Using Alloy to Analyze a Spatio-Temporal Access Control Model Supporting Delegation *

Manachai Toahchoodee and Indrakshi Ray
Computer Science Department, Colorado State University, Fort Collins CO
Email: {toahchoo,iray}@cs.colostate.edu

Abstract

Pervasive computing applications use the knowledge of the environment to provide better services and functionality to the end user. Access control for such applications needs to use contextual information. Towards this end, we proposed an access control model based on RBAC that uses the environmental contexts time and location to determine whether a user can get access to some resource. The model also supports delegation which is important for dynamic applications where a user is unavailable and permissions may have to be transferred temporarily to another user/role in order to complete a specific task. Such a model typically has numerous features to support the requirements of various applications. The features may interact in subtle ways to produce conflicts. Here, we propose an automated approach using Alloy for detecting such conflicts. Alloy is supported by a software infrastructure that allows automated analysis of models and has been used to verify industrial applications. The results obtained from the analysis will enable the users of the model to make informed decisions.

1 Introduction

With the increase in the growth of wireless networks and sensor and mobile devices, we are moving towards an era of pervasive computing. The growth of this technology will spawn applications such as, the Aware Home [10] and CMU's Aura [12], that will make life easier for people. However, before such applications can be widely deployed, it is important to ensure that no authorized users can access the resources of the application and cause security and privacy breaches. Traditional access control models, such as, Bell-LaPadula (BLP) and Role-Based Access Control (RBAC), do not work well for pervasive computing applications because the applications are dynamic in nature and do not take into account environmental factors while making access decisions. Consequently, new access control models are needed for such applications.

Access control for pervasive applications need to take into account environmental factors, such as location and time, before making access decisions. For instance, we may want access to a computer be enabled when a user enters

*This work was supported in part by AFOSR under contract number FA9550-07-1-0042.

a room and it to be disabled when he leaves the room. Pervasive computing applications are dynamic in nature and the set of users and resources are not known in advance. It is possible that a user/role for doing a specific task is temporarily unavailable and another user/role must be granted access during this time to complete it. This necessitates that the model be able to support delegation. Moreover, different types of delegation need to be supported because of the unpredictability of the application.

In an earlier work [24], we proposed a formal access control model for pervasive computing applications. Since RBAC is policy-neutral, simplifies access management, and widely used by commercial applications, we based our work on it. We extended RBAC to incorporate environmental contexts, such as time and location. We also described the different types of delegation that are supported by our model. Some of these are constrained by temporal and spatial conditions. We also showed how spatio-temporal information is used for making access decisions. The various features supported by the model were specified using logical constraints. These features often interact in subtle ways resulting in inconsistencies and conflicts. Consequently, it is important to analyze and understand these interactions before such models can be widely deployed. Manual analysis is tedious and error-prone. Analyzers based on theorem proving are hard to use, require expertise, and need manual intervention. Model checkers are automated but are limited by the size of the system they can verify.

In this paper, we advocate the use of Alloy [14], which supports automated analysis, for checking access control models. Alloy is a modeling language capable of expressing complex structural constraints and behavior. Moreover, it has been successfully used in the modeling and analysis of real-world systems [11, 30]. Alloy is supported by an automated constraint solver called Alloy Analyzer that searches instances of the model to check for satisfaction of system properties. The model is automatically translated into a Boolean expression, which is analyzed by SAT solvers embedded within the Alloy Analyzer. A user-specified scope on the model elements bounds the domain, making it possible to create finite Boolean formulas that can be evaluated by the SAT-solver. When a property does not hold, a counter example is produced that demonstrates how it has been violated. This paper illustrates how the spatio-temporal role-based access control model supporting delegation can be specified and analyzed using Alloy. The analysis demonstrates the features of the model that may conflict with each other.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 shows the relationship of each component of Core RBAC with time and location. Sections 4, 5, 6 and 7 propose different types of hierarchies, separation of duty constraints, and delegation that we can have in our model. Section 8 discusses how the model can be analyzed using Alloy. Section 9 concludes the paper with some pointers to future directions.

2 Related Work

Location-based access control has been addressed in other works not pertaining to RBAC [12, 18, 20]. Atluri and Chun [2, 3] proposed the Geospatial Data Authorization Model (GSAM) which is an authorization model for the geospatial information. The requester can get access to geospatial information provided his credentials and time of access matches the credential and temporal expressions defined in the authorization policy. Ardagna et al. [1] present the Location-

Based Access Control (LBAC) model where the access is contingent upon the location information of the user and his credentials. Yu et al. [33] proposed LTAM a location-temporal authorization model which focuses on controlling user access to the different locations. Pu et al. [19] present the context access control model, called CACM, which integrates the context information to the $UCON_{ABC}$ usage control model. Context Sensitive Access Control (CSAC) [13] proposed by Hulsebosch et al. focus on using context information such as time, location, velocity to control the accessibility of services while preserving the privacy of user information. Hengartner et al. [12] discuss how location information pertaining to a user can be securely accessed.

Researchers have also worked on extending RBAC to support ubiquitous computing applications. Sampemane et al. [25] present a new access control model for active spaces which denote the computing environment integrating physical spaces and embedded computing software and hardware entities. Environmental aspects are adopted into the access control model for active spaces, and the space roles are introduced into the implementation of the access control model based on RBAC. Covington et al. [10] introduce environment roles in a generalized RBAC model (GRBAC) to help control access to private information and resources in ubiquitous computing applications. The environments roles differ from the subject roles in RBAC but do have similar properties including role activation, role hierarchy and separation of duty. Environmental roles are also associated with permissions, and the environmental roles are activated when the environmental conditions change. In a subsequent work [9], Covington et al. describe the Context-Aware Security Architecture (CASA) which is an implementation of the GRBAC model. Ya-Jun et al. [32] and Chakraborty et al. [6] proposed trust based access control models, based on extending RBAC, for applications where users are not known in advance.

Other extensions to RBAC include the Temporal Role-Based Access Control Model (TRBAC) proposed by Bertino et al. [4] which adds the time dimension to the RBAC model. Joshi et al.[17] extend this work by proposing the Generalized Temporal Role Based Access Control Model (GTRBAC) where they introduce the concept of time-based role hierarchy and time-based separation of duty. In a subsequent work [15], Joshi and Bertino extend GTRBAC to support delegation operation in presence of the different types of temporal role-hierarchy. Researchers have also extended RBAC to incorporate spatial information [5, 21]. Incorporating both time and location in RBAC is addressed by several works [7, 23, 26]. Chandran's work combines the main features of GTRBAC and GEO-RBAC. Here again, role is enabled by time constraints. The user can activate the role if the role is enabled and the user satisfies the location constraints associated with role activation. Samuel et al. [26] propose GST-RBAC which incorporates topological spatial constraints to the existing GTRBAC model. In [23], we propose Spatio-Temporal RBAC which is an extension of RBAC model supporting the temporal and spatial constraints. Although the goal of this work is similar to those proposed by Chandran et al. [7] and Samuel et al. [26], the model can express some real-world constraints that are not possible in the other ones. In a subsequent work [24], we extended the model to support the delegation operation. Chen and Crampton use a graph based representation for defining a precise semantics for a spatio-temporal RBAC model [8].

Many works appear that analyze RBAC specifications. The formal analysis of the different types of time-based hybrid hierarchy [15, 17] is proposed by Joshi et al. in [16]. Some researchers have used formal specification languages, such as Z and UML, to analyze RBAC specifications [22, 21, 34]; however, these languages do not support automated

analysis. Towards this end, researchers have advocated the use of Alloy for modeling RBAC specifications. In [35], Zao et al. model basic features of RBAC, role hierarchy, and static separation of duties. Schaad et al. model user-role assignment, role-permission assignment, role hierarchy, and static separation of duties features of RBAC extension using Alloy in [28]. The authors do not model role activation hierarchy, dynamic separation of duties or the delegation operation. The authors briefly describe how to analyze conflicts in the context of the model. Samuel et al. [26] also illustrate how GST-RBAC can be specified in Alloy. They describe how the various GST-RBAC functionalities, that is, user-role assignment, role-permission assignment, and user-role activation, can be specified by Alloy. However, this work does not focus on how to identify interactions between features that result in conflicts. Our recent work [31] fills this gap. We propose the methodology to verify the specification of spatio-temporal RBAC model by using Alloy Analyzer tool. The work however, does not support the delegation operation. The current work adapts this approach to verify a more complex model – spatio-temporal RBAC with delegation.

3 Relationship of Core-RBAC Entities with Time and Location

We begin by discussing how time and location are related in our model, and then discuss how the different entities of core RBAC, namely, *Users*, *Roles*, *Sessions*, *Permissions*, *Objects* and *Operations*, are associated with location and time.

Representing Location and Time

Locations that correspond to the physical world are referred to as the physical locations. *Physical location* $PLoc_i$ is a non-empty set of points $\{p_i, p_j, \ldots, p_n\}$ where a point p_k is represented by three co-ordinates. Physical locations are grouped into symbolic representations that are used by applications and referred to as logical locations. Examples of logical locations are Fort Collins, Colorado etc. *Logical location* is an abstract notion for one or more physical locations. We assume the existence of two translation functions, m and m', that convert from logical locations to physical locations and vice-versa. In this work, we focus on the *containment* relation between locations that formalizes the idea whether one location is contained within another. A physical location $ploc_j$ is said to be *contained* in another physical location $ploc_k$, denoted as, $ploc_j \subseteq ploc_k$, if the following condition holds: $\forall p_i \in ploc_j, p_i \in ploc_k$. A logical location $lloc_m$ is contained in $lloc_n$, denoted as, $lloc_m \subseteq lloc_n$, if and only if the physical location corresponding to $lloc_m$ is contained within that of $lloc_m$, that is $m'(lloc_m) \subseteq m'(lloc_n)$. We assume the existence of a logical location called *universe* that contains all other locations. In the rest of the paper, the locations referred to are logical locations.

Our model requires representing time both at the instant level and at the interval level. A *time instant* is one discrete point on the time line. The exact granularity of a time instant will be application dependent. A *time interval* is a set of time instances. We use the notation $t_i \in d$ to mean that t_i is a time instance in the time interval d. A special case of relation between two time intervals that we use is referred to as *containment*. A time interval tv_i is contained in another interval tv_j, denoted as $tv_i \subseteq tv_j$, if the set of time instances in tv_i is a subset of those in tv_j. We introduce a special time interval, which we refer to as *always*, that includes all other time intervals.

Users

We assume that each valid user, interested in doing some location-sensitive operation, carries a locating device which is

able to track his location. The location of a user changes with time. The relation $UserLocation(u,t)$ gives the location of the user at any given time instant t. Since a user can be associated with only one location at any given point of time, we have the following constraint:

$UserLocation(u,t) = l_i \wedge UserLocation(u,t) = l_j \Leftrightarrow (l_i \subseteq l_j) \vee (l_j \subseteq l_i)$

We define a similar function $UserLocations(u,d)$ that gives the location of the user during the time interval d. Note that, a single location can be associated with multiple users at any given point of time.

Objects

Objects can be physical or logical. Example of a physical object is a computer. Files are examples of logical objects. Physical objects have devices that transmit their location information with the timestamp. Logical objects are stored in physical objects. The location and timestamp of a logical object corresponds to the location and time of the physical object containing the logical object. We assume that each object is associated with one location at any given instant of time. Each location can be associated with many objects. The function *ObjLocation(o,t)* takes as input an object o and a time instance t and returns the location associated with the object at time t. Similarly, the function *ObjLocations(o,d)* takes as input an object o and time interval d and returns the location associated with the object.

Roles

We have three types of relations with roles. These are user-role assignment, user-role activation, and permission-role assignment. We begin by focusing on user-role assignment. Often times, the assignment of user to roles is location and time dependent. For instance, a person can be assigned the role of U.S. citizen only in certain designated locations and at certain times only. To get the role of conference attendee, a person must register at the conference location during specific time intervals. Thus, for a user to be assigned a role, he must be in designated locations during specific time intervals. In our model, a user must satisfy spatial and temporal constraints before roles can be assigned. We capture this with the concept of *role allocation*. A role is said to be *allocated* when it satisfies the temporal and spatial constraints needed for role assignment. A role can be assigned once it has been allocated. $RoleAllocLoc(r)$ gives the set of locations where the role can be allocated. $RoleAllocDur(r)$ gives the time interval where the role can be allocated. Some role s can be allocated anywhere, in such cases $RoleAllocLoc(s) = universe$. Similarly, if role p can be assigned at any time, we specify $RoleAllocDur(p) = always$.

Some roles can be activated only if the user is in some specific locations. For instance, the role of audience of a theater can be activated only if the user is in the theater when the show is on. The role of conference attendee can be activated only if the user is in the conference site while the conference is in session. In short, the user must satisfy temporal and location constraints before a role can be activated. We borrow the concept of *role-enabling* [4, 17] to describe this. A role is said to be *enabled* if it satisfies the temporal and location constraints needed to activate it. A role can be activated only if it has been enabled. $RoleEnableLoc(r)$ gives the location where role r can be activated and $RoleEnableDur(r)$ gives the time interval when the role can be activated.

The predicate $UserRoleAssign(u,r,d,l)$ states that the user u is assigned to role r during the time interval d and location l. For this predicate to hold, the location of the user when the role was assigned must be in one of the locations where the role allocation can take place. Moreover, the time of role assignment must be in the interval when role allo-

cation can take place.

$UserRoleAssign(u,r,d,l) \Rightarrow (UserLocation(u,d) = l) \wedge (l \subseteq RoleAllocLoc(r)) \wedge (d \subseteq RoleAllocDur(r))$

The predicate $UserRoleActivate(u,r,d,l)$ is true if the user u activated role r for the interval d at location l. This predicate implies that the location of the user during the role activation must be a subset of the allowable locations for the activated role and all times instances when the role remains activated must belong to the duration when the role can be activated and the role can be activated only if it is assigned.

$UserRoleActivate(u,r,d,l) \Rightarrow (l \subseteq RoleEnableLoc(r)) \wedge (d \subseteq RoleEnableDur(r)) \wedge UserRoleAssign(u,r,d,l)$

The additional constraints imposed upon the model necessitates changing the preconditions of the functions *AssignRole* and *ActivateRole*. The permission role assignment is discussed later.

Sessions

In mobile computing or pervasive computing environments, we have different types of sessions that can be initiated by the user. Some of these sessions can be location-dependent, others not. Thus, sessions are classified into different types. Each instance of a session is associated with some type of a session. The type of session instance s is given by the function $Type(s)$. The type of the session determines the allowable location. The allowable location for a session type st is given by the function $SessionLoc(st)$. When a user u wants to create a session si, the location of the user for the entire duration of the session must be contained within the location associated with the session. The predicate $SessionUser(u,s,d)$ indicates that a user u has initiated a session s for duration d.

$SessionUser(u,s,d) \Rightarrow (UserLocation(u,d) \subseteq SessionLoc(Type(s)))$

Since sessions are associated with locations, not all roles can be activated within some session. The predicate $SessionRole(u,r,s,d,l)$ states that user u initiates a session s and activates a role for duration d and at location l.

$SessionRole(u,r,s,d,l) \Rightarrow UserRoleActivate(u,r,d,l) \wedge l \subseteq SessionLoc(Type(s)))$

Permissions

The goal of our model is to provide more security than their traditional counterparts. This happens because the time and location of a user and an object are taken into account before making the access decisions. Our model also allows us to model real-world requirements where access decision is contingent upon the time and location associated with the user and the object. For example, a teller may access the bank confidential file if and only if he is in the bank and the file location is the bank secure room and the access is granted only during the working hours. Our model should be capable of expressing such requirements.

Permissions are associated with roles, objects, and operations. We associate three additional entities with permission to deal with spatial and temporal constraints: user location, object location, and time. We define three functions to retrieve the values of these entities. $PermRoleLoc(p,r)$ specifies the allowable locations that a user playing the role r must be in for him to get permission p. $PermObjLoc(p,o)$ specifies the allowable locations that the object o must be in so that the user has permission to operate on the object o. $PermDur(p)$ specifies the allowable time when the permission can be invoked.

We define another predicate which we term *PermRoleAcquire(p,r,d,l)*. This predicate is true if role r has permission p for duration d at location l. Note that, for this predicate to be true, the time interval d must be contained in the dura-

tion where the permission can be invoked and the role can be enabled. Similarly, the location l must be contained in the places where the permission can be invoked and role can be enabled.

$PermRoleAcquire(p,r,d,l) \Rightarrow (l \subseteq (PermRoleLoc(p,r) \cap RoleEnableLoc(r))) \wedge (d \subseteq (PermDur(p) \cap RoleEnableDur(p)))$

The predicate $PermUserAcquire(u,o,p,d,l)$ means that user u can acquire the permission p on object o for duration d at location l. This is possible only when the permission p is assigned some role r which can be activated during d and at location l, the user location and object location match those specified in the permission, the duration d matches that specified in the permission.

$PermRoleAcquire(p,r,d,l) \wedge UserRoleActivate(u,r,d,l)$

$\wedge (ObjectLocation(o,d) \subseteq PermObjectLoc(p,o)) \Rightarrow PermUserAcquire(u,o,p,d,l)$

4 Impact of Time and Location on Role-Hierarchy

This organization structure is reflected in RBAC in the form of a role hierarchy [27] which is a transitive, anti-symmetric relation among roles. Senior roles can inherit the permissions of junior roles, or a senior role can activate a junior role, or do both depending on the nature of the hierarchy. Joshi et al. [17] identify two basic types of hierarchy. The first is the permission inheritance hierarchy where a senior role x inherits the permission of a junior role y. The second is the role activation hierarchy where a user assigned to a senior role can activate a junior role. Each of these hierarchies may be constrained by location and temporal constraints. Consequently, we have a number of different hierarchical relationships in our model that are described below.

[Unrestricted Permission Inheritance Hierarchy] A senior role inherits the junior roles permissions but not the spatial and temporal constraints associated with it. For example, *account auditor* role inherits the permissions from the *accountant* role but he can use the permissions at any time and at any place. If x and y be roles such that $x \geq y$, that is, senior role x has an unrestricted permission-inheritance relation over junior role y, then x inherits y's permissions but not the locations and time associated with it.

$(x \geq y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,always,universe)$

[Unrestricted Activation Hierarchy] A user who can activate a senior role can also activate a junior role at any time and at any place. For example, a *project manager* can activate the *code developer* role at any time and at any place. If x and y be roles such that $x \succcurlyeq y$, that is, senior role x has a role-activation relation over junior role y, then a user assigned to role x can activate role y at any time and at any place.

$(x \succcurlyeq y) \wedge UserRoleActivate(u,x,d,l) \Rightarrow UserRoleActivate(u,y,always,universe)$

[Time Restricted Permission Inheritance Hierarchy] A senior role inherits the junior role's permissions but the duration when the permissions are valid are those that are associated with the junior role. For example, a *contact author* can inherit the permissions of the *author* until the paper is submitted. If x and y be roles such that $x \geq_t y$, that is, senior role x has a time restricted permission-inheritance relation over junior role y, then x inherits y's permissions together with the temporal constraints associated with the permission.

$(x \geq_t y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,d,universe)$

[Time Restricted Activation Hierarchy] A user who can activate a senior role can activate a junior role only during the time when the junior role can be activated. For example, a *program chair* can activate a *reviewer* role only during the review period. If x and y be roles such that $x \succcurlyeq_t y$, that is, senior role x has a role-activation relation over junior role y, then a user assigned to role x can activate role y only at the time when role y can be enabled.

$(x \succcurlyeq_t y) \wedge UserRoleActivate(u,x,d,l) \wedge d \subseteq RoleEnableDur(y) \Rightarrow UserRoleActivate(u,y,d,universe)$

[Location Restricted Permission Inheritance Hierarchy] A senior role inherits the junior roles permissions but these permissions are restricted to the locations imposed on the junior roles. For example, a *top secret scientist* inherits the permission of *top secret citizen* only when he is in top secret locations. If x and y be roles such that $x \geq_l y$, that is, senior role x has a location restricted permission-inheritance relation over junior role y, then x inherits y's permissions together with the location constraints associated with the permission.

$(x \geq_l y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,always,l)$

[Location Restricted Activation Hierarchy] A user who can activate a senior role can also activate a junior role but the activation is limited to the place where the junior role can be activated. For example, a *Department Chair* can activate a *Staff* role only when he is in the Department. If x and y be roles such that $x \succcurlyeq_l y$, that is, senior role x has a role-activation relation over junior role y, then a user assigned to role x can activate role y only at the places when role y can be enabled.

$(x \succcurlyeq_l y) \wedge UserRoleActivate(u,x,d,l) \wedge l \subseteq RoleEnableLoc(y) \Rightarrow UserRoleActivate(u,y,always,l)$

[Time Location Restricted Permission Inheritance Hierarchy] A senior role inherits the junior roles permissions together with the spatial and temporal constraints imposed upon those of the junior role. For example, *daytime doctor* role inherits permission of *daytime nurse* role only when he is in the hospital during the daytime. If x and y be roles such that $x \geq_{tl} y$, that is, senior role x has a time-location restricted permission-inheritance relation over junior role y, then x inherits y's permissions together with the temporal and location constraints associated with the permission.

$(x \geq_{tl} y) \wedge PermRoleAcquire(p,y,d,l) \Rightarrow PermRoleAcquire(p,x,d,l)$

[Time Location Restricted Activation Hierarchy]: A user who can activate a senior role can also activate a junior role but must obey the temporal and spatial constraints imposed on the activation of the junior role. For example, user who has a role of *mobile user* can activate the *weekend mobile user* role only if he/she is in the US during the weekend. If x and y be roles such that $x \succcurlyeq_{tl} y$, that is, senior role x has a role-activation relation over junior role y, then a user assigned to role x can activate role y only at the places and during the time when role y can be enabled.

$(x \succcurlyeq_{tl} y) \wedge UserRoleActivate(u,x,d,l) \wedge d \subseteq RoleEnableDur(y) \ \wedge l \subseteq RoleEnableLoc(y) \Rightarrow UserRoleActivate(u,y,d,l)$

5 Impact of Time and Location on Static Separation Of Duties

Separation of duties (SoD) enables the protection of the fraud that might be caused by the user [29]. SoD can be either static or dynamic. Static Separation of Duty (SSoD) comes in two varieties. First one is with respect to user role assignment. The second one is with respect to permission role assignment. In the first case, the SoD is specified as a relation between roles. The idea is that the same user cannot be assigned to the same role. Due to the presence of temporal and spatial constraints, we can have different flavors of separation of duties – some that are constrained by

temporal and spatial constraints and others that are not. In the following we describe the different separation of duty constraints.

[Weak Form of SSoD - User Role Assignment] The same user cannot be assigned to two conflicting roles during the same time and at the same location. For example, a user should not be assigned the *audience* role and *mobile phone user* role at the same time and location. Let x and y be two roles such that $x \neq y$ and $x, y \in SSOD_w(ROLES)$, that is, x and y are two distinct roles that are related by the $SSOD_w$ relation. A user u assigned to role x during time d and location l cannot be assigned to role y at the same time and location if x and y are related by $SSOD_w$.

$(x,y) \in SSOD_w(ROLES) \Rightarrow UserRoleAssign(u,x,d,l) \wedge UserRoleAssign(u,y,d,l) = False$

[Strong Temporal Form of SSoD - User Role Assignment] The same user cannot be assigned to two conflicting roles at the same location at any time. The *consultant for oil company A* will never be assigned the role of *consultant for oil company B* in the same country. Let x and y be two roles such that $x \neq y$ and $(x,y) \in SSOD_t(ROLES)$, that is, x and y are two distinct roles that are related by the $SSOD_t$ relation. A user u assigned to role x during time d and location l cannot be assigned to role y at any time d' in the same location if x and y are related by $SSOD_t$.

$(x,y) \in SSOD_t(ROLES) \Rightarrow UserRoleAssign(u,x,d,l) \wedge UserRoleAssign(u,y,d',l) = False$

[Strong Spatial Form of SSoD - User Role Assignment] The same user cannot be assigned to two conflicting roles at any location during the same time. For example, a person cannot be assigned the roles of *realtor* and *instructor* at the same time. Let x and y be two roles such that $x \neq y$ and $(x,y) \in SSOD_l(ROLES)$, that is, x and y are two distinct roles that are related by the $SSOD_l$ relation. A user u assigned to role x during time d and location l cannot be assigned to role y at the same time at any location l' if x and y are related by $SSOD_l$.

$(x,y) \in SSOD_l(ROLES) \Rightarrow UserRoleAssign(u,x,d,l) \wedge UserRoleAssign(u,y,d,l') = False$

[Strong Form of SSoD - User Role Assignment] The same user cannot be assigned to two conflicting roles. For example, The same person cannot be assigned the role of *minority candidate* and *regular candidate* in a job application. Let x and y be two roles such that $x \neq y$ and $(x,y) \in SSOD_s(ROLES)$, that is, x and y are two distinct roles that are related by the $SSOD_s$ relation. A user u assigned to role x during time d and location l cannot be assigned to role y at any time d' or at any location l' if x and y are related by $SSOD_s$.

$(x,y) \in SSOD_s(ROLES) \Rightarrow UserRoleAssign(u,x,d,l) \wedge UserRoleAssign(u,y,d',l') = False$

We next consider the second form of static separation of duty that deals with permission role assignment. The idea is that the same role should not acquire conflicting permissions. The same manager should not make a request for funding as well as approve it.

[Weak Form of SSoD - Permission Role Assignment] The same role cannot be assigned two conflicting permissions during the same time and at the same location. For example, a passenger role should not be assigned the permission to *go aboard the plane* and *use the cell phone* at the same place and time. Let p and q be two permissions such that $p \neq q$ and $(p,q) \in SSOD\_PRA_w$, that is, p and q are two distinct permissions that are related by the $SSOD\_PRA_w$ relation. A role x which has permission p at time d and location l cannot be assigned permission q at the same time and location if p and q are related by $SSOD\_PRA_w$.

$(p,q) \in SSOD\_PRA_w \Rightarrow PermRoleAcquire(p,x,d,l) \wedge PermRoleAcquire(q,x,d,l) = False$

[Strong Temporal Form of SSoD - Permission Role Assignment] The same role cannot be assigned two conflicting permissions at the same location at any time. In the top secret base, if any role has a permission to *access the high confidential information*, the permission to *store or distribute information* should not be granted to that role. Let p and q be two permissions such that $p \neq q$ and $(p,q) \in SSOD\_PRA_t$, that is, p and q are two distinct permissions that are related by the $SSOD\_PRA_t$ relation. A role x which has permission p at time d and location l cannot be assigned permission q at the any time d' in the same location if p and q are related by $SSOD\_PRA_t$.

$(p,q) \in SSOD\_PRA_t \Rightarrow PermRoleAcquire(p,x,d,l) \wedge PermRoleAcquire(q,x,d',l) = False$

[Strong Spatial Form of SSoD - Permission Role Assignment] The same role cannot be assigned two conflicting permissions at any location during the time. For example, the permission to *access the exam paper* and *access the answer key* should not be assigned for the same time Let p and q be two permissions such that $p \neq q$ and $(p,q) \in SSOD\_PRA_l$, that is, p and q are two distinct permissions that are related by the $SSOD\_PRA_l$ relation. A role x which has permission p at time d and location l cannot be assigned permission q at the same time at any location l' if p and q are related by $SSOD\_PRA_l$.

$(p,q) \in SSOD\_PRA_l \Rightarrow PermRoleAcquire(p,x,d,l) \wedge PermRoleAcquire(q,x,d,l') = False$

[Strong Form of SSoD - Permission Role Assignment] The same role cannot be assigned two conflicting permissions. For example, The permission to *issue check* and permission to *authorize check* must not be assign to the same role. Let p and q be two permissions such that $p \neq q$ and $(p,q) \in SSOD\_PRA_s$, that is, p and q are two distinct permissions that are related by the $SSOD\_PRA_s$ relation. A role x which has permission p at time d and location l cannot be assigned permission q at any time d' or at any location l' if p and q are related by $SSOD\_PRA_s$.

$(p,q) \in SSOD\_PRA_s \Rightarrow PermRoleAcquire(p,x,d,l) \wedge PermRoleAcquire(q,x,d',l') = False$

6 Impact of Time and Location on Dynamic Separation of Duties

Static separation of duty ensures that a user does not get assigned conflicting roles or a role is not assigned conflicting permissions. Dynamic separation of duty addresses the problem that a user is not able to activate conflicting roles during the same session.

[Weak Form of DSoD] This allows the same user to activate two conflicting roles in the same session but not during the same time and in the same location. A user can activate a sales assistant role and a customer role in the same session but not during the same time and in the same location.

Let x and y be two roles such that $x \neq y$ and $(x,y) \in DSOD_s$, that is, two distinct roles x and y are related by $DSOD_w$. If roles x and y are related through weak DSoD and if user u has activated role x in some session s for duration d and location l, then u cannot activate role y during the same time d and in the same location l in session s.

$(x,y) \in DSOD_w \Rightarrow SessionRole(u,x,s,d,l) \wedge SessionRole(u,y,s,d,l) = False$

[Strong Temporal Form of DSoD] This allows the same user to activate two conflicting roles in the same session but not in the same location at any time. For example, in a teaching session in a classroom, a user cannot activate the the grader role and the student role at any time. Let x and y be two roles such that $x \neq y$ and $(x,y) \in DSOD_t$, that is, two

distinct roles x and y are related by $DSOD_t$. If roles x and y are related through strong temporal DSoD and if user u has activated role x in some session s, then u can never activate role y at any time d' at the same location in the same session.

$(x,y) \in DSOD_t \Rightarrow SessionRole(u,x,s,d,l) \wedge SessionRole(u,y,s,d',l) = False$

[Strong Spatial Form of DSoD] This allows the same user to activate two conflicting roles in the same session but not at the same time in any location. If a user has activated the *Graduate Teaching Assistant* role in his office, he cannot activate the *Lab Operator* role at the same time in any location. Let x and y be two roles such that $x \neq y$ and $(x,y) \in DSOD_l$, that is, two distinct roles x and y are related by $DSOD_l$. If roles x and y are related through strong DSoD and if user u has activated role x in some session s, then u can never activate role y in session s during the same time in any location.

$(x,y) \in DSOD_l \Rightarrow SessionRole(u,x,s,d,l) \wedge SessionRole(u,y,s,d,l') = False$

[Strong Form of DSoD] A user can never activate the roles related through strong DSoD. For example, a user cannot be both a *code developer* and a *code tester* in the same session. Let x and y be two roles such that $x \neq y$ and $(x,y) \in DSOD_s$. If roles x and y are related through strong DSoD and if user u has activated role x in some session s, then u can never activate role y in the same session.

$(x,y) \in DSOD_s \Rightarrow SessionRole(u,x,s,d,l) \wedge SessionRole(u,y,s,d',l') = False$

7 Impact of Time and Location on Delegation

Many situations require the temporary transfer of access rights to accomplish a given task. For example, in a pervasive computing application, a doctor may give certain privilege to a trained nurse, when he is taking a short break. In such situations, the doctor can give a subset of his permission to the nurse for a given period of time. There are a number of different types of delegation. The entity that transfers his privileges temporarily to another entity is often referred to as the delegator. The entity who receives the privilege is known as the delegatee. The delegator (delegatee) can be either an user or a role. Thus, we may have four types of delegations: *user to user* (U2U), *user to role* (U2R), *role to role* (R2R), and *role to user* (R2U). System administrators are responsible for overseeing delegation when the delegator is a role. Individual users administer delegation when the delegator is an user. When a user is the delegator, he can delegate a subset of permissions that he possesses by virtue of being assigned to different roles. When a role is the delegator, he can delegate either a set of permissions or he can delegate the entire role. We can therefore classify delegation on the basis of role delegation or permission delegation. We identify the following types of delegation.

[U2U Unrestricted Permission Delegation] In this type of delegation, the delegatee can invoke the delegator's permissions at any time and at any place where the delegator could invoke those permissions. The illness of the company president caused him to delegate his email reading privilege to his secretary. Let $DelegateU2U\_P_u(u,v,Perm)$ be the predicate that allows user u to delegate the permissions in the set $Perm$ to user v without any temporal or spatial constraints. This will allow v to invoke the permissions at any time or at any place.

$\forall p \in Perm, DelegateU2U\_P_u(u,v,Perm) \wedge PermUserAcquire(u,o,p,d,l) \Rightarrow PermUserAcquire(v,o,p,d,l)$

[U2U Time Restricted Permission Delegation] The delegator places time restrictions on when the delegatee can in-

voke the permissions. However, no special restrictions are placed with respect to location – the delegatee can invoke the permission at any place that the delegator could do so. The professor can delegate his permission to proctor an exam to the teaching assistant while he is on travel. Let $DelegateU2U\_P_t(u,v,Perm,d')$ be the predicate that allows user u to delegate the permissions in the set $Perm$ to user v for the duration d'.

$\forall p \in Perm, DelegateU2U\_P_t(u,v,Perm,d') \wedge PermUserAcquire(u,o,p,d,l) \wedge (d' \subseteq d) \Rightarrow PermUserAcquire(v,o,p,d',l)$

[U2U Location Restricted Permission Delegation] A delegator can place spatial restrictions on when the delegatee can invoke the permissions. However, the only temporal restriction is that the delegatee can invoke the permissions during the period when the original permission is valid. The teaching assistant can delegate the permission regarding lab supervision to the lab operator only in the Computer Lab. Let $DelegateU2U\_P_l(u,v,Perm,l')$ be the predicate that allows user u to delegate the permissions in the set $Perm$ to user v in the location l'.

$\forall p \in Perm, DelegateU2U\_P_l(u,v,Perm,l') \wedge PermUserAcquire(u,o,p,d,l) \wedge (l' \subseteq l) \Rightarrow PermUserAcquire(v,o,p,d,l')$

[U2U Time Location Restricted Permission Delegation] In this case, the delegator imposes a limit on the time and the location where the delegatee can invoke the permission. A nurse can delegate his permission to oversee a patient while he is resting in his room to a relative. Let $DelegateU2U\_P_{tl}(u,v,Perm,d',l')$ be the predicate that allows user u to delegate the permissions in the set $Perm$ to user v in the location l' for the duration d'.

$\forall p \in Perm, DelegateU2U\_P_{tl}(u,v,Perm,t',l') \wedge PermUserAcquire(u,o,p,d,l) \wedge (d' \subseteq d) \wedge (l' \subseteq l) \Rightarrow PermUserAcquire(v,o,p,d',l')$

[U2U Unrestricted Role Delegation] The delegator delegates a role to the delegatee. The delegatee can activate the roles at any time and place where the delegator can activate those roles. A manager before relocating can delegate his roles to his successor in order to train him. Let $DelegateU2U\_R_u(u,v,r)$ be the predicate that allows user u to delegate his role r to user v.

$DelegateU2U\_R_u(u,v,r) \wedge UserRoleActivate(u,r,d,l) \Rightarrow UserRoleActivate(v,r,d,l)$

[U2U Time Restricted Role Delegation] In this case, the delegator delegates a role to the delegatee but the role can be activated only for a more limited duration than the original role. A user can delegate his role as a teacher to a responsible student while he is in a conference. Let $DelegateU2U\_R_t(u,v,r,d')$ be the predicate that allows user u to delegate his role r to user v for the duration d'.

$DelegateU2U\_R_t(u,v,r,d') \wedge UserRoleActivate(u,r,d,l) \wedge (d' \subseteq RoleEnableDur(r)) \wedge (d' \subseteq d) \Rightarrow UserRoleActivate(v,r,d',l)$

[U2U Location Restricted Role Delegation]: In this case, the delegator delegates a role to the delegatee but the role can be activated in more limited locations than the original role. A student can delegate his lab supervision role to another student in a designated portion of the lab only. Let $DelegateU2U\_R_l(u,v,r,l')$ be the predicate that allows user u to delegate his role r to user v in the location l'.

$Delegate\_R_l(u,v,r,l') \wedge UserRoleActivate(u,r,d,l) \wedge (l' \subseteq RoleEnableLoc(r)) \wedge (l' \subseteq l) \Rightarrow UserRoleActivate(v,r,d,l')$

[U2U Time Location Restricted Role Delegation] The delegator delegates the role, but the delegatee can activate the role for a limited duration in limited places. A student can delegate his lab supervision role to another student only in the lab when he leaves the lab for emergency reasons. Let $DelegateU2U\_R_{tl}(u,v,r,d',l')$ be the predicate that allows user u to delegate his role r to user v in location l' and time d'.

$DelegateU2U\_R_l(u,v,r,d',l') \wedge UserRoleActivate(u,r,d,l) \wedge (l' \subseteq RoleEnableLoc(r)) \wedge (d' \subseteq RoleEnableDur(r)) \wedge (d' \subseteq$

$d) \wedge (l' \subseteq l) \Rightarrow UserRoleActivate(v, r, d', l')$

[R2R Unrestricted Permission Delegation] All users assigned to the delegatee role can invoke the delegator role's permissions at any time and at any place where the user of this delegator role could invoke those permissions. The Smart Home owner role may delegate the permission to check the status of security sensors of the home to the police officer role, so all police officers can detect the intruder at any time at any place. Let $DelegateR2R\_P_u(r_1, r_2, Perm)$ be the predicate that allows role r_1 to delegate the permissions in the set $Perm$ to role r_2 without any temporal or spatial constraints. This will allow users in the role r_2 to invoke the permissions at any time or at any place.

$\forall p \in Perm, DelegateR2R\_P_u(r_1, r_2, Perm) \wedge PermRoleAcquire(p, r_1, d, l) \wedge (d \subseteq RoleEnableDur(r_2)) \wedge (l \subseteq RoleEnableLoc(r_2))$
$\Rightarrow PermRoleAcquire(p, r_2, d, l)$

[R2R Time Restricted Permission Delegation] The delegator role can place temporal restrictions on when the users of the delegatee role can invoke the permissions. No special restrictions are placed with respect to location i.e. the delegatee role's users can invoke the permissions at any place that the delegator role's users could do so. CS599 teacher role can grant the permission to access course materials to CS599 student role for the specific semester. Let $DelegateR2R\_P_t(r_1, r_2, Perm, d')$ be the predicate that allows role r_1 to delegate the permissions in the set $Perm$ to role r_2 for the duration d'.

$\forall p \in Perm, DelegateR2R\_P_t(r_1, r_2, Perm, d') \wedge (d' \subseteq d)\ PermRoleAcquire(p, r_1, d, l) \wedge$
$(l' \subseteq l) \wedge (d' \subseteq RoleEnableDur(r_2)) \wedge (l \subseteq RoleEnableLoc(r_2)) \Rightarrow PermRoleAcquire(p, r_2, d', l)$

[R2R Location Restricted Permission Delegation] The delegator role places spatial constraints on where the users of the delegatee role can invoke the permissions. No special temporal constraints are placed, that is, the delegatee role's users can invoke the permissions at any time that the delegator role's users could do so. The librarian role may grant the permission to checkout the book to the student role only at the self-checkout station. Let $DelegateR2R\_P_l(r_1, r_2, Perm, l')$ be the predicate that allows role r_1 to delegate the permissions in the set $Perm$ to role r_2 in the location l'.

$\forall p \in Perm, DelegateR2R\_P_l(r_1, r_2, Perm, l') \wedge PermRoleAcquire(p, r_1, d, l) \wedge$
$(d \subseteq RoleEnableDur(r_2)) \wedge (l' \subseteq RoleEnableLoc(r_2)) \wedge (l' \subseteq l) \Rightarrow PermRoleAcquire(p, r_2, d, l')$

[R2R Time Location Restricted Permission Delegation] The delegator role imposes a limit on the time and the location where the delegatee role's users could invoke the permissions. The daytime doctor role may delegate the permission to get his location information to the nurse role only when he is in the hospital during the daytime. Let $DelegateR2R\_P_{tl}(r_1, r_2, Perm, d', l')$ be the predicate that allows role r_1 to delegate the permissions in the set $Perm$ to role r_2 in the location l' for the duration d'.

$\forall p \in Perm, DelegateR2R\_P_{tl}(r_1, r_2, Perm, d', l') \wedge PermRoleAcquire(p, r_1, d, l) \wedge$
$(d' \subseteq RoleEnableDur(r_2)) \wedge (l' \subseteq RoleEnableLoc(r_2)) \wedge (d' \subseteq d) \wedge (l' \subseteq l) \Rightarrow PermRoleAcquire(p, r_2, d', l')$

[R2R Unrestricted Role Delegation] All users assigned to the delegatee role can activate the delegator role at any time and at any place where the user of this delegator role could activate the role. In the case of company reorganization, the manager role can be delegated to the manager successor role in order to train him. Let $DelegateR2R\_R_u(r_1, r_2)$ be the predicate that allows role r_1 to be delegated to role r_2.

$DelegateR2R\_R_u(r_1, r_2) \wedge UserRoleActivate(u, r_2, d, l) \wedge (d \subseteq RoleEnableDur(r_1)) \wedge (l \subseteq RoleEnableLoc(r_1)) \Rightarrow UserRoleActivate(u, r$

[R2R Time Restricted Role Delegation] The delegator places temporal constraints on when the users of the delegatee role can activate the delegator role. No special spatial constraints are placed i.e. the delegatee role's users can activate the delegator role at any place that the delegator role's users could do so. The permanent staff role can be delegated to the contract staff role during the contract period. Let $DelegateR2R\_R_t(r_1, r_2, d')$ be the predicate that allows role r_1 to be delegated to role r_2 for the duration d'.

$DelegateR2R\_R_t(r_1, r_2, d') \wedge UserRoleActivate(u, r_2, d', l) \wedge (d \subseteq RoleEnableDur(r_1)) \wedge (l \subseteq RoleEnableLoc(r_1)) \wedge (d' \subseteq d) \Rightarrow UserRoleActivate(u, r_1, d', l)$

[R2R Location Restricted Role Delegation] The delegator role can place spatial restrictions on where the users of the delegatee role can activate the delegator role. No special restrictions are placed with respect to time i.e. the delegatee role's users can activate the delegator role at any time that the delegator role's users could do so. The researcher role can be delegated to the lab assistant role at the specific area of the lab. Let $DelegateR2R\_R_l(r_1, r_2, l')$ be the predicate that allows role r_1 to be delegated to role r_2 in the location l'.

$DelegateR2R\_R_l(r_1, r_2, l') \wedge UserRoleActivate(u, r_2, d, l') \wedge (d \subseteq RoleEnableDur(r_1)) \wedge (l \subseteq RoleEnableLoc(r_1)) \wedge (l' \subseteq l) \Rightarrow UserRoleActivate(u, r_1, d, l')$

[R2R Time Location Restricted Role Delegation] In this case, the delegator role imposes a limit on the time and the location where the delegatee role's users could activate the role. The full-time researcher role can be delegated to the part-time researcher role only during the hiring period in the specific lab. Let $DelegateR2R\_R_{tl}(r_1, r_2, d', l')$ be the predicate that allows role r_1 to be delegated to role r_2 in the location l' for the duration d'.

$$DelegateR2R\_R_{tl}(r_1, r_2, d', l') \wedge UserRoleActivate(u, r_2, d', l') \wedge (d' \subseteq d) \wedge (l' \subseteq l) \wedge$$
$$(d \subseteq RoleEnableDur(r_1)) \wedge (l \subseteq RoleEnableLoc(r_1)) \wedge (d' \subseteq d) \wedge (l' \subseteq l) \Rightarrow UserRoleActivate(u, r_1, d', l')$$

8 Model Analysis

An Alloy model consists of *signature* declarations, *fields*, *facts* and *predicates*. Each signature consists of a set of *atoms* which are the basic entities in Alloy. Atoms are *indivisible* (they cannot be divided into smaller parts), *immutable* (their properties do not change) and *uninterpreted* (they do not have any inherent properties). Each field belongs to a signature and represents a relation between two or more signatures. A relation denotes a set of tuples of atoms. 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.

The basic types in the access control model, such as, *User*, *Time*, *Location*, *Role*, *Permission* and *Object* are represented as signatures. For instance, the declarations shown below define a set named *User* and a set named *Role* that represents the set of all users and the set of all roles in the system. Inside the *Role* signature body, we have four relations, namely, *RoleAllocLoc*, *RoleAllocDur*, *RoleEnableLoc*, and *RoleEnableDur* which relates *Role* to other signatures.

```
sig User{}
sig Role{
     RoleAllocLoc: Location,
```

```
RoleAllocDur: Time,
RoleEnableLoc: Location,
RoleEnableDur: Time}
```

The different relationships between the model components are also expressed as signatures. *RoleEnable* has a field called *member* that maps to a cartesian product of *Role*, *Time* and *Location*. *UserRoleAssignment*, *RolePermissionAssignment*, *UserLocation*, *ObjLocation*, *UserRoleActivate* and *PermRoleAcquire* are specified similarly. *PermUserAcquire* has a field called *member* that maps to a cartesian product of *User*, *Object*, *Permission* and *TimeLoc*. Note that, for *PermUserAcquire*, instead of declaring it as a cartesian product of product of *User*, *Object*, *Permission*, *Time* and *Location*, we have to define a special signature called *TimeLoc* which consists of two fields called *dur* and *loc* representing Time and Location, respectively. The rationale behind this indirect declaration is to overcome the limitation of Alloy, which limits the dimension of cartesian product to 4. And finally, *RoleHierarchy* has a field *RHmember* that represents a relationship between *Role* and *Role*. Note that we use the *abstract* signature to represent role hierarchy, and the different types of role hierarchy are modeled as the subsignatures of *RoleHierarchy*. The analyzer will recognize that role hierarchy consists of only these different types of role hierarchy, and nothing else.

```
one sig RoleEnable {member : Role-> Time ->Location}
one sig PermUserAcquire{member : User->Object->Permission->TimeLoc}
abstract sig RoleHierarchy{member : Role -> Role}
sig UPIH, TPIH, LPIH, TLPIH, UAH, TAH, LAH, TLAH extends RoleHierarchy{}
```

The various invariants in the access control model are represented as facts in Alloy. For instance, the fact *URActivate* states that for user *u* to activate role *r* during the time interval *d* and location *l*, this user has to be assigned to role *r* in location *l* during time *d*. Moreover, the location of the user must be a subset of the locations where the role is enabled, and the time must be in the time interval when role *r* can be enabled. This is specified in Alloy as shown below. Other invariants are modeled in a similar manner.

```
fact URActivate{
all u: User, r: Role, d: Time, l: Location, uras: UserRoleAssignment,
urac: UserRoleActivate |
((u->r->d->l) in urac.member) => (((u->r->d->l) in uras.member) &&
(l in r.RoleEnableLoc) && (d in r.RoleEnableDur))}
```

Alloy's *fact* feature is used to represent the effects of the different hierarchies. The fact *UPIHFact* represents the Unrestricted Permission Inheritance Hierarchy's property. The fact states that senior role `sr` can acquire all permission assigned to itself together with all permissions assigned to junior role `jr`. Note that, the permission assigned to junior role must never be assigned to the senior role.

```
//Unrestricted Permission Inheritance Hierarchy
fact UPIHFact{
```

```
    all sr, jr: Role, p: Permission, d: Time, l: Location, upih: UPIH,
        rpa: RolePermissionAssignment, pra: PermRoleAcquire |
        ((sr->jr in upih.member) && (jr->p->d->l in pra.member) &&
        (sr->p !in (rpa.member).Location.Time)) =>
             (sr->p->sr.RoleEnableDur->sr.RoleEnableLoc) in pra.member}
```

The separation of duty constraints are modeled as predicates. Consider the Weak form of Static Separation of Duties User Role Assignment. This constraint says that a user u assigned to role $r1$ du ring time d and location l cannot be assigned to its conflicts role $r2$ at the same time and location.

```
pred W_SSoD_URA(u: User, disj r1, r2: Role,
ura: UserRoleAssignment.member, d: Time, l: Location){
     ((u->r1->d->l) in ura) => ((u->r2->d->l) not in ura)}
```

The different types of delegation are also modeled as predicates. Consider the U2U Unrestricted Permission Delegation. This type of delegation says that a user dtr delegates his permission p to user dte. User dte can invoke the delegator's permission at any time and at any place where the delegator could invoke the permission.

```
//U2U Unrestricted Permission Delegation
pred u2uUPD(disj dtr, dte: User, p: Permission){
     all o: Object, tl: TimeLoc, puacq: PermUserAcquire |
         (dtr->o->p->tl in puacq.member) =>
               (dte->o->p->tl in puacq.member)}
```

Finally, to check for conflicts, we create an *assertion* that specifies the properties we want to verify and then let the ALLOY analyzer validate the assertion using the *check* command. If our assertion does not hold in the specified scope, ALLOY analyzer will generate a counterexample. For example, to check the interaction of the Weak form of SSOD Permission Role Assignment and the R2R Unrestricted Permission Delegation, we make the assertion shown below. The assertion does not hold as illustrated by the counterexample shown in Figure 1.

```
// WSSoD_PRA violation in the present of R2R Unrestricted
// Permission Delegation
assert TestConflict14_1{
     all disj rdtr, rdte: Role, disj p, q: Permission, d: Time,
     l: Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_1
```

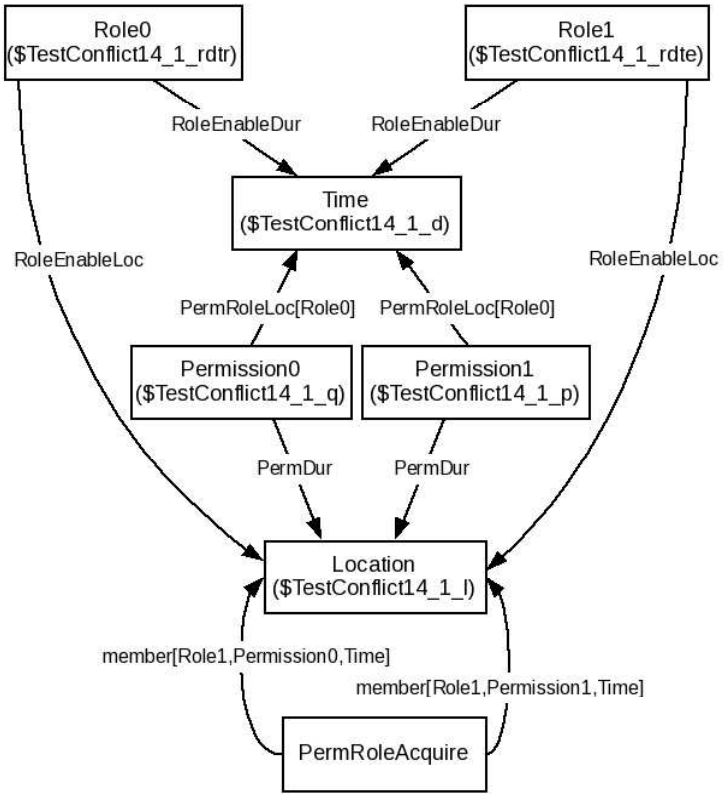

Figure 1: Counterexample for assertion TestConflict14_1

The counterexample shows one violation instance: $Role = \{Role0, Role1\}, Permission = \{Permission0, Permission1\}$ $Time = d$, $Location = l$ $PermRoleAcquire = \{Role1 \rightarrow Permission0 \rightarrow Time \rightarrow Location, Role1 \rightarrow Permission1 \rightarrow Time \rightarrow Location\}$ Substituting $rdtr$, $rdte$, p, q, d, and l in r2rUPD and W_SSoD_PRA predicates with $Role0$, $Role1$, $Permission0$, $Permission1$, d and l respectively, we get the violation. We checked the assertion on a HP-xw4400-Core2Duo-SATA with two Core2Duo 1.86Ghz CPU and 2 Gb memory running Linux 64. We used Version 4.1.2 Alloy Analyzer. The time taken to check this assertion was 20,572 ms. We created assertions and tested for other sources of conflicts. Our analysis revealed the various types of conflicts. Examples include conflicts of permission inheritance hierarchy with SSoD constraints, activation hierarchy with DSoD constraints, role delegation with DSoD constraints, and permission delegation with SSoD permission role assignments.

9 Conclusion and Future Work

Traditional access control models which do not take into account environmental factors before making access decisions may not be suitable for pervasive computing applications. Towards this end, researchers have proposed spatio-temporal role based access control models. However, such models have numerous features restricted by spatio-temporal constraints that may interact producing conflicts and inconsistencies. We have shown how such a spatio-temporal model

that supports delegation can be specified and automatically analyzed using Alloy. Our analysis reveals the potential conflicts that may occur in our model.

A lot of work remains to be done. Our analysis reveals undesirable interactions only for those case that we tested; the analysis is therefore not complete. We plan to investigate new techniques that will reveal all types of undesirable interactions without requiring input from the analysts. We also plan to investigate how to verify dynamic access control models where the entities and relationships are changed on the fly and the analysis must be done in real-time. Since pervasive computing applications are typically modeled as dynamic workflows, we need to extend our access control model to support them. Moreover, it is unlikely that a single access control model can fulfill all the needs for pervasive computing applications. Towards this end, we plan to investigate how our model can be composed with other models, and the result verified to give assurance that the resulting behavior is acceptable.

References

[1] Claudio A. Ardagna, Marco Cremonini, Ernesto Damiani, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Supporting location-based conditions in access control policies. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, pages 212–222, Taipei, Taiwan, March 2006.

[2] Vijayalakshmi Atluri and Soon Ae Chun. An authorization model for geospatial data. *IEEE Transactions on Dependable and Secure Computing*, 1(4):238–254, October-December 2004.

[3] Vijayalakshmi Atluri and Soon Ae Chun. A geotemporal role-based authorisation system. *International Journal of Information and Computer Security*, 1(1/2):143–168, January 2007.

[4] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: a temporal role-based access control model. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control*, pages 21–30, Berlin, Germany, July 2000.

[5] Elisa Bertino, Barbara Catania, Maria Luisa Damiani, and Paolo Perlasca. GEO-RBAC: a spatially aware RBAC. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, pages 29–37, Stockholm, Sweden, June 2005.

[6] Sudip Chakraborty and Indrajit Ray. TrustBAC: integrating trust relationships into the RBAC model for access control in open systems. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, pages 49–58, Lake Tahoe, CA, USA, June 2006. ACM.

[7] Suroop Mohan Chandran and James B. D. Joshi. *LoT-RBAC*: A Location and Time-Based RBAC Model. In *Proceedings of the 6th International Conference on Web Information Systems Engineering*, pages 361–375, New York, NY, USA, November 2005.

[8] Liang Chen and Jason Crampton. On Spatio-Temporal Constraints and Inheritance in Role-Based Access Control. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, pages 205–216, Tokyo, Japan, March 2008.

[9] Michael J. Covington, Prahlad Fogla, Zhiyuan Zhan, and Mustaque Ahamad. A Context-Aware Security Architecture for Emerging Applications. In *Proceedings of the Annual Computer Security Applications Conference* , pages 249–260, Las Vegas, NV, USA, December 2002.

[10] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind Dey, Mustaque Ahamad, and Gregory Abowd. Securing Context-Aware Applications Using Environment Roles. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, pages 10–20, Chantilly, VA, USA, May 2001.

[11] Geri Georg, James Bieman, and Robert B. France. Using Alloy and UML/OCL to Specify Run-Time Configuration Management: A Case Study. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists.*, volume P-7 of *LNI*, pages 128–141, 2001.

[12] Urs Hengartner and Peter Steenkiste. Implementing Access Control to People Location Information. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies*, pages 11–20, Yorktown Heights, NY, USA, June 2004.

[13] R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma. Context sensitive access control. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, pages 111–119, New York, NY, USA, 2005.

[14] Daniel Jackson. *Alloy 3.0 reference manual*. At http://alloy.mit.edu/reference-manual.pdf, 2004.

[15] James B. D. Joshi and Elisa Bertino. Fine-grained role-based delegation in presence of the hybrid role hierarchy. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, pages 81–90, Lake Tahoe, California, USA, June 2006.

[16] James B. D. Joshi, Elisa Bertino, Arif Ghafoor, and Yue Zhang. Formal foundations for hybrid hierarchies in GTRBAC. *ACM Transactions on Information and System Security*, 10(4):1–39, January 2008.

[17] James B. D. Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A Generalized Temporal Role-Based Access Control Model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, January 2005.

[18] Ulf Leonhardt and Jeff Magee. Security Consideration for a Distributed Location Service. *Imperial College of Science, Technology and Medicine, London, UK*, 1997.

[19] Fang Pu, Daoqin Sun, Qiying Cao, Haibin Cai, and Fan Yang. Pervasive Computing Context Access Control Based on $UCON_{ABC}$ Model. In *Proceedings of the 2nd International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 689–692, Pasadena, CA, December 2006.

[20] Indrakshi Ray and Mahendra Kumar. Towards a Location-Based Mandatory Access Control Model. *Computers & Security*, 25(1):36–44, February 2006.

[21] Indrakshi Ray, Mahendra Kumar, and Lijun Yu. LRBAC: A Location-Aware Role-Based Access Control Model. In *Proceedings of the 2nd International Conference on Information Systems Security*, pages 147–161, Kolkata, India, December 2006.

[22] Indrakshi Ray, Na Li, Robert France, and Dae-Kyoo Kim. Using UML to Visualize Role-Based Access Control Constraints. In *Proceedings of the 9th ACM symposium on Access Control Models and Technologies*, pages 115–124, Yorktown Heights, NY, USA, June 2004.

[23] Indrakshi Ray and Manachai Toahchoodee. A Spatio-temporal Role-Based Access Control Model. In *Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 211–226, Redondo Beach, CA, July 2007.

[24] Indrakshi Ray and Manachai Toahchoodee. A Spatio-Temporal Access Control Model Supporting Delegation for Pervasive Computing Applications. In *Proceedings of the 5th International Conference on Trust, Privacy & Security in Digital Business*, pages 48–58, Turin, Italy, September 2008.

[25] Geetanjali Sampemane, Prasad Naldurg, and Roy H. Campbell. Access Control for Active Spaces. In *Proceedings of the Annual Computer Security Applications Conference* , pages 343–352, Las Vegas, NV, USA, December 2002.

[26] Arjmand Samuel, Arif Ghafoor, and Elisa Bertino. A Framework for Specification and Verification of Generalized Spatio-Temporal Role Based Access Control Model. Technical report, Purdue University, February 2007. CERIAS TR 2007-08.

[27] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.

[28] Andreas Schaad and Jonathan D. Moffett. A Lightweight Approach to Specification and Analysis of Role-Based Access Control Extensions. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, pages 13–22, Monterey, CA, USA, June 2002.

[29] Richard Simon and Mary Ellen Zurko. Separation of Duty in Role-based Environments. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 183–194, Rockport, MA, USA, June 1997.

[30] Mana Taghdiri and Daniel Jackson. A Lightweight Formal Analysis of a Multicast Key Management Scheme. In *Formal Techniques for Networked and Distributed Systems - FORTE 2 003*, volume 2767 of *Lecture Notes in Computer Science*, pages 240–256, 2003.

[31] Manachai Toahchoodee and Indrakshi Ray. On the Formal Analysis of a Spatio-Temporal Role-Based Access Control Model. In *Proceedings of the 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 17–32, London, U.K., July 2008.

[32] Guo Ya-Jun, Hong Fan, Zhang Qing-Guo, and Li Rong. An Access Control Model for Ubiquitous Computing Application. In *Proceedings of the 2nd International Conference on Mobile Technology, Applications and Systems*, pages 1–6, Guangzhou, China, November 2005.

[33] Hai Yu and Ee-Peng Lim. LTAM: A Location-Temporal Authorization Model. In *Secure Data Management*, volume 3178 of *Lecture Notes in Computer Science*, pages 172–186, Toronto, Canada, August 2004.

[34] Chunyang Yuan, Yeping He, Jianbo He, and Zhouyi Zhou. A Verifiable Formal Specification for RBAC Model with Constraints of Separation of Duty. In *Proceedings of the 2nd SKLOIS Conference on Information Security and Cryptology*, pages 196–210, Beijing, China, November 2006.

[35] John Zao, Hoetech Wee, Jonathan Chu, and Daniel Jackson. *RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis*. At http://alloy.mit.edu/publications.php, 2002.

Appendix: Alloy Specification of the Spatio-Temporal Role-Based Access Control Model

```
module STRBAC

sig Time{}
sig Location{}

// For solving the dimension limitation problem
sig TimeLoc{
     dur : Time,
     loc : Location}

sig User{}
sig Role{
     RoleAllocLoc: Location,
     RoleAllocDur: Time,
     RoleEnableLoc: Location,
     RoleEnableDur: Time}
sig Permission{
     PermRoleLoc: Role->Location,
     PermObjLoc: Object->Location,
     PermDur: Time
```

```
}
sig Object{}

one sig RoleEnable {member : Role-> Time ->Location}
one sig UserRoleAssignment{member : User -> Role ->Time ->Location}
one sig RolePermissionAssignment{member : Role-> Permission ->Time->Location}
one sig UserLocation{member : User->Time->Location}
one sig ObjLocation{member : Object->Time->Location}
one sig UserRoleActivate{member : User-> Role->Time->Location}
one sig PermRoleAcquire{member : Role->Permission->Time->Location}
one sig PermUserAcquire{member : User->Object->Permission->TimeLoc}

abstract sig RoleHierarchy{member : Role -> Role}
sig UPIH, TPIH, LPIH, TLPIH, UAH, TAH, LAH, TLAH extends RoleHierarchy{}

fact ULoc{
     all u: User, uloc: UserLocation, d: Time, l1, l2: Location |
          (((u->d->l1) in uloc.member) && ((u->d->l2) in uloc.member)) <=>
          ((l1 in l2) || (l2 in l1))}

fact ObjLoc{
     all o: Object, oloc: ObjLocation, d: Time, l1, l2: Location |
          (((o->d->l1) in oloc.member) && ((o->d->l2) in oloc.member)) <=>
          ((l1 in l2) || (l2 in l1))}

// Each user must has role assigned to him
fact UserRole{
     all u: User, uras: UserRoleAssignment | some r: Role |
          u->r in (uras.member).Location.Time}

fact URAssign{
     all u: User, r: Role, d: Time, l: Location, ura: UserRoleAssignment,
     uloc: UserLocation |
          ((u->r->d->l) in ura.member) => (((u->d->l) in uloc.member) &&
          (l in r.RoleAllocLoc) && (d in r.RoleAllocDur))}
```

```
fact URActivate1{
     all u: User, sr, jr: Role, d: Time, l: Location, uras: UserRoleAssignment,
     urac: UserRoleActivate,
     uah: UAH, tah: TAH, lah: LAH, tlah: TLAH |
          ((u->jr->d->l) in urac.member) && (u->sr in (uras.member).Location.Time) &&
          (jr !in sr.^((uah + tah + lah + tlah).member)) =>
          (((u->jr->d->l) in uras.member) && (l in jr.RoleEnableLoc) &&
          (d in jr.RoleEnableDur))}

fact URActivate2{
     all u: User, sr, jr: Role, d: Time, l: Location, uras: UserRoleAssignment,
     uract: UserRoleActivate, uah: UAH, tah: TAH, lah: LAH, tlah: TLAH |
          ((u->jr->d->l in uract.member) && (u->sr in (uras.member).Location.Time) &&
          (u->jr !in (uras.member).Location.Time)) =>
          (jr in sr.^((uah + tah + lah + tlah).member))}

fact NocycleRH{
     all r: Role, RH: RoleHierarchy| r !in r.^(RH.member)}


// All types of hierarchy are disjointed
fact ScopeRH{
     all rh: RoleHierarchy, upih: UPIH, tpih: TPIH, lpih: LPIH, tlpih: TLPIH, uah: UAH,
     tah: TAH, lah: LAH, tlah: TLAH |
          (upih.member = rh.member - (tpih.member + lpih.member + tlpih.member +
                         uah.member + tah.member + lah.member + tlah.member)) &&
          (tpih.member = rh.member - (upih.member + lpih.member + tlpih.member +
                         uah.member + tah.member + lah.member + tlah.member)) &&
          (lpih.member = rh.member - (upih.member + tpih.member + tlpih.member +
                         uah.member + tah.member + lah.member + tlah.member)) &&
          (tlpih.member = rh.member - (upih.member + tpih.member + lpih.member +
                         uah.member + tah.member + lah.member + tlah.member)) &&
          (uah.member = rh.member - (upih.member + tpih.member + lpih.member +
                         tlpih.member + tah.member + lah.member + tlah.member)) &&
          (tah.member = rh.member - (upih.member + tpih.member + lpih.member +
                         tlpih.member + uah.member + lah.member + tlah.member)) &&
```

```
          (lah.member = rh.member - (upih.member + tpih.member + lpih.member +
                          tlpih.member + uah.member + tah.member + tlah.member)) &&
          (tlah.member = rh.member - (upih.member + tpih.member + lpih.member +
                          tlpih.member + uah.member + tah.member + lah.member))}


// Each role must has at least one permission assigned to it
fact RoleFact{
     all r: Role, rpa: RolePermissionAssignment |
          r in (rpa.member).Location.Time.Permission}

// All permissions assigned to roles can be acquired
fact RPAFact{
     all disj r: Role, p: Permission, d: Time, l : Location,
     rpa: RolePermissionAssignment, pra : PermRoleAcquire |
          (r->p->d->l in rpa.member) =>
             (r->p->d->l in pra.member)}

// All roles can acquire only their own assigned or inherited permissions
fact PRAFact{
     all disj r1, r2: Role, p: Permission, d1, d2: Time, l1, l2 : Location,
     rpa: RolePermissionAssignment, pra : PermRoleAcquire,
     upih: UPIH, tpih: TPIH, lpih: LPIH, tlpih: TLPIH |
          (r1->p->d1->l1 in pra.member) =>
             ((r1->p->d1->l1 in rpa.member) ||
             ((r2->p->d2->l2 in rpa.member) &&
             (r1->r2 in ((upih + tpih + lpih + tlpih).member)))))}

// Permission User Acquire
fact PUAFact{
     all r: Role, p: Permission, u: User, d: Time, l : Location, o: Object,
     tl: TimeLoc, pra : PermRoleAcquire, puacq: PermUserAcquire,
     ol: ObjLocation, urac: UserRoleActivate |
          ((r->p->d->l in pra.member) &&
          (u->r->d->l in urac.member) &&
          (o->d->l in ol.member) &&
```

```
          (o->l in p.PermObjLoc) &&
          (tl.dur = d) && (tl.loc = l)) =>
              (u->o->p->tl in puacq.member)
}

//Unrestricted Permission Inheritance Hierarchy
fact UPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, upih: UPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in upih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
              (sr->p->sr.RoleEnableDur->sr.RoleEnableLoc) in pra.member}

//Time Restricted Permission Inheritance Hierarchy
fact TPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, tpih: TPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in tpih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
              (sr->p->d->sr.RoleEnableLoc) in pra.member}

//Location Restricted Permission Inheritance Hierarchy
fact LPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, lpih: LPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in lpih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
              (sr->p->sr.RoleEnableDur->l) in pra.member}

//Time Location Restricted Permission Inheritance Hierarchy
fact TLPIHFact{
     all sr, jr: Role, p: Permission, d: Time, l: Location, tlpih: TLPIH,
         rpa: RolePermissionAssignment, pra: PermRoleAcquire |
         ((sr->jr in tlpih.member) && (jr->p->d->l in pra.member) &&
         (sr->p !in (rpa.member).Location.Time)) =>
              (sr->p->d->l) in pra.member}
```

```
//Unrestricted Activation Hierarchy
fact UAHFact{
     all disj sr, jr: Role, u: User, d: Time, l: Location, uah: UAH,
         uras: UserRoleAssignment, uract: UserRoleActivate |
         ((sr->jr in uah.member) && (u->sr->d->l in uract.member) &&
         (u->jr !in (uras.member).Location.Time) && (d in sr.RoleEnableDur) &&
         (l in sr.RoleEnableLoc)) =>
              (u->jr->d->l) in uract.member}

//Time Restricted Activation Hierarchy
fact TAHFact{
     all disj sr, jr: Role, u: User, d, d': Time, l: Location, tah: TAH,
         uras: UserRoleAssignment, uract: UserRoleActivate |
         ((sr->jr in tah.member) && (u->sr->d->l in uract.member) &&
         (u->jr !in (uras.member).Location.Time) && (d in sr.RoleEnableDur) &&
         (d' in jr.RoleEnableDur) && (l in sr.RoleEnableLoc)) =>
              (u->jr->d'->l) in uract.member}

//Location Restricted Activation Hierarchy
fact LAHFact{
     all disj sr, jr: Role, u: User, d: Time, l, l': Location, lah: LAH,
         uras: UserRoleAssignment, uract: UserRoleActivate |
         ((sr->jr in lah.member) && (u->sr->d->l in uract.member) &&
         (u->jr !in (uras.member).Location.Time) && (d in sr.RoleEnableDur) &&
         (l in sr.RoleEnableLoc) && (l' in jr.RoleEnableLoc)) =>
              (u->jr->d->l') in uract.member}

//Time Location Restricted Activation Hierarchy
fact TLAHFact{
     all disj sr, jr: Role, u: User, d, d': Time, l, l': Location, tlah: TLAH,
         uras: UserRoleAssignment, uract: UserRoleActivate |
         ((sr->jr in tlah.member) && (u->sr->d->l in uract.member) &&
         (u->jr !in (uras.member).Location.Time) && (d in sr.RoleEnableDur) &&
         (d' in jr.RoleEnableDur) && (l in sr.RoleEnableLoc) &&
         (l' in jr.RoleEnableLoc)) =>
```

```
              (u->jr->d'->l') in uract.member}

//Weak Form of SSoD-User Role Assignment
pred W_SSoD_URA(u: User, disj r1, r2: Role, ura: UserRoleAssignment.member,
d: Time, l: Location){
     ((u->r1->d->l) in ura) => ((u->r2->d->l) not in ura)}

//Strong Temporal Form of SSoD-User Role Assignment
pred ST_SSoD_URA(u: User, disj r1, r2: Role, ura: UserRoleAssignment.member,
d, d': Time, l: Location){
     ((u->r1->d->l) in ura) => ((u->r2->d'->l) not in ura)}

//Strong Spatial Form of SSoD-User Role Assignment
pred SS_SSoD_URA(u: User, disj r1, r2: Role, ura: UserRoleAssignment.member,
d: Time, l, l': Location){
     ((u->r1->d->l) in ura) => ((u->r2->d->l') not in ura)}

//Strong Form of SSoD-User Role Assignment
pred S_SSoD_URA(u: User, disj r1, r2: Role, ura: UserRoleAssignment.member,
d, d': Time, l, l': Location){
     ((u->r1->d->l) in ura) => ((u->r2->d'->l') not in ura)}

//Weak Form of SSoD-Permission Role Assignment
pred W_SSoD_PRA(r: Role, disj p, q : Permission,
d: Time, l: Location){
     all pra: PermRoleAcquire.member |
          ((r->p->d->l) in pra) => ((r->q->d->l) not in pra)}

//Strong Temporal Form of SSoD-Permission Role Assignment
pred ST_SSoD_PRA(r: Role, disj p, q : Permission,
d, d': Time, l: Location){
     all pra: PermRoleAcquire.member |
          ((r->p->d->l) in pra) => ((r->q->d'->l) not in pra)}

//Strong Spatial Form of SSoD-Permission Role Assignment
pred SS_SSoD_PRA(r: Role, disj p, q : Permission,
```

```
d: Time, l, l': Location){
     all pra: PermRoleAcquire.member |
          ((r->p->d->l) in pra) => ((r->q->d->l') not in pra)}

//Strong Form of SSoD-Permission Role Assignment
pred S_SSoD_PRA(r: Role, disj p, q : Permission,
d, d': Time, l, l': Location){
     all pra: PermRoleAcquire.member |
          ((r->p->d->l) in pra) => ((r->q->d'->l') not in pra)}

//Weak Form of DSoD
pred W_DSoD(u: User, disj r1, r2: Role, d: Time, l: Location){
     all urac: UserRoleActivate.member |
          ((u->r1->d->l) in urac) => ((u->r2->d->l) not in urac)}

//Strong Temporal Form of DSoD
pred ST_DSoD(u: User, disj r1, r2: Role, d, d': Time, l: Location){
     all urac: UserRoleActivate.member |
          ((u->r1->d->l) in urac) => ((u->r2->d'->l) not in urac)}

//Strong Spatial Form of DSoD
pred SS_DSoD(u: User, disj r1, r2: Role, d: Time, l, l': Location){
     all urac: UserRoleActivate.member |
          ((u->r1->d->l) in urac) => ((u->r2->d->l') not in urac)}

//Strong Form of DSoD
pred S_DSoD(u: User, disj r1, r2: Role, d, d': Time, l, l': Location){
     all urac: UserRoleActivate.member |
          ((u->r1->d->l) in urac) => ((u->r2->d'->l') not in urac)}

//U2U Unrestricted Permission Delegation
pred u2uUPD(disj dtr, dte: User, p: Permission){
     all o: Object, tl: TimeLoc, puacq: PermUserAcquire |
         (dtr->o->p->tl in puacq.member) =>
               (dte->o->p->tl in puacq.member)}
```

```
//U2U Time Restricted Permission Delegation
pred u2uTPD(disj dtr, dte: User, p: Permission, d': Time){
     all o: Object, tl, tl': TimeLoc, puacq: PermUserAcquire |
          ((d' in tl.dur) && (d' != tl.dur) && (tl'.dur = d') &&
          (tl'.loc = tl.loc) && (dtr->o->p->tl in puacq.member)) =>
                (dte->o->p->tl' in puacq.member)}

//U2U Location Restricted Permission Delegation
pred u2uLPD(disj dtr, dte: User, p: Permission, l': Location){
     all o: Object, tl, tl': TimeLoc, puacq: PermUserAcquire |
          ((tl'.dur = tl.dur) && (l' in tl.loc) && (l' != tl.loc) && (tl'.loc = l') &&
          (dtr->o->p->tl in puacq.member)) =>
                (dte->o->p->tl' in puacq.member)}

//U2U Time Location Restricted Permission Delegation
pred u2uTLPD(disj dtr, dte: User, p: Permission, d': Time, l': Location){
     all o: Object, tl, tl': TimeLoc, puacq: PermUserAcquire |
          ((d' in tl.dur) && (d' != tl.dur) && (l' in tl.loc) && (l' != tl.loc) &&
          (tl'.dur = d') && (tl'.loc = l') &&
          (dtr->o->p->tl in puacq.member)) =>
                (dte->o->p->tl' in puacq.member)}

//U2U Unrestricted Role Delegation
pred u2uURD(disj dtr, dte: User, r: Role){
     all d: Time, l: Location, urac: UserRoleActivate |
          (dtr->r->d->l in urac.member) =>
                (dte->r->d->l in urac.member)}

//U2U Time Restricted Role Delegation
pred u2uTRD(disj dtr, dte: User, r: Role, d': Time){
     all d: Time, l: Location, urac: UserRoleActivate |
          ((dtr->r->d->l in urac.member) && (d' in r.RoleEnableDur) &&
          (d' in d)) =>
                (dte->r->d'->l in urac.member)}

//U2U Location Restricted Role Delegation
```

```
pred u2uLRD(disj dtr, dte: User, r: Role, l': Location){
     all d: Time, l: Location, urac: UserRoleActivate |
         ((dtr->r->d->l in urac.member) && (l' in r.RoleEnableLoc) &&
         (l' in l)) =>
               (dte->r->d->l' in urac.member)}

//U2U Location Restricted Role Delegation
pred u2uTLRD(disj dtr, dte: User, r: Role, d': Time, l': Location){
     all d: Time, l: Location, urac: UserRoleActivate |
         ((dtr->r->d->l in urac.member) && (d' in r.RoleEnableDur) &&
         (d' in d) && (l' in r.RoleEnableLoc) && (l' in l)) =>
               (dte->r->d->l' in urac.member)}

//R2R Unrestricted Permission Delegation
pred r2rUPD(disj rdtr, rdte: Role, p: Permission){
     all d: Time, l: Location, pracq: PermRoleAcquire |
         ((rdtr->p->d->l in pracq.member) && (d in rdte.RoleEnableDur) &&
         (l in rdte.RoleEnableLoc)) =>
               (rdte->p->d->l in pracq.member)}

//R2R Time Restricted Permission Delegation
pred r2rTPD(disj rdtr, rdte: Role, p: Permission, d': Time){
     all d: Time, l: Location, pracq: PermRoleAcquire |
         ((rdtr->p->d->l in pracq.member) && (d' in d) &&
         (d' in rdte.RoleEnableDur) &&
         (l in rdte.RoleEnableLoc)) =>
               (rdte->p->d'->l in pracq.member)}

//R2R Location Restricted Permission Delegation
pred r2rLPD(disj rdtr, rdte: Role, p: Permission, l': Location){
     all d: Time, l: Location, pracq: PermRoleAcquire |
         ((rdtr->p->d->l in pracq.member) && (l' in l) &&
         (d in rdte.RoleEnableDur) &&
         (l' in rdte.RoleEnableLoc)) =>
               (rdte->p->d->l' in pracq.member)}
```

```
//R2R Time Location Restricted Permission Delegation
pred r2rTLPD(disj rdtr, rdte: Role, p: Permission, d': Time, l': Location){
     all d: Time, l: Location, pracq: PermRoleAcquire |
         ((rdtr->p->d->l in pracq.member) && (d' in d) && (l' in l) &&
         (d' in rdte.RoleEnableDur) &&
         (l' in rdte.RoleEnableLoc)) =>
               (rdte->p->d'->l' in pracq.member)}

//R2R Unrestricted Role Delegation
pred r2rURD(disj rdtr, rdte: Role){
     all u: User, d: Time, l: Location, urac: UserRoleActivate |
         ((u->rdte->d->l in urac.member) && (d in rdtr.RoleEnableDur) &&
         (l in rdtr.RoleEnableLoc))=>
               (u->rdtr->d->l in urac.member)}

//R2R Time Restricted Role Delegation
pred r2rTRD(disj rdtr, rdte: Role, d': Time){
     all u: User, l: Location, urac: UserRoleActivate |
         ((u->rdte->d'->l in urac.member) && (d' in rdtr.RoleEnableDur) &&
         (l in rdtr.RoleEnableLoc))=>
               (u->rdtr->d'->l in urac.member)}

//R2R Location Restricted Role Delegation
pred r2rLRD(disj rdtr, rdte: Role, l': Location){
     all u: User, d: Time, urac: UserRoleActivate |
         ((u->rdte->d->l' in urac.member) && (d in rdtr.RoleEnableDur) &&
         (l' in rdtr.RoleEnableLoc))=>
               (u->rdtr->d->l' in urac.member)}

//R2R Time Location Restricted Role Delegation
pred r2rTLRD(disj rdtr, rdte: Role, d': Time, l': Location){
     all u: User, urac: UserRoleActivate |
         ((u->rdte->d'->l' in urac.member) && (d' in rdtr.RoleEnableDur) &&
         (l' in rdtr.RoleEnableLoc))=>
               (u->rdtr->d'->l' in urac.member)}
```

```
// Conflicts with the Weak Form of SSOD-User Role Assignment: Condition 1
assert TestConflict1_1{
     no u: User, disj x, y: Role, upih: UPIH,
     d: Time, l: Location, ura: UserRoleAssignment |
          ((x->y in ^(upih.member))  &&
            (u->x->d->l in ura.member)) =>
               W_SSoD_URA[u, x, y, u->(x+y)->d->l, d, l]
}
check TestConflict1_1

// Conflicts with the Weak Form of SSOD-User Role Assignment: Condition 2
assert TestConflict1_2{
     all u: User, disj x, y: Role, tpih: TPIH, d: Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(tpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               W_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->y.RoleAllocDur->l), d, l]
}
check TestConflict1_2

// Conflicts with the Weak Form of SSOD-User Role Assignment: Condition 3
assert TestConflict1_3{
     all u: User, disj x, y: Role, lpih: LPIH, d: Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(lpih.member)) && (u->x->d->l in ura.member) &&
          (l in y.RoleAllocLoc)) =>
               W_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->y.RoleAllocLoc), d, l]
}
check TestConflict1_3

// Conflicts with the Weak Form of SSOD-User Role Assignment: Condition 4
assert TestConflict1_4{
     all u: User, disj x, y: Role, tlpih: TLPIH, d: Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(tlpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur) && (l in y.RoleAllocLoc)) =>
```

```
          W_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->y.RoleAllocDur->y.RoleAllocLoc), d, l]
}
check TestConflict1_4

// Conflicts with the Strong Temporal Form of SSOD-User Role Assignment: Condition 1
assert TestConflict2_1{
     all u: User, disj x, y: Role, upih: UPIH, d, d': Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(upih.member)) && (u->x->d->l in ura.member) &&
          (l in y.RoleAllocLoc)) =>
               ST_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->l), d, d', l]
}
check TestConflict2_1

// Conflicts with the Strong Temporal Form of SSOD-User Role Assignment: Condition 2
assert TestConflict2_2{
     all u: User, disj x, y: Role, tpih: TPIH, d, d': Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(tpih.member)) && (u->x->d->l in ura.member) &&
          (l in y.RoleAllocLoc)) =>
               ST_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->y.RoleAllocDur->l), d, d', l]
}
check TestConflict2_2

// Conflicts with the Strong Temporal Form of SSOD-User Role Assignment: Condition 3
assert TestConflict2_3{
     all u: User, disj x, y: Role, lpih: LPIH, d, d': Time, l: Location,
     ura: UserRoleAssignment |
          ((y in x.^(lpih.member)) && (u->x->d->l in ura.member) &&
          (l in y.RoleAllocLoc)) =>
               ST_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->y.RoleAllocLoc), d, d', l]
}
check TestConflict2_3

// Conflicts with the Strong Temporal Form of SSOD-User Role Assignment: Condition 4
assert TestConflict2_4{
```

```
     all u: User, disj x, y: Role, tlpih: TLPIH, d, d': Time, l: Location,
     ura: UserRoleAssignment |
           ((y in x.^(tlpih.member)) && (u->x->d->l in ura.member) &&
           (l in y.RoleAllocLoc)) =>
                ST_SSoD_URA[u, x, y,
                     (u->x->d->l) + (u->y->y.RoleAllocDur->y.RoleAllocLoc), d, d', l]
}
check TestConflict2_4

// Conflicts with the Strong Spatial Form of SSOD-User Role Assignment: Condition 1
assert TestConflict3_1{
     all u: User, disj x, y: Role, upih: UPIH, d: Time, l, l': Location,
     ura: UserRoleAssignment |
          ((y in x.^(upih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               SS_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->l), d, l, l']
}
check TestConflict3_1

// Conflicts with the Strong Spatial Form of SSOD-User Role Assignment: Condition 2
assert TestConflict3_2{
     all u: User, disj x, y: Role, tpih: TPIH, d: Time, l, l': Location,
     ura: UserRoleAssignment |
          ((y in x.^(tpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               SS_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->y.RoleAllocDur->l), d, l, l']
}
check TestConflict3_2

// Conflicts with the Strong Spatial Form of SSOD-User Role Assignment: Condition 3
assert TestConflict3_3{
     all u: User, disj x, y: Role, lpih: LPIH, d: Time, l, l': Location,
     ura: UserRoleAssignment |
          ((y in x.^(lpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               SS_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d->y.RoleAllocLoc), d, l, l']
```

```
}
check TestConflict3_3

// Conflicts with the Strong Spatial Form of SSOD-User Role Assignment: Condition 4
assert TestConflict3_4{
     all u: User, disj x, y: Role, tlpih: TLPIH, d: Time, l, l': Location,
     ura: UserRoleAssignment |
          ((y in x.^(tlpih.member)) && (u->x->d->l in ura.member) &&
          (d in y.RoleAllocDur)) =>
               SS_SSoD_URA[u, x, y,
                    (u->x->d->l) + (u->y->y.RoleAllocDur->y.RoleAllocLoc), d, l, l']
}
check TestConflict3_4

// Conflicts with the Strong Form of SSOD-User Role Assignment
assert TestConflict4{
     all u: User, disj x, y: Role, d, d': Time, l, l': Location,
     ura: UserRoleAssignment,
     upih: UPIH, tpih: TPIH, lpih: LPIH, tlpih: TLPIH |
          ((y in x.^(upih.member + tpih.member + lpih.member + tlpih.member)) &&
          (u->x->d->l in ura.member)) =>
               S_SSoD_URA[u, x, y, (u->x->d->l) + (u->y->d'->l'), d, d', l, l']
}
check TestConflict4

// Conflicts with the Weak Form of SSOD-Permission Role Assignment: Condition 1
assert TestConflict5_1{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, upih: UPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(upih.member))) =>
               W_SSoD_PRA[x, p, q, d, l]
}
check TestConflict5_1

// Conflicts with the Weak Form of SSOD-Permission Role Assignment: Condition 2
```

```
assert TestConflict5_2{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tpih: TPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tpih.member)) &&
          (l & l' != none)) =>
               W_SSoD_PRA[x, p, q, d, l]
}
check TestConflict5_2

// Conflicts with the Weak Form of SSOD-Permission Role Assignment: Condition 3
assert TestConflict5_3{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, lpih: LPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(lpih.member)) &&
          (l & l' != none)) =>
               W_SSoD_PRA[x, p, q, d, l]
}
check TestConflict5_3

// Conflicts with the Weak Form of SSOD-Permission Role Assignment: Condition 4
assert TestConflict5_4{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tlpih: TLPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tlpih.member)) &&
          (l & l' != none) && (d & d' != none)) =>
               W_SSoD_PRA[x, p, q, d, l]
}
check TestConflict5_4

// Conflicts with the Strong Temporal Form of SSOD-Permission Role Assignment: Condition 1
assert TestConflict6_1{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, upih: UPIH|
```

```
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(upih.member))) =>
               ST_SSoD_PRA[x, p, q, d, d', l]
}
check TestConflict6_1

// Conflicts with the Strong Temporal Form of SSOD-Permission Role Assignment: Condition 2
assert TestConflict6_2{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tpih: TPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tpih.member))) =>
               ST_SSoD_PRA[x, p, q, d, d', l]
}
check TestConflict6_2

// Conflicts with the Strong Temporal Form of SSOD-Permission Role Assignment: Condition 3
assert TestConflict6_3{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, lpih: LPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(lpih.member)) &&
          (l & l' != none)) =>
               ST_SSoD_PRA[x, p, q, d, d', l]
}
check TestConflict6_3

// Conflicts with the Strong Temporal Form of SSOD-Permission Role Assignment: Condition 4
assert TestConflict6_4{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tlpih: TLPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tlpih.member)) &&
          (l & l' != none)) =>
               ST_SSoD_PRA[x, p, q, d, d', l]
}
```

```
check TestConflict6_4

// Conflicts with the Strong Spatial Form of SSOD-Permission Role Assignment: Condition 1
assert TestConflict7_1{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, upih: UPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(upih.member))) =>
               SS_SSoD_PRA[x, p, q, d, l, l']
}
check TestConflict7_1

// Conflicts with the Strong Spatial Form of SSOD-Permission Role Assignment: Condition 2
assert TestConflict7_2{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, tpih: TPIH |
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tpih.member)) &&
          (d & d' != none)) =>
               SS_SSoD_PRA[x, p, q, d, l, l']
}
check TestConflict7_2

// Conflicts with the Strong Spatial Form of SSOD-Permission Role Assignment: Condition 3
assert TestConflict7_3{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member, lpih: LPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(lpih.member))) =>
               SS_SSoD_PRA[x, p, q, d, l, l']
}
check TestConflict7_3

// Conflicts with the Strong Spatial Form of SSOD-Permission Role Assignment: Condition 4
assert TestConflict7_4{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
```

```
     rpa: RolePermissionAssignment.member, tlpih: TLPIH|
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) && (y in x.^(tlpih.member)) &&
          (d & d' != none)) =>
               SS_SSoD_PRA[x, p, q, d, l, l']
}
check TestConflict7_4

// Conflicts with the Strong Form of SSOD-Permission Role Assignment
assert TestConflict8{
     all x, y: Role, disj p, q: Permission, d, d': Time, l, l': Location,
     rpa: RolePermissionAssignment.member,
     upih: UPIH, tpih: TPIH, lpih: LPIH, tlpih: TLPIH |
          (((x->p->d->l) in rpa) &&  (x->q not in rpa.Location.Time) &&
          (y->q->d'->l' in rpa) &&
          (y in x.^(upih.member + tpih.member + lpih.member + tlpih.member))) =>
               S_SSoD_PRA[x, p, q, d, d', l, l']
}
check TestConflict8

// Conflicts with the Weak Form of DSOD: Condition 1
assert TestConflict9_1{
     all u: User, disj x, y: Role, d: Time, l: Location,
     ura: UserRoleActivate.member, uah: UAH|
          (((u->x->d->l) in ura) && (y in x.^(uah.member))) =>
               W_DSoD[u, x, y, d, l]
}
check TestConflict9_1

// Conflicts with the Weak Form of DSOD: Condition 2
assert TestConflict9_2{
     all u: User, disj x, y: Role, d: Time, l: Location,
     ura: UserRoleActivate.member, tah: TAH|
          (((u->x->d->l) in ura) && (y in x.^(tah.member)) &&
          (d in y.RoleEnableDur)) =>
               W_DSoD[u, x, y, d, l]
```

```
}
check TestConflict9_2

// Conflicts with the Weak Form of DSOD: Condition 3
assert TestConflict9_3{
     all u: User, disj x, y: Role, d: Time, l: Location,
     ura: UserRoleActivate.member, lah: LAH|
          (((u->x->d->l) in ura) && (y in x.^(lah.member)) &&
          (l in y.RoleEnableLoc)) =>
               W_DSoD[u, x, y, d, l]
}
check TestConflict9_3

// Conflicts with the Weak Form of DSOD: Condition 4
assert TestConflict9_4{
     all u: User, disj x, y: Role, d: Time, l: Location,
     ura: UserRoleActivate.member, tlah: TLAH|
          (((u->x->d->l) in ura) && (y in x.^(tlah.member)) &&
          (d in y.RoleEnableDur) && (l in y.RoleEnableLoc)) =>
               W_DSoD[u, x, y, d, l]
}
check TestConflict9_4

// Conflicts with the Strong Temporal Form of DSOD: Condition 1
assert TestConflict10_1{
     all u: User, disj x, y: Role, d, d': Time, l: Location,
     ura: UserRoleActivate.member, uah: UAH|
          (((u->x->d->l) in ura) && (y in x.^(uah.member))) =>
               ST_DSoD[u, x, y, d, d', l]
}
check TestConflict10_1

// Conflicts with the Strong Temporal Form of DSOD: Condition 2
assert TestConflict10_2{
     all u: User, disj x, y: Role, d, d': Time, l: Location,
     ura: UserRoleActivate.member, tah: TAH|
```

```
          (((u->x->d->l) in ura) && (y in x.^(tah.member))) =>
               ST_DSoD[u, x, y, d, d', l]
}
check TestConflict10_2

// Conflicts with the Strong Temporal Form of DSOD: Condition 3
assert TestConflict10_3{
     all u: User, disj x, y: Role, d, d': Time, l: Location,
     ura: UserRoleActivate.member, lah: LAH|
          (((u->x->d->l) in ura) && (y in x.^(lah.member)) &&
          (l in y.RoleEnableLoc)) =>
               ST_DSoD[u, x, y, d, d', l]
}
check TestConflict10_3

// Conflicts with the Strong Temporal Form of DSOD: Condition 4
assert TestConflict10_4{
     all u: User, disj x, y: Role, d, d': Time, l: Location,
     ura: UserRoleActivate.member, tlah: TLAH|
          (((u->x->d->l) in ura) && (y in x.^(tlah.member)) &&
          (l in y.RoleEnableLoc)) =>
ST_DSoD[u, x, y, d, d', l]
}
check TestConflict10_4

// Conflicts with the Strong Spatial Form of DSOD: Condition 1
assert TestConflict11_1{
     all u: User, disj x, y: Role, d: Time, l, l': Location,
     ura: UserRoleActivate.member, uah: UAH|
          (((u->x->d->l) in ura) && (y in x.^(uah.member))) =>
          SS_DSoD[u, x, y, d, l, l']
}
check TestConflict11_1

// Conflicts with the Strong Spatial Form of DSOD: Condition 2
assert TestConflict11_2{
```

```
     all u: User, disj x, y: Role, d: Time, l, l': Location,
     ura: UserRoleActivate.member, tah: TAH|
          (((u->x->d->l) in ura) && (y in x.^(tah.member)) &&
          (d in y.RoleEnableDur)) =>
               SS_DSoD[u, x, y, d, l, l']
}
check TestConflict11_2

// Conflicts with the Strong Spatial Form of DSOD: Condition 3
assert TestConflict11_3{
     all u: User, disj x, y: Role, d: Time, l, l': Location,
     ura: UserRoleActivate.member, lah: LAH|
          (((u->x->d->l) in ura) && (y in x.^(lah.member))) =>
               SS_DSoD[u, x, y, d, l, l']
}
check TestConflict11_3

// Conflicts with the Strong Spatial Form of DSOD: Condition 4
assert TestConflict11_4{
     all u: User, disj x, y: Role, d: Time, l, l': Location,
     ura: UserRoleActivate.member, tlah: TLAH|
          (((u->x->d->l) in ura) && (y in x.^(tlah.member)) &&
          (d in y.RoleEnableDur)) =>
               SS_DSoD[u, x, y, d, l, l']
}
check TestConflict11_4

// Conflicts with the Strong Form of DSOD
assert TestConflict12{
     all u: User, disj x, y: Role, d, d': Time, l, l': Location,
     ura: UserRoleActivate.member, uah: UAH, tah: TAH, lah: LAH, tlah: TLAH|
          (((u->x->d->l) in ura) && (y in x.^((uah + tah + lah + tlah).member)) &&
          (d in y.RoleEnableDur)) =>
               S_DSoD[u, x, y, d, d', l, l']
}
check TestConflict12
```

```
// Conflicts in Permission Role Assignment
assert TestConflict13{
     all p: Permission, r: Role, d: Time, l: Location, rpa: RolePermissionAssignment,
     re: RoleEnable |
          (r->p->d->l in rpa.member) => (r->d->l in re.member)
}
check TestConflict13

// Conflicts between r2rUPD and the Weak Form of SSOD-PRA
assert TestConflict14_1{
     all rdtr, rdte: Role, disj p, q: Permission, d: Time, l: Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_1

// Conflicts between r2rTPD and the Weak Form of SSOD-PRA
assert TestConflict14_2{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l: Location |
          (r2rTPD[rdtr, rdte, p, d'] && r2rTPD[rdtr, rdte, q, d']) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_2

// Conflicts between r2rLPD and the Weak Form of SSOD-PRA
assert TestConflict14_3{
     all rdtr, rdte: Role, disj p, q: Permission, d: Time, l, l': Location |
          (r2rLPD[rdtr, rdte, p, l'] && r2rLPD[rdtr, rdte, q, l']) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_3

// Conflicts between r2rTLPD and the Weak Form of SSOD-PRA
assert TestConflict14_4{
```

```
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTLPD[rdtr, rdte, p, d', l'] && r2rTLPD[rdtr, rdte, q, d', l']) =>
               W_SSoD_PRA[rdte, p, q, d, l]
}
check TestConflict14_4

// Conflicts between r2rUPD and the Strong Temporal Form of SSOD-PRA
assert TestConflict15_1{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l: Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               ST_SSoD_PRA[rdte, p, q, d, d', l]
}
check TestConflict15_1

// Conflicts between r2rTPD and the Strong Temporal Form of SSOD-PRA
assert TestConflict15_2{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l: Location |
          (r2rTPD[rdtr, rdte, p, d'] && r2rTPD[rdtr, rdte, q, d']) =>
               ST_SSoD_PRA[rdte, p, q, d, d', l]
}
check TestConflict15_2

// Conflicts between r2rLPD and the Strong Temporal Form of SSOD-PRA
assert TestConflict15_3{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rLPD[rdtr, rdte, p, l'] && r2rLPD[rdtr, rdte, q, l']) =>
               ST_SSoD_PRA[rdte, p, q, d, d', l]
}
check TestConflict15_3

// Conflicts between r2rTLPD and the Strong Temporal Form of SSOD-PRA
assert TestConflict15_4{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTLPD[rdtr, rdte, p, d', l'] && r2rTLPD[rdtr, rdte, q, d', l']) =>
               ST_SSoD_PRA[rdte, p, q, d, d', l]
}
```

```
check TestConflict15_4

// Conflicts between r2rUPD and the Strong Spatial Form of SSOD-PRA
assert TestConflict16_1{
     all rdtr, rdte: Role, disj p, q: Permission, d: Time, l, l': Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               SS_SSoD_PRA[rdte, p, q, d, l, l']
}
check TestConflict16_1

// Conflicts between r2rTPD and the Strong Spatial Form of SSOD-PRA
assert TestConflict16_2{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTPD[rdtr, rdte, p, d'] && r2rTPD[rdtr, rdte, q, d']) =>
               SS_SSoD_PRA[rdte, p, q, d, l, l']
}
check TestConflict16_2

// Conflicts between r2rLPD and the Strong Spatial Form of SSOD-PRA
assert TestConflict16_3{
     all rdtr, rdte: Role, disj p, q: Permission, d: Time, l, l': Location |
          (r2rLPD[rdtr, rdte, p, l'] && r2rLPD[rdtr, rdte, q, l']) =>
               SS_SSoD_PRA[rdte, p, q, d, l, l']
}
check TestConflict16_3

// Conflicts between r2rTLPD and the Strong Spatial Form of SSOD-PRA
assert TestConflict16_4{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTLPD[rdtr, rdte, p, d', l'] && r2rTLPD[rdtr, rdte, q, d', l']) =>
               SS_SSoD_PRA[rdte, p, q, d, l, l']
}
check TestConflict16_4

// Conflicts between r2rUPD and the Strong Form of SSOD-PRA
assert TestConflict17_1{
```

```
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rUPD[rdtr, rdte, p] && r2rUPD[rdtr, rdte, q]) =>
               S_SSoD_PRA[rdte, p, q, d, d', l, l']
}
check TestConflict17_1

// Conflicts between r2rTPD and the Strong Form of SSOD-PRA
assert TestConflict17_2{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
           (r2rTPD[rdtr, rdte, p, d'] && r2rTPD[rdtr, rdte, q, d']) =>
               S_SSoD_PRA[rdte, p, q, d, d', l, l']
}
check TestConflict17_2

// Conflicts between r2rLPD and the Strong Form of SSOD-PRA
assert TestConflict17_3{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rLPD[rdtr, rdte, p, l'] && r2rLPD[rdtr, rdte, q, l']) =>
               S_SSoD_PRA[rdte, p, q, d, d', l, l']
}
check TestConflict17_3

// Conflicts between r2rTLPD and the Strong Form of SSOD-PRA
assert TestConflict17_4{
     all rdtr, rdte: Role, disj p, q: Permission, d, d': Time, l, l': Location |
          (r2rTLPD[rdtr, rdte, p, d', l'] && r2rTLPD[rdtr, rdte, q, d', l']) =>
               S_SSoD_PRA[rdte, p, q, d, d', l, l']
}
check TestConflict17_4

// Conflicts between r2rURD and the Weak Form of DSOD
assert TestConflict18_1{
     all u: User, disj rdtr, rdte: Role, d: Time, l: Location |
        r2rURD[rdtr, rdte] => W_DSoD[u, rdtr, rdte, d, l]
}
check TestConflict18_1
```

```
// Conflicts between r2rTRD and the Weak Form of DSOD
assert TestConflict18_2{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l: Location |
        r2rTRD[rdtr, rdte, d'] => W_DSoD[u, rdtr, rdte, d, l]
}
check TestConflict18_2

// Conflicts between r2rLRD and the Weak Form of DSOD
assert TestConflict18_3{
     all u: User, disj rdtr, rdte: Role, d: Time, l, l': Location |
          r2rLRD[rdtr, rdte, l'] => W_DSoD[u, rdtr, rdte, d, l]
}
check TestConflict18_3

// Conflicts between r2rTLRD and the Weak Form of DSOD
assert TestConflict18_4{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTLRD[rdtr, rdte, d', l'] => W_DSoD[u, rdtr, rdte, d, l]
}
check TestConflict18_4

// Conflicts between r2rURD and the Strong Temporal Form of DSOD
assert TestConflict19_1{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l: Location |
     r2rURD[rdtr, rdte] => ST_DSoD[u, rdtr, rdte, d, d', l]
}
check TestConflict19_1

// Conflicts between r2rTRD and the Strong Temporal Form of DSOD
assert TestConflict19_2{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l: Location |
     r2rTRD[rdtr, rdte, d'] => ST_DSoD[u, rdtr, rdte, d, d', l]
}
check TestConflict19_2
```

```
// Conflicts between r2rLRD and the Strong Temporal Form of DSOD
assert TestConflict19_3{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rLRD[rdtr, rdte, l'] => ST_DSoD[u, rdtr, rdte, d, d', l]
}
check TestConflict19_3

// Conflicts between r2rTLRD and the Strong Temporal Form of DSOD
assert TestConflict19_4{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTLRD[rdtr, rdte, d', l'] => ST_DSoD[u, rdtr, rdte, d, d', l]
}
check TestConflict19_4

// Conflicts between r2rURD and the Strong Spatial Form of DSOD
assert TestConflict20_1{
     all u: User, disj rdtr, rdte: Role, d: Time, l, l': Location |
          r2rURD[rdtr, rdte] => SS_DSoD[u, rdtr, rdte, d, l, l']
}
check TestConflict20_1

// Conflicts between r2rTRD and the Strong Spatial Form of DSOD
assert TestConflict20_2{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTRD[rdtr, rdte, d'] => SS_DSoD[u, rdtr, rdte, d, l, l']
}
check TestConflict20_2

// Conflicts between r2rLRD and the Strong Spatial Form of DSOD
assert TestConflict20_3{
     all u: User, disj rdtr, rdte: Role, d: Time, l, l': Location |
          r2rLRD[rdtr, rdte, l'] => SS_DSoD[u, rdtr, rdte, d, l, l']
}
check TestConflict20_3

// Conflicts between r2rTLRD and the Strong Spatial Form of DSOD
```

```
assert TestConflict20_4{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTLRD[rdtr, rdte, d', l'] => SS_DSoD[u, rdtr, rdte, d, l, l']
}
check TestConflict20_4

// Conflicts between r2rURD and the Strong Form of DSOD
assert TestConflict21_1{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rURD[rdtr, rdte] => S_DSoD[u, rdtr, rdte, d, d', l, l']
}
check TestConflict21_1

// Conflicts between r2rTRD and the Strong Form of DSOD
assert TestConflict21_2{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTRD[rdtr, rdte, d'] => S_DSoD[u, rdtr, rdte, d, d', l, l']
}
check TestConflict21_2

// Conflicts between r2rLRD and the Strong Form of DSOD
assert TestConflict21_3{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rLRD[rdtr, rdte, l'] => S_DSoD[u, rdtr, rdte, d, d', l, l']
}
check TestConflict21_3

// Conflicts between r2rTLRD and the Strong Form of DSOD
assert TestConflict21_4{
     all u: User, disj rdtr, rdte: Role, d, d': Time, l, l': Location |
          r2rTLRD[rdtr, rdte, d', l'] => S_DSoD[u, rdtr, rdte, d, d', l, l']
}
check TestConflict21_4
```