An Approach to Engineer and Enforce Context Constraints in an RBAC Environment

Gustaf Neumann
gustaf.neumann@wu-wien.ac.at

Mark Strembeck
mark.strembeck@wu-wien.ac.at

Department of Information Systems, New Media Lab
Vienna University of Economics and BA, Austria

ABSTRACT

This paper presents an approach that uses special purpose RBAC constraints to base certain access control decisions on context information. In our approach a *context constraint* is defined as a dynamic RBAC constraint that checks the actual values of one or more contextual attributes for predefined conditions. If these conditions are satisfied, the corresponding access request can be permitted. Accordingly, a *conditional permission* is an RBAC permission which is constrained by one or more context constraints. We present an engineering process for context constraints, that is based on goal-oriented requirements engineering techniques, and describe how we extended the design and implementation of an existing RBAC service to enable the enforcement of context constraints. With our approach we aim to preserve the advantages of RBAC, and offer an additional means for the definition and enforcement of fine-grained context-dependent access control policies.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications - Elicitation methods, Methodologies; D.2.9 [**Software Engineering**]: Management - Life cycle, Software process models; D.4.6 [**Operating Systems**]: Security and Protection - Access controls; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection - Unauthorized access

General Terms

Security, Design, Management

1. INTRODUCTION

The evolution of software and hardware technologies for interactive networked applications is progressing at a high pace. This poses high demands on access control services that are deployed in interconnected and interactive environments. In particular, such services often need to consider context information to enforce fine-grained access control policies, that rely on information like time, location, process-state, or access history, for instance. Therefore permissions and permission assignment often depend on such context information. One possibility to deal with a dynamically changing context is to rapidly modify permission assignment relations according to the changes in the environment. An other possibility is to define conditional permissions, i.e. permissions that consider certain context conditions in access control decisions, and thus are context-aware to a certain degree. Either way it is sensible to adapt existing access control models and technologies in order to meet the needs of networked interactive applications, as offered by web-based services and pervasive computing devices for example. Thus we think that an access control mechanism with context constraints should be based on well known models and techniques, and should offer a path from "traditional" to context-dependent access control policies.

RBAC [15, 32] provides an access control model that enables the enforcement of many different access control policies. A central idea is to support constraints on almost all parts of an RBAC model (e.g. permissions, roles, or assignment relations) to achieve much higher flexibility. Static and dynamic separation of duties are two of the most common types of RBAC constraints (see e.g. [2]). However, as mentioned above, it is often required to consider various context information in authorization decisions, especially in highly interconnected and interactive environments. Moreover in many real-world applications it is necessary to enforce fine-grained policies where permissions are directly assigned to certain individuals (see e.g. [1, 17]).

In this paper we propose a process for the engineering of context constraints. This process is designed as an extension to the scenario-driven role engineering process (see [26]). Moreover, we describe how we extended the design and implementation of the xoRBAC component (cf. [25]) to enable the enforcement of context constraints. With this extension xoRBAC provides an access control service that preserves the advantages of role-based access control (see e.g. [32]), and allows for the definition of "traditional" RBAC policies. Additionally it adds further flexibility through the specification of fine-grained context-dependent access control policies via context constraints.

1.1 Motivation

In recent years software-based appliances and applications rapidly evolved from relatively isolated standalone computer workstations to interconnected and highly flexible devices that are used in almost any part of human life. Together

SACMAT'03, June 2–3, 2003, Como, Italy.

with the widespread deployment of the respective technologies corresponding security requirements arose to protect sensitive services and information objects that are (potentially) accessed by many interacting users and/or machines. One important demand in this respect is the enforcement of customized context-based access control policies.

Some motivating examples of applications that inevitably need to consider context information in authorization decisions are sketched below:

- In the area of computer supported cooperative work (CSCW) such as workflow management, or groupware applications, context, may, for example, consist of the interacting persons, the processed documents, the daytime, the logical and/or physical location of a person, etc. Researchers have already investigated related access control (and other security) issues for more than two decades and achieved many improvements (see e.g. [7, 9, 17]). However, even this relatively well known area still offers a rich field for further research.

- Mobile code applications range from comparatively simple downloaded Java applets to proactive mobile agents that gather information from distributed sources, and/or autonomously travel in a computer network and react on certain events. The protection of host computers from malicious mobile applications, as well as the protection of mobile applications from malicious hosts, results in many access control related problems where context information needs to be considered, for example the owner of an agent, the owner of a host, the access/travel history of an agent etc. (see e.g. [14, 20]).

- Another example is the purchase of digital goods over the internet, e.g. downloading research articles from digital libraries, purchasing music files directly from an artist, or subscribing to a video streaming channel. Digital goods may pass into the possession of the respective customer, where the owner may use the corresponding products as often, or as long, as she likes to. However, one may also sell only a restricted number of uses, or limit the authorized users to some explicitly named individuals. This and other context information may be captured in special digital contracts for instance. Although some recent contributions describe sophisticated approaches for specific sub-domains (e.g. [1]) the whole field is still young and a large number of open research questions remains.

- Hardware technologies for wireless communications as Bluetooth, Wireless LAN (IEEE 802.11), or mobile phone related technologies distribute quickly. Moreover, middleware standards such as CORBA or the simple object access protocol (SOAP), and software technologies for dynamic service lookup and ad hoc networking like Jini, universal plug and play (UPNP), or E-Speak evolve (see e.g. [24]). With these technologies the vision of ubiquitous and pervasive computing [39, 40] is about to become reality. They enable the realization of novel applications based on mobile devices (see e.g. [33]). For example customized location-based services, distance monitoring of medical parameters, direct and ad hoc interactions through mobile devices, or an intelligent/aware home that responds to particular events and actively controls the access to certain services (see e.g. [12]).

The impressive technical opportunities yet give also an enormous rise to complexity of information security in general, and of access control in particular. For example, publicly offered services (commercial as well as non-profit) must be protected so that only authorized users may access specific resources. Furthermore, user-related information needs to be protected from illegal accesses, no matter if the respective information is stored on a user's mobile device, through an intelligent home environment, or by a publicly available service (such as connection or movement logs of cell phones). Among other things the fulfillment of these requirements is certainly essential to protect the privacy of users in a pervasive computing environment.

1.2 Different Categories of RBAC Constraints

In principle RBAC supports the definition of arbitrary constraints on the different parts of an RBAC model (cf. [32]). However, at first research efforts concerning RBAC constraints focused primarily on separation of duties constraints. With the increasing interest in RBAC in general and constraint-based RBAC in particular, research pertaining to other types of RBAC constraints also gained in importance (see e.g. [8, 19]). In this paper we especially deal with context constraints in an RBAC environment. Subsequently we describe some dimensions for the categorization of RBAC constraints, that are relevant for the purposes of this paper. Then we use these dimensions to explain our definition of context constraints. At first we differentiate between static and dynamic constraints:

- *Static constraints* refer to constraints that can be evaluated directly at design time of an RBAC model (e.g. static separation of duties).

- *Dynamic constraints* can only be checked at runtime according to the actual values of specific attributes, or with respect to characteristics of the current session (e.g. dynamic separation of duties, or time constraints)

Another criterion to classify RBAC constraints is the distinction of endogenous and exogenous factors:

- *Endogenous constraints* are constraints that relate to intrinsic properties of an RBAC model, and inherently affect the structure and construction of a concrete instance of an RBAC model. For example, a static separation of duties (SSD) constraint on two mutual exclusive permissions prohibits an assignment of these permissions to the same role. Moreover, it also influences the definition of the respective role-hierarchy since it further prohibits that two distinct roles to which these permissions are assigned can have a common senior role. Otherwise a common senior could acquire both (mutual exclusive) permissions and thereby violate the corresponding SSD constraint. Similar effects can be observed for cardinality constraints for instance.

- *Exogenous constraints* are constraints that apply to attributes that do not belong to the core elements of an RBAC model, but are defined as "side conditions"

for certain operations or decisions of an access control service. An example can be time constraints that restrict role activation to a specific time interval, or allow access operations for a particular resource only on a specific weekday.

Beside the categorization as static/dynamic, and endogenous/exogenous, constraints can also be subdivided in authorization constraints and assignment constraints:

- *Authorization constraints* are constraints that place additional controls on access control decisions. Thus, even if a subject is in possession of a permission that grants a certain access request, the access can only be allowed if the corresponding authorization constraints are fulfilled at the same time. For example, such constraints can be applied to implement access control policies based on access histories, as in chinese wall policies for instance.
- *Assignment constraints* are constraints that control the assignment of permissions and roles (e.g. maximum and minimum cardinalities, or separation of duty constraints). On the source code level assignment constraints may be implemented by using the same means as applied for authorization constraints (e.g. as an authorization constraint on the "assign role" permission). We think, however, that it is sensible to discriminate assignment and authorization constraints on the design level since both types address distinguishable intentions when engineering an RBAC policy.

The above categories are not completely orthogonal, and do not claim to provide a complete classification framework for all possible types of RBAC constraints. Nevertheless, these categories consider different aspects that can be observed individually, and facilitate the communication about RBAC constraints.

The remainder of this paper is structured as follows. In Section 2 we introduce the notion of *context constraints* as used in this paper. Subsequently we describe an engineering process for the elicitation and specification of context constraints on the requirements level (Section 3). In Section 4 we then describe the conceptual structure of the xoRBAC component that can be implemented using any suitable programming environment. Especially we describe how context information, which is captured by special xoRBAC context functions, can be used to define context constraints. Afterwards we show how we used specific object-oriented techniques to implement a respective extension to xoRBAC in Section 5, before we discuss related work in Section 6. Section 7 concludes the paper.

2. CONTEXT CONSTRAINTS

In the first place a *context constraint* is an abstract concept on the modeling level (like other types of constraints, or the role concept are). A context constraint specifies that certain context attributes must meet certain conditions in order to permit a specific operation. With respect to the categories mentioned in Section 1.2 we thus define context constraints as *dynamic exogenous authorization* constraints. While context constraints can (in principle) also be applied as assignment constraints (cf. Section 1.2), our hitherto experiences concerning the modeling and enforcement of context constraints are primarily based on the usage of context constraints as dynamic exogenous authorization constraints. As authorization decisions are based on the permissions a particular subject/role possesses, context constraints are associated with RBAC permissions (see Figure 1).

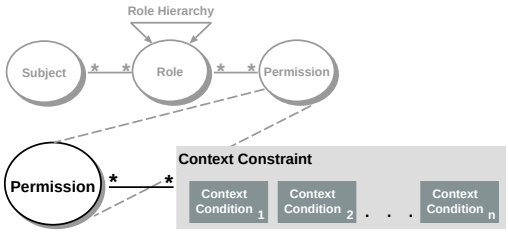

Figure 1: RBAC permission with context constraint

A context constraint is defined through the terms context attribute, context function, and context condition:

- A *context attribute* represents a certain property of the environment whose actual value might change dynamically (like time, date, or session-data for example), or which varies for different instances of the same abstract entity (e.g. location, ownership, birthday, or nationality). Thus, context attributes are a means to make (exogenous) context information explicit. On the programming level each context attribute CA represents a variable that is associated with a $domain_{CA}$ which determines the type and range of values this attribute may take (e.g. date, real, integer, string).
- A *context function* is a mechanism to obtain the current value of a specific context attribute (i.e. to explicitly capture context information). For example, a function $date()$ could be defined to return the current date. Of course a context function can also receive one or more input parameters. For example, a function $age(subject)$ may take the subject name out of the $\langle subject, operation, object\rangle$ triple to acquire the age of the subject which initiated the current access request, e.g. the age can be read from some database.
- A *context condition* is a predicate (a Boolean function) that compares the current value of a context attribute either with a predefined constant, or another context attribute of the same domain. The corresponding comparison operator must be an operator that is defined for the respective domain. All variables must be ground before evaluation. Therefore each context attribute is replaced with a constant value by using the according context function prior to the evaluation of the respective condition. Examples for context conditions can be $cond_1 : date() \leq$ "2003/01/01", $cond_2 : date() == birthday(subject)$, or $cond_3 : age(subject) > 21$.
- A *context constraint* is a clause containing one or more context conditions. It is satisfied iff all its context conditions hold. Otherwise it returns false.

Context constraints are used to define conditional permissions. With respect to the terms defined above a *conditional permission* is a permission that is associated with one or more context constraints, and grants access if and only

if (iff) each corresponding context constraint evaluates to "true". Therefore conditional permissions grant an access operation iff the actual values of the context attributes captured from the environment fulfill the attached context constraints. The relation between context constraints and permissions is a many-to-many relation (see Figure 1). Thereby a number of permissions can be associated with the same context constraint if necessary.

```
% Definition of the access control function
check_access(Subject,Operation,Object) :-
   assigned_role(Subject,Role),
   has_permission(Role,Operation,Object),
   check_context_constraint(Subject,Operation,Object).

% Permission checking for roles and role hierarchies
has_permission(Role,Operation,Object) :-
   assigned_permission(Role,Operation,Object),
   permission(Operation,Object).
has_permission(Role,Operation,Object) :-
   super_role(Role,Super),
   has_permission(Super,Operation,Object).

% Evaluation of context constraints
check_context_constraint(Subject,Operation,Object) :-
   associated_cc(Operation,Object,CC),
   not violated(CC,Subject,Operation,Object).
```

Figure 2: Excerpt from a Datalog specification

Figure 2 shows an excerpt from a (stratified) Datalog specification [4] for role-based access control decisions in the presence of context constraints. The `check_access` predicate examines if an access request identified by the classical $\langle subject, operation, object \rangle$ triple can be granted or must be denied. The `assigned_role` and `has_permission` predicates detect the roles and permissions the subject possesses. The `check_context_constraint` predicate determines the context constraints associated with a specific permission (an $\langle operation, object \rangle$ pair), and subsequently checks that none of these constraints is violated.

Endogenous constraints, as separation of duties constraints, or cardinalities for example, can often be derived from the business rules of a particular organization, e.g. constraints like: the roles "accounting clerk" and "controller" must be statically mutual exclusive, or the minimum user cardinality for the role "controller" is one. In contrast to that it is, in our experiences, more complicated to specify exogenous (context) constraints. In Section 3 we therefore propose an engineering process for context constraints.

3. ELICITATION AND SPECIFICATION OF CONTEXT CONSTRAINTS

Context is an elusive concept which has many different meanings to different people and communities. A definition for the meaning of *context* found in *Merriam-Webster's Collegiate Dictionary* is: "(1) the parts of a discourse that surround a word or passage and can throw light on its meaning (2) the interrelated conditions in which something exists or occurs."

In the area of *ubiquitous and pervasive computing* context is defined as: "... any information that can be used to characterize the situations of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." (cf. [13]).

In other words: context in general may consist of almost every available information that describes a specific situation. That is, context on the one hand consists of relatively static environment characteristics like, a person's nationality, affiliation to an organization, or the salary of a certain employee. On the other hand context also includes dynamic and often changing attributes like time, the location of a person or a device (physical and logical), proximity of other devices or proximity of a specific human being, history information stored in a log-file or database, the current CPU or network load, memory consumption of a specific device and so on.

With respect to access control one has to ask first which parts of these unmanageable quantities of context information are relevant for a specific authorization decision, and how the corresponding information may be elicited and defined on the modeling level. In this section we therefore suggest a process for the elicitation and specification of context constraints. This process is based on goal-oriented requirements engineering techniques (see [3, 36]), and is designed as an extension to the scenario-driven role engineering process for RBAC roles presented in [26]. Prior to describing the engineering of context constraints in detail, we give some background information concerning the scenario-driven role engineering process.

In the scenario-driven role engineering process usage scenarios of an information system are used to derive permissions and to define tasks. In general a *scenario* describes an action and event sequence, for example to register a new patient in a hospital information system. Thus each scenario consists of several steps, and a subject performing a scenario must own all permissions that are needed to complete the different steps of this scenario. In turn a *task* consists of one or more scenarios, and tasks are combined to form work profiles. A *work profile* comprises all tasks that a certain type of subject is allowed to perform. In a hospital environment different work profiles for physicians, nurses, and clerks are needed for instance. In the role engineering process work profiles are then used together with the permission catalog and the constraint catalog to define a concrete RBAC model. However, the scenario-driven approach presented in [26] only provides general guidance for the sub-process of defining (exogenous) constraints. This fact and our aim to specify and enforce context constraints in an RBAC environment led us to the definition of the process extension proposed in this section.

3.1 Description of the Engineering Process

Figure 3 depicts an activity diagram for the engineering (sub-)process. Like the role engineering process as a whole, the engineering of context constraints is in essence a requirements engineering process. To elicit context constraints we especially use goals which are a familiar concept in the area of requirements engineering (see e.g. [3, 36]). Furthermore, goals are well-suited to be applied in combination with scenarios in order to elicit and define requirements, and to drive a requirements engineering process (see e.g. [22, 31]). In general, a *goal* is an objective that the system under consideration should or must achieve. Goals can be defined on different levels of abstraction, ranging from high-level business goals to low-level technical concerns. Furthermore, goals may be used to represent functional as well as non-functional aspects, like performance for instance. An *obsta-*

cle is an undesired condition which obstructs the fulfillment of one or more goals. Thus, obstacles can be seen as the opposite of goals. In the area of requirements engineering obstacles are a valuable means to define more complete and more realistic requirements (see e.g. [37]).

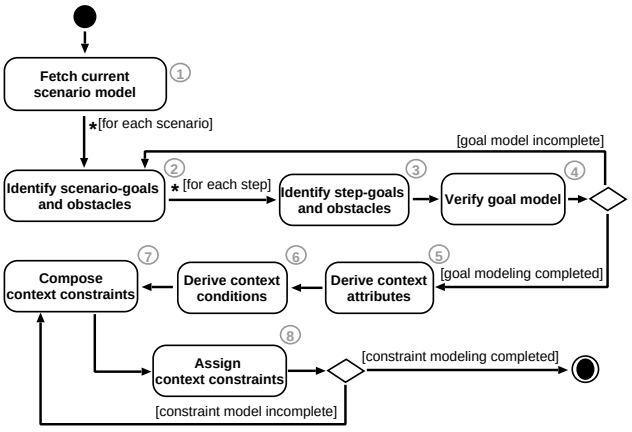

Figure 3: A process for the elicitation and specification of context constraints

Scenarios and the scenario model serve as the basis for the scenario-driven role engineering process [26]. The first step of the constraint engineering sub-process shown in Figure 3 is thus to *fetch the current scenario model*. The succeeding activities are now described in more detail:

- *Identify scenario-goals and obstacles*: In this activity the goal(s) and obstacle(s) associated with each scenario are identified and explicitly modeled by filling out a small goal-template (obstacle-template), which consists of attributes like name, sub-goal-of, super-goal-of, and associated-with-scenario.

- *Identify step-goals and obstacles*: For each step within a scenario the associated goal(s) and obstacle(s) are identified and attached to the goal model. Each step-goal is a natural sub-goal of the corresponding scenario goal(s).

- *Verify goal model*: Here the goal model produced in the preceding steps is verified and further elaborated. This activity is essential for the purpose of defining stable goals which reflect the (security) demands on the system under consideration. To accomplish this task security engineers rely on the assistance of domain experts, for example, a bond dealer, an executive officer, and a clerk for a banking information system. The activities two to four are repeated until the goal model is completed (see Figure 3), i.e. until the security engineers and domain experts define the model as adequate.

- *Derive context attributes*: Each goal (and obstacle) is examined to derive the context attributes that are needed to describe/fulfill this particular goal (e.g. daytime, a user's nationality, or the IP address of the host computer a specific service is requested from). Each context attribute is given a descriptive name, and explicitly stored together with a link to the goal(s) or obstacle(s) it has been derived from. Though it is often possible to straightforwardly derive context attributes and context conditions from goals (obstacles) in a single step, we model each as an own sub-activity to ensure that it is not omitted. In the further course of the process context attributes are used to decide if the access control service that should be applied is able to enforce context conditions based on a particular context attribute, e.g. time information, or the access history of a particular subject (see Section 4).

- *Derive context conditions*: The goals and obstacles are now used to specify context conditions. Each goal and obstacle is a potential source of an access control relevant context condition and is thus analyzed individually. Since obstacles describe what should *not* happen, they are particularly useful in the derivation of context conditions. At this stage of the process (which is still focused on requirements engineering) we make no demands on the way context conditions are specified. For example they can be defined as short sentences like "the IP address of the requesting computing device must have the value x", or "the request can only be granted if the requesting subject has already finished the processing of document b". However, the examples above can also be defined in a much shorter form like "IP address = x", and "access-history = document b" for instance. Each context condition is then stored together with a link to the originating goal/obstacle and scenario. Moreover each context condition is classified if it can be enforced by the corresponding access control service or not, i.e. if it can be mapped to the functions offered by a concrete access control service (see Section 4). In our experiences good reasons exist to model context conditions (and context constraints) even if they can not (yet) be enforced on a technical level. The aim to specify and maintain a comprehensive, and preferably complete, access control policy for an information system is perhaps the most important reason. Such a "complete" policy provides a valuable source of information for the corresponding security engineers. For example, it is then possible to identify which subset of an organization's access control policy can (already) be enforced by the runtime system, and which security goals can not be achieved yet. These data can be applied to thoroughly configure the respective access control service, and to avoid security breaches that could result from unavailable information. Furthermore, a "complete" description of an access control policy on the requirements level can drive the technical evolution of access control services to close the gap between an abstract (complete) access control policy and its enforceable subset.

- *Compose context constraints*: In this activity previously defined context conditions are composed to form context constraints. Each context constraint comprises one or more context conditions. A context constraint that consists of two or more context conditions thereby defines that all of the included context conditions must hold simultaneously in order to fulfill this particular constraint. The context constraints are stored in a constraint catalog.

- *Assign context constraints*: Each constraint can be traced back to the context condition(s), goal(s)/obstacle(s), and scenario(s) it originates from. Since we derive permissions from scenarios and compose work profiles of tasks/scenarios (cf. [26]), we can identify the permission(s) a context constraint could sensibly be assigned to in a straightforward manner. The activities seven and eight are repeated until the constraint model is completed (see Figure 3), i.e. until the security engineers define the model as adequate.

3.2 A small Example

In this section we give an example for the engineering of context constraints as described in the previous section. Since a detailed case study would fill its own paper we chose a simplified example that, however, provides additional insights into the process, and allows for an intuitive understanding of the corresponding activities.

Figure 4 shows a scenario for online examinations, depicted as message sequence chart. For example, online examinations are sensible for tests where students have to show their ability to use certain software tools (e.g. a programming language compiler, or a CASE tool), or for tests that should be analyzed (semi)automatically in a subsequent step.

In our example a student first sends a "fetch" request for an exam document together with her matriculation number to the exam-server (for the sake of simplicity we assume that a proper authentication procedure already took place, e.g. by using a Kerberos based mechanism). The exam-server then generates an individualized scrambling of the exercises to counteract cheating, or cooperation attempts of students (we presume, that all students take the exam within an invigilated PC pool). Afterwards the exam document is dispatched to the client. Following the student edits the exam document, and finally dispatches the completed exam document back to the server.

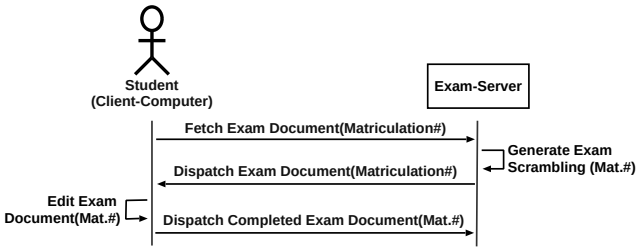

Figure 4: A simple scenario for online examinations

By applying the permission derivation procedure described in [26] we can identify the following student permissions as ⟨operation, object⟩ pairs: ⟨fetch exam⟩, ⟨edit exam⟩, ⟨dispatch exam⟩. In other words: a student (resp. the student role) needs to be equipped with these permissions in order to successfully perform the scenario shown in Figure 4.

Subsequently we conduct the engineering process for the elicitation and definition of context constraints as described in Section 3.1. This results in the following (condensed and simplified) goals and obstacles (we use a leading G for goals and a leading Ob for obstacles):

G_1 Enable online examinations.

- $G_{1.1}$ Provide an individual scrambling for each student.
- $G_{1.2}$ Ensure that students can edit their individual exam only.
- $G_{1.3}$ Ensure that students can fetch and dispatch their individual exam documents.
- $G_{1.4}$ Ensure that only specifically registered PCs can be used to access the exam-server.
- $G_{1.5}$ Ensure that student access to the exam-server is limited to a specific date and a specific time interval.
- $Ob_{1.1}$ Student X can read or write the exam document of student Y.
- $Ob_{1.2}$ The exam-server can be accessed from an unregistered client PC.
- $Ob_{1.3}$ Student X is able to access the exam-server prior to, or after, the specified date and time interval.

According to the process shown in Figure 3 we now derive the context attributes and context conditions from the above goals and obstacles (see Figure 5).

Context Attributes

1) todays_date	5) client_IP_address
2) examination_date	6) registered_IP_address
3) current_time	7) matriculation_number
4) exam_time_interval	8) exam_document_number

Context Conditions

1) todays_date = examination_date

2) current_time in exam_time_interval

3) client_IP_address is-a registered_IP_address

4) matriculation_number = exam_document_number

Figure 5: Context attributes and conditions

Subsequently the context conditions are used to compose context constraints which are then assigned to permissions (see Figure 3). According to the above goals and obstacles we compose three context constraints, and assign them to the permissions derived from the scenario depicted in Figure 4. For the sake of simplicity the context constraints are written as a list of conditions (cf. Figure 5) surrounded by curly brackets:

- ⟨fetch exam⟩ {Cond_1, Cond_2, Cond_3}
- ⟨edit exam⟩ {Cond_2, Cond_3, Cond_4}
- ⟨dispatch exam⟩ {Cond_1, Cond_3, Cond_4}

Note that in the most simple case each context constraint consists of exactly one context condition. Context constraints composed of more than one condition are used to explicitly express the coherence and need for simultaneous validity of several conditions when performing a certain operation i.e. when using a specific permission.

The role and constraint engineering processes result in a concrete RBAC model. The elements of this RBAC model are roles and role-hierarchies, permissions, and (context) constraints (see also [26]). The xoRBAC component [25] provides an RBAC service that (among other things) supports role-hierarchies, static separation of duties, and cardinality constraints for both roles and permissions. Nevertheless, in order to actually enforce RBAC policies which make use of context constraints on a technical level, a respective RBAC service must provide means to map modeling level context constraints on concrete implementation structures. Section 4 and Section 5 describe how we extended the xoRBAC component to enable the definition and enforcement of context constraints.

4. XORBAC: CONCEPTUAL STRUCTURE

Figure 6 depicts the conceptual structure of the xoRBAC component, which is an extension of the structure presented in [25]. xoRBAC is associated with a metadata service that records logging and audit information, and enables the serialization and recreation of xoRBAC runtime instances by using XML encoded RDF models as serialization format. Moreover the xoRBAC component can be bound to arbitrary authentication services (such as Kerberos, or a service based on X.509 certificates for example) and does not demand on a particular authentication mechanism, it simply requires that a means exists to authenticate subjects within the system (for further details see [25]).

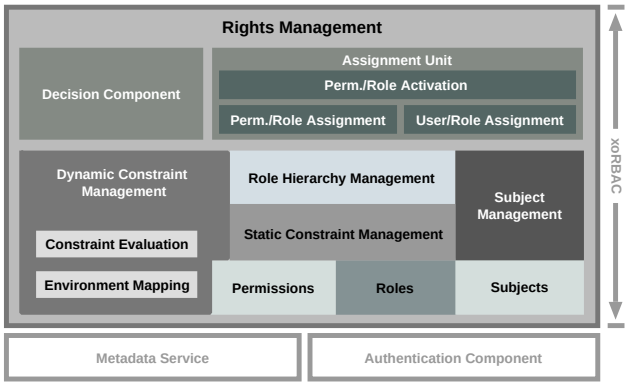

Figure 6: xoRBAC: conceptual structure

The xoRBAC component comprises static and dynamic constraint management as individual sub-systems (see Figure 6). The *dynamic constraint management* sub-system is the most significant increment to [25] with respect to this paper. It comprises the *environment mapping* which captures context information via sensors, and the *constraint evaluation* which checks if the collected values match the context constraints associated with a certain conditional permission. Context constraints can be defined for "ordinary" access permissions as well as for administrative permissions such as assignment or revocation operations for example.

Fundamentally the *environment mapping* component comprises the *sensor library* of the xoRBAC access control service (see Figure 7). It manages all sensors connected to xoRBAC. Therefore every sensor must be registered in the sensor library before it can be used within xoRBAC. Each sensor provides one or more context functions.

A context function is a mechanism to obtain actual values for specific context attributes (i.e. to explicitly capture context information). In other words: context functions are used to filter environment information and to make the current value of a relevant attribute available, so that it can be used by xoRBAC. Therefore each context attribute that can be provided by a respective context function can be used to define xoRBAC context conditions (cf. Figure 7).

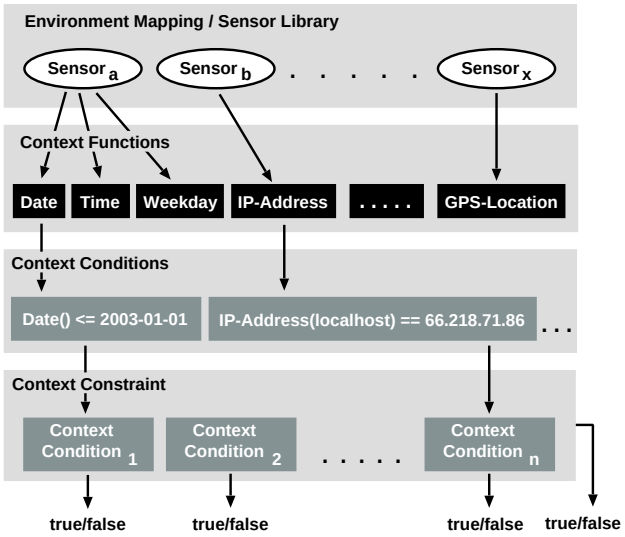

Figure 7: From sensors to context constraints

In general, the return value of an xoRBAC context function may be either a string or a numerical value. Moreover, a context function may return a single value (e.g. the current weekday, daytime, or IP-address), or a vector/list of values (e.g. a list of all users currently logged into the local host, all employees working in a specific project, or all hardware devices within a certain radius). The sensor library of xoRBAC can be extended with arbitrary new sensors, respectively their corresponding software interface. This means that (in principle) xoRBAC can be connected to physical as well as logical sensors to sense context attributes (see Section 5.5).

The *constraint evaluation* component checks if the topical sensor values match the corresponding context constraints, and returns either `true` or `false` depending on the result of the evaluation. In this sense context constraints provide sensor fusion, i.e. they combine and interrelate the measurements of several sensors.

Thus in xoRBAC sensors and context conditions represent two different layers. A sensor (resp. the corresponding context functions) only captures "raw" context information from the environment and makes the respective context attribute available as string or numerical value. In turn, context conditions use the actual values of context attributes to check predefined conditions, and to decide if the corresponding access can be granted.

Figure 8 shows the definition process of concrete xoRBAC context conditions as activity diagram. For each modeling level context condition (see Section 3.1) it is examined if the corresponding (abstract) context attribute(s) can be captured by an actual context function of an xoRBAC sensor. If so, a respective concrete context condition is specified (see Section 5.3). If, however, no appropriate context function is

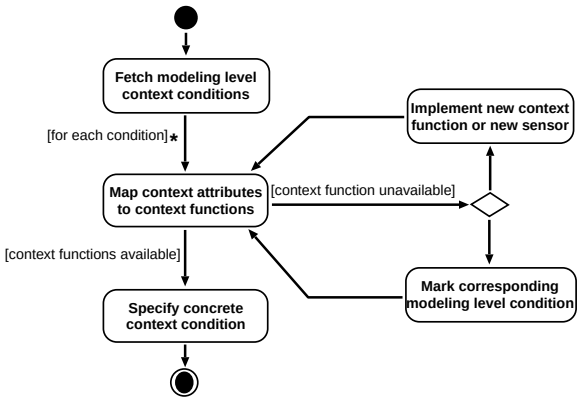

Figure 8: Definition of concrete context conditions

available, one can either implement a new context function or a new sensor in order to enforce the corresponding modeling level condition, or the corresponding modeling level condition is marked as "not yet enforceable". In our experiences good reasons exist to model context conditions (and context constraints) even if they can not (yet) be enforced on a technical level (cf. Section 3.1).

5. XORBAC: IMPLEMENTATION

The xoRBAC access control component is implemented with XOTcl (eXtended Object Tcl) [27]. XOTcl is a general purpose object-oriented programming language that can be dynamically loaded into every Tcl compatible environment and is embeddable in C programs. As a Tcl extension, all Tcl commands [29] are directly accessible in XOTcl. XOTcl preserves the flexibility of Tcl and adds new language constructs to provide a highly flexible OO programming environment.

In the implementation of xoRBAC we especially used the dynamic object aggregation feature and the per-object-mixin language construct of XOTcl. *Dynamic object aggregation* enables the dynamic aggregation and disaggregation of objects at runtime. A *per-object mixin* (POM) is a class which is inserted at the beginning of the precedence order for a particular object. In other words, POMs are inserted in front of the precedence order induced by the class-hierarchy from which the object was instantiated. Thus POMs are a means to extend every single object with additional behavior or capabilities dynamically at runtime (see [25, 27]). However, as already mentioned, the design and abstract architecture of xoRBAC presented in this paper can of course be implemented using other programming languages as well.

Some important features of xoRBAC are: definition of arbitrary role-hierarchies (permission-inheritance), user-role review, user-permission review, permission-role review, definition of static separation of duties constraints, and definition of maximum and minimum cardinalities (for details see [25]). In this section we describe an extension of xoRBAC that enables the definition and enforcement of context constraints on permissions (see also Section 2 and Section 4).

5.1 Static Design-time Structures

Figure 9 depicts the essential design level class relations within the xoRBAC component. Several design patterns (see [16]) are used in the implementation of xoRBAC. For example, the `RightsManager` class serves as *Facade* for the xoRBAC component, i.e. it hides xoRBAC internal structures from other components that use the xoRBAC component. Thus every external component uses the xoRBAC component through a well-defined API offered by the `RightsManager` class. At runtime an `Audit` object can be registered for the `RightsManager` object according to the *Observer* pattern. The user-role assignment and the permission-role assignment relations are implemented using the *Decorator* pattern (for more details see [25]).

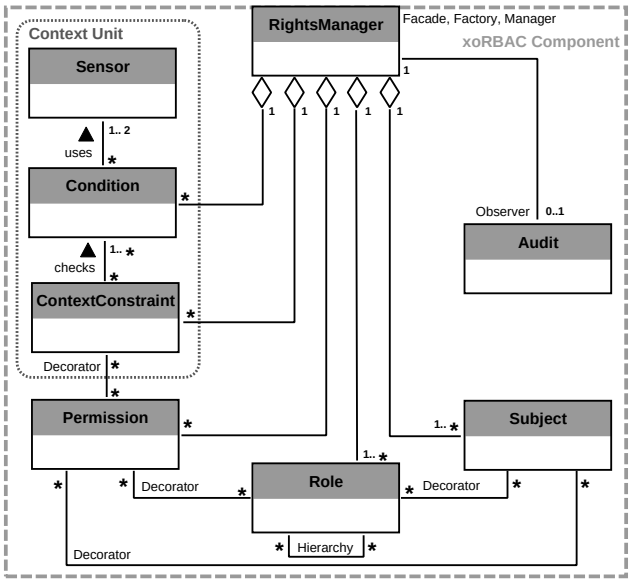

Figure 9: xoRBAC **component: class relations**

As shown in Figure 9 the xoRBAC component basically consists of eight classes. The classes `Sensor`, `Condition`, and `ContextConstraint` form the *Context Unit* of xoRBAC. The context unit extends xoRBAC with functions that allow for the specification and enforcement of context constraints as described in Section 4.

The `ContextConstraint` class is defined as a meta-class, which means that its instances are regular classes (for further information on XOTcl meta-classes see also [27]). Thus, for each conceptual context constraint that was defined during the engineering process (see Section 3) a respective `ContextConstraint` instance is created. At runtime each of these instances checks *exactly one* (modeling level) context constraint. Further on, each actual `ContextConstraint` checks one or more `Condition` objects. And each `Condition` object uses either one or two `Sensor` objects to implement a specific modeling level context condition.

5.2 Dynamic Runtime Structures

In xoRBAC new sensors, conditions and context constraints (i.e. instances of the `Sensor`, `Condition`, and `ContextConstraint` classes) can be dynamically defined. In other words: the sensor-library and the pool of context conditions and constraints can be dynamically extended.

Figure 10 depicts a `RightsManager` object at runtime, it shows the dynamic object aggregation of `Subject`, `Role`, `Permission`, `ContextConstraint`, and `Condition` instances, and their encapsulation within a respective namespace. In xoRBAC POMs are used to assign roles to subjects and

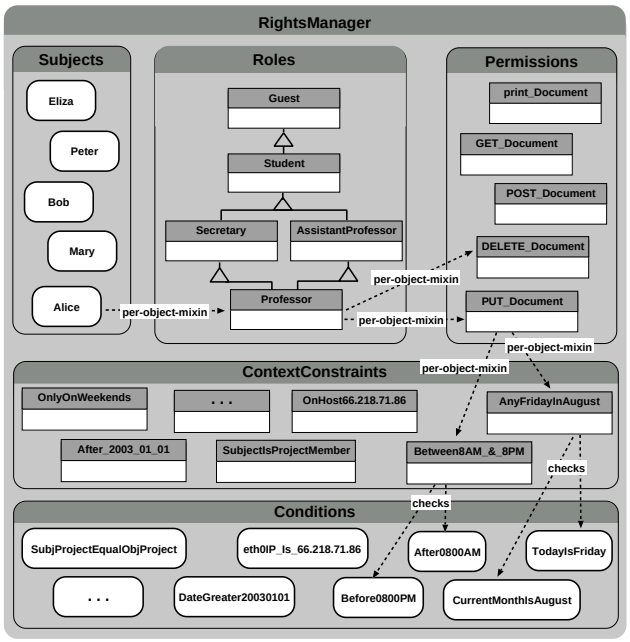

Figure 10: A `RightsManager` object at runtime

permissions to roles [25]. In the same way xoRBAC uses POMs to associate permissions with context constraints. The `ContextConstraint`s are associated with `Permission` objects according to the *Decorator* pattern (see Figure 9). The use of POMs allows to dynamically (de)register `ContextConstraint`s for `Permission` objects at arbitrary times.

```
ContextConstraint instproc notViolated {subj op obj} {
  [self] instvar conditions
  foreach condition $conditions {
    if {![$condition isSatisfied $subj $op $obj]} {
      return 0
    }
  }
  return 1
}
```

Figure 11: `ContextConstraint notViolated` method

For each `ContextConstraint` the `notViolated` method checks if all `Condition` objects that are registered for this particular constraint are satisfied (see Figure 11). Figure 12 depicts two context constraints `constraint1` and `constraint2` which are registered as POMs for `permission1`. Through the next-path each method call to `permission1` is at first directed to its POMs `constraint1` and `constraint2` prior to invoking the respective method in `permission1`. This feature is called method combination or *method chaining* [27] and is a well-known approach to handle dynamic class structures.

In particular Figure 12 shows a simple example of a next-path resulting from a call of the `notViolated` method on `permission1`. The `notViolated` call to `permission1` is passed along the next-path to its POMs `constraint1` and `constraint2` and finally back to `permission1`. Regarding the `notViolated` method, the permissions and context constraints of xoRBAC form a *chain of responsibility* (see [16]). This means: a `notViolated` call is passed along the next-

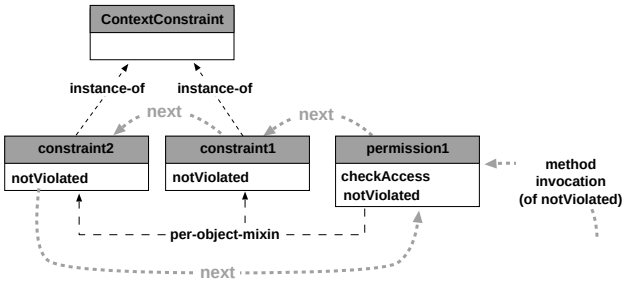

Figure 12: Next-path for the call of `notViolated`

path until a context constraint is violated and denies the request by returning `0` (false). However, if all context constraints return `1` (true) the call is finally passed back to `permission1` which then grants the corresponding access request.

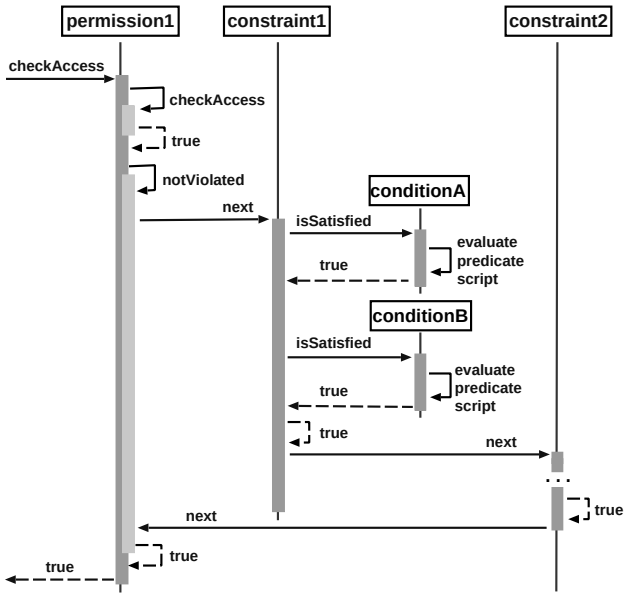

Figure 13: `notViolated` MSC for the return of `true`

Figure 13 shows a message sequence chart (MSC) of a `notViolated` call for the return of `true`. Section 5.4 provides a detailed description for access control decisions with conditional permissions in xoRBAC.

5.3 Specification of Context Constraints

Each `Condition` object implements one particular context condition (see Section 2). In essence, a `Condition` object consists of a *left operand*, an *operator*, and a *right operand* (for binary operators). The left operand is always represented by a particular context function which captures the current value of a specific context attribute. The right operand can be either a constant value, or an other context function. The operator is used to compare the left and right operands. This means, a `Condition` object either compares the results of two context functions, or the result of a context function and a constant value. For this purpose the `isSatisfied` method is called (see Figure 14).

Since each context condition represents a predicate, a con-

```
Condition instproc isSatisfied {subj op obj} {
  [self] instvar predicate_script
  if {[info exists predicate_script]} {
    return [eval $predicate_script]
  } else {
    return 0
  }
}
```

Figure 14: The `Condition isSatisfied` method

crete `Condition` object returns either true or false as result of an `isSatisfied` call (see also Section 2). In particular the `isSatisfied` method checks the predicate script of a specific `Condition` object. This predicate script is stored in the `predicate_script` instance variable of the `Condition` object. Each predicate script represents a piece of XOTcl source code which is automatically generated from the left operand, operator, and right operand instance variables of a `Condition` (see Figure 15). After the predicate script of a `Condition` object is generated, it can be evaluated for arbitrary times to check the corresponding context condition. Nevertheless, if an operand or operator instance variable of a `Condition` object is modified, the corresponding `predicate_script` is adapted accordingly, of course.

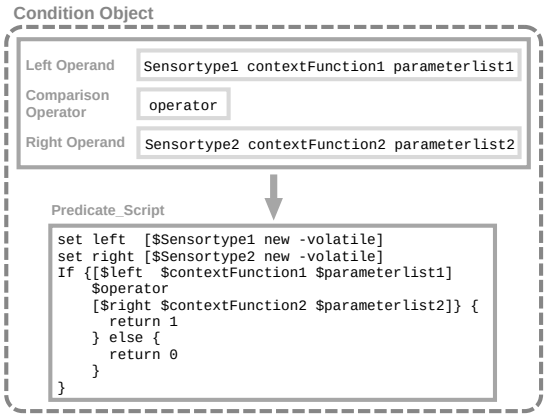

Figure 15: Generation of a predicate script

When a predicate script is evaluated at runtime (the `[eval $predicate_script]` call in Figure 14) it first instantiates the `Sensor` objects needed to check the corresponding condition (the `[$Sensortype -new volatile]` calls in Figure 15). Thus, in xoRBAC every `Condition` object employs its own volatile `Sensor` objects to capture a consistent snapshot of the relevant context attributes (cf. Section 4). Next, the two context functions are executed (all variables must be ground before evaluation - see Section 2), and the respective results are compared using the corresponding comparison operator. If the right operand is a constant value the corresponding predicate script is even more simple. In this case only one `Sensor` object needs to be instantiated, and the result of the respective context function is compared with the corresponding constant value. A predicate script always delivers either a return value of `1` (true), or `0` (false).

Figure 16 shows a simple example of an actual (automatically generated) predicate script. This script uses the `GenericXPathSensor` and the `LocalhostSensor` of xoRBAC (see Section 5.5). The `xPathTextNodeQuery` method is used

```
set gxps [GenericXPathSensor new -volatile]
set lhs  [LocalhostSensor new -volatile]
if {[$gxps xPathTextNodeQuery exam.xml //exam//date]
    ==
    [$lhs clock %Y%m%d]} {
    return 1
  } else {
    return 0
}
```

Figure 16: Example of a predicate script

to read the examination date from the `exam.xml` document. The `clock` method is applied to read the current date in a YYYY MM DD format (parameters/format string `%Y%m%d`). If and only if both of these values are equal (comparison operator `==`) the corresponding context condition is satisfied (return value `1`). In order to actually compare two (date) values in a predicate script these values need to be formatted identically. This, however, can be achieved straightforwardly by adapting the corresponding format string passed to a specific context function (e.g. from `%Y%m%d` to `%d%m%Y` in the example above, to change from a YYYY MM DD format to a DD MM YYYY format).

```
set lhs  [LocalhostSensor new -volatile]
if {[$lhs eth0IPAddress] == "66.218.71.86"} {
    return 1
  } else {
    return 0
}
```

Figure 17: Script with a constant as right operand

Figure 17 depicts an example of a predicate script with a constant value as right operand. Here the `eth0IPAddress` context function reads the IP address which is bound to the eth0 interface of the corresponding local-host. The predicate script returns `1` (true) iff this IP address is equal to "66.218.71.86", and `0` (false) otherwise.

To conveniently manage xoRBAC, we developed a graphical tool that allows for the administration of xoRBAC runtime instances. Moreover, this tool provides support for the scenario-driven role engineering process, and the constraint engineering process presented in Section 3.1. Besides, it controls a number of internal integrity rules, for example the administration tool only allows for the definition of context conditions that either compare two string-values or two numerical values. Furthermore, it is possible to allow only specific comparison operators for a particular context attribute type. For example it could be defined that an IP address may only be compared with an other value through a == operator but not through one the $>, <, \geq, \leq$ operators. xoRBAC can also be controlled "directly" via its API.

5.4 Access Control Decisions

The access control function of xoRBAC is implemented by the `checkAccess` method. This method requires the traditional access control triple $\langle subject, operation, object \rangle$ as input attributes. Thereby xoRBAC provides a clear interface to other components that use xoRBAC as their access control service.

In xoRBAC permissions are always positive, i.e. a permission always grants a certain access right and does not deny it. The processing of "ordinary" access requests is explicitly described in [25]. Therefore the focus in this section is on ac-

cess control decisions with conditional permissions. Figure 18 depicts a message sequence chart of an action and event sequence which occurs if an access request is granted by a context constraint. Here `constraint1` is an instance of the `ContextConstraint` class, `conditionA` and `conditionB` are instances of the `Condition` class, and `sensorX` and `sensorY` are actual `Sensor` objects (see Figure 9).

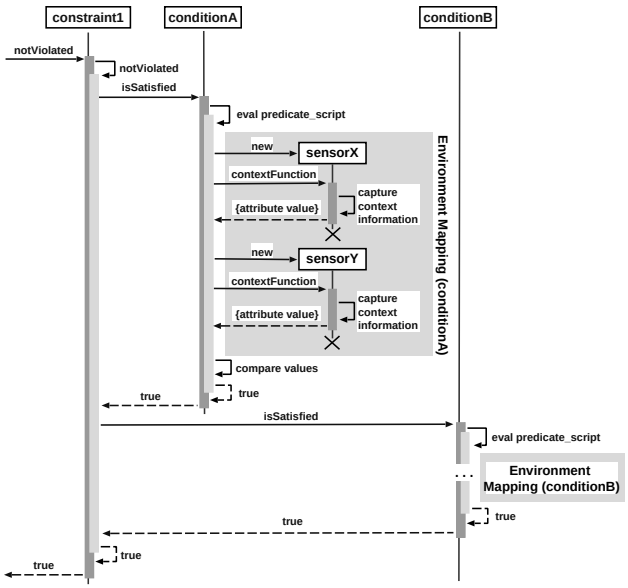

Figure 18: Evaluation of a context constraint

Initially `constraint1` receives a `notViolated` call (see Figure 13 and Figure 18). Next, `constraint1` calls the `isSatisfied` method of all `Condition` objects associated with `constraint1` (see also Figure 11). At first the `isSatisfied` method of `conditionA` is called. Then `conditionA` evaluates its predicate script to decide if this particular context condition is fulfilled or not (see also Figure 15). During evaluation of the predicate script `conditionA` draws a snapshot of the relevant context attributes by using the two (volatile) sensor objects `sensorX` and `sensorY` (note that the call of `contextFunction` in Figure 18 serves as a placeholder for any call of an actual context function - cf. Section 5.3 and Section 5.5). After `conditionA` has drawn a snapshot of the relevant context attributes it compares the obtained values by applying the corresponding comparison operator (cf. Section 5.3), and returns either `true` or `false` according to the result of this comparison. In Figure 18 the `isSatisfied` method of `conditionA` returns `true`. The same procedure is repeated for `conditionB`. The `isSatisfied` method of `conditionB` also results in a return value of `true`. Since all context conditions associated with `constraint1` are satisfied, `constraint1` finally returns `true` as result of the `notViolated` call.

Note that the `notViolated` method of a `ContextConstraint` returns `true` iff each `Condition` object that is associated with this `ContextConstraint` is satisfied. Moreover, for a call of `notViolated` the corresponding `Permission` object is always the last object within the next-path and thus the last object within the chain of responsibility (cf. Section 5.2). This means: if no `ContextConstraint` previously denies the requested access by returning `false` the `notViolated` call is finally passed back to the respective `Permission` object which then returns `true` to indicate that the corresponding access request can be granted (cf. Figure 13).

5.5 Sensor Library

We differentiate sensors in two coarse-grained categories: *physical sensors* (i.e. pieces of hardware) which capture information on a host's physical environment (e.g. current GPS location, temperature, noise-level, light, or proximity of an other device), and *logical sensors* which consist only of software components and are used to gather information that can be extracted from system internal sources (e.g. the IP-address of a certain device, information stored in databases or log-files, the status of other applications or services, CPU ID, CPU state, network load, etc.).

In principle both sensor types can be used to capture access control relevant context information. However, for the time being, we concentrate especially on the use of logical sensors in xoRBAC. The use of logical sensors is sensible for the purpose of access control, since relevant access control related information is often stored using software-based services. For example, information like birthday, nationality, ownership, or physician to patient relations, can be gathered from specific databases or documents, as birth certificates, contracts, passports, or patient records. Therefore (for the time being) it is convenient to read such information directly from the respective electronic sources by using logical sensors, e.g. through a context function that executes a certain database query, or a context function that reads a specific information from XML documents. Likewise, most system internal attributes can be conveniently captured through logical sensors that query the status of a certain software service, or look for specific log-file entries for example.

In general a context attribute that can be captured by a context function can be used as an operand for the specification of xoRBAC context constraints (see Section 4). The different sensors and their context functions thus provide the "operand-vocabulary" of the xoRBAC component. All sensors described in this section are logical sensors. Any sensor in xoRBAC can be extended with additional functions, and new sensors can be defined and registered at runtime. Subsequently we briefly describe the sensors which are currently available from the xoRBAC sensor library.

- The *Localhost Sensor* provides functions that capture context information directly from the device xoRBAC is running on (the "local host"). Currently the `LocalhostSensor` class exports the following context functions:
  - the *clock* context function uses the standard C library function `clock` and can provide several time-related attributes. For example the current local time in 12 or 24-hour format, the current local date, the full weekday name (Monday, Tuesday, etc.), or the full month name (January, February, etc.).
  - the *loggedOnUsers* context function returns a redundance-free list of all users who are currently logged on (i.e. who have an active session on the local host).

  - the *eth0IPAddress* context function returns the IP-address that is bound to the `eth0` interface of the local host.

- The *Database Sensor* is a generic database sensor that uses SQL commands to query relational databases. Currently the sensor can connect to MySQL and PostgreSQL databases. In order to query domain specific context information from a database, like physician-to-patient relations, or project membership for instance, one needs in-depth knowledge of the underlying database schema of course. However, this is true for any approach that reads context information from databases using SQL commands (see e.g. [17, 18]).

- The *XPath Sensor* uses an XPath C-library to access arbitrary XML documents. XPath [10] is a sophisticated compact query syntax developed by the W3 Consortium to easily extract information from XML documents in a automated manner. XPath operates on the logical tree structure of XML documents. It allows to query a single node or a group of nodes at once. The XPath sensor can be used to query arbitrary XML documents independent of the DTD or XML Schema definition for a particular document. This feature, however, stems from the abilities offered by XPath [10], respectively the corresponding XPath implementation.

- The *Flatfile Sensor* uses regular expressions to allow the retrieval of information from text-based files, e.g. access histories stored in log-files.

The xoRBAC component can be reused for applications on Unix or Windows with a C or Tcl linkage (see also [25]). However, some sensors may access platform-specific system functions, for example to read the local-host's IP address. Therefore we install different sensor libraries depending on the platform xoRBAC is used on. Nevertheless the sensor interfaces are (of course) platform independent. This means that two sensors which provide the same function on different platforms offer the same interface but access different implementations. A sensor's interface thus hides the implementation details from the xoRBAC component. Thereby the platform xoRBAC is used on is not relevant when accessing sensor functions.

All sensors of xoRBAC are currently *passive sensors*. That means that the corresponding context functions do not permanently provide xoRBAC with topical context attribute values but are selectively polled to provide a "snapshot" of the context attributes that are needed for a particular authorization decision.

6. RELATED WORK

This section is not intended to give an exhaustive survey of all existing approaches to include context information in authorization decisions. Nevertheless we think it provides a good overview of relevant related contributions.

Adam et al. introduce a sophisticated authorization model that was specifically designed to meet access control requirements of digital libraries [1]. In particular their model allows for the consideration of additional user and object attributes aside from unique identifiers. Here, so called credentials represent attributes that describe certain characteristics and qualifications of users, like age, salary, nationality, or current project involvement for example. Likewise, attributes describing the content of digital library objects are stored (e.g. taxation, civil law, information system research), and digital library objects are divided in different segments, like authors, abstract, sections, bibliography. These information are then used to define fine grained access control policies. That enables the definition of access rights on individual objects (based on their IDs), or on specific parts of a set of objects (like the abstract and author information of all research papers), or on all objects that comprise certain contents, e.g. all objects concerning import taxes. Similarly users may acquire permissions explicitly (through their ID), or implicitly through their characteristics/credentials, e.g. all users with a specific age or nationality.

In [28] Nitsche et al. give a high-level description of a system extension they implemented to consider context information for authorization decisions in medical workflows. Their workflow system is based on a central database which contains patient records as well as information concerning the actual state of medical workflows. Nitsche et al. use the process descriptions, and state information available through the central database to decide whether a particular user may perform a certain action with regard to the actual process-state.

Beside the example above there are various contributions concerning access control in collaborative environments esp. for groupware and workflow systems. For example Thomas and Sandhu introduced TBAC [35], a family of models that support the specification of active security models where permissions are actively (de)activated according to the current task/process-state. In [23] an approach for access control in inter-organizational workflows is suggested, and in [9] Bertino et al. present a well-elaborated language and algorithms to express and enforce constraints to ensure that all tasks within a workflow are performed only by predefined users/roles. A more general language that allows for the specification of different access control policies which can coexist in the same system is presented in [21]. One similarity of all of these approaches is that they allow to use some context information, e.g. the execution history of individuals/roles, or the current process-state, to make assignment, activation, or authorization decisions.

In [17] Georgiadis et al. introduce the Context-based Team Access Control model (C-TMAC) as an extension of the TMAC approach presented by Thomas [34]. Here a team is defined as a group of users acting in different roles with the objective of corporately completing a certain task. Thus in C-TMAC the team concept is used to associate users with contexts, like roles are used to associate users with permissions. Georgiadis et al. give a formal description of C-TMAC and provide an example how their model can be implemented with dynamic SQL statements. They give an example from the health care domain where they use three types of context information: the patient name/ID, the location/area a team works in, and the actual time an access operation takes place. The problem of information sharing and security in dynamic coalitions (see [11, 30]) is related to C-TMAC. A (dynamic) coalition consists of two or more different organizations (resp. their employees) that temporary work together to achieve a common goal. Coalitions can be formed dynamically, and each coalition member may share a number of information resources with other coalition

members. However, each party must be able to individually tailor the access rules for their own content according to the status of other coalition members.

In [38] Wang presents an approach to realize context and role-based access control for a hypermedia environment. He uses three different role categories: roles that represent the job position of a user like "software engineer", team roles to express team membership, and personal roles that are assigned to individuals, but may, in exceptional cases, be assigned to other persons in order to transfer individual job responsibilities. On the other hand, hypermedia objects are organized using so called wrappers. Wrappers serve as containers for several hypermedia objects. Every wrapper represents a particular system/process state and objects may be moved into another wrapper to represent a changing process state. Access control lists can be defined for wrappers as well as for individual objects within a wrapper. In this approach roles are therefore used to reflect structures that remain relatively stable during a project, while ACLs are used to depict dynamically changing access rights on certain objects. Context information is only implicitly included through the notion of team membership, and process states represented by the wrapper a certain object is actually in.

Edjlali et al. present a history-based access control mechanism called Deeds [14]. They propose to utilize the access history of (mobile) Java programs as context information, to protect a host computer from potentially insecure/dangerous operation sequences. For example, a mobile Java program may not do both, open a specific file for reading and open a socket for writing. Each security relevant operation (called a security-event in Deeds) is associated with a handler that maintains an event-history for this operation and decides whether a particular request may be granted according to certain, user defined, constraints.

Jaeger et al. [20] introduce a system architecture for the control of downloaded executable content. The underlying model was built to support both, system and application specific access control policies. Simplified: administrators are able to define system wide mandatory access control policies, while individual users may perform additional discretionary access controls for specific sub-domains. The proposed architecture consists of several services that, among other things, authenticate the content provider and the downloaded executable content. Then the downloaded content is assigned to a so called protection domain to enforce its respective permissions. As context information Jaeger et al. particularly use the identity of the content provider and the content itself, the identity of the downloading principal, and the actual state of the associated application.

In [6] Barkley et al. suggest a model to consider relationships between real-world entities during RBAC access decisions. They propose to apply the resource access decision facility (RAD) defined by the Object Management Group (OMG) to combine the access decisions of two (or more) access control services. In specific they combine an RBAC service and a service that evaluates the relationships between real-world subjects and/or objects. An example from the health care domain is used to describe the approach. If a physician tries to access a certain patient record, the RBAC service decides whether the access can be granted according to the physician's roles/permissions. In addition, a so called dynamic attribute service decides if the access can be granted according to the relationship between the physician and the patient, i.e. if the physician is the attending physician of this particular patient. Both decisions are sent to a so called decision combinator which assembles a final decision by applying a certain combination policy.

Role templates as proposed by Giuri and Iglio [18] can be used to consider certain types of context information when defining roles and permissions. In particular, roles and permissions are parameterized to gain more flexibility compared to treating them as fixed entities. For example, instead of defining an own role-hierarchy for different projects, a single generic hierarchy may be defined. When assigning a certain user to a specific project-role the name of the concrete project or department the user works in is used as a parameter for the assignment operation. Hence, according to the concrete parameter value, a user may only access resources that are allocated to her/his project or department.

Covington et al. describe an approach that uses two different kinds of roles to assign rights to users, and to include context information in an intelligent/aware home environment [12]. They suggest to use classical RBAC roles to provide subjects with permissions. Besides, they introduce the notion of environment roles that are automatically (de)activated by the aware home to depict the actual environmental context. Environment roles are bound to environment conditions that can be captured by the (hardware) sensors within the aware home, like time, room temperature, or location of a user. Environment roles are activated according to these conditions and are used together with subject roles to reach an authorization decision.

The TRBAC model presented by Bertino et al. [8] allows for the periodic (de)activation of roles, and for the definition of temporal dependencies among the events that (de)activate roles. They use so called role triggers to define temporal dependencies between activation events. Role triggers and periodic activations may be associated with a priority to resolve possible conflicts that could result from simultaneous (de)activation requests. Thus, time in general, and time intervals between activation events in specific, are used as context information in TRBAC. Bertino et al. describe a specification language for TRBAC and provide a mathematical proof that the correct use of their language guarantees the absence of ambiguities and inconsistencies.

In [41] Yao et al. describe the support of active security in the OASIS role-based access control architecture. In OASIS role activation is governed by rules that are specified in logic. The corresponding rules may also specify certain preconditions that must be fulfilled in order to activate a particular role. These conditions are bound to events which cause that a role is deactivated as soon as a condition becomes false. Likewise, rules specifying access to objects or services can be bound to conditions/attributes that need to be evaluated each time a rule is applied. The time of day, or a user's membership in a certain group are examples for such environmental attributes. In [5] Bacon et al. present an approach to express OASIS policy rules as pseudo-natural language statements that can be translated into first-order logic with side conditions.

7. CONCLUSION AND FUTURE WORK

This paper introduced a framework for a special kind of RBAC constraints, namely context constraints, which are defined as dynamic exogenous authorization constraints. We specified the required terminology and provided a definition

for context constraints. We defined an elicitation process to derive context constraints during role engineering, and described an implementation that extends an existing RBAC system to enable the enforcement of context constraints.

The presented process for the elicitation and specification of context constraints is based on goal-oriented requirements engineering techniques. This process is designed as an extension to the scenario-driven role engineering process for RBAC roles [26]. The overall process provides guidance for security engineers and allows for the specification of concrete RBAC models including context constraints. Moreover, we implemented a graphical tool that supports the role engineering process in general, and the specification of context constraints in particular. In order to actually enforce the context constraints that are defined on the modeling level, we extended the design and implementation of the xoRBAC component. Thereby xoRBAC provides a flexible RBAC service that preserves the advantages of role-based access control and additionally offers functions for the definition and enforcement of fine-grained context-dependent access control policies. In particular it allows for the definition of conditional permissions.

While it is possible to define (in principle) any kind of context constraint on the modeling level, the enforcement of such constraints is clearly limited to the functionality which is provided by a concrete RBAC service. Currently xoRBAC provides context functions for: time and date information, IP addresses, information on user sessions provided by the operating system, and information stored in flat-files, XML files, or MySQL or PostgreSQL databases. Nevertheless, the xoRBAC sensor-library can be extended with additional sensors, and each context attribute that can be captured by an xoRBAC sensor/context function can be used in the definition of context constraints.

Our graphical role engineering and administration tool is implemented with Tcl/Tk and can thus be directly applied on different platforms including Unix and Windows. The xoRBAC component can be reused for applications on Unix or Windows with a C or Tcl linkage (see also [25]). Nevertheless, while some sensors may access platform-specific system functions we install different sensor libraries depending on the platform xoRBAC is used on. However, the sensor interfaces are (of course) platform independent.

The abstract design of xoRBAC is generic and can be used to extend arbitrary (traditional) RBAC services with context constraints. Our reference implementation of xoRBAC presented in this paper can be flexibly extended with reasonable efforts, and thereby allows for the consideration of previously "unknown" context information. Approaches for context-dependent access control could be implemented in many other ways as the one suggested in this paper, of course (see Section 6). However, in our opinion the approach presented in this paper has yet a good potential to investigate the consideration and significance of context information in access control.

Novel applications using pervasive computing techniques, and the vision of ubiquitous internet access, e.g. in cars or planes, yet give only a rough idea of the upcoming related security issues. Thus, we hope to increase the knowledge and understanding of *context* with respect to access control, to enable the enforcement of tailored context-dependent access control policies. This is, however, a wide open ground and is likely to provide research questions for many years.

In our experiences, context constraints, as defined in this paper, are intuitively understandable and are a suitable means to model dynamic context-dependent constraints. Furthermore, they allow for the dynamic evolution of access control policies, and the alignment to changing environment conditions as they frequently occur in interactive networked environments. Although context constraints can be modeled and used straightforwardly they (potentially) give an enormous rise to complexity of access control policies. On the other hand they add much flexibility and expressiveness, and allow for the definition of fine-grained access control policies as they are often needed in real world applications. So far we especially gained experiences with xoRBAC in the domain of web-based collaborative applications. However, we are conducting more case studies to further improve the role engineering process, and to investigate the applicability of context constraints in different application domains. We are especially interested in the enforcement of RBAC policies that include context constraints in an ad hoc computing environment.

The xoRBAC component is publicly available from `http://www.xotcl.org`.

8. REFERENCES

[1] N.R. Adam, V. Atluri, E. Bertino, and E. Ferrari. A Content-Based Authorization Model for Digital Libraries. *IEEE Transactions on Knowledge and Data Engineering*, 14(2), March/April 2002.

[2] G.J. Ahn and R. Sandhu. Role-based Authorization Constraints Specification. *ACM Transactions on Information and System Security*, 3(4), November 2000.

[3] A.I. Antón. Goal-Based Requirements Analysis. In *Proc. of the IEEE International Conference on Requirements Engineering (ICRE)*, April 1996.

[4] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers, 1988.

[5] J. Bacon, M. Lloyd, and K. Moody. Translating Role-Based Access Control Policy within Context. In *Proc. of the International Workshop on Policies for Distributed Systems and Networks, LNCS 1995, Springer Verlag*, January 2001.

[6] J. Barkley, K. Beznosov, and J. Uppal. Supporting Relationships in Access Control Using Role Based Access Control. In *Proc. of ACM Workshop on Role Based Access Control*, 1999.

[7] E. Bertino, P.A. Bonatti, and E. Ferrari. TRBAC: A Temporal Role-based Access Control Model. In *Proc. of the ACM Workshop on Role-Based Access Control*, 2000.

[8] E. Bertino, P.A. Bonatti, and E. Ferrari. TRBAC: A Temporal Role-based Access Control Model. *ACM Transactions on Information and System Security*, 4(3), August 2001.

[9] E. Bertino, E. Ferrari, and V. Atluri. The Specification and Enforcement of Authorization Constraints in Workflow Management Systems. *ACM Transactions on Information and System Security*, 2(1), February 1999.

[10] J. Clark and S. DeRose. XML Path Language

(XPath). http://www.w3.org/TR/xpath, November 1999. W3 Consortium Recommendation.

[11] E. Cohen, R.K. Thomas, W. Winsborough, and D. Shands. Models for Coalition-based Access Control (CBAC). In *Proc. of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2002.

[12] M.J. Covington, W. Long, S. Srinivasan, A.K. Dey, M. Ahamad, and G.D. Abowd. Securing Context-Aware Applications Using Environment Roles. In *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, May 2001.

[13] A.K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing, Springer Verlag*, 5(1), 2001.

[14] G. Edjlali, A. Acharya, and V. Chaudhary. History-based Access Control for Mobile Code. In *Proc. of the Fifth ACM Conference on Computer and Communications Security (CCS)*, November 1998.

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

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

[17] C.K. Georgiadis, I. Mavridis, G. Pangalos, and R.K. Thomas. Flexible Team-Based Access Control Using Contexts. In *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, May 2001.

[18] L. Giuri and P. Iglio. Role Templates for Content-Based Access Control. In *Proc. of the ACM Workshop on Role-Based Access Control*, 1997.

[19] T. Jaeger. On the Increasing Importance of Constraints. In *Proc. of the ACM Workshop on Role-Based Access Control*, 1999.

[20] T. Jaeger, A. Prakash, J. Liedtke, and N. Islam. Flexible Control of Downloaded Executable Content. *ACM Transactions on Information and System Security*, 2(2), May 1999.

[21] S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems*, 26(2), June 2001.

[22] M. Jarke, X.T. Bui, and J.M. Carroll. Scenario management: An interdisciplinary approach. *Requirements Engineering Journal*, 3(3/4), 1998.

[23] M.H. Kang, J.S. Park, and J.N. Froscher. Access Control Mechanisms for Inter-Organizational Workflow. In *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2001.

[24] W. Kim, S. Graupner, A. Sahai, D. Lenkov, C. Chudasama, S. Whedbee, Y. Luo, B. Desai, H. Mullings, and P. Wonng. Web E-Speak: Facilitating Web-Based E-Services. *IEEE Multimedia*, 9(1), 2002.

[25] 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.

[26] 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.

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

[28] U. Nitsche, R. Holbein, O. Morger, and S. Teufel. Realization of a Context-Dependent Access Control Mechanism on a Commercial Platform. In *Proc. of the 14th International Information Security Conference (IFIP/SEC)*, September 1998.

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

[30] C.E. Phillips, T.C. Ting, and S.A. Demurjian. Information Sharing and Security in Dynamic Coalitions. In *Proc. of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2002.

[31] C. Rolland, G. Grosz, and R. Kla. Experience with Goal-Scenario coupling in Requirements Engineering. In *Proc. of the IEEE International Symposium on Requirements Engineering (RE)*, 1998.

[32] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2), February 1996.

[33] A. Schmidt, M. Beigl, and H.W. Gellersen. There is more to context than location. *Computers & Graphics, Elsevier*, 23(6), December 1999.

[34] R.K. Thomas. Team-based Access Control (TMAC): A Primitive for Applying Role-based Access Controls in Collaborative Environments. In *Proc. of the ACM Workshop on Role Based Access Control*, 1997.

[35] R.K. Thomas and R.S. Sandhu. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization management. In *Proc. of the IFIP WG11.3 Conference on Database Security*, August 1997.

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

[37] A. van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Transactions on Software Engineering*, 26(10), October 2000.

[38] W. Wang. Team-and-Role-Based Organizational Context and Access Control for Cooperative Hypermedia Environments. In *Proc. of the ACM Conference on Hypertext and Hypermedia*, 1999.

[39] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3), September 1991.

[40] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7), July 1993.

[41] W. Yao, K. Moody, and J. Bacon. A Model of OASIS Role-Based Access Control and its Support for Active Security. In *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2001.