Systematic Scenario-Based Analysis of UML Design Class models

Lijun Yu, Robert B. France, Indrakshi Ray, Wuliang Sun
Department of Computer Science, Colorado State University, USA
{lijun, france, iray, sunwl}@cs.colostate.edu

Abstract

Scenario-based UML Design Analysis (SUDA) is a lightweight technique for analyzing behavior specified in UML design class models. A person (verifier) charged with verifying a design class model developed by an independent group of modelers can use SUDA to analyze the design class model against a set of scenarios manually created by the verifier. In this paper we describe an extension of SUDA that allows a verifier to automatically generate scenarios satisfying criteria defined by the verifier. We illustrate the extended SUDA technique by analyzing a subset of the behavior specified in a Location-aware Role-Base Access Control (LRBAC) class model.

1. Introduction

In previous work we described the Scenario-based UML Design Analysis (SUDA) technique [Yu08] [Yu07]. SUDA can be used to rigorously analyze UML [UML] design class models that include operations specified using the Object Constraint Language (OCL) [OCL]. A UML design class model is analyzed against a given set of independently developed scenarios that describe desired and undesired behaviors. The intent is to uncover inconsistencies between an independent verifier's understanding of required behavior as described in scenarios, and the behavior specified in a design class model developed by a designer. The technique leverages the USE tool [USE] in the analysis. The analysis technique is lightweight in that it analyzes behavior specified in a UML design class model against a set of scenarios. It is static because it does not require that the UML design class model be executable. The technique was developed specifically for the analysis of UML design models of sequential systems.

There are two roles involved in the use of SUDA: Designer and (Independent) Verifier. The designer creates a UML design class model that includes invariants and operation specifications specified using OCL. The verifier independently creates a set of scenarios that will be used to analyze the UML design. A *scenario* consists of (1) a sequence of operation calls with parameter values, and (2) a scenario operation definition for each operation called in the scenario. A *scenario operation definition* defines the effect an operation has on a state produced in the scenario.

The SUDA technique consists of three major steps (see Figure 1).

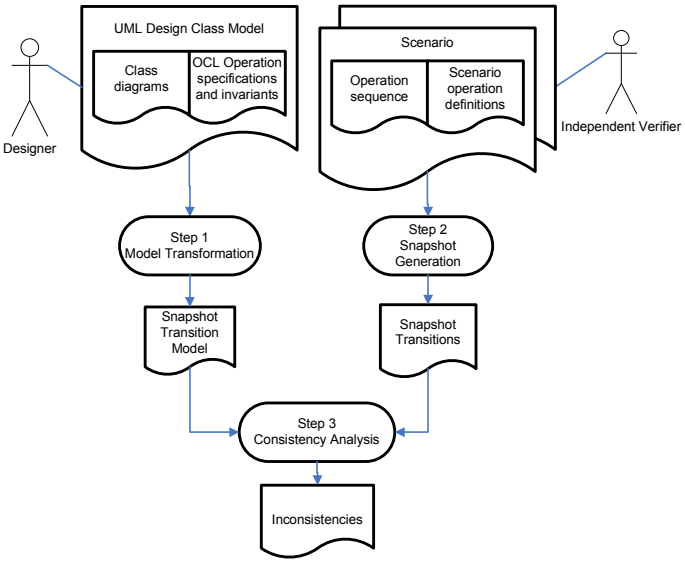

Figure 1. Scenario-based UML design analysis (SUDA) technique

In the first step, the UML design class model is automatically transformed to a *snapshot transition model*. A *snapshot transition model* is a UML class model that specifies valid *snapshot transitions*, that is, changes to object configurations (snapshots) triggered by the execution of operations. A *snapshot transition* describes the effect an operation has on a state and consists of (1) the name and parameter values of the operation that triggered 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, an independently developed scenario that describes either desired or undesired functionality from the perspective of a verifier is transformed to a *snapshot transition sequence*.

In the third step, the USE tool is used to check if the snapshot transition sequence conforms to the snapshot transition model. If the scenario describes invalid behavior and is found to be consistent with the design class model then there is either an error in the design model or the verifier made an error in defining the invalid scenario. If the scenario describes valid behavior and is inconsistent with the design model, the output of SUDA is a set of inconsistencies. The USE tool reports these inconsistencies as failed constraints. The inconsistencies indicate that either the UML design model or the verifier's scenarios are incorrect.

The ability of SUDA to uncover errors depends on the scenarios used to analyze the design model. In our previous work, a verifier manually created the scenarios. In this paper, we describe a technique for generating scenarios that satisfy scenario generation criteria provided by the verifier. Scenario generation criteria are expressed as operation sequence patterns that describe the operation call sequence and object configurations that should be covered by scenarios used to analyze the UML design.

The scenario generation technique transforms scenario operation definitions and the UML class model to an Alloy model, and uses Alloy to produce snapshot transition sequences (scenarios) that satisfy the scenario generation criteria. The generated scenarios are used to analyze the design model as described earlier.

The rest of the paper is organized as follows. In Section 2 we describe a key component of SUDA, the snapshot transition model, and we also describe the Alloy specification language and the Alloy analyzer. In Section 3 we give an overview of the proposed scenario generation technique and demonstrate the technique by applying it to a Location-aware Role-Based Access Control system. In Section 4 we discuss related work, and in Section 5 we summarize our contributions and future directions.

2. Background

In this section we first describe the model used to illustrate the scenario generation technique. We also describe the form of snapshot transition models and give an overview of Alloy.

2.1 The Location-aware Role-Based Access Control Model

Role-Based Access Control (RBAC) is a policy-neutral access control model used to protect sensitive information resources [Ferraiolo01]. In core RBAC permissions are granted to roles and roles are assigned to users. The assigned roles that a user activates in a session determine the resources that the user can access in the session. In hierarchical RBAC, roles are organized into hierarchies of junior and senior roles. A senior role inherits all the permissions of its junior roles, and a junior role inherits all the assigned users of the senior role. The separation of duty (SOD) constraint enforces conflict of interest policies. There are two forms of SOD constraints: A static separation of duty (SSD) constraint prohibits the *assignment* of conflict-of-interest roles to the same user, and a dynamic separation of duty (DSD) constraint prohibits the simultaneous *activation* of conflict-of-interest roles by the same user.

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

In the LRBAC design class model shown in Figure 2, the *User*, *Role, Session* and *Permission* classes represent users, roles, sessions and permissions in standard RBAC. 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 the dominated roles indirectly assigned to the user. The *activeRoles* association end identifies 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.

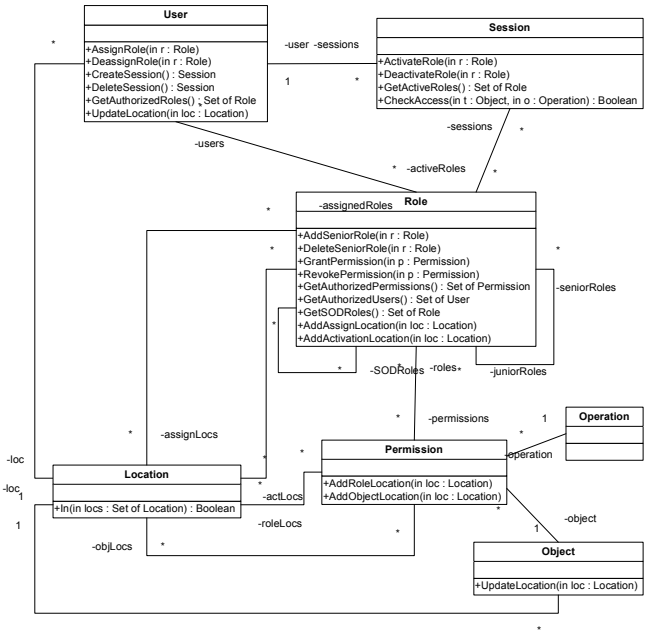

Figure 2. The LRBAC UML design class diagram

The *Location* class describes the location entity introduced 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 represent the locations of the user and object, respectively. The operation *Location::In()* checks whether the location is contained in a set of locations. The *assignLocs* and *actLocs* association ends describe the set of assign and activation locations, respectively, of the role. The *roleLocs* and *objLocs* association ends describe the set of role and object locations of the permission.

The OCL operation specifications we will use to illustrate the scenario generation technique 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().GetAuthorizedPermis
sions()->exists (p | p.object = t and
p.operation = o and
self.user.loc.In(p.roleLocs) and
o.loc.In(p.objLocs)))
```

2.2. Snapshot transition models

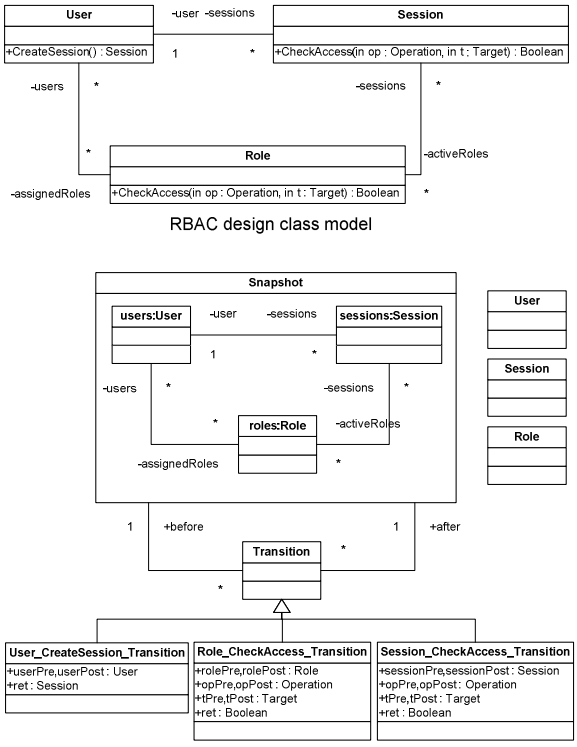

Figure 3. Partial RBAC class model and its snapshot transition model

The *snapshot transition model* is a key artifact in the SUDA technique. In our previous work we described how a snapshot transition model can be mechanically generated from a UML design class model. Figure 3 shows the snapshot transition model generated from a part of the UML design model for RBAC.

The snapshot transition model consists of a structured *Snapshot* class that characterizes snapshots (object configurations) and a *Transition* class that relates before and after snapshots for an operation. The three specializations of the *Transition* class characterize transitions triggered by the three operations in the RBAC design class diagram shown above the snapshot transition model. For example, the class *User_CreateSession_Transition* characterizes transitions triggered by invocations of the *CreateSession()* operation defined in the *Session* class. The attributes in the *Transition* subclasses represent scalar operation parameters and references to before and after states of the objects accessed by the operations. For example, the attribute *userPost* is a reference to the start state of the *User* object created in an invocation of *CreateSession()*. The created session object is stored in *ret*.

Operation specifications in the UML design model are transformed to invariants on the snapshot transition model. This transformation is described in a previously published paper [Yu08].

2.3. Alloy

Alloy is a formal notation used to describe structural properties of software systems [Alloy]. The Alloy notation is based on first-order relational logic. A key component in an Alloy model is a signature. A signature defines a set of objects. It can also contain fields that each describes a relation. In Alloy invariants can be defined as *facts* which are constraints that always hold in a valid system state. A predicate is a named constraint. An assert is a constraint that is intended to hold in a model. A function is a named expression with zero or more arguments. Operations can be specified as predicates or functions.

Properties to be checked by the Alloy Analyzer are expressed as either predicates or assertions. For properties expressed as predicates, Alloy attempts to find an instance that satisfies the property. For properties expressed as assertions, the Alloy Analyzer attempts to find a counterexample that violates the property. The Alloy Analyzer finds the example or counterexample within a scope that is limited by the number of instances of each entity in the system.

3. The scenario generation technique

The scenario generation technique (see Figure 4) 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.

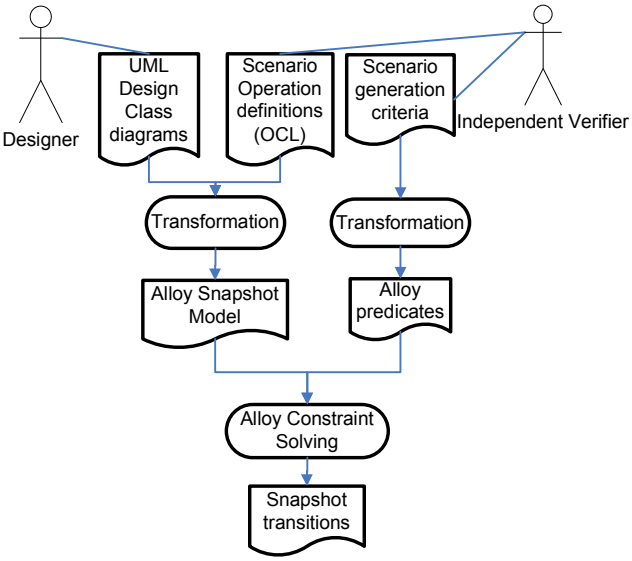

Figure 4. Generating transition sequences

In section 3.1 we describe the types of scenario generation criteria that verifiers can define. In section 3.2 we give examples of scenario operation definitions and in section 3.3 we describe how scenarios are generated.

3.1. Defining scenario generation criteria

In the extended SUDA 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. 5.

Structural coverage criteria. The verifier defines a criterion stating that the snapshot before the *CheckAccess()* operation in Fig. 5 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 one granted permission, and that the set of user locations is not equal to the role set of permission role locations.

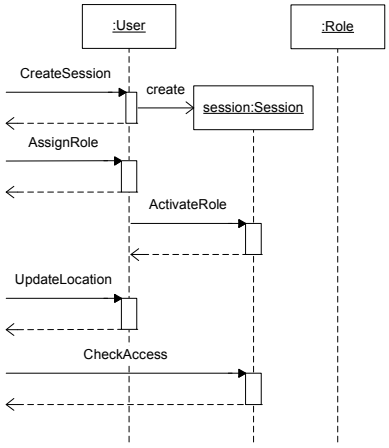

Figure 5. The analysis operation sequence

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 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)
}
```

3.2 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: (1) a user is in a location in which he cannot activate any roles, and (2) he 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.assignL
ocs)
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
```

3.3 Generating scenarios

There are four major steps in the scenario generation process.

Step 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 (see Fig. 2). 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
  …
}
```

Step 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'}
```

Step 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)
```

Step 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. 6 and Fig. 7. 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:

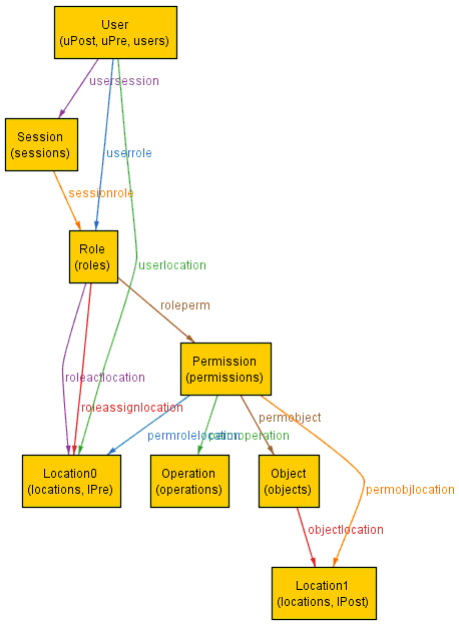

Figure 6. Snapshots before User_UpdateLocation_Transition

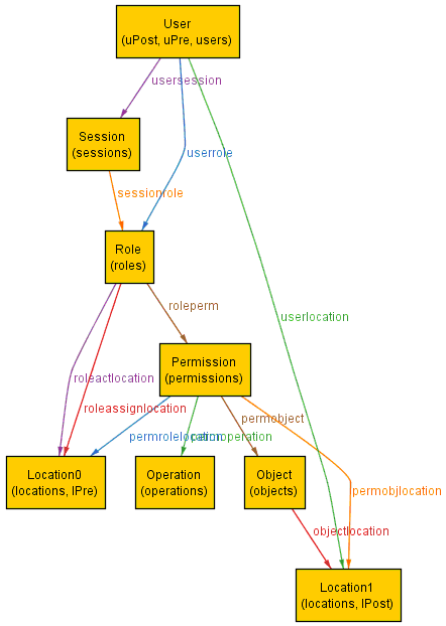

Figure 7. Snapshots after User_UpdateLocation_Transition

```
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)))
```

4. Related work

Formal analysis tools such as the Alloy Analyzer [Alloy] and model checking tools [Clarke99] do exhaustive search in a constrained state space to find solutions that satisfy constraints or verify given properties. Formal theorem proving tool such as Isabelle can be used to reason about modeled properties in the model in an interactive mode [Brucker08]. In order to use these tools to analyze UML class models, the models must be transformed to tool-specific input languages so that the verifier must be familiar with these notations before doing 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. Also the technique is lightweight in that it analyzes UML design class models in the scope of a set of scenarios.

The UML animation and testing approach (UMLAnT) [Trung05] and other analysis approaches that rely on executable models can be used to test designs by executing them on a set of inputs. In SUDA, operations need only be specified using OCL. It is a static analysis technique in which the UML design class diagram is checked against the functionality of scenarios to find design errors.

UML animation techniques can be used to generate scenarios. Oliver and Kent propose a UML animation technique to validate a UML design [Oliver99]. In their work UML design class diagrams are animated by mapping OCL constraints to operations on snapshots. In another piece of work Krieger and Knapp use a SAT solver to find new system state that satisfies operation post-conditions [Krieger08]. Compared with UML animation techniques that generate next system state from OCL specifications, the scenario generation technique in this paper focuses on generating scenarios from the verifier's scenario coverage criteria and OCL scenario operation definitions. The advantage of the approach described in this paper is that it analyzes the UML designs from an independent verifier's point of view. Tools and techniques that are based on first-order logic or linear temporal logic have been used to

generate system traces that reach a goal state [Reiter01] [Margaria07] [Margaria09]. Compared with these techniques, the Alloy scenario generation technique in this paper does not expose users to notation outside of the UML and OCL. We do plan to investigate whether these techniques can be integrated with SUDA as back-end analysis techniques that are not exposed to modelers.

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. OCLE also checks the consistency of OCL invariants on objects or object diagrams. A limitation of OCLE is that it currently does not support analysis of scenarios against operation specifications in class models. The latest version of USE tool validates pre and post-conditions in interactive command mode. However, neither of these tools analyzes behavior against scenarios. The SUDA technique extends the capabilities of these tools by providing the means to statically analyze behavior.

5. Conclusions and future plans

This paper presents a systematic scenario generation technique for scenario-based analysis of UML design class models. To automatically generate scenarios, the verifier defines scenario coverage criteria and specifies effects of operations using OCL. The technique generates scenarios that satisfy the verifier's scenario coverage criteria.

We have implemented the snapshot transformation tool in Kermeta [Kermeta], a meta-modeling environment. The tool transforms an EMF [EMF] Ecore UML design class models to a snapshot transition model in USE. We are now implementing the transformation from UML snapshot transition model to Alloy and transformation from Alloy analysis results to USE instances based on the existing work of UML2Alloy [Anastasakis10] [Shah09]. After the implementation is done, we will do a formal empirical study of the SUDA technique to evaluate the effectiveness and the capability of the technique.

We will further enhance the scenario coverage criteria and implement a tool that translates the verifier's definitions to Alloy predicates and transforms Alloy instances to UML object diagrams. We will also evaluate the effectiveness of the scenario generation technique with respect to the ability of the generated scenarios to uncover errors.

6. References

[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.

[Anastasakis10] 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).

[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.

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

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

[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.

[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.

[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 2.0.

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

[Margaria07] Tiziana Margaria, Bernhard Steffen, "LTL Guided Planning: Revisiting Automatic Tool Composition in ETI," Software Engineering Workshop, Annual IEEE/NASA Goddard, pp. 214-226, *31st IEEE Software Engineering Workshop (SEW 2007)*, 2007.

[Margaria09] Tiziana Margaria, Marco Bakera, Christian Kubczak, Stefan Naujokat and Bernhard Steffen, "Automatic Generation of the SWS- Challenge Mediator with jABC/ABC", *SEMANTIC WEB SERVICES CHALLENGE, Semantic Web and Beyond*, 2009, Volume 8, I, 119-138, DOI: 10.1007/978-0-387-72496-6_7

[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.

[Reiter01] Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press (2001)

[Shah09] Seyyed M. A. Shah, Kyriakos Anastasakis, Behzad Bordbar: From UML to Alloy and Back Again. *MoDELS Workshops 2009*: 158-171.

[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.

[UML] Object Management Group, Unified Modeling Language: Superstructure, vers 2.0 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.

[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 International Conference on Engineering Complex Computer Systems*, Potsdam, Germany, June 2009.