



Deliverable AP 2.2

# Requirements for the Policy Language

31.01.2007

Christopher Alm, HITeC / University of Hamburg  
Thomas Apel, Eurosec  
Michael Drouineaud, TZI / University of Bremen  
Mathias Kohler, SAP

ORKA is funded by the German Ministry of Education and Research (BMBF) as part of its Software Engineering 2006 program.

© 2006 ORKA Consortium



**Internal document information:**

\$Id: del-ap2.2.tex 372 2007-04-10 09:09:40Z alm \$

# Contents

- 1 Introduction 4**
  
- 2 Expressive Power (Enterprise View) 5**
  - 2.1 Results of the Case Studies A to D 5
    - 2.1.1 Case Study A - Updating a Law in Austria 5
    - 2.1.2 Case Study B - Loan Origination Process in Banking Domain 5
    - 2.1.3 Case Study C - An Inpatient Health Care Scenario 6
    - 2.1.4 Case Study D - Cross Organizational Engineering Process 6
  - 2.2 Results of the Case Study 2 6
  - 2.3 Case Study Conclusions 7
  
- 3 Expressive Power (Access Control Model View) 7**
  - 3.1 Role-Based Access Control 7
  - 3.2 RBAC Extensions 7
    - 3.2.1 Role Hierarchies 7
    - 3.2.2 RBAC Authorization Constraints 8
    - 3.2.3 Delegation 8
    - 3.2.4 Workflows 8
    - 3.2.5 Separation of Duty 8
  - 3.3 RBAC Authorization Constraints 8
    - 3.3.1 Classification of Authorization Constraints 9
    - 3.3.2 Required Authorization Constraints 10
  - 3.4 RBAC Authorization Policy and Policy Validity 12
  
- 4 Validation Requirements 13**
  - 4.1 The advantages of first-order linear temporal logic 13
  - 4.2 Validation with (first-order) LTL 14
  - 4.3 Model Checkers and Theorem Provers 14
  - 4.4 Validation Requirements 15
  
- 5 Enforcement Requirements 15**
  - 5.1 Mandatory Enforcement Requirements 15
    - 5.1.1 Machine Readability 15
    - 5.1.2 Support for distributed authorization architecture 16
    - 5.1.3 Link between policies, business process and roles/subjects 16
  - 5.2 Optional Enforcement Requirements 17
    - 5.2.1 Structural Requirements 17
    - 5.2.2 Conflict Solution 17
  
- 6 Administrative Requirements 17**
  
- 7 Additional Design Requirements 18**
  
- 8 Conclusions 19**

# 1 Introduction

The goal of ORKA is to develop a fine-grained *authorization architecture* that goes together with enterprise IT infrastructures including workflow environments and service-oriented architectures. Whenever a user attempts to perform a certain task or attempts to access a certain object it has to be decided, as part of the authorization architecture, whether such a request is granted or denied. These so-called *access decisions* are made according to an *authorization policy* that provides all information and all conditions necessary to come to an access decision. In ORKA, the authorization policy is represented by a dedicated policy specification language. This deliverable is intended for collecting the requirements of the ORKA authorization policy language.

This document is structured according to a classification of requirements. On the most abstract level, we classify the requirements according to their level in language engineering. We distinguish between the level of *language expressiveness* treated in Sections 2 and 3 and the level of *language design* treated in Sections 4, 5, 6, and 7. Within the level of language expressiveness we describe the required range of policies the language is able to express. For example, what kind of constraints or what kind of access control models should be supported. The level of language design encompasses the required language properties, for example, whether it is a formal language or whether the language is easy to learn. These two levels are further subdivided according to the following schema.

- Language Expressiveness
  - Enterprise View: the expressiveness requirements are formulated here from the perspective of an enterprise. The enterprise view refers only to the required organizational control principles such as separation of duty or delegation of rights that the language should be able to express in a policy. Hence, this view is quite abstract and does not care about the underlying access control model on which the actual policy is based.
  - Access Control Model View: this view expresses the expressiveness requirements collected in the enterprise view the by means of an access control model. For example, the principle of separation of duty can be expressed through mutual exclusion constraints placed on the set of users in the role-based access control model.
- Language Design
  - Validation Requirements: design requirements emerging from the goal that it should be possible for ORKA policies to be subject to some automated or semi-automated validation procedure. This procedure includes consistency checks, i.e. checking whether a policy contains inherent contradictions, completeness checks, i.e. checking whether a given policy is sufficient to express some required organizational control principle, and security property checks, i.e. checking whether a given policy adheres to some security property such as the principle of least privilege.
  - Enforcement Requirements: enforcement requirements are design requirements resulting from the mechanism that are responsible for the enforcement of the policy.
  - Administrative Requirements: administrative requirements are design requirements resulting from the perspective of policy administration.

- Additional Design Requirements: all design requirements not yet classified fall into this category.

Independent from the structure of this document, requirements can be optionally classified according to the following criteria:

- According to Bradner [?], design requirements may be classified as MUST, SHOULD, MAY, MUST NOT, SHOULD NOT, or MAY requirements.
- According to Sommerville [?], it is possible to distinguish functional and non-functional requirements. Functional requirements are required features or functions that the language is required to support such as obligation or delegation support. Non-functional requirements are required properties that emerge from the language as a whole such as scalability and usability.

## **2 Expressive Power (Enterprise View)**

### **2.1 Results of the Case Studies A to D**

#### **2.1.1 Case Study A - Updating a Law in Austria**

The following requirements are the access control requirements from Case Study A extracted in general form:

1. One person should not be allowed to perform two exclusive tasks.
2. One person who performed one or more specific tasks should not be allowed to perform another specific task.
3. A person is only allowed to perform n tasks out of a set of tasks within a workflow instance.
4. A person performed task t is obliged to perform task t'.

#### **2.1.2 Case Study B - Loan Origination Process in Banking Domain**

The following requirements are the access control requirements from Case Study B extracted in general form (generalized requirements from the previous case study with the same meaning are not included):

1. Two tasks get exclusive depending on an objects attribute.
2. A person may not activate two exclusive roles within one workflow instance.
3. Two tasks are exclusive depending on previous task results (application dependent).
4. A principal may be a member of the two exclusive roles and the complete set of authorizations acquired over these roles may cover a critical authorization set, but a principal must not use all authorizations on the same object(s).

5. A task's role-permission assignment is dependent on an object's attribute.

### **2.1.3 Case Study C - An Inpatient Health Care Scenario**

The following requirements are the access control requirements from Case Study C extracted in general form (generalized requirements from the previous case study with the same meaning are not included):

1. A task can be delegated to another person.
2. A role has the permission to perform a certain task.
3. A role has the permission to perform a certain task depending on the object.

### **2.1.4 Case Study D - Cross Organizational Engineering Process**

This case study does not deliver requirements for the policy language rather than for the design of the authorization architecture of the workflow engine. These requirements are therefore not considered in this work package.

## **2.2 Results of the Case Study 2**

Case study 2 from ORKA work package 1.1 uses Microsoft Windows environments in small and medium-sized companies for presenting different scenarios where mechanisms for organizational control apply. With respect to the policy definition language, the following high-level requirements can be derived from the different case study scenarios:

1. It should be possible to specify that a person who executed task A must not be able to perform task B on the same object. Performing task B on other objects however might be allowed.
2. Slightly similar, it should be possible to demand that a person who executed task A must not be able to perform task B within the same workflow. Performing task B within other workflows however may be allowed.
3. The language should allow to specify event-driven permissions. For example it should be possible to express that the permission to perform certain tasks automatically gets transferred to a substitute while an employee is on vacation.
4. It should be possible to define roles that bundle permissions that are necessary for performing certain tasks.
5. A policy should be able to force a user to perform action A if event B occurred.
6. It should be possible to demand that action A by user B requires approval by user C.
7. The language should allow the specification of audit requirements or logging actions for certain events or actions.

## 2.3 Case Study Conclusions

From the case studies we conclude that the ORKA language should be able to express the principles listed in the following:

- Separation of duty: various types of separation of duty including simple dynamic separation of duty, object-based separation of duty, and operational separation of duty, e.g. for tasks in a workflow.
- Prerequisites: a prerequisite is a precondition that has to be fulfilled in order for some access request to be granted.
- Support for workflows: it should be possible to pose restrictions based on the state of a workflow.
- Order of events in workflows: as a particular requirement, prerequisites in workflows should be supported.
- Delegation and revocation of rights.
- Context-sensitive restrictions: a context-sensitive restriction is a restriction that depends on information available only at runtime such as information about a user's session.
- Obligation: a system supporting *obligation* is capable of forcing a subject to perform a certain task or action in case that a certain event occurs or a certain condition becomes true. For example, a subject could be obliged to perform a task after obtaining or exercising a right. Obligation is particularly important for workflow environments. Note that this definition is in line with the notion of obligation in Ponder [?] and in UCON [?].
- History-based restrictions.
- Time-based restrictions.

## 3 Expressive Power (Access Control Model View)

### 3.1 Role-Based Access Control

We require that the authorization policies expressible by our language are based on the role-based access control (RBAC) model plus some extensions of RBAC, which are defined in the Subsections 3.2 and 3.3. We define the term authorization policy within the realm of RBAC in Subsection 3.4. It should be noted that it is possible to simulate other access control models using RBAC such as lattice-based access control [?].

### 3.2 RBAC Extensions

#### 3.2.1 Role Hierarchies

A role hierarchy should be supported. The hierarchy extension of RBAC is already included in the RBAC standard [?].

### 3.2.2 RBAC Authorization Constraints

Since RBAC authorization constraints are such a major extension to the RBAC model in terms of expressiveness, we treat it in the dedicated Subsection 3.3.

### 3.2.3 Delegation

In order to provide a concept for delegation and revocation of rights, RBAC needs to be extended. In particular, it should be possible in ORKA that a user is able to grant rights to another user and even to transfer a right to another user. It should be noted that a delegation concept is not part of the RBAC standard [?], yet.

### 3.2.4 Workflows

One goal of the ORKA architecture is to provide an authorization service for a workflow environment. Thus it is necessary that the policy language can express a workflow related authorization policy. However, it has to be decided whether it is sufficient to add workflow related authorization constraints or whether a coherent model including workflows and RBAC is needed. While the latter case would make an additional RBAC extension necessary, we can implement the former case using RBAC authorization constraints treated in Subsection 3.3.

### 3.2.5 Separation of Duty

Similar to workflow related policies, an RBAC policy supporting the principle of separation of duty could make an RBAC extension beyond authorization constraints necessary. For example, sets of conflicting roles or permissions could be needed in the model.

## 3.3 RBAC Authorization Constraints

As we pointed out in the previous subsection, a common means to extend the expressiveness of the role-based access control model is *authorization constraints*. Before we can define authorization constraints, we need to summarize some facts about RBAC<sup>1</sup>.

The RBAC model with its extensions, except for constraints, consists of so-called *RBAC elements* which are some sets (e.g. for users and permissions), relations (e.g. for the user assignment to roles), and functions (e.g. for the mapping of a session to its user or for status queries including access decision requests). An instance of an RBAC model, which defines the state of all sets, relations, and functions, is called an *RBAC configuration*. A *state transition* occurs if one RBAC configuration changes to another. Examples for state transitions are adding users, creating a session, and granting a permission to a role. Delegating a right by a user is also an example, however, there is no appropriate concept in the RBAC standard, yet. We distinguish between *user initiated state transitions* and *administrative state transitions*. A user initiated

---

<sup>1</sup>A precise, mathematical definition of role-based access control will be made available in another document within the ORKA project. Here, we can only refer to the ANSI standard [?].

state transition is a state transition that can be performed by a normal user such as the delegation of a right or the activation of a role. An administrative state transition is a state transition that can only be performed by the policy administrator such as the assignment of a user to a role.

An *authorization constraint* is a necessary condition for *validity*<sup>2</sup> of an RBAC configuration that is placed on an arbitrary RBAC element. An authorization constraint can be represented by a formula of an appropriate logic such as first-order logic (FOL) or linear temporal first-order logic (first-order LTL). Hence, if one authorization constraint is not satisfied the configuration is not valid.

Having defined authorization constraints in this way, this concept is clearly separated from the rest of the RBAC model. As various examples below in this subsection illustrate, authorization constraints provide a flexible and extensible approach to build fine-grained authorization policies on top of RBAC.

After classifying authorization constraints in the next subsection, we give the list of authorization constraints that are necessary in order for the language to be able to express the required principles of Section 2.

### 3.3.1 Classification of Authorization Constraints

The following classification of authorization constraints is derived from the work of Strembeck and Neumann [?].

- An authorization constraint is either **dynamic** or **static**.
  - A *dynamic constraint* is an authorization constraint that can be evaluated only at runtime.
  - A *static constraint* is an authorization constraint that is not a dynamic constraint.
- An authorization constraint is either **endogenous** or **exogenous**.
  - An *endogenous constraint* is an authorization constraint that depends only on information included in the current RBAC configuration.
  - An *exogenous constraint* is an authorization constraint that may also depend on information from resources external to RBAC such as time or location.
- An authorization constraint can be an **access decision constraint**.

An *access decision constraint* is an authorization constraint that is placed on the function *checkaccess* [?, p.15] of the RBAC model. One common form of access decision constraints for a session  $s_0 \in S$ , an operation  $op_0 \in OPS$ , and an object  $ob_0 \in OBS$  is

$$checkaccess(s_0, op_0, ob_0) = 1 \Rightarrow \phi$$

where  $\phi$  is some formula of an appropriate logic.

---

<sup>2</sup>Since it is not the scope of this subsection, we postpone the definition of validity for RBAC configurations to Subsection 3.4.

It is important to emphasize that the constraint consists of the whole expression and not only of the formula  $\phi$ . However, it is necessary to evaluate  $\phi$  to ensure that such a constraint holds, at worst, upon each access request. If  $\phi$  is false then the value of *checkaccess* is 0, corresponding to access denied.

### 3.3.2 Required Authorization Constraints

The following list of authorization constraints should be supported by the ORKA language in order to provide fine-grained authorization policies.

- All types of *separation of duty* discovered by Simon and Zurko [?] should be supported. These are static separation of duty, simple dynamic separation of duty, object-based separation of duty, Chinese Wall (i.e. object-class-based separation of duty), operational separation of duty (i.e. workflow-based separation of duty), and history-based separation of duty. The appropriate constraints can be implemented using mutual exclusion constraints on both permissions and roles. As mentioned above in this section, it could be necessary to extend the RBAC model with sets of conflicting roles or permissions.
- A *prerequisite constraint* is an endogenous authorization constraint that places a precondition on an RBAC element such that certain RBAC configurations are allowed only if the precondition is fulfilled. A common example is a role  $r_1 \in R$  that can only be assigned or activated by a user  $u$  if  $u$  has already been assigned to or activated respectively role  $r_0 \in R$ . The first part of this example can be formalized as

$$(u, r_1) \in UA \Rightarrow (u, r_0) \in UA.$$

Prerequisite constraints for permission assignments are also very common.

- An *activation constraint* is a, mostly endogenous, authorization constraint placed on the relation associating roles and sessions. Thereby a user can be restricted from activating an arbitrary set of roles in a session.
- A *role assignment constraint* is a, mostly endogenous, authorization constraint that is placed on the user assignment relation  $UA$ , which is used assign users to roles. For example, there could be constraint such that every user must be assigned to role  $r_0$ .
- A *permission assignment constraint* is a, mostly endogenous, authorization constraint that is placed on the permission assignment relation  $PA$ , which is used to assign permissions to roles.
- A *role hierarchy constraint* is a, mostly endogenous, authorization constraint that is placed on the role hierarchy relation  $RH$ , which is used to model the inheritance of rights. As mentioned above, it is possible to simulate various types of lattice-based access control in RBAC. This is done by using activation constraints, role assignment constraints, permission assignment constraints, and role hierarchy constraints [?].
- A *cardinality constraint* is an endogenous authorization constraint that restricts the maximum number of elements for a set. For example,

$$|U| \leq 100$$

which would restrict the maximum number of users to 100. For a given  $u \in U$

$$|\{s \in S | SU(s) = u\}| \leq 2$$

would restrict the number of sessions that  $u$  can have at a time to 2. The function  $SU$  is denoting the mapping from sessions to users. In contrast to the former example, the latter example is considered to be dynamic, since the set of sessions  $S$  is not part of the static policy.

It should be possible to place cardinality constraints on any set that is part of the RBAC model. Since relations and functions are particularly sets, a cardinality constraint could be placed on any RBAC element in theory.

- Once RBAC is extended with a delegation concept, it is required to have a means to restrict delegation. A *delegation constraint* is an authorization constraint placed on any RBAC element that is involved in the delegation concept. For example, if *CanDelegate* is a relation specifying the set of roles that can be delegated by delegator, as defined by Crampton and Khambhammettu [?], a delegation constraint on this relation could prohibit certain roles to be added to this relation. As a consequence, these roles are non-delegable by any user.
- A *context constraint* is a dynamic, mostly exogenous, access decision constraint. In particular, the following context constraints should be supported.
  - A *history-based constraint* is a context constraint that depends on information about out-dated RBAC configurations or even on information about events and actions that have been performed in the past. For example:
    - \* User  $u$  is allowed to exercise permission  $p_1$  only if  $u$  has already exercised  $p_0$ .
    - \* User  $u$  can only be assigned to role  $r_1$  if  $u$  once was assigned to role  $r_0$ .
    - \* User  $u$  must not exercise right  $p$  more than two times.

Note that it depends on the RBAC model, which history information is considered to be RBAC external. For example, it is external, if history information has to be provided through an external database. However, it is internal if, for example, an out-dated RBAC configuration is available within the model (and its implementation).

To express history-based constraints, a temporal logic such as LTL is very useful.

- A *time constraint* is a context constraint that involves time. Typically, for a session  $s_0 \in S$ , an object  $ob_0 \in OBS$ , and an operation  $op_0 \in OPS$  a time constraint looks like

$$checkaccess(s_0, op_0, ob_0) = 1 \Rightarrow f(t)$$

where the function  $f$  gives a boolean value for the point in time  $t$ . Using  $f$ , it is possible, for example, to express that this access request can only be granted between 9am and 5pm.

In combination with history-based information a time constraint could express that a certain right can be exercised at most once an hour by a given user.

- Context constraints can be used to express workflow-related authorization policies since they may depend on workflow-related information such as information about a current task.
- To support obligation as defined in Subsection 2.3, the authorization policy should be able to express *obligation-supporting constraints*<sup>3</sup>. An *obligation-supporting constraint* is an access decision constraint that allows a given subject to perform only the obligated operation on a given object or a given set of objects.

There are two types of obligation-supporting constraints.

1. The system using the ORKA authorization service (for example a workflow management system) could place (and remove) a *static obligation-supporting constraint*. Say, the system obliges session  $s_0 \in S$  to perform  $op_0 \in OPS$  on  $ob_0 \in OBS$ . A constraint supporting this obligation could be represented as

$$\forall op \in OPS, \forall ob \in OBS : (checkaccess(s_0, op, ob) = 1 \Rightarrow (op = op_0 \wedge ob = ob_0)).$$

2. In contrast to this, a *dynamic obligation-supporting constraint* triggers automatically an obligation in case that some condition becomes true. For example, this could be a condition about RBAC configurations (out-dated plus current) so that we would speak then of a history-based constraint. Another example is to make a dynamic obligation-supporting constraint depend on some external event, where the condition becomes true if the event occurs. For some  $s_0 \in S$ ,  $op_0 \in OPS$ , and  $ob_0 \in OBS$  such constraints could be represented as

$$\phi(s_0, op_0, ob_0) \Rightarrow checkaccess(s_0, op, ob) = 1 \Rightarrow (op = op_0 \wedge ob = ob_0)$$

where  $\phi$  could be a function representing the state of an event or a condition about an RBAC configuration.

### 3.4 RBAC Authorization Policy and Policy Validity

The goal of this subsection is to bring together the terms authorization policy and policy validity. In particular, we strive for answering the question how security and policy validity are related.

An *RBAC authorization policy*, for whose **full representation** our language will be designed, consists of the current RBAC configuration and a set of authorization constraints. The part of a policy that can be affected by user initiated state transitions such as the relation between sessions and roles is referred to as the *dynamic part* of the policy. Everything that is not included in the dynamic part of a policy is referred to as the *static part* of the policy. Hence, the static part of the policy is the part that can only be affected by administrative state transitions or by changes of the set of authorization constraints. It has to be decided whether the ORKA policy language should also be able to express the dynamic part of the policy in addition the the static part.

In an abstract sense, the policy is *valid* if it satisfies the security requirements of the customer. However, this is the only sufficient condition we can give for policy validity and thus it is

---

<sup>3</sup>We chose the term *obligation-supporting constraint* on the one hand in favor of not getting confused with the notion of Ahn's *obligation constraints* [?]. On the other hand we would like to express that this kind of constraints provides the concept of obligation to the model instead of that obligation is a characteristic of these constraints.

not possible to determine precisely whether a given policy is valid. We can only list important necessary conditions for policy validity. This includes *policy consistency*, i.e. the policy does not contain inherent contradictions, *policy completeness*, i.e. the policy is sufficient to express the required organizational control principles, and *secure policy design*, i.e. the policy adheres to the required security properties such as the principle of least privilege. In contrast to policy validity, we call an RBAC configuration of a given RBAC policy *valid* if all authorization constraints of the policy are fulfilled. RBAC configuration validity is also a necessary condition for RBAC policy validity. It should be noted that we give no information on how to proceed in case that an RBAC authorization policy becomes invalid.

The job of *policy validation* is twofold. On the one hand the goal of validation is to create an RBAC authorization policy which is as valid as possible<sup>4</sup>. In particular, an appropriate set of authorization constraints has to be established. For example, a certain security requirement may be modeled as a theorem which is proven using a certain set of authorization constraints as axioms. Note that such theorems need not to be expressed by the ORKA policy language. On the other hand validation must ensure that all authorization constraints of the policy are fulfilled at any time (in order for the RBAC configuration to be valid). I.e. checking necessary conditions for policy validity. In order to achieve these two goals, every RBAC policy change – either a state transition or a change of the set of authorization constraints – must preserve policy validity; in particular, authorization constraints must be preserved. And, if there are dynamic authorization constraints that may violate validity (i.e. become unsatisfied) without the occurrence of a policy change, additional checks are needed, for example, upon each access request. In this context, the computability of constraints is important.

## 4 Validation Requirements

### 4.1 The advantages of first-order linear temporal logic

First-order linear temporal logic (short: first-order LTL) is a solid quite expressive formalism possessing a mathematically defined syntax and semantics (Kripke semantics<sup>5</sup>). Indeed, first-order logic is powerful enough to serve as a formalism for many parts of mathematics (including set theory). First-order LTL augments first-order logic with a linear ordered temporal structure. Thus it is very suitable for reasoning about temporal invariants (e.g. RBAC security policies having authorization constraints such as history-based separation of duty). On the other hand, due to first-order properties first-order LTL formulas potentially define an infinite class of models. This makes first-order LTL rather appropriate to specify RBAC (security) policies, since such policies are essentially finite sets of schematic rules (cp. AP 4.1). This holds especially for temporal properties such as history-based separation of duty.

Altogether, first-order LTL is a formalism offering sufficient expressiveness, high flexibility (schematic rules), a mathematically defined syntax and semantics and a linear ordered temporal structure. Hence it is very apt to reason about (temporal) properties of RBAC policies. First-order LTL and several RBAC theories have been encoded in the theorem prover Isabelle.

---

<sup>4</sup>Once policy validity is assured during policy design, one could call a policy *valid up to trustworthiness*.

<sup>5</sup>For a precise definition of ‘Kripke semantics’ we refer the reader to AP 4.1 or the glossary.

## 4.2 Validation with (first-order) LTL

For validation RBAC (security) policies are specified as finite sets of closed <sup>6</sup> first-order LTL formulas. Properties and consequences of a given RBAC policy can be derived as lemmata resp. theorems from the corresponding set of (closed) first-order LTL formulas.

On the other hand, it has to be assured that a given RBAC policy is contradiction-free. This can be done by giving a (non-trivial) Kripke model for the corresponding set of first-order LTL formulas, since by a fundamental result of logic the existence of a model rules out contradictions. There may still be unwanted consequences, but contradictions are excluded.

## 4.3 Model Checkers and Theorem Provers

Since regular RBAC configurations (cp. AP 4.1) can be finitely described using a *propositional* LTL, the consistency of an RBAC policy can be checked by a model checker. For these purposes one has to give a (regular) RBAC configuration to the model checker in question and the schematic rules of the policy as properties to check. This can be done automatically without human intervention (automated validation). Model checking can thus be used to check security requirements of an RBAC policy in an IT system at runtime while deduction-based validation is generally used during the design phase. At the first look automated validation at runtime may seem like the perfect solution, but the computational complexity may be very high. So it could slow down the actual IT system to an intolerable extent.

This can be avoided by proving, that the security requirements are properties of the considered RBAC policy using the inference rules of first-order LTL (presuming that the policy has been designed accordingly and been specified in first-order LTL). The correctness of the given proofs can then be verified by the theorem prover Isabelle (cf. subsection 4.1) under the assumption, that Isabelle itself works correctly w.r.t. the specifications of a theorem prover. Therefore theorem provers like Isabelle can be very helpful tools for the design of an RBAC (security) policy. Of course this will only help if checking the rules of the policy in question has a notably lower computational complexity than checking the security requirements itself. This has also to be considered while designing an RBAC policy.

Supposed the above assumptions hold, it is sufficient to enforce the rules of the RBAC policy in order to ensure the security requirements, which can be done with distinctively lower computational effort than directly checking the security requirements.

Finally we conclude, that there are good reasons for model checking as well as for theorem proving and that model checking and theorem proving should be regarded rather as complementary than as competing methods.

Model checking can help to assure that a given RBAC policy is contradiction-free resp. consistent. Furthermore it can be used to test an RBAC policy for unwanted consequences. Theorem proving helps to make sure, that security requirements follow from a considered RBAC policy. Beyond that a lot of other properties (e.g. safety properties) might be proven (as far as they can be expressed in first-order LTL). And in the end theorem proving can be used to verify, that for a given RBAC policy and given security requirements certain transformations of the policy in question will not violate the security requirements (provided they are valid for the considered policy) <sup>7</sup>.

---

<sup>6</sup>A closed first-order LTL formula is a formula without **free** variables.

<sup>7</sup>Such transformations are denoted **validity preserving** transformations.

## 4.4 Validation Requirements

From the perspective of validation a policy description language should have an unequivocal translation to (first-order) LTL. This means, it should be able to express a linear temporal order and/or quantification. So it should be possible to translate a specified RBAC policy into (first-order) LTL by a theory morphism. On that account there would also be mathematically defined semantics (namely a Kripke semantics) for a core of the policy description language.

## 5 Enforcement Requirements

An essential element for authorization is a system's capability to enforce policies. This also requires certain demands for the expressiveness of an policy specification language. This section describes the requirements for the policy language with respect to the enforcement of policies within the system to be developed in this project.

The aspects to be identified as enforcement requirements split up into two main groups.

The first group comprises mandatory requirements. Without their realization in the policy language, the later developed system won't work. We will show which requirements are mandatory and explain why.

The second group of requirements won't prevent the system from working if not realized. Therefore they are to be seen as optional and aim mainly on the systems scalability. If implemented, they will enhance the systems performance in comparison to if not realized.

In our requirement analysis we assumed that the architecture to be developed in this project will be placed in an distributed environment. Therefore the enforcement requirements for the policy language will reflect this.

In the following we describe the single requirements we identified to be of relevance. We start describing the mandatory requirements followed by the description of the optional requirements.

### 5.1 Mandatory Enforcement Requirements

This section lists and describes the mandatory requirements for the policy language.

#### 5.1.1 Machine Readability

A common architecture as described in [?] is that a policy enforcement component enforces the access control decision made by the policy decision component. The decision component evaluates authorization requests according to the policies specified with a policy language. Consequently, the policy component must be able to read in or parse the policies.

Hence, one of the most important points is that the authorization policies are available in a machine readable way. In other words, for instance, a parser and compiler bundle should be able parse and compile the policies for the later policy evaluation and subsequent enforcement.

Also taking into account that there is a request for human readability (see section 6) a semi-formal language is required to express authorization policies.

### **5.1.2 Support for distributed authorization architecture**

The system to be developed in ORKA is assumed to work in a distributed environment. The following requirements express the necessary aspects needed for the language to support distributed systems.

- Distributed policy locations:

There should be the possibility to process policies at distributed locations. This implies that policy specifications must be able to be stored at different locations and also, for instance, internal references within a policy specification document have to support this. Distributed policy locations should also be able to be used for redundancy reasons such as backups or prevention of single point of failure effects as well as for performance reasons if, for instance, policy specifications can be stored as close to the decision component as possible.

- Enforcement independent policy location:

The location of the policy specifications should be independent from the location for the final enforcement. It should also be independent from the location of the policy decision component.

### **5.1.3 Link between policies, business process and roles/subjects**

There is the need that specified tasks within a business process and their accompanying policies must be unequivocally be linked. This requirement can be subdivided into the following four points.

- Unequivocal assignment of policies to a business process:

There should be a unequivocal assignment of a policy or a collection of policies to a business process possible. This is necessary to identify the policies belonging to a certain business process for which an authorization request has to be evaluated and eventually enforced.

- Unequivocal assignment of policies to a single task:

There should also be a unequivocal assignment of a policy or a collection of policies to a single task of a business process possible. This is necessary to identify the policies which affect authorization requests for a certain task.

- Clear assignment of roles or subjects to a certain policy:

It is necessary that every policy definition clearly states which roles or subjects are affected by a policy.

- Policy identifier:

Each policy of a policy specification should have a unique identifier. This is necessary for logging which policy has eventually been enforced. This makes auditing of authorization requests possible.

## 5.2 Optional Enforcement Requirements

This section lists and describes the optional requirements for the policy language.

### 5.2.1 Structural Requirements

There are several optional requirements with respect to performance enhancement.

- Structural optimization for analyzing the question 'Has role/subject x the rights to perform task y?':

A request like this will possibly be one of the most processed requests performed by the decision component. Therefore, the policy specification should support this type of request with respect to its document and policy structure.

- Structural optimization for analyzing the question 'Which users have the rights to perform task y?':

To assign a certain task to a subject this type of request has to be performed by the decision component. Therefore, the policy specification document also should support this type of request.

Possible starting points for both previous structural requirements might be that the policy specification language supports nesting or prioritizing of policies that it is possible to policies with greater impact on which roles or subjects are affected before policies are checked where less users can be excluded. Another possible starting point is to separate policies according to different areas such as 'policies for administration' and 'policies for workflow authorization'.

### 5.2.2 Conflict Solution

There might always be the possibility of conflicting situations or situations where error handling is necessary. There should be an entry for a policy or a collection of policies which states how such situations should be handled.

## 6 Administrative Requirements

An aspect that is essential for the effectiveness of policy-based authorization systems is the proper administration of the policies. This applies to both the initial policy definition and the necessary policy maintenance during its lifecycle.

In order to archive this goal, the design of the policy language should help to keep efforts for creating and maintaining policy definitions as low as possible. This results in the following administrative requirements for the policy definition language in the ORKA project:

- **Machine writability:** The policy language must not only be machine readable, it must also be machine writable. Machine writability is necessary for applications that aid users with creating appropriate policies by offering wizards or graphical point and click interfaces. Such applications must be able to generate specifications using the policy language.
- **Human writability and readability (Controllability):** Humans must be able to read and write policy specifications without too much effort. In general, human administrators must be able to have control over their policy definitions by being able to understand definitions and to express demands. This requirement is somewhat reduced in scenarios where a (possibly graphical) policy administration tool offers an easy to use interface for specifying policies. Nevertheless, even in such scenarios, the underlying language must be comprehensible to skilled users for trouble-shooting and development purposes.
- **Comments:** The language should allow to add comments within a policy specification. Comments are necessary for storing meta information like explanations why a certain rule is necessary and what it is intended for. Moreover, comments allow to keep information why, when, and by whom certain parts of the specification have been changed.
- **Hierarchical definitions:** The language should allow the application of definitions to hierarchical structures including inheritance.
- **Distributed administration:** To allow distributed administration of policies, it should be possible to split a policy into multiple parts that contain references to other policy resources (e.g. files).
- **Macros:** The policy language may support macro expansion including parameter substitution to allow central maintenance of recurring definitions. However, this may also be implemented by using an additional macro language for that purpose. Moreover, including macro functionalities might complicate policy validation.
- **Logging and auditing:** The language should be able to express logging requirements for arbitrary events. Besides defining who can perform what actions under what circumstances, the policy should be able to mandate the generation of log entries whenever an action was allowed or denied.
- **Maintainability:** In general the policy language should allow easy maintenance of the policy definitions. This requirement subsumes all requirements that are necessary for easy policy maintenance but are not explicitly mentioned in the previous requirements. For example the language should be text-based in order to allow the application of standard text-editing tools and version control systems.

## 7 Additional Design Requirements

The design of the ORKA policy language should take into account the following, mainly non-functional, requirements.

- The **extensibility** of the language is required so that it is possible to adapt the language to unforeseen needs arising in the future.
- The language should have a **modular design** to support extensibility, flexibility, and controllability.
- While designing the language, the **scalability** of the whole ORKA authorization architecture should be kept in mind as one of the most important requirements of the architecture.
- The language should have a **formal definition** (syntax and semantics). This is also required as a validation requirement in that there should be an unequivocal translation from the ORKA policy language to LTL.
- A **type system** may support readability, writability<sup>8</sup>, and validation.
- The language should have a **declarative design** as defined in the deliverable document of AP2.1.
- The **efficiency** of evaluation, interpretation, or compilation of the policy or of policy expressions is an important requirement in the ORKA authorization architecture and the policy language should be designed to support it.

## 8 Conclusions

Having elicited the requirements for the policy language in this document, we do not see any major conflicts between them.

At this point, we believe this language will be special in that it brings together validation capabilities, ability to express a wide range of organizational control principles in a direct, elegant, and concise way, and suitability for daily use in an enterprise setting. To fully accomplish the latter point we strive additionally for developing an easy-to-learn policy editor supporting policy input and administration (see Ap2.5).

Before deciding whether a new language needs to be developed or whether enhancing an existing language satisfies these requirements, this deliverable is followed by a broad analysis of existing policy languages (see Ap2.3).

---

<sup>8</sup>Readability and writability of a language are defined in the deliverable document of AP2.1.

## References

- [1] G.-J. Ahn. *The RCL 2000 language for specifying role-based authorization constraints*. PhD thesis, George Mason University, Fairfax, Virginia, 1999.
- [2] Anne H. Anderson. A comparison of two privacy policy languages: Epal and xacml. In *SWS '06: Proceedings of the 3rd ACM workshop on Secure web services*, pages 53–60, New York, NY, USA, 2006. ACM Press.
- [3] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels, 1997. RFC 2119, <http://rfc.net/rfc2119.html>.
- [4] J. Crampton and H. Khambhammettu. Delegation in Role-Based Access Control. In *ES-ORICS 06: Proceedings of 11th European Symposium on Research in Computer Security*, pages 174–191, 2006.
- [5] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder Policy Specification Language. *LNCS*, 1995:18–39, 2001.
- [6] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House Publishers, 2003.
- [7] Jaehong Park and Ravi Sandhu. Towards Usage Control Models: Beyond Traditional Access Control. In *SACMAT 2002: Proceedings of the seventh ACM Symposium on Access Control Models and Technologies*, pages 57–64. ACM Press, 2002.
- [8] Ravi Sandhu. Role hierarchies and constraints for lattice-based access control. In *ES-ORICS 1996: Proceedings of the Fourth European Symposium on Research in Computer Security*, pages 65–79. Springer-Verlag, 1996.
- [9] Richard T. Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *IEEE Computer Security Foundations Workshop*, pages 183–194, 1997.
- [10] Ian Sommerville. *Software Engineering*. Addison-Wesley, seventh edition edition, 2004.
- [11] American National Standard. *Role Based Access Control*. 2004. ANSI INCITS 359-2004, American National Standards Institute.
- [12] 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):392–427, 2004.