69. In: UML - Modeling Languages and Applications.

[Clark99] E. Clark, O. Grumberg, and D. Peled. Model Checking. The MIT Press, 1999.

[Clarke01] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2001. Progress on the State Explosion Problem in Model Checking. In *informatics - 10 Years Back. 10 Years Ahead.*

[Conradi03] Reidar Conradi, Parastoo Mohagheghi, Tayyaba Arif, Lars Christian Hedge, Geir Arne Bunde, and Anders Pedersen. Object-oriented reading techniques for inspection of UML models – an

industrial experiment. In *Proceedings of ECOOP'03*, volume 2749 of LNCS, pages 483–501. Springer, July 2003.

[EMF] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/?project=emf

[Eshuis06] Eshuis, R. 2006. Symbolic model checking of UML activity diagrams. *ACM Trans. Softw. Eng. Methodol*. 15, 1 (Jan. 2006), 1-38.

[Fagan76] M.E., Fagan (1976). "Design and Code inspections to reduce errors in program development". *IBM Systems Journal* 15 (3): pp. 182–211.

[Ferraiolo01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.* 4, 3 (August 2001), 224-274.

[Garcia07] Miguel Garcia, "How to process OCL Abstract Syntax Trees", Technische Universität Hamburg-Harburg (Germany), June 2007.

[IEEE1028] IEEE std 1028-1988, IEEE Standard for Software Reviews and Audits (ANSI).

[Isabelle02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic", Springer-Verlag, Berlin, Heidelberg, 2002.

[Jacobson92] I. Jacobson, M. Christerson, P. Jonsson, G. .vergaard: Object Oriented Software Engineering: A Use Case Driven Approach. Amsterdam: Addison-Wesley, 1992.

[Kermeta] Kermeta language reference manual, http://www.kermeta.org/

[Krieger08] Krieger, M. P. & Knapp, A. Executing Underspecified OCL Operation Contracts with a SAT Solver. *ECEASST*, 2008, 15.

[Kundu09] Debasish Kundu and Debasis Samanta, "A Novel Approach to Generate Test Cases from UML Activity Diagrams", *Journal of Object Technology*, Volume 8, no. 3 (May 2009), pp. 65-83.

[Lilius99] J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. Proc. of the *International Conference on the Unified Modelling Language: Beyond the Standard (UML'99)*, volume 1723 of Lecture Notes in Computer Science, pages 430-445, USA, 1999. Springer-Verlag.

[MOF] Meta Object Facility (MOF) Core Specification, Object Management Group, Version 2.0.

[Muller05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel, "Weaving Executability into Object-Oriented Meta-Languages", Proceedings of *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Jamaica, 2-7 October 2005.

[Nebut06] Cle'mentine Nebut, Franck Fleurey, Yves Le Traon, Jean-Marc Je'ze' quel, "Automatic Test Generation: A Use Case Driven Approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140-155, Mar. 2006, doi:10.1109/TSE.2006.22.

[NuSMV99] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri. In N. Halbwachs and D. Peled, editors: "NuSMV: a new symbolic model verifier", *Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, In Lecture Notes in Computer Science, number 1633, pages 495-499, Trento, Italy, July 1999. Springer.

[OCLE] D. Chiorean, M. Pasca, A. Carcu, C. Botiza, S. Moldovan, "Ensuring UML Models Consistency Using the OCL Environment", *Electronic Notes in Theoretical Computer Science*, Volume 102, Nov. 2004, pages 99-110.

[OCL] Object Management Group, Object Constraint Language Specification, Version 2.3.

[Oliver99] Iam Oliver, Stuart Kent, "Validation of Object Oriented Models using Animation," euromicro, vol. 2, pp.2237, *25th Euromicro Conference (EUROMICRO '99)*-Volume 2, 1999.

[Ray05] Indrakshi Ray and Lijun Yu, "Short Paper: Towards a Location-Aware Role-Based Access Control Model", Proceedings of the *1st IEEE Conference on Security and Privacy for Emerging Areas in Communication Networks*, Athens, Greece, September 2005.

[Ray06] Indrakshi Ray, Mahendra Kumar, and Lijun Yu, "LRBAC: A Location-Aware Role-Based Access Control Model", Proceedings of the *2nd International Conference on Information Systems Security*, Kolkata, India, December 2006.

[Ray07] Indrakshi Ray, Manachai Toahchoodee: A Spatio-temporal Role-Based Access Control Model. *DBSec* 2007: 211-226.

[Shah09] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. 2009. From UML to Alloy and back again. In Proceedings of the *6th International Workshop on Model-Driven Engineering,*

Verification and Validation (MoDeVVa '09), ACM, New York, NY, USA, , Article 4 , 10 pages. DOI=10.1145/1656485.1656489 http://doi.acm.org/10.1145/1656485.1656489

[Steinberg09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, EMF Eclipse Modeling Framework, Second Edition, Addison-Wesley, 2009.

[Sutcliffe98] Alistair G. Sutcliffe, Neil A.M. Maiden, Shailey Minocha, Darrel Manuel, "Supporting Scenario-Based Requirements Engineering," *IEEE Transactions on Software Engineering*, pp. 1072-1088, December, 1998.

[Torlak07] Emina Torlak, Daniel Jackson: "Kodkod: A Relational Model Finder", in *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference,* 632-647, March 2007.

[Travassos02] Travassos, G.H., Shull, F., Carver, J., Basili, V.R.: Reading Techniques for OO Design Inspections. *University of Maryland Technical Report CS-TR-4353*. April 2002, http://www.cs.umd.edu/Library/TRs/CS-TR-4353/CS-TR-4353.pdf.

[Travassos99] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. 1999. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '99)*, A. Michael Berman (Ed.). ACM, New York, NY, USA, 47-56. DOI=10.1145/320384.320389 http://doi.acm.org/10.1145/320384.320389 3.1. Sample selection and training and grouping.

[Trung05] T. Dinh-Trong, N. Kawane, S. Ghosh, R. B. France, and A. A. Andrews. "A Tool-Supported Approach to Testing UML Design Models", Proceedings of the *10th IEEE International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society Press, pp.519-528, Shanghai, China, June 16-20, 2005.

[UML2Alloy] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, Indrakshi Ray: On challenges of model transformation from UML to Alloy. *Software and System Modeling* 9(1): 69-86 (2010).

[UML] Object Management Group, Unified Modeling Language: Superstructure, vers 2.4, Final Adopted Standard.

[USE] Gogolla, M., Büttner, F., and Richters, M. 2007. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program*. 69, 1-3, December 2007.

[Valmari98] Valmari, A. 1998. The State Explosion Problem. In Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets W. Reisig and G. Rozenberg, *Eds. Lecture Notes In Computer Science*, vol. 1491. Springer-Verlag, London, 429-528.

[Whittle03] Jonathan Whittle. 2003. Formal approaches to systems analysis using UML: an overview. In *Advanced topics in database research vol. 1, Keng Siau (Ed.)*. IGI Publishing, Hershey, PA, USA 324-341.

[Wohlin12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, Anders Wesslén, Experimentation in Software Engineering, Springer, 2012, ISBN: 978-3642290435.

[Yu07] Lijun Yu, Robert B. France, Indrakshi Ray, and Kevin Lano, "A Light-Weight Static Approach to Analyzing UML Behavioral Properties", Proceedings of the *12th IEEE International Conference on Engineering of Complex Computer Systems*, Auckland, New Zealand, July 2007.

[Yu08] Lijun Yu, Robert France, Indrakshi Ray, "Scenario-based Static Analysis of UML Class Models", *Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France, Sep. 28-Oct.3, 2008.

[Yu09] Lijun Yu, Robert France, Indrakshi Ray, Sudipto Ghosh, "A Rigorous Approach to Uncovering Security Policy Violations in UML Designs", *Proceedings of the 14th International Conference on Engineering Complex Computer Systems*, Potsdam, Germany, June 2009.

[Yu12] lijun Yu, Robert B. France, Indrakshi Ray, Wuliang Sun: "Systematic Scenario-Based Analysis of UML Design Class Models", *Proceedings of the 17th International Conference on Engineering Complex Computer Systems*, Paris, France, July 2012.