# Supporting Iterative Development of Robust Operation Contracts in UML Requirements Models

Wuliang Sun, Robert B. France, and Indrakshi Ray
*Department of Computer Science*
*Colorado State University*
*Fort Collins, USA*
{*sunwl, france, iray*}*@cs.colostate.edu*

***Abstract*—Developing adequate system operation contracts at the requirements level can be challenging. A specifier needs to ensure that a contract allows an operation to be invoked in different usage contexts without putting the system in an invalid state. Specifiers need usable rigorous analysis techniques that can help them develop more robust contracts, that is, contracts that are neither too restrictive nor too permissive. In this paper we describe an iterative approach to developing robust operation contracts. The approach supports rigorous robustness analysis of operation contracts against a set of scenarios that provide usage contexts for the operation. We illustrate the approach by developing a robust operation contract for a functional feature in a Location-aware Role-Based Access Control (LRBAC) model.**

***Keywords*-Requirement Modeling; Operation Contract;**

## I. INTRODUCTION

A UML requirements model of a system typically consists of a class model that specifies required system state properties and informal use case descriptions of *system operations* that provide services to users. In more rigorous UML-based requirements modeling approaches, system operations are associated with *contracts* expressed in the Object Constraint Language (OCL) [19]. A contract specifies the conditions under which the associated operation will produce a desired result (the precondition) and the conditions that exist after the operation has completed its task (the postcondition). If the precondition is not satisfied when the operation is invoked, the operation ends with no effect on the modeled system state (i.e., the state of the system just after the operation has completed its execution is the same as the state just before the operation started to execute). A system operation is invoked by an external entity and thus the system cannot control when the operation will be invoked. A modeler thus has to ensure that the contract associated with a system operation is robust enough to cover the different usage scenarios in which the operation may be called. Here, a usage scenario is a sequence of system operation calls initiated by users of the system. A contract is said to be robust with respect to a set of usage scenarios if it allows an operation to produce desired results when invoked in the different scenarios.

The following requirements modeling scenario illustrates one type of problem that can occur if system operation contracts are not robust: A requirements model for a role based access control (RBAC) [16] system is being developed. In RBAC, a user can activate assigned roles in a session. Only assigned roles can be activated by a user in a session. A user can thus be linked to a role in two ways: via a *role assignment* relationship and via a *role activated* relationship. Furthermore, if a user is linked to a role via the *role activation* relationship, it must also be linked to the same role via the *role assignment relationship*. Consider the following contract for a $DeassignRole()$ operation initially developed by a modeler (written in natural language to ease readability). The purpose of the $DeassignRole()$ operation is to remove the specified role, $r$, from a set of roles assigned to a specified user, $u$:

*// Deassign a role from a user*
***Context*** *DeassignRole(u:User, r:Role)*
***Precondition****: role r is assigned to user u and r is not activated*
***Postcondition****: role r is removed from a set of roles assigned to user u*

When viewed on its own the contract seems adequate, but when used in the context of other system operations a problem can arise. Consider a case in which the following happens: (1) a system administrator (mistakenly) assigns a role, $r$, to user $u$, (2) user $u$ creates a session $s$ and activates role $r$ in session $s$, (3) user $u$ retrieves information that he is allowed to access in the role, $r$, and (4) the administrator invokes the $Deassign$ operation to remove the role from the user to prevent the user from accessing other information that can be accessed via the role. Note that in the last step the intent is that the role be deassigned, but the operation contract does not allow this (it will not deassign the role because the role is activated). To address the problem raised by this scenario the contract precondition can be weakened by removing the clause *r is not activated*, and the postcondition can be strengthened to include a constraint stating that the role is also removed from the set of roles activated by the user.

Figure 1: An Overview of the Approach

Figure 2: LRBAC Requirements Class Model

The above scenario illustrates the need for analysis techniques that provide modelers with feedback on the robustness of operation contracts. In this paper, we describe such an analysis technique. The following are the inputs to the analysis technique (see Figure 1): (1) a UML requirements class model, (2) operation contracts expressed using the OCL, (3) a set of usage scenarios (sequences of operation calls), and (4) specifications of the intended effects of each scenario. In the RBAC usage scenario described above, the specification of intended effect would state that at the end of the scenario the role is removed from the assignment and activation role sets of the user.

The inputs are used to produce an Alloy model [10] of required behavior. If the Alloy Analyzer determines that the intended effect is not realized then there exists an operation call (or calls) in the usage scenario that prevented the production of the intended effect. The modeler can then examine the operation contracts to determine how they can be made more robust.

The rest of the paper is organized as follows. Section II gives an overview of the analysis technique. Section III presents a Location-aware Role-Based Access Control (LRBAC) model that we use to demonstrate the approach in this paper. Section IV illustrates the approach using the LRBAC model. Section V discusses related work, and Section VI concludes the paper with an overview of future directions.

## II. Approach Overview

The approach described in this paper builds upon our previous work on rigorous analysis of UML design class models [18]. A designer can use the previous technique to check whether a system can move from a specified valid state to a specified invalid state by invoking a sequence of operations. The approach proposed in this paper extends the applicability of our previous work to requirements modeling and provides support for checking that the intended effects of scenarios are supported by operation contracts.

Figure 1 presents an overview of the robustness analysis technique. The first step of the technique transforms a class model that characterizes valid states (a state is referred to as a snapshot), to a class model, called a *snapshot transition model*, that specifies valid sequences of states that represent the behaviors supported by the design class model. This transformation is the same as the one used in the design analysis technique [18], except that the system operations at the requirements level are specified separately (i.e., they are not specified in the context of classes as is done in a design class model).

The second step of the technique converts the snapshot transition model to an Alloy model [11] using the transformation algorithm described in [18]. The third step of the approach generates an Alloy predicate, referred to as the *robustness analysis predicate*, from each scenario and its intended effect. Each predicate specifies that the operations must adhere to the invocation order described in the scenario, and that end states are valid with respect to the scenario (i.e., end states satisfying the intended effect of the scenario).

The robustness analysis predicate is added to the Alloy model generated from the snapshot transition model. The resulting Alloy model is fed into the Alloy Analyzer and the predicate is evaluated. The Alloy Analyzer uses the predicate to determine if the intended effect of the scenario is supported by operations that abide by their contracts. If the Alloy Analyzer returns a sequence representing the scenario specified by the predicate, the contracts of the analyzed operations are robust enough with respect to the scenario.

## III. Location-aware Role-Based Access Control Model

The Location-aware Role-Based Access Control (LRBAC) model (see Figure 2), proposed by Ray et. al. [14] [12] [13], is used in this paper to demonstrate the analysis technique. LRBAC is an access control model used to protect sensitive information resources based on Role-Based Access Control (RBAC) model [16]. LRBAC extends RBAC by incorporating the $location$ concept. In an RBAC model, $permissions$ are granted to $roles$, and $roles$ are assigned to $users$. The assigned $roles$ that a $user$ activates in a $session$

determine the resources ($objects$) that the $user$ can access ($operate$) in the $session$.

In the LRBAC requirements class model shown in Fig. 2, the class $Location$ is associated with $User$, $Object$, $Role$, and $Permission$ classes. A $user$ can be in exactly one $location$ at any given time, while a $location$ can be associated with multiple $users$. $UserLoc$ is the association between $User$ and $Location$. Given a $user$, $user.UserLoc$ returns the $location$ of the $user$. Similarly, an $object$ is associated with one $location$ only, while a $location$ can have many $objects$. $Roles$ are associated with $locations$ by two relationships: $AssignLoc$ and $ActivateLoc$. Given a $role$, $role.AssignLoc$ returns the set of $locations$ in which that $role$ can be assigned, while $role.ActivateLoc$ returns the set of $locations$ in which that $role$ can be activated. A $role$ can be assigned to a $user$ only if $user.UserLoc$ is a member of $role.AssignLoc$. Similarly, a $user$ can activate a $role$ only if $user.UserLoc$ is a member of $role.ActivateLoc$.

Ray et. al. [13] used the Z specification language to formally specify the LRBAC model. In this paper, we use OCL to specify LRBAC operations. For example, system operations, $UpdateLoc$, $AssignRole$ and $AddAssignLoc$, are specified using OCL as follows:

*// Move user $u$ into a new location $l$*
***Context*** *UpdateLoc(u:User, l:Location)*
*// Precondition: user $u$ is not in location $l$ and user $u$ is not*
*//assigned any role*
***Pre:*** *u.UserLoc→excludes(l) and u.UserAssign→isEmpty()*
*// Postcondition: user $u$ is in location $l$*
***Post:*** *u.UserLoc→includes(l)*

*// Assign a role $r$ to user $u$*
***Context*** *AssignRole(u:User, r:Role)*
*// Precondition: user $u$ is not assigned role $r$ and user $u$ is in*
*// a location in which role $r$ can be assigned to him*
***Pre:*** *u.UserAssign→excludes(r) and*
*r.AssignLoc→includes(u.UserLoc)*
*// Postcondition: user $u$ is assigned role $r$*
***Post:*** *u.UserAssign→includes(r)*

*// Allow role $r$ to be assigned to any user in location*
*// $l$*
***Context*** *AddAssignLoc(r:Role, l:Location)*
*// Precondition: role $r$ cannot be assigned to any user in location $l$*
***Pre:*** *r.AssignLoc→excludes(l)*
*// Postcondition: role $r$ can be assigned to any user in location $l$*
***Post:*** *r.AssignLoc→includes(l)*

## IV. Applying the Analysis Approach

In this section, we analyze the contract of a system operation, $UpdateLoc$, in the LRBAC requirements model described in Section III.

The system operation will be invoked in three different scenarios to determine if the intended effects of scenarios are supported by the operation contract. In the first scenario, (1) user $u$ is initially in a location, $l1$, and assigned a role, $r1$, (2) a system administrator removes role $r1$ from user $u$ ($DeassignRole$) before he leaves location $l1$, (3) user $u$ moves to location $l2$ ($UpdateLoc$), (4) the system administrator assigns role $r2$ to user $u$ ($AssignRole$). The intented effect of the scenario is that the user $u$ be in location $l2$ and be assigned role $r2$.

In the second scenario, (1) user $u$ is initially in a location, $l1$, (2) a system administrator assigns a role, $r$, to user $u$ ($AssignRole$), (3) user $u$ needs to retrieve information in a new location, $l2$, but rather than removing role $r$ from user $u$ before he leaves location $l1$, the system administrator allows role $r$ to be assigned to user $u$ in location $l2$ ($AddAssignLoc$). The intented effect of the scenario is that the user $u$ be in location $l2$.

In the third scenario, (1) user $u$ is initially in a location, $l1$, (2) a system administrator assigns a role, $r$, to user $u$ ($AssignRole$), (3) user $u$ needs to retrieve information in a new location, $l2$, and thus user $u$ moves to location $l2$ with his assigned role $r$ being removed after he leaves location $l1$ ($UpdateLoc$), (4) a system administrator assigns role $r$ to user $u$ in location $l2$ ($AssignRole$). The intented effect of the scenario is that the user $u$ be in location $l2$, and be assigned role $r$.

Although the LRBAC developers formally described operation $UpdateLoc$ using the Z specification language, we were able to uncover significant errors in the formalization when analyzing the operation contract with respect to different scenarios using the approach.

In the remainder of this section we describe the robustness analysis performed on the system operation in the following steps.

### *A. Transforming a UML Requirements Class Model with System Operation Contracts to a Snapshot Model*

Software behavior can be represented as a sequence of state transitions, where each transition is triggered by an operation invocation. Yu et. al. [22][20][21] proposed a scenario-based static analysis approach, called SUDA, that allows a developer to check whether a particular sequence of state transitions is supported by a design class model in which operations are specified in OCL. In SUDA, a design class model with operation specifications is transformed to a static model of behavior, called a snapshot transition model. A snapshot represents a system object configuration at a particular time. A snapshot transition describes the behavior of an operation in terms of how system state changes after

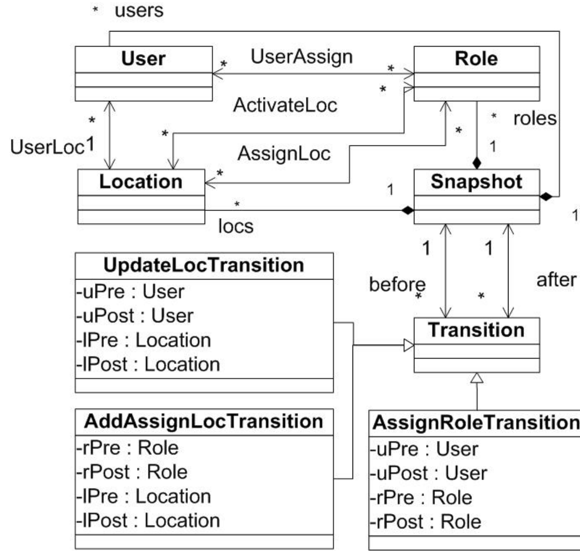



Figure 3: Partial LRBAC Snapshot Transition Model

the invoked operation has completed its task. It consists of a before state, an after state, and the operation invocation that triggers the transition. An operation invocation is described by the operation name and the parameter values used in the invocation.

The LRBAC requirements class model described in Figure 2 together with a set of system opereations (e.g., $UpdateLoc$, $AssignRole$ and $AddAssignLoc$) are transformed to the snapshot transition model partially shown in Figure 3 using the transformation algorithm described in [22][20][21].

Two classes, $Snapshot$ and $Transition$, are added into the snapshot transition model. The instances of class $Snapshot$, are snapshots, and the instances of class $Transition$ are transitions that each relates a *before* snapshot with an *after* snapshot. A snapshot consists of linked instances of classes in a requirements model (i.e., an object configuration). Class $Snapshot$ and $Transition$ are connected by $before$ and $after$ associations.

System operations are transformed to specializations of class $Transition$. For example, operation $UpdateLoc(u : User,\ l : Location)$ is transformed to a specialization of class $Transition$ (e.g., $UpdateLocTransition$). Its parameters (e.g., $u$ and $l$) are transformed into references (shown as attributes) in the $Transition$ specialization. Moreover, if a parameter has a class type, it is transformed into two references. For example, the parameter $l$ in the operation is transformed into $lPre : Location$ and $lPost : Location$, one of which specifies the parameter's state before the execution of the operation ($lPre$) and the other specifies the parameter's state after the execution of the operation ($lPost$).

System operation contracts are transformed into transition invariants that precisely specify the before and after snapshots that are associated with $Transition$ instances. For example, the operation contract for $UpdateLoc$ is transformed to the following transition invariant on the $Transition$ class $UpdateLocTransition$:

***Context*** *UpdateLocTransition*
***inv****:*
*// Generated from precondition*
*before.users→includes(uPre) and*
*before.locs→includes(lPre) and*
*uPre.UserLoc→excludes(lPre) and*
*uPre.UserAssign→isEmpty() and*
*// Generated from postcondition*
*after.users→includes(uPost) and*
*after.locs→includes(lPost) and*
*uPost.UserLoc→includes(lPost) and*
*// Unchanged parts of object configuration*
*after.users→excluding(uPost)=before.users→excluding(uPre)*
*after.locs→excluding(lPost)=before.locs→excluding(lPre)*

*B. Transforming a Snapshot Transition Model to an Alloy Model*

Alloy [11] is a textual modeling language based on first-order relational logic. An Alloy model consists of $signature$ declarations, $fields$, $facts$ and $predicates$. Each $field$ belongs to a $signature$ and represents a relation between two or more $signatures$. $Facts$ are statements that define constraints on the elements of the model. $Predicates$ are parameterized constraints that can be invoked from within $facts$ or other $predicates$.

A snapshot transition model is transformed to an Alloy model using the following transforming algorithm described in [18]. Each class that is part of the $Snapshot$ class in the class model is transformed to a signature in Alloy. For example, class $Role$ in Fig. 3 is transformed to a signature $sig\ Role\{\}$. If a class has attributes, its attributes are transformed to fields of the signature corresponding to the class.

The $Snapshot$ class is transformed to a $Snapshot$ signature containing fields that specify the object configuration within a snapshot. Two groups of fields in the $Snapshot$ signature are used to specify object configurations: fields defining a set of objects (e.g. $roles$:$set\ Role$), and fields defining links between objects (e.g. $UserAssign$: $User\ set{\rightarrow}set\ Role$). Linked objects in a snapshot must be in the domain defined by the $Snapshot$ signature. This constraint is expressed as a fact associated with the $Snapshot$ signature. For example, the fact $UserAssign = UserAssign :> roles$ & $users <: UserAssign$ specifies that linked objects either belong to $roles$ or $users$. The Snapshot signature also includes a field, $OperID$, that is used to identify the

operation that causes a transition to the snapshot when the snapshot is part of a sequence of transitions.

Each $Transition$ specialization in the snapshot model is transformed to a predicate in Alloy. If a $Transition$ specialization has attributes, its attributes are transformed to parameters of the predicate. Two more parameters, $before$ and $after$ with the type $Snapshot$, are added to each predicate to represent the system states before and after the transition. OCL invariants associated with each $Transition$ specialization in the snapshot model are transformed into the body of the predicate corresponding to the $Transition$ specialization using UML2Alloy [3][2][4]. Objects and links that are not changed during the transition are explicitly specified in the predicate. For example, the Alloy predicate $UpdateLocPred$ partially shown below is generated from the transition class $UpdateLocTransition$ in the snapshot model:

```
pred UpdateLocPred[disj before, after: Snapshot, uPre,
uPost: User, lPre, lPost: Location] {
after.OperID = ID_UpdateLoc
// Precondition
uPre in before.users
lPre in before.locs
lPre not in uPre.(before.UserLoc)
uPre.(before.UserAssign) = none
// Postcondition
uPost in after.users
lPost in after.locs
lPost in uPost.(after.UserLoc)
// Unchanged objects
after.users - uPost = before.users - uPre
after.locs - lPost = before.locs - lPre
...
// Unchanged links
after.AssignLoc = before.AssignLoc
after.ObjLoc = before.ObjLoc
... }
```

### *C. Generating a Robustness Analysis Predicate from a Scenario and its Intended Effect*

Alloy provides a trace mechanism that associates the transitions triggered by operation invocations with states defined by $signatures$. The trace mechanism uses an $ordering$ type that casts a set of $states$ into a sequence of $states$ (e.g., open util/ordering[$Snapshot$] as $SnapshotSequence$). A $trace$ predicate that defines $states$ that are reachable through the invocation of sequence of operations, can be used to specify that the operations analyzed by the Alloy Analyzer must adhere to the invocation order described in the scenario. The $trace$ predicate can be extended to a robustness analysis predicate by adding an Alloy statement specifying the intended effect of the scenario. An example of a robustness analysis predicate, generated from the first scenario described in Section IV, is shown below:

```
pred Scenario1{
let first=SnapshotSequence/first | // Get the first snapshot
let second=SnapshotSequence/next[first] |
let third=SnapshotSequence/next[second] |
let fourth=SnapshotSequence/next[third] |
some disj r1, r2: Role | some l: Location | some u: User |
// Specifying the invocation order described in the first
// scenario
DeassignRole[first, second, u, u, r1, r1] and
UpdateLoc[second, third, u, u, l, l] and
AssignRole[third, fourth, u, u, r2, r2] and
// Specifying the intended effects of the scenario
u.(fourth.UserLoc) = l and r2 in u.(fourth.UserAssign)
}
```

This predicate specifies a scenario that starts from the first snapshot and ends in the fourth snapshot as result of invocations of $DeassignRole$, $UpdateLoc$ and $AssignRole$. The intent effect of the scenario is that the user $u$ be in location $l$ and be assigned role $r2$.

### *D. Analyzing an Operation Contract in Different Usage Scenarios*

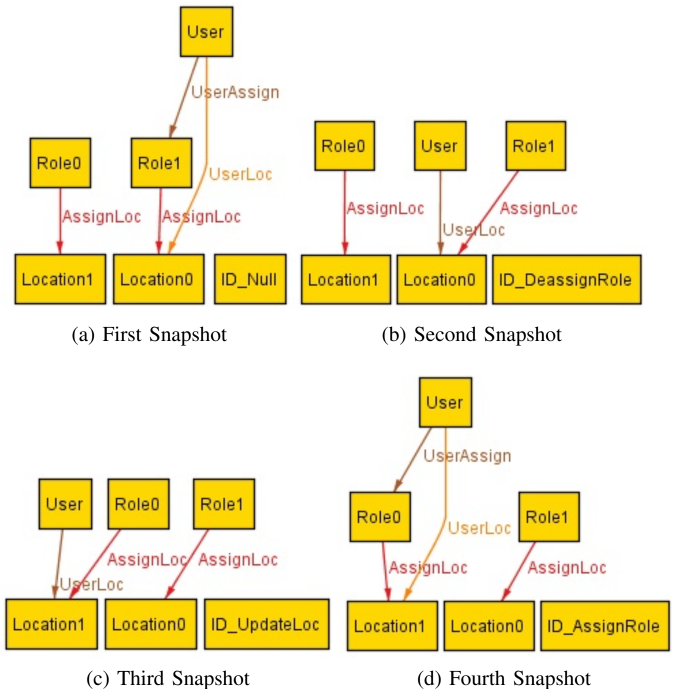



Figure 4: Invoking Operation $UpdateLoc$ in the First Scenario

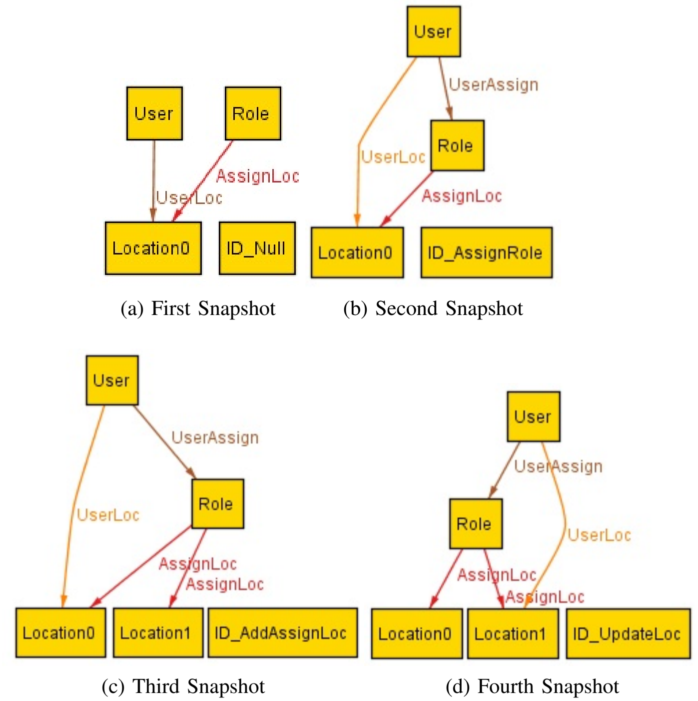
By querying the predicate, the Alloy Analyzer returns an instance that satisfies the predicate. Figure 4 shows four

snapshots produced by the Analyzer for the predicate shown in Section IV-C. The first snapshot in Fig. 4a shows that in the initial state user $User$ is in location $Location0$, and has been assigned role $Role1$. $Role0$ can be assigned to any user in location $Location1$. Note that since this is the start state, $OperID$ has the value $ID_Null$. The second snapshot in Fig. 4b shows that role $Role1$ has been removed from user $User$. $OperID$ is $ID_DeassginRole$, indicating that operation $DeassignRole$ caused the transition from the first snapshot to the second snapshot. The third snapshot in Fig. 4c shows that user $User$ has moved into location $Location1$. $OperID$ is $ID_UpdateLoc$, indicating that operation $UpdateLoc$ caused the transition. The fourth snapshot in Fig. 4d shows that user $User$ has bee assigned roel $Role0$ in location $Location1$. $OperID$ indicates that operation $UpdateLocation$ caused the transition from the third snapshot to the fourth snapshot.

The analysis suggests that the contract of operation $UpdateLoc$ is robust enough with respect to the first scenario since the Alloy Analyzer generates an instance simulating the operation invocations described in the fist scenario. The operation is then analyzed with respect to the second scenario described in Section IV. A new robustness analysis predicate is generated from the second scenario and its intended effect, and fed into the Alloy Analyzer. The Analyzer returns no instance satisfying the new predicate, suggesting that the intended effect of the second scenario cannot be satisfied by the contract of operation $UpdateLoc$.

By analyzing the contract of operation $UpdateLoc$, we have found that its precondition may need to be weakened since it did not allow a user to move to a new location without the assigned roles being removed, even if these roles can be assigned to any user in the new location. To improve the contract, we replace partial precondition of the operation (u.UserAssign→isEmpty()) with a clause that allows a user to move to a new location if every role, that is assigned to the user before, can be assigned to the user in the new location ( $u.UserAssign{\rightarrow}forAll(r : Role|r.AssignLoc{\rightarrow}includes(l))$). When the LRBAC model with this modified operation contract is analyzed by the Alloy Analyzer, an instance (see Figure 5) satisfying the new robustness analysis predicate is generated.

Figure 5 shows four snapshots produced by the Analyzer for the new predicate. The first snapshot in Fig. 5a shows that in the initial state user $User$ is in location $Location0$, and role $Role$ can be assigned to any user in location $Location0$. The second snapshot in Fig. 5b shows that user $User$ has been assigned role $Role$. $OperID$ is $ID_AssignRole$, indicating that operation $AssignRole$ caused the transition. The third snapshot in Fig. 5c shows that role $Role$ has been allowed to be assigned to any user in location $Location1$. $ID_AddAssignLoc$ indicates that operation $AddAssignLoc$ caused the transition. The fourth snapshot in Fig. 5d shows that user $User$ has moved into a new location, $Location1$.




(a) First Snapshot (b) Second Snapshot

(c) Third Snapshot (d) Fourth Snapshot

Figure 5: Invoking Operation $UpdateLoc$ in the Second Scenario

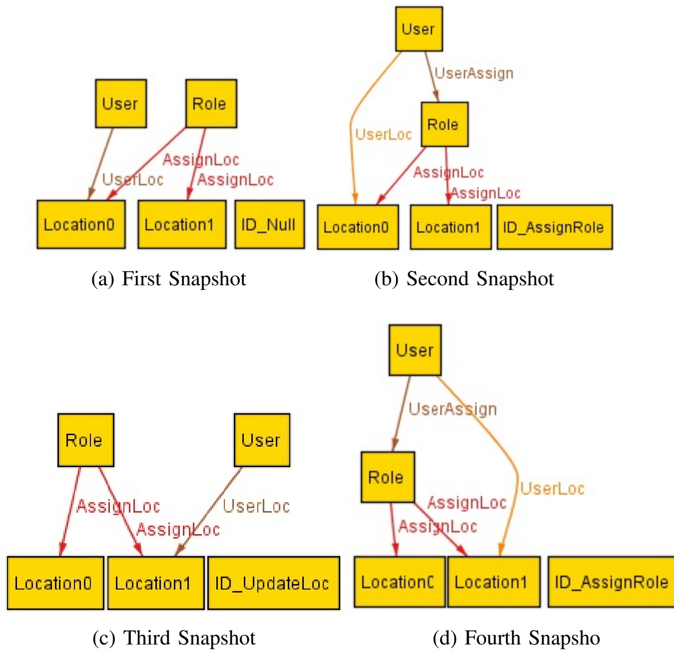
The analysis suggests that the contract of operation $UpdateLoc$ is robust enough with respect to the second scenario. The operation is then analyzed against the third scenario described in Section IV. No instance satisfying the intended effect of the third scenario can be found by the Alloy Analyzer.

When reviewing the contract of operation $UpdateLoc$, we have found that its precondition is still too strict for the third scenario, and its postcondition may need to be strengthened since it did not ensure that the user's assigned roles must be removed before the user moves to a new location. To improve the contract, the clause, $u.UserAssign{\rightarrow}forAll(r : Role|r.AssignLoc{\rightarrow}includes(l))$ is removed from the precondition, and the postcondition is strengthened by adding a clause specifying that the user's assigned roles have been removed after the operation completes its execution.

When the LRBAC model with the modified operation contract given below,

***Context*** *UpdateLoc(u:User, l:Location)*
***Pre:*** *u.UserLoc→excludes(l)*
***Post:*** *u.UserLoc→includes(l)and u.UserAssign→isEmpty()*

is analyzed by the Alloy Analyzer, an instance (see Figure 6) satisfying the intended effect of the third scenario is generated.




(a) First Snapshot (b) Second Snapshot

(c) Third Snapshot (d) Fourth Snapsho

Figure 6: Invoking Operation $UpdateLoc$ in the Third Scenario

Figure 6 shows four snapshots produced by the Analyzer for the third scenario. The first snapshot in Fig. 6a shows that in the initial state user $User$ is in location $Location0$, and role $Role$ can be assigned to any user in both location $Location0$ and $Location1$. The second snapshot in Fig. 6b shows that user $User$ has been assigned role $Role$. The third snapshot in Fig. 6c shows that role $Role$ has moved into location $Location1$ and role $Role$ has been removed from user $User$. The fourth snapshot in Fig. 6d shows that user $User$ has been assigned role $Role$ in location $Location1$.

## V. Related Work

OCLE [8] can be used to analyze structural properties of UML models, but it provides very poor support for analyzing functional properties. USE [9] [17] [15] generates an object configuration conforming to the class model and evaluates the snapshot against OCL constraints. However, it can only check *one* snapshot transition at a time, unlike the approach described in this paper that generates sequences of snapshot transitions triggered by the operation invocations and provides the feedback to determine the robustness of the operation contract.

Cabot et. al. [7][1] proposed an approach to derive operation contracts from UML class diagrams. However, their approach cannot verify the operation contracts against user-defined properties. UMLtoCSP [5] is a tool for the formal verification of UML/OCL models, that transforms a UML model into a Constraint Satisfaction Problem (CSP) and uses the constraint solver to check the correctness properties of the model (e.g., is there an instance of a model without violating any constraint?). [6] extends UMLtoCSP by providing support for verifying an operation's contracts against properties such as executability (e.g., are there two instances of a model, one of which satisfies the precondition of an operation and the other satisfies the postcondition of the operation?). However, UMLtoCSP cannot be used to analyze the robustness of operation contracts against different usage scenarios.

## VI. Conclusion and Future work

In this paper, we described a rigorous approach to iterative development of a system operation contract in the requirements model. The approach uses the Alloy Analyzer to analyze the robustness of an operation contract against different scenarios that provide usage contexts for the operation. The approach builds upon our previous work that involves transforming UML models to Alloy models with traces to support analysis of sequences of operation executions, and extends it by providing support for checking that the intended effects of scenarios are supported by operation contracts.

We applied the approach to a LRBAC specification to demonstrate that the requirements modeler can use the approach to ensure that the contract of a system operation is robust enough to cover different usage scenarios in which the operation may be invoked. The approach is not limited to analyzing access control models, and thus the requirements modeler can use the same technique to analyze other models that are specified using the UML requirements class notations. In future, we plan to validate our approach by applying it on real-world problems, and investigate how the scenarios are selected to provide sufficient test coverages for a system operation.

## Acknowledgment

The work described in this report was supported by the National Science Foundation grant CCF-1018711.

## References

[1] M. Albert, J. Cabot, C. Gómez, and V. Pelechano. Generating operation specifications from uml class diagrams: A model transformation approach. *Data & Knowledge Engineering*, 2011.

[2] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. *Model Driven Engineering Languages and Systems*, pages 436–450, 2007.

[3] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.

[4] B. Bordbar and K. Anastasakis. UML2Alloy: A tool for lightweight modelling of Discrete Event Systems. In *IADIS International Conference in Applied Computing*, volume 1, pages 209–216. Citeseer, 2005.

[5] J. Cabot, R. Clarisó, and D. Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548. ACM, 2007.

[6] J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL operation contracts. In *Integrated Formal Methods*, pages 40–55. Springer, 2009.

[7] J. Cabot and C. Gómez. Deriving operation contracts from uml class diagrams. *Model Driven Engineering Languages and Systems*, pages 196–210, 2007.

[8] D. Chiorean et al. Object constraint language environment (OCLE), version 2.02, 2004.

[9] M. Gogolla, F. Buttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1-3):27–34, 2007.

[10] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):290, 2002.

[11] D. Jackson. Software Abstractions: Logic, Language, and Analysis. *The MIT Press*, 2006.

[12] I. Ray and M. Kumar. Towards a location-based mandatory access control model. *Computers & Security*, 25(1):36–44, 2006.

[13] I. Ray, M. Kumar, and L. Yu. LRBAC: A location-aware role-based access control model. *Information Systems Security*, pages 147–161, 2006.

[14] I. Ray and L. Yu. Short paper: Towards a location-aware role-based access control model. In *Security and Privacy for Emerging Areas in Communications Networks, 2005. SecureComm 2005. First International Conference on*, pages 234–236. IEEE, 2006.

[15] M. Richters. The USE tool: A UML-based specification environment, 2001. *Internet: http://www. db. informatik. uni-bremen. de/projects/USE*, 20:133–147.

[16] RS Sandhu, EJ Coyne, HL Feinstein, and CE Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[17] K. Sohr, G.J. Ahn, M. Gogolla, and L. Migge. Specification and validation of authorisation constraints using UML and OCL. *Computer Security–ESORICS 2005*, pages 64–79, 2005.

[18] W. Sun, R. France, and I. Ray. Rigorous Analysis of UML Access Control Policy Models. In *2011 IEEE International Symposium on Policies for Distributed Systems and Networks*. IEEE, 2011.

[19] J. Warmer and A. Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[20] L. Yu, R. France, and I. Ray. Scenario-Based Static Analysis of UML Class Models. *Model Driven Engineering Languages and Systems*, pages 234–248.

[21] L. Yu, R. France, I. Ray, and S. Ghosh. A Rigorous Approach to Uncovering Security Policy Violations in UML Designs. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 126–135. IEEE, 2009.

[22] L. Yu, RB France, I. Ray, and K. Lano. A light-weight static approach to analyzing UML behavioral properties. In *12th IEEE International Conference on Engineering Complex Computer Systems, 2007*, pages 56–63, 2007.