A light-weight static approach to analyzing UML behavioral properties

Lijun Yu, Robert B. France, Indrakshi Ray
Colorado State University, USA
{lijun, france, iray}@cs.colostate.edu
Kevin Lano
Kings College, UK.
kcl@dcs.kcl.ac.uk

Abstract

Identifying and resolving design problems in the early design phase can help ensure software quality and save costs. There are currently few tools for analyzing designs expressed using the Unified Modeling Language (UML). Tools such as OCLE and USE support analysis of static structural properties. These tools provide mechanisms for checking instance models against invariant properties expressed using the Object Constraint Language (OCL). In this paper we propose an approach to analyzing behavioral properties of UML models that can utilize static analysis tools. The approach includes a technique for generating a class model of behavior from operation specifications expressed in a restricted form of OCL Behavioral properties are expressed as invariants defined in the class model of behavior. Static analysis tools such as USE and OCLE can be used to check object models describing series of snapshots. Most of the analysis can be automated. We illustrate our approach by analyzing static separation of duty and dynamic separation of duty properties of a hierarchical role-based access control model (HRBAC).

1. Introduction

Tools supporting rigorous analysis of design models can enhance the ability of developers to identify potentially costly design errors earlier. Defects that are introduced in the design phase can be more expensive to find and fix after the implementation is built. Correcting design errors early also minimizes the time-wasting process of implementing faulty designs. Design models must be expressed in an analyzable form to support analysis.

The Unified Modeling Language (UML) [UML2] can be used to formally describe structural properties of systems. The structural properties are expressed in class models that may contain invariants expressed in the Object Constraint Language (OCL). Static analysis tools such as OCLE and USE can be used to analyze class models [Chiorean] [Gogolla]. For example, the OCLE [Chiorean] tool can be used to check that object diagrams conform to a class model, that is, that invariants specified in the class model hold in the object diagrams. These static analysis tools are currently not capable of analyzing behavioral properties captured by operation specifications in class models.

In this paper we describe a technique for describing behavioral properties in class modeling terms to enable the use of tools such as OCLE and USE in the analysis of these properties. The approach takes a class model describing the structural aspects of a system and transforms it into a *snapshot class model* that specifies behavior in terms of sequences of snapshots. A snapshot represents the state of a system at some point in time and can be described by an object diagram. Constraints on sequences of snapshots are expressed using the OCL. Modeling behavior in this way allows one to use tools such as USE and OCLE to analyze behavior.

In this paper, we describe an approach to generating snapshot class models from application class models that contain operation specifications expressed in a restricted form of OCL. Behavioral properties are expressed as invariants in the snapshot class models. A sequence of snapshots described as an object diagram can be checked against the snapshot model to determine if the behavior represented by the sequence is compliant with the snapshot class model. We illustrate the approach using a snapshot model generated from a Role-Based Access Control (RBAC) model.

The rest of the paper is organized as follows. In Section 2 we give an overview of our approach. In Section 3 we introduce the RBAC model and the behavioral properties to be analyzed. In Section 4 we provide the steps for generating a snapshot class model. We illustrate each step using the RBAC example. In Section 5 we discuss related work. In

Section 6 we summarize our contributions and give pointers to future directions.

2. Overview

In the approach a design model consists of a design class model and sequence models. Each class has fully defined attributes (that is, each attribute has a defined type, cardinality, and visibility) and each operation is associated with a specification consisting of pre- and post-conditions. Invariants and operation specifications are expressed in the OCL.

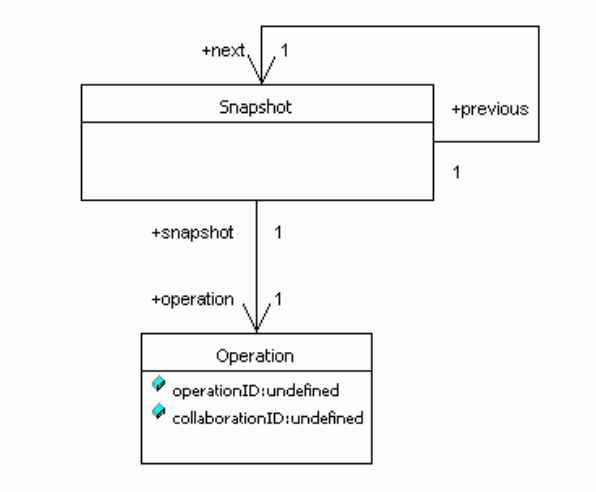

Figure 1. Snapshot class model

The approach is used to systematically produce a snapshot model from a UML application design model. In the snapshot model shown in Figure 1 a Snapshot class is used to record the system state. Its instances are snapshots. Each snapshot is associated with an operation instance that represents a call to the next operation to be performed on the snapshot. We assume operations are executed sequentially. The Snapshot class is thus associated with an Operation class representing operation calls. An instance of Operation represents an invocation of an operation. The attribute *operationID* is a reference to the operation description in the application class model. Note that a single operation can be called many times and thus there may be more than one operation instance with the same *operationID*. The before and after snapshots associated with an operation instance can be obtained using the *next-previous* relationship between snapshots.

Sequence diagrams are used to describe the sequence of operation calls that are to be analyzed. A sequence of operation calls produces a sequence of snapshots, where each pair in the sequence represents before and after states for an operation execution that is a consequence of an operation call. A Snapshot invariant is generated for each call in a sequence. We also represent behavioral properties as invariants in the Snapshot model.

Object diagrams are used to describe sequences of snapshots representing the effects of a sequence of operation calls. We can use tools such as OCLE and USE to check that a snapshot object diagram conforms to a Snapshot class model. An object diagram conforms to a Snapshot model if the invariants specified in the model hold in the object diagram.

3. The Role-Based Access Control example

We illustrate our approach by analyzing the static separation of duty (SSD) and dynamic separation of duty (DSD) properties of the hierarchical role-based access control model (HRBAC) [Ferraiolo]. The analysis of the properties is done using the OCLE modeling environment.

The HRBAC model used in this paper is shown in Figure 2. Users are assigned to roles that determine the permissions they have. Users activate sessions and they user activate a subset of their assigned roles in a session. We do not model permissions because the properties we analyze do not involve them.

The children-parent role hierarchy (RH) association models the dominance relationship between roles. The dominance relationship is reflexive, anti-symmetric and transitive. In RH relationships, dominating roles inherit assigned users of dominated roles and dominating roles inherit permissions of dominated roles. To simplify the modeling and analysis, dominated roles and dominating roles are separately assigned and activated, i.e. when activating or de-activating a role, all its dominated roles and dominating roles are not activated or de-activated automatically.

The static separation of duty (SSD) constraint is described as an invariant of the User class. It restricts the assignment of conflicting roles to one user. The dynamic separation of duty (DSD) constraint is modeled as an invariant of the Session class. It restricts the activation of conflicting roles to one session created by a user.

To illustrate the approach we will analyze SSD and DSD properties for a sequence of operation calls defined in an application which uses the hierarchical RBAC model. The application has several users. Two such users are Alice and Bob. The roles are Cashier, Accountant, Senior Accountant, Senior Cashier and Teller. Senior Accountant role dominates the role of Accountant.

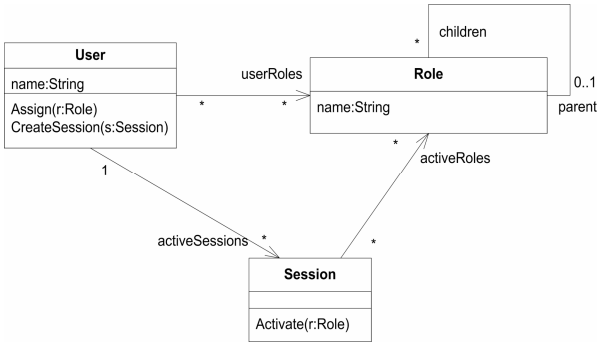

Figure 2. RBAC class model

The SSD property in this application is that the role Accountant and Cashier cannot be assigned to the same user. Any role dominating the Accountant role and any role dominating the Cashier role cannot be assigned to the same user.

The DSD property allows the assignment of Accountant and Cashier to the same user. However, they can not be activated by the same user in one session. Thus any senior Accountant role and any senior Cashier role cannot be activated by the same user in one session.

The invariants in the HRBAC model are given below.

[OCL invariants in the RBAC model]
context User
//Static separation of duty constraint
inv SSD: not (self.assignedRoles->exists(r | r.name = "Accountant") and self.assignedRoles->exists(r| r.name = "Cashier"))
//Static separation of duty constraint with role hierarchies
inv SSD_RH: not (self.assignedRoles->exists(r | r.isDominating("Accountant")) and self.assignedRoles ->exists(r | r.isDominating("Cashier")))

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

//Query method to determine the role hierarchy relationship
context Role::isDominating(roleName:String):Boolean
pre:true
post:
if self.name = roleName then
  result = true
else
  if self.dominating->size() = 0 then
    result = false
  else result =
    (self.dominating.isDominating(roleName))
  endif
endif

//pre- and post- conditions of Activate method
context Session::Activate(role:Role)
pre: self.activatedRoles->forAll(r | r.name <> role.name) and self.user.assignedRoles->exists(r | r.name = role.name)
post: self.activatedRoles->exists(r | r.name = role.name) and self.activatedRoles@pre->forAll(r1 | self.activatedRoles->exists(r2 | r1.name = r2.name)) and (self.activatedRoles->size() = self.activatedRoles@pre->size() + 1)

context Session
//Dynamic separation of duty constraint
inv DSD: not (self.activatedRoles ->exists(r | r.name = "Accountant") and self.activatedRoles ->exists(r| r.name = "Cashier"))
//Dynamic separation of duty constraint with role hierarchies
inv SSD_RH: not (self.activatedRoles ->exists(r | r. isDominating("Accountant")) and self.activatedRoles ->exists(r | r.isDominating("Cashier")))

The behavior that will be analyzed is described by a sequence diagram that describes the following interactions in the order given (space does not allow us to show the sequence diagram but its form can be inferred from what follows): (1) user Bob creates a session using *CreateSession()*, (2) Bob is assigned Accountant and SeniorCashier roles through two calls to the *Assign()* operation, (3) Bob activates the assigned roles in the session by calling *Activate()*. We will illustrate the approach by showing how it is used to evaluate the behavior described by this sequence of operation calls.

4. Generating snapshot models

In this chapter we describe how snapshot models are generated from design class models. We use the (hierarchical) RBAC example described in the previous section to illustrate each step of the

generation process. The process consists of four steps which are described in the remainder of this section.

Step 1 Generate basic snapshot class model

The basic snapshot class model does not contain OCL constraints. The snapshot model is formed by (1) introducing Snapshot and Operation classes to the application class model, (2) specifying the links between the snapshots and the system state. Here, system state is an object configuration that conforms to the application class model. In the case of the RBAC model a system state is a collection of users, roles and sessions that are linked as specified in the RBAC class model. Each collection of objects in the system state (e.g., users) is modeled as a linked list in the snapshot model.

In the following we describe the automatable process for creating basic snapshot class models. We refer to the application class model as the *system design model*.

[Snapshot class model generation algorithm]
***Input**: System design class diagram*
***Output**: Snapshot class diagram*
Process:
1. For each class in the system design model: Organize the instances in a linked list by introducing a self association that relates an instance to the next instance in the list. Associate Snapshot with each class such that each snapshot is linked to exactly one object of the class (the first element in the linked list).
2. For each operation in each class:
a. Generate ClassName_OperationName as an Operation enumeration. The enumerations are the values that can be assigned to the operationID attribute.
b. For each parameter of the operation: Generate ClassName_OperationName_ParameterName as an attribute of the Operation class.

Figure 3 shows the result of applying the process to the RBAC model. The formal parameters in the *Assign()*, *CreateSession()*, and *Activate()* operations are represented as attributes in the *Operation* class. The *operationID* attribute references the operation names used in the RBAC model as reflected in the use of the *OperationName* enumeration type.

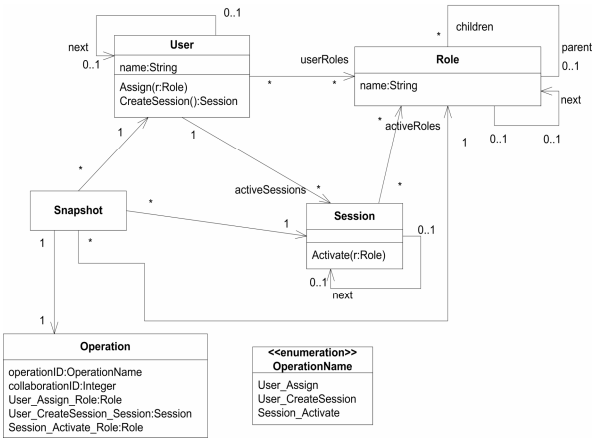

Figure 3. RBAC snapshot model

Step 2 Generate snapshot transition diagrams

To analyze behavioral properties for a sequence of operation calls made in a sequence diagram, we generate a sequence of snapshot transitions in which the order of the transitions is determined by the order in which operations are called in the sequence diagram.

A snapshot transition diagram describes how a system state transits to a next system state as a consequence of an operation invocation. Snapshot transitions are described in an object diagram called a *collaboration*. A collaboration has two snapshot instances which record the system states before and after the operation invocation, and an operation instance which represents the operation context.

Some operation invocations spawn calls to sub-operations. For example, a *Transfer* operation in a *Bank* class may call a sub-operation *Withdraw* to take out money from the source account and then call a sub-operation *Deposit* to add the money to the destination account, before returning a result to the caller. In our approach we treat an externally invoked operation and its associated sub-operation as one atomic operation, that is, the calls to the sub-operations are considered to be internal actions that are not visible during analysis. This allows us to more readily generate behavioral constraints.

The Snapshot collaboration generation algorithm generates snapshot transitions from the operation invocation sequence given in the operation invocation sequence diagram.

[Snapshot collaboration generation algorithm]
***Input**: Snapshot class diagram, Operation invocation sequence diagram, initial system state*
***Output**: Snapshot transitions for each atomic operation*

***Process**:*
1. Partition the operation invocation sequence into atomic operations:
Trace each externally initiated operation call through its sub-operations until a result is returned to the initiator; the trace is treated as a single operation call.
2. Generate snapshot collaborations:
For each atomic operation call in the sequence diagram

- *Generate a unique collaborationID*
- *Create an uninitialized snapshot object (instance of Snapshot) and an unintialized operation object (instance of Operation) .*

current operation. Since our approach verifies behavioral properties by checking invariants against snapshot (system state) transitions, we must make sure the next system state is determined precisely. To ensure this, we require that the operation specifications be deterministic and complete. A complete operation specification is one that specifies its effect on all attributes in class. To shorten the presentation of complete specifications, it is assumed that if an effect is not specified for an attribute, then the attribute values in the before and after states are identical.

We also require that operation specifications be operative, that is, one should be able to implement

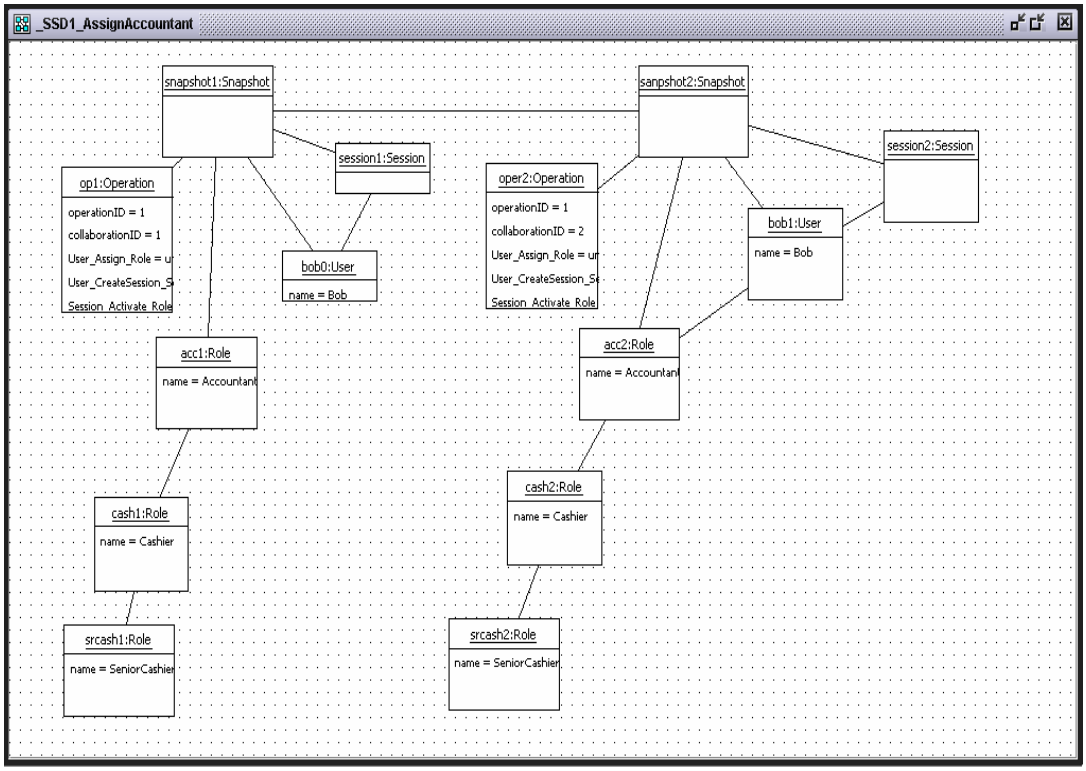

Figure 4. Assign Accountant role to Bob collaboration

- *Instantiate the operation object. The values used to instantiate the attributes representing parameters are the parameter values given in the sequence diagrams.*
- *Create linked list of system state instances. These are the elements in the before system state for the operation.*
- *Generate the after system state; see [next system state generation algorithm].*

The *next system state generation algorithm* is called to generate the next system state. We generate the next system state from the pre- and post- conditions of their effect using a process that completes its task in a finite time. For example, quantifying values over an infinite type space is disallowed. If operation specifications do not have the properties mentioned above then the after state must be produced manually.

[Next system state generation algorithm]
***Assumptions:** The post-condition of the current operation is complete and operative. We restrict the post-condition to assignments involving single-valued and multi-valued attributes.*
***Input**: Current operation, System state before current operation*

***Output**: System state after current operation is executed and results are returned*
***Process**:*
1. Evaluate pre-condition of current operation:
If the before system state does not satisfy the pre-condition, it means that either the system state is not valid with respect to the operation or the pre-condition is not correct. The algorithm halts and the user is notified of the problem.
2. Convert post-condition into a procedural form:
Convert single-value assignment into simple assignment in a procedural language (e.g., the Java programming language).
Convert multi-value assignments or assignment with forall quantifier into iterative statements in algorithmic language.
3. Run the procedure to generate the next system state.

The sequence of RBAC behavior described in Section 3 involves an invocation of the *CreateSession()* operation, two invocations of the *Assign()* operation and two invocations of the *Activate()* operation. For illustration purposes we focus on four snapshot collaborations in this paper: The two *Assign()* invocations and the two *Activate()* invocations. The collaboration for one of the *Assign()* operations is shown in Figure 4. To make Figure 4 more compact we use "1" to represent the value *User_Assign* that is assigned to *operationID*. This value indicates that the operation *Assign()* in *User* provides the context for the transition. The system states in the collaboration are created manually because the post-condition of *Assign()* is not fully operative.

Step 3 Generate snapshot constraints

In this step operation specifications are transformed to snapshot invariants. The step is performed for each collaboration produced in the previous step. Note that the transformation of the same operation on different snapshot collaborations generates different snapshot invariants because the operation may be performed on different objects.

[Snapshot invariant transformation algorithm]
***Input**: Snapshot transitions, operation specifications*
***Output**: Snapshot invariant for each snapshot transition*
***Process**:*
For each snapshot transition (Snapshot collaboration) with operation call op, create a Snapshot invariant from the pre- and post- conditions of op in the class model in the following way:

1. *Replace each object reference in the pre- and post- conditions with a reference to the objects in the associated linked lists. This involves finding the object in the linked list.*
2. *For each object reference in the pre- and post-conditions in the form of X@pre, replace it with X.*
3. *For each object reference in the post-conditions in the form of X, replace it with self.next.X.*
4. *Form the conjunction of the pre and post conditions generated from the above steps and include in the body of the Snapshot invariant.*
5. *Add equalities for operationID and collaborationID in the pre-condition of the Snapshot invariant. The final invariant will have the following form:*

   Invariant: operationID = op_enumeration and collaborationID = current_colaborationID implies Snapshot invariant body

The following are the snapshot invariants generated for the four RBAC collaborations mentioned in the previous step:

[Constraints of the RBAC Snapshots]

```
context Snapshot
inv Collaboration1-Assign1:
  (self.operation.collaborationID = 1 and
self.operation.operationID = 1) implies
  (self.userInstance.assignedRoles->forAll(r | r.name
<> self.roleInstance.name) and
  self.next.userInstance.assignedRoles->exists(r |
r.name = self.roleInstance.name)) and
  self.userInstance.assignedRoles->forAll(r1 |
     self.next.userInstance.assignedRoles->exists(r2 |
r1.name = r2.name)) and
 (self.next.userInstance.assignedRoles->size() =
self.userInstance.assignedRoles->size() + 1)

inv Collaboration2-Assign2:
  (self.operation.collaborationID = 2 and
self.operation.operationID = 1) implies
  (self.userInstance.assignedRoles->forAll(r | r.name
<> self.roleInstance.next.next.name) and
  self.next.userInstance.assignedRoles->exists(r |
r.name = self.roleInstance.next.next.name)) and
  self.userInstance.assignedRoles->forAll(r1 |
     self.next.userInstance.assignedRoles->exists(r2 |
r1.name = r2.name)) and
 (self.next.userInstance.assignedRoles->size() =
self.userInstance.assignedRoles->size() + 1)

inv Collaboration3-Activate1:
  (self.operation.collaborationID = 3 and
self.operation.operationID = 2) implies
```

```
   (self.sessionInstance.activatedRoles->forAll(r |
r.name <> self.roleInstance.name) and
   self.sessionInstance.user.assignedRoles->exists(r |
r.name = self.roleInstance.name) and
   self.next.sessionInstance.activatedRoles->exists(r |
r.name = self.roleInstance.name)) and
   self.sessionInstance.activatedRoles->forAll(r1 |
      self.next.sessionInstance.activatedRoles-
>exists(r2 | r1.name = r2.name)) and
  (self.next.sessionInstance.activatedRoles->size() =
self.sessionInstance.activatedRoles->size() + 1)

inv Collaboration4-Activate2:
   (self.operation.collaborationID = 4 and
self.operation.operationID = 2) implies
   (self.sessionInstance.activatedRoles->forAll(r |
r.name <> self.roleInstance.next.next.name) and
   self.sessionInstance.user.assignedRoles->exists(r |
r.name = self.roleInstance.next.next.name) and
    self.next.sessionInstance.activatedRoles->exists(r |
r.name = self.roleInstance.next.next.name)) and
   self.sessionInstance.activatedRoles->forAll(r1 |
      self.next.sessionInstance.activatedRoles-
>exists(r2 | r1.name = r2.name)) and
  (self.next.sessionInstance.activatedRoles->size() =
self.sessionInstance.activatedRoles->size() + 1)
```

Step 4 Analyze behavioral properties

We analyzed the SSD constraint based on the first two *Assign* Snapshot transitions. The OCLE tool was used to perform the analysis and it reported the error arising from the assignment of two conflicting roles to the same user. Note that *SeniorCashier* dominates *Cashier* and *Cashier* conflicts with *Accountant*.

In our second analysis we removed the SSD constraints and analyzed the snapshot sequence involving the two Activation Snapshot transitions against the DSD constraints. As we expected OCLE reported the error arising from the activation of two conflicting roles in the same session created by *Bob*.

5. Related work

Existing UML modeling tools like OCLE [Chiorean] and USE [Gogolla] support for validating syntactic and structural properties. OCLE [Chiorean] 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. The limitation of these tools is they do not evaluate pre- and post-conditions of operations.

The Alloy Analyzer [Jackson] developed by the Software Design Group of MIT generates examples or counter-examples of certain properties by exploring a search space given by limiting the number of entities in the model. Alloy models systems in a structural modeling language based on first-order logic. Alloy has been used to check abstract system designs and specification consistencies. The checking of UML models with Alloy requires the transformation of UML models to Alloy models. Such transformations must be validated if they are to be trusted and one must e able to trace errors in Alloys to errors in UML models. These are currently challenging problems.

Model checking has been applied to automate the verification of the safety and correctness of finite state-based systems [Clark]. Zhang et. al. [Zhang] developed a model checking approach to evaluating access control policies. As with Alloy, a translation process is needed to convert the UML specifications into a model that can be verified by the model checker.

Static model analysis is useful in verifying design models. Ray et. al. [Ray] model RBAC and MAC access control policies with parameterized UML and compose the models. The static analysis on the composed models can find undesirable violations to the access control policies. However, the proposed approach is manual.

Testing techniques are applied to UML models to check design faults. Dinh-Trong [Dinh-Trong] proposes a systematic approach to testing UML designs. In the approach test cases are generated from a set of test adequacy criteria on the UML design model. Java is used to formally define execution semantics of UML. The design model is then transformed to executable models that exercise the test cases and evaluate the test results. Model execution is another way to check dynamic model behaviors. Harel et. al. [Harel] propose a model execution framework Rhapsody that can translate class diagram and state charts into executable code and execute it.

6. Conclusions and future work

In this paper we propose an approach to modeling behavioral properties using class models of behavior that can be analyzed using existing static analysis tools such as OCLE and USE. The approach takes a design model consisting of design class models with operation specifications and sequence models and generates a snapshot model. It analyzes the behavior specified by sequence diagrams against the snapshot model. It does this by checking object diagrams describing sequences of state changes against invariants in the snapshot model. We demonstrate the approach by analyzing RBAC SSD and DSD properties for a sequence of operation calls in an application model.

The approach limits the form of operation specifications that can be present in a design class model. Models that do not conform to these restrictions require significant human effort to produce the snapshot model. Another limitation is that it can only be used to verify systems where the operations occur sequentially. Our future work aims to remove as much of these limitations while preserving the level of automation to the extent possible.

Acknowledgement

This work was partially supported by a grant from the AFOSR under Contract No. FA9550-04-1-0102.

7. References

[Chiorean] D. Chiorean, M. Pasca, A. Cârcu, 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.

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

[Dinh-Trong] T. T. Dinh-Trong, "A Systematic Approach to Testing UML Design Models", *Doctoral Symposium, 7th International Conference on the Unified Modeling Language (UML)*, Lisbon, Portugal, 2004.

[Ferraiolo] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. "Proposed NIST Standard for Role-Based Access Control". *ACM Transactions on Information and Systems Security*, 4(3), Aug. 2001.

[Gogolla] M. Gogolla, J. Bohling, and M. Richters. "Validating UML and OCL Models in USE by Automatic Snapshot Generation". *Journal on Software and System Modeling*, 4(4):386-398, 2005.

[Harel] D. Harel and E. Gery, "Executable Object Modelling with Statecharts", *IEEE Computer*, 30(7): 31-42, 1997.

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

[Ray] I. Ray, N. Li, D-K. Kim, R. France, "Using Parameterized UML to Specify and Compose Access Control Models", *Proceedings of the 6th IFIP WG 11.5 Working Conference on Integrity and Internal Control in Information Systems*, Lausanne, Switzerland, Nov. 2003.

[UML2] Object Management Group, *Unified Modeling Language: Superstructure*, vers 2.0 Final Adopted Standard.

[Zhang] N. Zhang, M. Ryan, and D. Guelev. "Evaluating Access Control Policies through Model Checking", *Proceedings of the 8th International Conference on Information Security*, Singapore, Sept. 2005.