A Role Engineering Tool for Role-Based Access Control

Mark Strembeck
Institute of Information Systems, New Media Lab
Vienna University of Economics and BA, Austria
mark.strembeck@wu-wien.ac.at

Abstract

Like every requirements engineering process, the process of role engineering for role-based access control depends significantly on human factors. For this reason, many elements of the process cannot be automated (or at most partially). Nevertheless, tool support is necessary to cope with the complexity of the process and to efficiently handle the different interrelated artifacts used and produced during the role engineering process. In this paper, we present the design and implementation of the XORET *software tool which provides tool support for the scenario-driven role-engineering process. Furthermore,* XORET *is capable to produce a policy rule set that can be imported by the* XORBAC *access control service.*

1 Introduction and Motivation

Role engineering for role-based access control (RBAC, cf. [4]) is focused on the elicitation, specification, and maintenance of a policy rule set for RBAC. In other words, *role engineering* for role-based access control is the process of defining roles, permissions, constraints, and role-hierarchies (see [2]).

The scenario-driven role engineering process presented in [8, 13] is based on the scenario and goal concepts (see e.g. [1, 14]). With our work we essentially aim at a systematic role engineering methodology that is flexible enough to be applicable for arbitrary organizations and information systems. An important requirement for a role engineering methodology is the support of change management activities to ease the propagation of changes that occur within the information system or its environment into all security relevant models and finally into the policy rule set enforced by a concrete policy monitor. However, the intricacy of real world information systems and the respective models that are used and produced during role engineering activities cannot be conveniently handled via a “manually operated” paper-based process. Therefore, we need special purpose software tools that support role engineering activities and facilitate the incorporation of changes into a configuration consisting of several RBAC related models.

In this paper, we present the design and implementation of the XORET software tool which provides *tool support for the scenario-driven role-engineering process*. It facilitates the specification and inspection of trace-relations to ease change management activities, and it is capable to propose a (preliminary) policy rule set. Moreover, policy rule sets produced with XORET can be directly imported by the XORBAC access control service (see [7, 13]). The XORET role engineering tool consists of the XORET core component and the XORET graphical user interface. XORET allows to create and maintain the different objects that are used and produced during the scenario-driven role engineering process. Figure 1 shows the main window of the XORET tool. The foreground of this figure depicts and describes the toolbar located at the top of the XORET main window. The XORET tool is implemented in XOTcl [9] and Tcl/Tk [11].

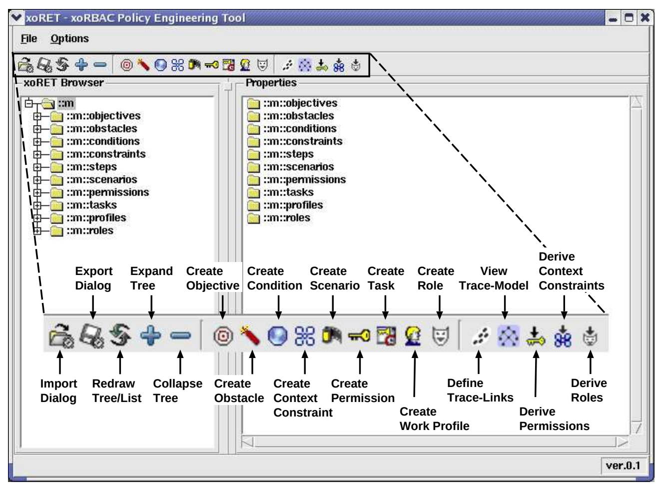


Figure 1. The XORET **GUI main window**

The paper is organized as follows. In Section 2 we give an overview of the scenario-driven role engineering process. Section 3 then goes into detail about the static design-time structures of the XORET core component. The subsequent sections give a description of the XORET graphical role engineering tool. However, due to the page limitation we can only describe a fraction of the functionality provided by XORET. Thus, we focus on the definition of processes/scenarios (Section 4), the specification of trace-relations between XORET objects and the inspection of the resulting trace model (Section 5). Furthermore, in Section 6 we go into detail about the (automatic) derivation of a (preliminary) policy rule set for RBAC.

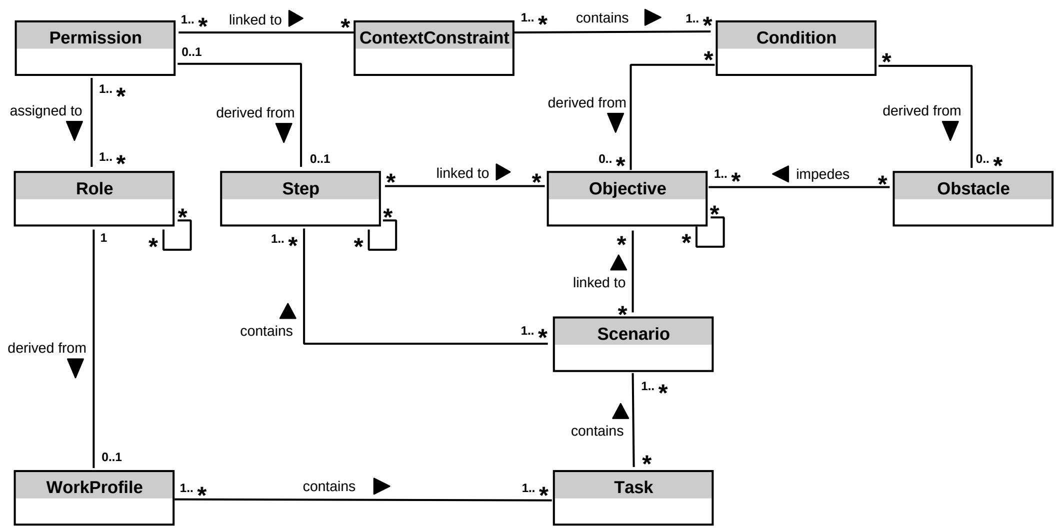


Figure 2. xoRET: main class relations

2 Scenario-driven Role Engineering

This section gives an overview of the scenario-driven role engineering process (for a more detailed description see [8, 13]). We model usage *scenarios* of an information system and use these scenarios/processes to derive permissions. Each scenario is in essence an action and event sequence consisting of several steps, for example to describe the processing of a damage event in an insurance company. Thus, to perform a certain scenario, a subject needs to be equipped with the exact number of permissions that are needed to complete each step of the respective scenario. A *task* consists of several scenarios and a *work profile* contains all tasks that a certain type of subject (e.g. a certain type of employee) is allowed to perform. In the further course of the role engineering process, work profiles are used to derive roles (see Section 6). Moreover, we apply *goals* and *obstacles* to define *context constraints* (see [13]). In our approach, a *control objective* is a goal specified by the authority which is responsible for the operation of a particular system. Thereby, control objectives define acceptable system behavior as intended by the system authority (see also [12]). Like other goals, control objectives can be defined on different levels of abstraction. The permission catalog, the constraint catalog, and the work profiles are used to define a concrete policy rule set for RBAC that is tailored to the needs of the corresponding information system (see also [12]). Furthermore, the role engineering process directly supports change management activities to allow for the correct and efficient propagation of changes into all (possibly) affected models (see also [8]).

3 xoRET: Static Design-time Structures

Figure 3 shows the classes of the xoRET core component. In particular, it contains ten "core classes" that represent the different artifacts which are needed to conduct the scenario-driven role engineering process (control objectives, obstacles, scenarios etc. - see also Section 2). Each core class is a subclass of the `xoRETObject` class. The `xoRETObject` class specifies common methods for all of its sub-classes, e.g. for the definition of traceability links (see Section 5).

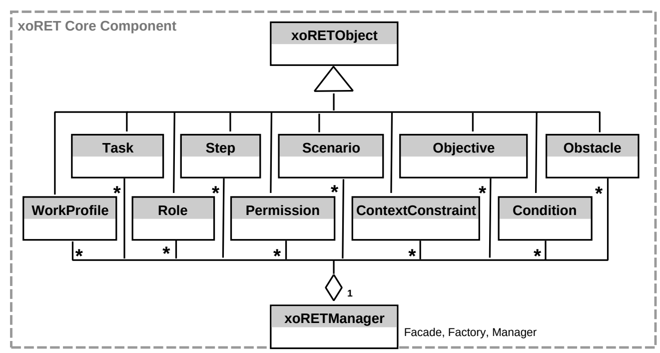


Figure 3. Classes of the xoRET core component

At runtime, each object that is instantiated from one of the ten core classes is aggregated within an `xoRETManager` object (see Figure 3). The `xoRETManager` class serves as *Facade* (see [5]) for the xoRET component. It hides internal xoRET structures and provides the external API of xoRET to other software components. Moreover, the `xoRETManager` class is a *Factory* (see [5]) and is responsible for the creation and management of its aggregated objects. For example, it controls the instantiation of new objects, writes status messages to log-files (if logging is activated), and observes the definition of traceability links between different xoRET objects.

Figure 2 depicts the main relations among the ten xoRET core classes in more detail. Scenario steps serve as a central means to derive `Permission` objects (see [8]). However, steps describing system internal actions sometimes do not need to be mapped to explicit user-level permissions. Nevertheless, depending on the intended abstraction level of the resulting policy rule set, it may be sensible to also model permis-

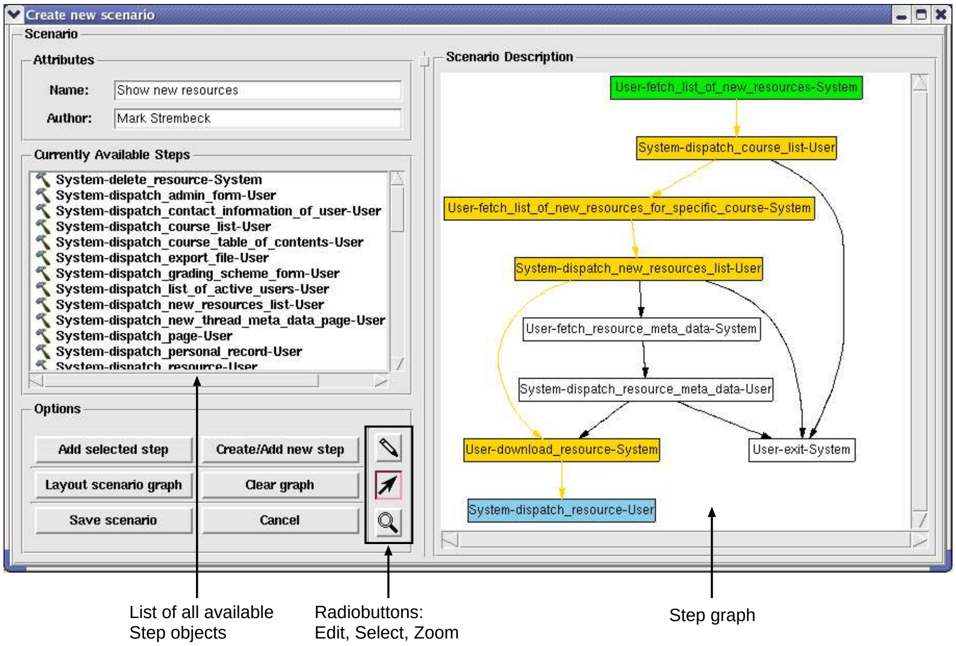


Figure 4. The XORET scenario creation dialog

sions pertaining to internal actions. Furthermore, to provide a flexible modeling environment, XORET allows to define additional `Permission` objects independent from scenario steps - resulting in a `0..1` relation between the `Permission` and `Step` classes (see Figure 2). Similarly, work profiles are the primary means to derive (preliminary) roles. However, in the course of the role engineering process derived roles are adapted, redundant roles can be deleted, and additional roles may be defined independently from the work profiles (see [8]). In other words, not every role resulting from the role engineering process necessarily originates (directly) from a work profile.

`Condition` objects are associated with `Objective` or `Obstacle` objects (see also [12, 13]). In particular, each `Objective` and `Obstacle` object contains a list of abstract conditions that are represented by a short, descriptive, and informal sentence respectively (e.g. "students may only edit their own exam", or "date greater than 2006/01/01"). In a later step, these abstract conditions are used to derive placeholder `Condition` objects (see Section 6). Each `ContextConstraint` object is linked to one or more `Condition` objects, and to one or more `Permission` objects (see Figure 2). Beside the (automatically recorded) implicit trace relations between XORET objects, it is possible to define explicit traces (see Section 5).

4 Definition of Scenarios

The scenario creation dialog shown in Figure 4 enables the definition of XORET scenarios. The primary element of a XORET scenario/process is a directed graph of steps. Although not depicted in Figure 4, it is possible to define labels for the edges in a step graph (e.g. to specify guard conditions). The left hand side of Figure 4 shows a (scrollable) list of all `Step` objects that are currently available. The right hand side shows the step graph of the current scenario/process. The step graph can be accessed via three different functions, each of which is activated through a respective radiobutton (see Figure 4):

- The *pencil-button* activates the *edit* mode which allows to *add* or *remove* steps or edges.
- The *pointer-button* activates the *select* mode. The select mode, again, offers two distinct functions:
  - First, one can choose a start node and an end node (by drawing a "virtual" edge between the two nodes) to find the *shortest path* between two nodes (i.e. between two steps). In particular, XORET performs a breadth-first-search to find the shortest path and colors the affected nodes as follows: the start node is colored green, the end node is colored blue, and all intermediate steps and edges are colored yellow. The example in Figure 4 shows the shortest path starting from the step "User, fetch list of new resources, from System" to the step "System, dispatch resource, to User" (note that each XORET step consists of triplet including an *actor*, performing an *action* directed at a specific *target*).
  - Second, one can compute the *spanning tree* for one particular node in the step graph. The spanning tree is computed, if, in select mode, a node is clicked without drawing an edge to another node. The spanning tree visualizes the reachability of other steps/nodes (i.e. it shows which nodes can be reached starting from a specific node).
- The *magnify-button* offers a *zoom* function for the graph.

As mentioned above, each XORET `Scenario` object contains a graph structure. The nodes of this graph are steps which are connected via labeled edges. Labeled edges can be used to model guard conditions (conditional branches) for step transitions. Thus, we can specify simple linear step sequences as well as complex scenarios/processes with an arbitrary number of fork and join points, and/or conditional branches between steps. Therefore, XORET scenarios can model scenarios/processes on an arbitrary level of abstraction.

5 Trace Definition and Inspection

The trace-management dialog allows for the specification and deletion of explicit trace relations between XORET objects (see Figure 5). The left hand side of the dialog offers five drop-down lists (comboboxes) allowing to define a specific trace relation. The *Start type* combobox chooses the object-type a specific trace starts from. Depending on the selected start-type, the *Start object* combobox is filled with the names of all instances of the corresponding start-type. The same procedure is applied for the *Destination type* and *Destination object* boxes. The *Trace type* box offers a (extensible) list of trace-relations that can be established between XORET objects.

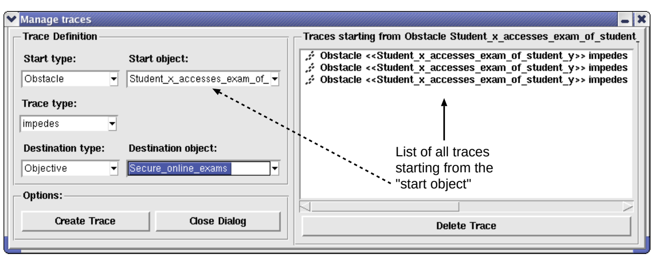


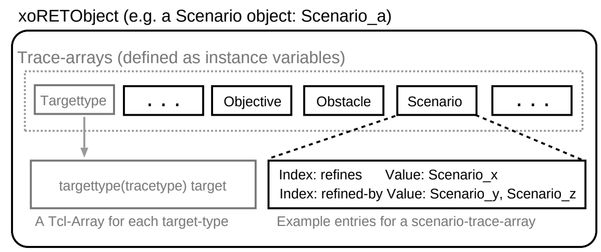
Figure 5. The XORET trace management dialog

The `addTraceRelation` method shown in Figure 6 defines bidirectional trace relations between XORET objects. In XORET each trace-type is associated with a *counterpart* which defines the name of the corresponding opposite trace-direction. Examples for such trace-type pairs are: [*sub-goal, super-goal*], [*contains, part-of*], or [*refines, refined-by*]. The `tracetype_counterparts` instance variable is an array that stores all valid XORET trace-type pairs. At any time, arbitrary new trace-type pairs can be defined via the `addTraceTypePair` method. To establish a bidirectional trace between two objects the `addTraceRelation` method establishes two directly opposed unidirectional traces.

```
xoRETManager instproc addTraceRelation {sourcetype source
                                        tracetype targettype target} {
  if {[my isValidTraceType $tracetype]} {
     if {[$source addTrace $tracetype $targettype $target]} {
        return [$target addTrace \
                  [my set tracetype_counterparts($tracetype)] $sourcetype $source]
     }
  }
  return 0
}
```

Figure 6. The `addTraceRelation` method

Each `xoRETObject` uses a number of internal arrays to store trace relations. The name of a particular trace-array corresponds to the object-type (i.e. the class-name) of the target-objects that are referenced via this array (see Figure 7). Each array index represents a particular trace-type (e.g. contains, refines, part-of, owns), and the value stored at a specific index position is a list consisting of one or more object-names. For example, a `Scenario` object $Scenario_a$ may have several traces to other `Scenario` objects. Thus, the corresponding trace-links are stored in an array named "Scenario" (cf. Figure 7). The traces depicted in Figure 7 define that $Scenario_a$ refines $Scenario_x$ and is refined by $Scenario_y$ and $Scenario_z$.




```
xoRETObject instproc addTrace {tracetype targettype target} {
  my instvar $targettype
  if {![my existTrace $tracetype $targettype $target]} {
    if {[info exists [set targettype]($tracetype)]} {
      set current [set [set targettype]($tracetype)]
      set new [lappend current $target]
      set [set targettype]($tracetype) $new
      return 1
    } else {
      lappend [set targettype]($tracetype) $target
      return 1
    }
  }
  return 0
}
```

Figure 7. The `addTrace` method

The lower part of Figure 7 shows the `addTrace` method defined in the `xoRETObject` class. It receives three mandatory input parameters - `tracetype`, `targettype`, and `target`. If the respective trace does not already exist, the method adds the corresponding trace relation as discussed above.

An alternative option to specify trace relations between different programming objects would be the definition of explicit trace-objects. However, in our experiences the above mentioned procedure to store traces as array variables of XORET objects has several advantages compared to explicit trace-objects. The array-based approach offers a straightforward way to query trace relations in many different ways, e.g. to fetch all traces associated with a particular XORET object, or to fetch all traces of a particular trace-type pointing from a given object to other XORET objects of a certain object-type; for example, all *fulfills* traces pointing from $Scenario_a$ to instances of the `Objective` class. In contrast to that, it is, in the general case, by far more complicated to perform similar queries in a set of explicit trace-objects (in general, one may either perform a brute force search traversing all trace-objects or define a respective query-language). An advantage of individual trace-objects could be the definition of additional information attached to a trace-relation itself. Nevertheless, if required, this could also be integrated with the array-based approach.

All trace relations together form the XORET trace model which can be visualized and inspected to analyze the dependencies between XORET objects. The respective dialog allows

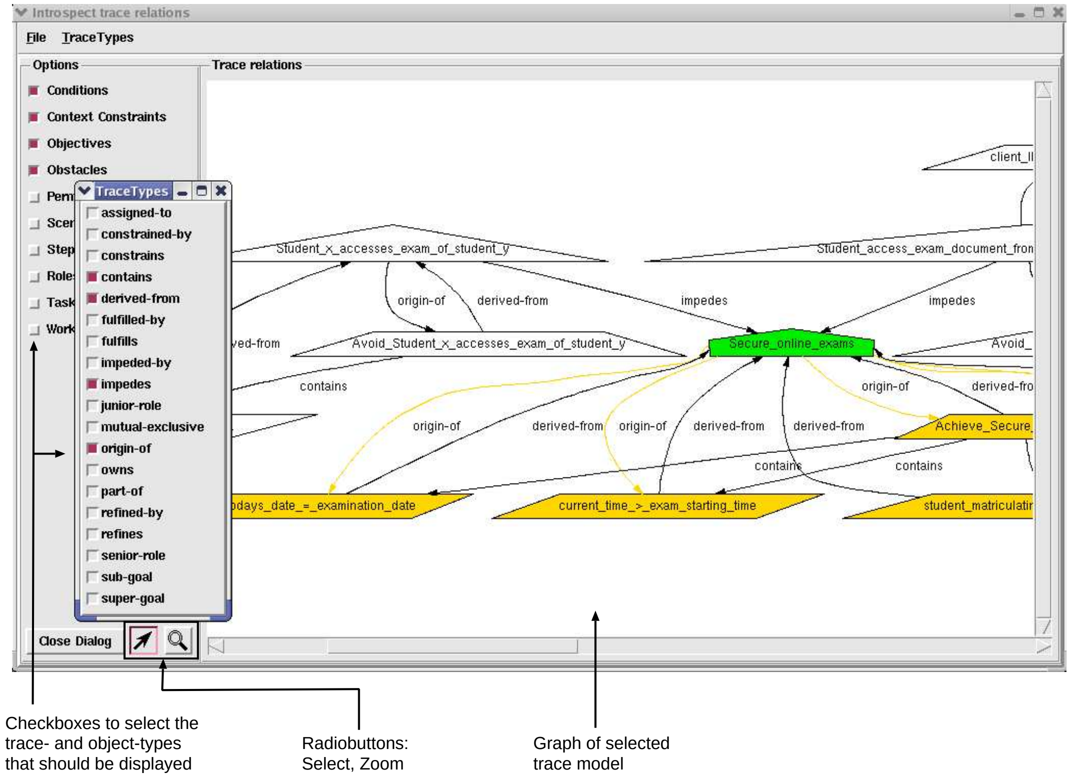


Figure 8. The xoRET trace model inspection dialog

to selectively display certain trace relations (see Figure 8). In particular, it allows to select the object types and the trace types to be visualized. Similar to the scenario definition dialog (see Section 4), the trace model dialog allows to *zoom* the corresponding model, to compute the *spanning tree* of a particular node, or to compute the *shortest path* between two nodes. The example depicted in Figure 8 shows the spanning tree for the control objective "Secure online exams". In xoRET the spanning tree is colored as follows: the start node is colored green and all reachable nodes, as well as the edges required for the spanning tree, are colored yellow.

6 Derivation of a Preliminary Policy Rule Set

Once the control objectives, obstacles, scenarios, tasks, and work profiles are defined, we can automatically derive a (preliminary) RBAC policy rule set (see also [8, 13]). In our approach an RBAC policy rule set consists of roles, permissions, context constraints, conditions, and the corresponding assignment relations. We now describe the xoRET methods that are applied to derive a preliminary policy rule set. The `derivePermissionFromStep` method (see Figure 9) receives three mandatory parameters to identify the respective step (`actor`, `action`, `target`). It uses the `lookup` method (described below) to determine the fully qualified name of the corresponding `Step` object and to create a new `Permission` object, if necessary. Subsequently, a [*derived-from Step*] trace is established between the two `Permission` and `Step` objects.

```
xoRETManager instproc derivePermissionFromStep {actor action target} {
  set step [Step lookup 0 [self] $actor $action $target]
  if {$step != "FAILED"} {
    set perm [Permission lookup 1 [self] $action $target]
    # establish trace-link if necessary
    if {![$perm existTrace derived-from Step $step]} {
       [self] addTraceRelation \
             Permission [$perm name] derived-from Step [$step name]
    }
    return 1
  }
  return 0
}
Permission proc lookup {create parent args} {
  set permname [eval join $args _]
  set permission ${parent}::permissions::$permname
  if {[$parent existPermission $permission]} {
    return $permission
  }
  if {$create} {
   return [Permission $permission -name      $permname \
                                  -operation [lindex $args 0] \
                                  -object    [lindex $args 1]]
  }
  return FAILED
  }
}
```

Figure 9. Derive permissions from steps

The `lookup` method is defined as an object-specific `proc` on each of the ten core classes (see Section 3). This method provides two functions depending on the value of the `create` parameter. Figure 9 shows the `lookup` method defined for the `Permission` class. If the `create` parameter is set to `0` (false) the `lookup` method checks if a respective object exists and either returns the fully qualified object name or the value "FAILED" (see Figure 9). In case the `create` parameter is set to `1` (true), the `lookup` method creates a respective object, if necessary.

The `deriveRoleFromWorkProfile` method is applied to automatically generate a `Role` object for each work profile (see

```
xoRETManager instproc deriveRoleFromWorkProfile {profile} {
  set profile [WorkProfile lookup 0 [self] $profile]
  if {$profile != "FAILED"} {
    set role [Role lookup 1 [self] "Role [$profile name]"]
    # establish trace-link if necessary
    if {![$role existTrace derived-from WorkProfile $profile]} {
       [self] addTraceRelation \
              Role [$role name] derived-from WorkProfile [$profile name]
    }
    # assign permissions if necessary
    foreach p [my getPermissionsLinkedToWorkProfile [$profile name]] {
      if {![$role ownsPerm $p]} {
        [self] permRoleAssign [$p name] [$role name]
      }
    }
    return 1
  }
  return 0
}
```

Figure 10. Derive roles from work profiles

Figure 10). The name of the new `Role` object consists of the respective work profile name with the prefix “Role”; so the name of a role derived from the “Teacher” work profile would be “Role_Teacher” for instance. If necessary, a corresponding `Role` object is created (via the `lookup` method), and a [*derived-from WorkProfile*] trace-link is established. Subsequently, the `getPermissionsLinkedToWorkProfile` method determines the permissions linked to the respective `WorkProfile` object (see Figure 10). To find the corresponding permissions the method follows the traceability links established between work profiles and tasks, tasks and scenarios, scenarios and steps, and steps and derived permissions (see also [8]). Finally, the respective `Permission` objects are assigned to the (new) role.

```
xoRETManager instproc derivePreliminaryRoleHierarchy {} {
  # check for redundancies and add junior-role relations
  foreach r1 [my getRoleList] {
    foreach r2 [my getRoleList] {
      if {$r1 != $r2} {
        if {[[self] equalPermissions [$r1 name] [$r2 name]]} {
           $r1 roleswithequalpermissions [concat [$r1 roleswithequalpermissions] $r2]
           $r2 roleswithequalpermissions [concat [$r2 roleswithequalpermissions] $r1]
        }
        if {[[self] role1PermsSupersetOfRole2Perms [$r1 name] [$r2 name]]} {
          if {![$r1 hasJuniorRole $r2]} {
            [self] addJuniorRoleRelation [$r1 name] [$r2 name]
          }
        }
      }
    }
  }
  # remove redundant junior-role relations
  [self] updateRoleHierarchy
  # remove redundant permissions
  [self] removeRedundantPermissionsFromRoles
}
```

Figure 11. Derive a preliminary role hierarchy

The `derivePreliminaryRoleHierarchy` method (cf. Figure 11) automatically derives a (preliminary) role-hierarchy (or hierarchies). Two roles owning an identical set of permissions may be redundant and are thus candidates for later refinement or deletion. In particular, each `Role` object has a `roleswithequalpermissions` instance variable to store links to “identical” roles (see Figure 11). For two distinct roles r_1 and r_2, the `role1PermsSupersetOfRole2Perms` method checks if the permissions of r_2 constitute a subset of the permissions of r_1. In such a case, we say that r_1 is greater than r_2 ($r_1 > r_2$). If the method finds a respective pair of roles, r_2 is defined as junior-role for r_1. After the junior-role derivation procedure is finished, we remove redundant junior-role relations and redundant permissions from each role.

To remove redundant junior-role relations the `updateRoleHierarchy` method invokes the `removeRedundantSuperclasses` method for each `Role` object aggregated by the corresponding xoRET instance (see Figure 12). In xoRET and xoRBAC roles are defined as classes which can be arranged in a class hierarchy to form a directed acyclic graph. The superclass relation between XOTcl classes is thus applied to straightforwardly define a role-hierarchy (for further details see [7]). The `removeRedundantSuperclasses` method uses the XOTcl introspection mechanism (see also [9, 15]) to gain knowledge about the superclasses of the corresponding `Role` object. It then removes all roles from the superclass list which are direct *and* indirect superclasses of the corresponding `Role` object at the same time (see Figure 12). Finally, it registers the new (redundance-free) superclass list for the respective `Role` object.

```
xoRETManager instproc updateRoleHierarchy {} {
  foreach role [my getRoleList] {
    $role removeRedundantSuperclasses
  }
}
Role instproc removeRedundantSuperclasses {} {
  set new [my info superclass]
  foreach supercl [my info superclass] {
    foreach class [$supercl info heritage] {
      set index [lsearch -exact $new $class]
      #if class is in "my superclass"
      #and "my superclass heritage"
      if {$index != -1} {
        #remove class from the superclass list of self
        set new [lreplace $new $index $index]
      }
    }
  }
  #set new superclass list
  [self] superclass $new
}
```

Figure 12. Removing redundant role relations

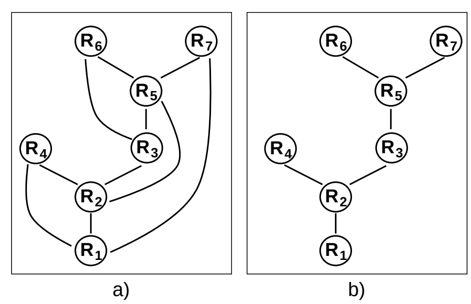


Figure 13. Redundant inheritance relations

Figure 13a) depicts a hierarchy with redundant inheritance relations. In particular, the relations between the following role pairs are redundant $[R_1, R_4]$, $[R_1, R_7]$, $[R_2, R_5]$, and $[R_3, R_6]$. For the example in Figure 13a) the `updateRoleHierarchy` method removes all redundant inheritance relations resulting in the hierarchy depicted in Figure 13b).

The `removeRedundantPermissionsFromRoles` method shown in Figure 14 identifies and revokes all permissions which are, at the same time, directly *and* indirectly (through the role-hierarchy) assigned to a xoRET `Role` object. In other words, we revoke all permissions that are directly assigned to a role and are also inherited from its junior-roles. When this step is finished we have defined a preliminary role-hierarchy (see also Figure 11).

Figure 15 shows the `deriveCCFromObstacle` method which defines how xoRET derives (preliminary) `ContextConstraint` objects from `Obstacle` objects (see also [12]). The corresponding XORET method to derive context constraints from `Objective` objects (`deriveCCFromObjective`) is similar to the derivation procedure for obstacles.

```
xoRETManager instproc removeRedundantPermissionsFromRoles {} {
  foreach role [my getRoleList] {
    foreach perm [$role getAllDirectlyAssignedPerms] {
      if {[$role ownsPermThroughInheritance $perm]} {
        $role revokePerm $perm
      }
    }
  }
}
```

Figure 14. Remove redundant permission-to-role assignment relations

The `deriveCCFromObstacle` method (cf. Figure 15) receives a mandatory parameter `obstacle` which identifies the corresponding `Obstacle` object. In XORET each `Obstacle` and each `Objective` object possess an instance variable to store the abstract conditions that are associated with the respective `Obstacle` (or `Objective`). In essence, this instance variable is a Tcl list, and each list entry represents an abstract condition (see also [13]). For each abstract condition the `deriveCCFromObstacle` method creates a corresponding `Condition` object and establishes a [*derived-from Obstacle*] trace-link, if necessary.

The `deriveCCFromObstacle` method then creates a `ContextConstraint` object (see Figure 15). The name of the `ContextConstraint` object is determined from the type and the name of the corresponding obstacle (an example of a respective `ContextConstraint` name is "Avoid_Access_Before_2006_01_01"). In the next step, the `Condition` objects are linked to the (new) `ContextConstraint` (see Figure 15). Note that the aforementioned procedure specifies exactly one `ContextConstraint` object which is linked to all `Condition` objects derived from the corresponding obstacle. Afterwards, the `getAllPermissionsInfluencedByObstacle` method uses the trace-links between XORET objects to detect all permissions that are influenced by the respective obstacle (see Figure 16). Finally, the (new) `ContextConstraint` is linked to each of these permissions (see Figure 15).

To find all permissions that are influenced by a certain obstacle, the `getAllPermissionsInfluencedByObstacle` method first determines all `Objective` objects that are linked to the corresponding `Obstacle` object via an [*impedes Objective*] trace-link (cf. Figure 16). Then, the `getAllPermissionsPotentiallyNeededForObjective` method is called to determine the permissions that are needed to perform the scenarios/processes which are associated with the respective objectives. In particular, the `getAllPermissionsNeededForScenario` method traverses all `Step` objects linked to a `Scenario` and assembles a list of the `Permisssion` objects linked to these steps via an [*origin-of Permission*] trace-link (see Figure 16). To assure that all permissions needed for a particular scenario are found, the `getAllPermissionsNeededForScenario` method recursively

```
xoRETManager instproc deriveCCFromObstacle {obstacle} {
  set obstacle [Obstacle lookup 0 [self] $obstacle]
  set conditions [$obstacle getAbstractConditionList]
  if {$conditions != ""} {
    foreach cond $conditions {
      set condition [Condition lookup 1 [self] $cond]
      # establish trace-link if necessary
      if {![$condition existTrace derived-from Obstacle $obstacle]} {
         [self] addTraceRelation \
               Condition [$condition name] \
               derived-from Obstacle [$obstacle name]
      }
      lappend clist $condition
    }
    set constraint [ContextConstraint lookup 1 [self] \
                   "[$obstacle type] [$obstacle name]"]
    # establish trace-link if necessary
    if {![$constraint existTrace derived-from Obstacle $obstacle]} {
       [self] addTraceRelation \
              ContextConstraint [$constraint name] \
              derived-from Obstacle [$obstacle name]
    }
    # link conditions to constraint if necessary
    foreach cond $clist {
      if {![$constraint hasCondition $cond]} {
         [self] linkConditionToContextConstraint [$cond name] [$constraint name]
      }
    }
    # link constraint to permissions if necessary
    foreach perm [my getAllPermissionsInfluencedByObstacle $obstacle] {
      if {![$perm hasContextConstraint $constraint]} {
         [self] linkContextConstraintToPerm [$constraint name] [$perm name]
      }
    }
  }
}
```

Figure 15. Derive `ContextConstraint` objects

seeks through the scenarios that are connected via a [*refined-by Scenario*] trace. A similar procedure is performed in the `getAllPermissionsPotentiallyNeededForObjective` method which recursively visits all `Objective` objects connected via a [*sub-goal Objective*] trace (see Figure 16).

```
xoRETManager instproc getAllPermissionsInfluencedByObstacle {obstacle} {
  set perms ""
  foreach obj [$obstacle getAllTraceTargets impedes Objective] {
    set perms [concat $perms \
              [my getAllPermissionsPotentiallyNeededForObjective $obj]]
  }
  return [lsort -dictionary -unique $perms]
}
xoRETManager instproc getAllPermissionsPotentiallyNeededForObjective {objective} {
  set perms ""
  foreach scenario [$objective getAllTraceTargets fulfilled-by Scenario] {
    set perms [concat $perms [my getAllPermissionsNeededForScenario $scenario]]
  }
  foreach obj [$objective getAllTraceTargets sub-goal Objective] {
    # recursively seek in sub-goals
    set perms [concat $perms \
              [my getAllPermissionsPotentiallyNeededForObjective $obj]]
  }
  return [lsort -dictionary -unique $perms]
}
xoRETManager instproc getAllPermissionsNeededForScenario {scenario} {
  set perms ""
  foreach step [$scenario getSteps] {
    set perms [concat $perms [$step getAllTraceTargets origin-of Permission]]
  }
  foreach rs [$scenario getAllTraceTargets refined-by Scenario] {
    # recursively seek in refinement scenarios
    set perms [concat $perms [my getAllPermissionsNeededForScenario $rs]]
  }
  return [lsort -dictionary -unique $perms]
}
```

Figure 16. Obstacles influence Permissions

The RBAC policy rule set that results from the derivation procedures described in this section serves as a basis for the specification of a refined and adapted policy rule set that can be applied to the corresponding information system. Further refinement procedures may include the definition of additional roles, the deletion of redundant roles, or the specification of additional context constraints for example. Moreover, the scenario-driven approach inherently supports change management activities. Thereby, it supports the management of RBAC policies and allows to tailor the policy rule set to changes on each relevant abstraction level (see also [8, 12]).

7 Related Work

In [10], Osborn et al. present the Role Graph Tool which supports the definition and administration of general RBAC role-hierarchies (directed acyclic graphs). The paper describes how they applied the tool to simulate RBAC policies in ACL-based Unix systems.

Kern et al. describe a commercial software tool (SAM Jupiter) that supports an RBAC variant called enterprise role-based access control model (ERBAC) [6]. An "enterprise role" is a role which subsumes access rights of several underlying software systems. For example, an enterprise role could provide simultaneous membership in a certain Unix group, in a database-specific role (e.g. of DB2 or Oracle), and in a group defined in a groupware system. A subject assigned to an enterprise role may thus receive several accounts in different software systems simultaneously. Thereby, enterprise roles allow for a centralized management of authorizations for resources located in several heterogeneous systems. Analogous features can also be integrated with existing standalone RBAC services, as XORBAC for example. For such an integration we especially need an implementation level interface to map ERBAC-level assignments of system-specific permissions or grouping mechanisms to the corresponding target systems. Kern et al. especially describe an administrative concept for ERBAC.

Another software tool supporting an ERBAC model is the Role Control Center (RCC) presented by Ferraiolo et al. [3]. The RCC tool supports general role hierarchies, static separation of duty constraints, and functions for permission review. It provides a feature called "role graph navigation". The corresponding navigation scheme demands that, at any time, one of the graph nodes is selected as "anchor", RCC then displays the n-tiered upward and downward projections of this anchor. With respect to role-hierarchies, Ferraiolo et al. distinguish "containment" and "inheritance" relations between roles. The inheritance relation defines that a senior-role inherits permissions from its junior-role(s), while containment refers to membership, i.e. all users of the senior-role are also users of the corresponding junior-role(s).

8 Conclusion

The scenario-driven role engineering process (see [8, 13]) supports the definition of a policy rule set for RBAC. In the course of the process, a permission catalog, a constraint catalog, and definitions of work profiles are produced. These models/documents serve as the foundation for the definition of a tailored policy rule set which consists of permissions, roles, and constraints. The XORET tool presented in this paper, was explicitly developed to provide tool support for the scenario-driven role-engineering process. To support change management activities, XORET facilitates the specification and inspection of trace-relations. Furthermore, XORET is capable to produce a (preliminary) policy rule set that can be further refined and directly imported by the XORBAC [7, 13] component.

With our work we aim to provide an integrated and tool-supported methodology to engineer, enforce, and maintain role-based access control policies. So far we described a systematic role engineering approach (see [8]), and an approach to model access control relevant context information and to use requirements engineering techniques to engineer (and enforce) context-dependent access control policies based on context constraints (see [13]). Moreover, we implemented the XORBAC access control service that can be used to enforce RBAC policies including context constraints (see [7, 13]). The XORET tool presented in this paper now provides tool support for the engineering and maintenance of role-based access control policies including context constraints.

References

[1] J. Carroll, editor. *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons, 1995.

[2] E. Coyne. Role engineering. In *Proc. of the ACM Workshop on Role-Based Access Control*, November 1995.

[3] D. Ferraiolo, G. Ahn, R. Chandramouli, and S. Gavrila. The Role Control Center: Features and Case Studies. In *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2003.

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

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] A. Kern, A. Schaad, and J. Moffett. An Administration Concept for the Enterprise Role-Based Access Control Model. In *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2003.

[7] G. Neumann and M. Strembeck. Design and Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language. In *Proc. of the 8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.

[8] G. Neumann and M. Strembeck. A Scenario-driven Role Engineering Process for Functional RBAC Roles. In *Proc. of 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2002.

[9] G. Neumann and U. Zdun. XOTcl, an Object-Oriented Scripting Language. In *Proc. of Tcl2k: 7th USENIX Tcl/Tk Conference*, February 2000.

[10] S. Osborn, Y. Han, and J. Liu. A Methodology for Managing Roles in Legacy Systems. In *Proc. of ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2003.

[11] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[12] M. Strembeck. Embedding Policy Rules for Software-Based Systems in a Requirements Context. In *Proc. of the IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, June 2005.

[13] M. Strembeck and G. Neumann. An Integrated Approach to Engineer and Enforce Context Constraints in RBAC Environments. *ACM Transactions on Information and System Security (TISSEC)*, 7(3), August 2004.

[14] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proc. of the 5th IEEE International Symposium on Requirements Engineering (RE)*, August 2001.

[15] XOTcl Homepage. http://www.xotcl.org, 2005.