ADQL: A Flexible Access Definition and Query Language to Define Access Control Models

Andreas Sonnenbichler and Andreas Geyer-Schulz
Karlsruhe Institute of Technology, Karlsruhe, Germany

Keywords: Access Control, ADQL, Formal Language, Meta Language, Access Control Service.

Abstract: We suggest a full specified formal language, the Access Definition and Query Language (ADQL). It has been designed to define access control models, facts, policies, and queries. ADQL, therefore, has the features of a meta language: It can be configured to act like known access control models e.g. as Bell-LaPadula, RBAC and its extensions and applications (e.g. SAP R/3), but also it can implement new models. Because of this, ADQL is highly flexible. Nevertheless, ADQL is not only a meta-language, but also allows to define facts, policies and queries. It has been implemented as a software service. It can be used as external authorization component for other applications and services. Through its flexibility many access control models can be supported.

1 INTRODUCTION AND RELATED WORK

Since the inception of the first time-sharing and multi-process operating systems, a large variety of access control mechanisms has been designed and implemented. When we look at the current landscape of access control mechanisms, we have the impression, that there are three major directions of development:

First, extremely flexible mechanisms with as many ways to formulate, check, and monitor access policies as possible, e.g. XACML (Rissanen, 2010) which has been discussed in the literature of the last years extensively (cf. (Gupta and Bhide, 2005; Crampton and Huth, 2010)). The drawback of such solutions usually is that the rule sets of a system can become intransparent very fast (e.g. (Li et al., 2009; Ni et al., 2009)).

Second, simpler models inspired by easily understandable concepts like the role-based access control (RBAC). Many applications and services in the internet make use of such generic RBAC-based models. E.g. Oracle DBMS (Notargiacomo, 1996), SAP (AG, 2008), Amazon Web Services[1], Drupal[2], the accelerator control for CERN (Geysin et al., 2007).

The drawback of these solutions is that the underlying concept of access control, the method or model, is always hard coded and can only be extended with large efforts later.

Third, support for freely recombining internet services in a privacy-aware and scalable manner requires the flexibility to adapt access control mechanisms to service interfaces, process and organization structures and collaboration styles. Factoring out access control mechanisms from internet services, considerably simplifies service design and implementation (e.g. (Yuan and Tong, 2005)).

Our hypothesis is that because the powerful mechanisms tend to be very complex and are often not understood by software engineers, administrators, and users, the simple mechanisms are reused over and over again. We see a huge gap between what really is used when it comes to software development and what is available. This gap is mainly caused by the complexity of the powerful mechanisms.

The unification of access control models through a meta-model has been researched by Barker (Barker, 2009). He demonstrated that such a meta-model can be defined and can represent multiple access controls as derieved special cases. ADQL's approach aims in the same direction.

Samarati and Vimercati provide a comprehensive and well written overview on access control models, mechanisms, and principles (Samarati and Vimercati, 2001).

Crampton and Huth (Crampton and Huth, 2010) suggest a XACML-like policy language which is syntactically simpler than XACML, but not less expressive. In XACML (sub-) policies may not be resolv-

[1] http://aws.amazon.com
[2] http://drupal.org

Sonnenbichler A. and Geyer-Schulz A..
ADQL: A Flexible Access Definition and Query Language to Define Access Control Models.
DOI: 10.5220/0004023903790386
In *Proceedings of the International Conference on Security and Cryptography* (SECRYPT-2012), pages 379-386
ISBN: 978-989-8565-24-2

able: XACML may return “not applicable” or “indeterminate”, which makes it difficult (impossible?) to decide if access shall be granted or not. The suggested language is resilient to such “failures”. This problem especially arises, when the requested resolution has to be done on distributed systems on sub-policies.

In (Damiani et al., 2002) Damiani et al. describe a fine-grained access control system for XML documents. The access control model used is based on subjects, objects, actions, and inheritance rules. Subjects are defined as users, IP addresses, or hierarchies of these. Objects are seen as URIs (see (Berners-Lee et al., 1998)). Policies are applied on XML objects identified through XPath.

A mechanism for access control for collaborative groups is described in (Wang et al., 2009). Users can tag each other. Based on these tags access control policies can be defined.

The interplay between access control and privacy, namely privacy-aware access control is discussed in a paper of Ardagna et al. (Ardagna et al., 2008).

2 STRUCTURE OF ADQL

The design of ADQL is structured in five layers depicted in figure 1.

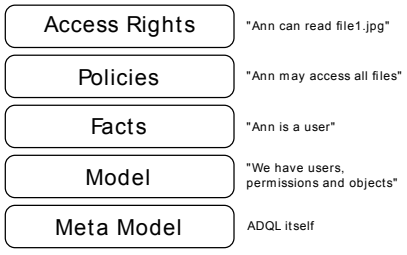

Figure 1: Structure of the Access Definition Query Language.

The *access rights layer* is a complete enumeration of all atomic conditions. Atomic is defined in the sense that no further elements of sets can be substituted to decide if access can be granted or not. The access rights layer corresponds to the Harrison, Ruzzo, and Ullman’s (HRU) access matrix model (Harrison et al., 1976).

Access rights can not be managed practically in such an atomic way: A combinatoric explosion occurs. The *policy layer* defines access conditions on a more general level. Complex policies with conditions beyond the typical triple (principal, permission, object) can be defined.

The *facts layer* describes the facts which are applied on the policies to calculate the atomic access rights. The policy “Ann may access any file” implies, that we have to know which files actually are meant (by an enumeration of the entities which are files currently present in the system) and what access means (e.g. read, write, delete, ...).

In the *model layer* of the concepts aof the access control model are described. In contrast to the facts layer, where existing entities and relationships are defined, the model layer says, what entity types and relationships between them are generally possible. E.g. concepts like “users”, “permissions”, “files”, “file owners”, and “days in a week” are defined. The policy and fact layers are heavily dependent on the model layer: Every concept used in these layers must have been introduced (and well defined) in the model layer.

The final layer of ADQL is the *meta model* of ADQL which describes the basic language elements of ADQL itself.

3 CONCEPTS OF ADQL

ADQL relies on a small amount of concepts. We define an *entity* as any “object in reality” modeled in the system.

Containers correspond to sets of entities. Containers can be used to form hierarchies (e.g. trees) or even networks (e.g. circles, graphs). It can be decided if an entity belongs to a container (hierarchy) or not. A container can be flattened or decomposed. The decomposition is a transformation of a hierarchical or linked container structure to a flat container. The decomposed container includes all entities being part of any level/container of the container structure.

An *n*-ary ADQL *relations* is define on the Cartesian product of *n* containers. It allows to link entities belonging to the underlying containers through the relation. ADQL relations can be reflexive, symmetric and transitive.

ADQL supports *projections*. In an ADQL projection $n-1$ elements of a *n*-ary relation are fixed, the result is a container of all entities belonging to the pivot container which are part of the relation. Example: Let $(file1, Alice)$ be a tuple of the relation *owner*. Then $projowner(.,Alice)$ is the projection of the relation owner with the fixed entity Alice. The result is file1. As a speciality in ADQL, projections always have to have exactly one target element, the projection target, and accordingly $n-1$ fixed tuple elements.

ADQL *variables* are always related to a specific container or container hierarchy. A variable can be bound to one or more container entities. Variables are denoted by the containers symbol enclosed in square brackets. $[user]$ is the variable corresponding to the

container ”user”. Please note, that this definition is no limitation of generality: In case two different variables are necessary for a single container, it is sufficient to define a second container including the the first and vice versa.

An ADQL *scope* is a collection of different variable bindings. E.g. $([user] = \{Alice\}, [file] = \{file1, file2\})$ is a scope of two variable bindings. The variable ”user” is bound to ”Alice” while ”file” is bound to ”file1” and ”file2”. A variable binding is only valid, if the bound entity belongs to the container of the variable. A scope is only valid, if all variables bindings are valid.

Tests are defined in ADQL as 2-argument logical operations supporting flexible logical operators evaluating to true or false.

Finally, *policies* are sets of tests. Logically a policy becomes true, if all tests are true. We immediately see, that a policy is a logical AND expression of its tests. ADQL expresses its policies in conjunctive normal form.

4 LANGUAGE INTRODUCTION

ADQL's syntax is divided into four major parts: creation statements, deletion statements, query statements, and access check statements. We will not describe deletion statements further in this section, but focus on creation statements. Nevertheless, consistency restrictions apply, so that entities can only be deleted, if they are not used in another term of ADQL. Query statements exist to provide information about the current model, facts and policies. With these statements the current definitions of the layers can be retrieved in order to manage model, facts and policies. We will also omit them here.

Creation Statements. We provide examples:

```
CREATE CONTAINERS users, permissions,
    files;
CREATE ENTITIES {Ann, Jim, Liz, read, write
    , f1};
CREATE ASSIGNMENTS users:{Ann, Jim, Liz},
  permissions:{read, write}, files:{f1};
```

The first statement creates three entities of the type container. This creation statement belongs to the model layer: We define, what types of entities may be defined subsequently; e.g. users, permissions, files.

The second statement creates 6 entities with so far no assignment to a container.

The third statement assigns Ann, Jim, and Liz to the container “users”. The entities read and write “are” permissions, the entity “f1” is a file. These definitions belong to the facts layer. While the model layer defines that concepts like “users”, “permissions” and “files” do exist, it is defined on the facts layer, that “Ann” is an entity of the concept “users”.

ADQL supports container hierarchies: It is possible to assign entities in two ways, directly or indirectly. If assigned directly, only the entity itself is assigned. If assigned indirectly not the entity itself is assigned but its value, the ”entity as a container“.

```
CREATE CONTAINERS usersA:{Ann, Jim},
       usersB:{Liz};
CREATE CONTAINERS
       alluser:{(usersA),(usersB)},
       usersets:{usersA, usersB};
```

The container *alluser* consists of two indirect assignments (note the round brackets). If resolved in depth the container *alluser* resolves to $\{Ann, Jim, Liz\}$. *usersets* resolves to $\{usersA, usersB\}$.

```
CREATE RELATIONS owner(files, users);
CREATE LINKS ON owner:
       {(f1, Ann),(f1, Jim)};
```

The 2-ary relation named “owner” is defined on the container “files” and “users”. The first element of any tuple of the relation “owner” has to be from the container “files”, the second from the container “users”. We define one or more owners for files. Actually, we define a n:m-relationship allowing that a user is the owner of several (or no) files and the files are owned by several (or no) users. Formally, the relation owner defines possible 2-tuples in the form (m, n) with $m \in files, n \in users$. The creation of the concept “owner” belongs to the model layer: It is defined there that such a concept as an owner exists. It does not say, who is the owner of which file.

The latter is defined by the second statement: “f1” is owned by “Ann” and “Jim”. This statement belongs to the facts base.

```
CREATE RELATIONS proxy(users, users):
       {(Ann, Jim)};
```

Our last example is a relation defined on $users \times users$. It says, that Ann is a proxy of Jim.

Let us now focus on the policy layer.

```
CREATE TESTS user_is_Ann:
       ([users],{Ann},theta);
```

A test is a boolean function. The definition of a test has three arguments, the first two are mandatory. The first two are expressions which are evaluated to a container. “users” (without any brackets) is a valid container name, which is evaluated to all entities within the container ”users”. “{users}” is a fixed

container, namely the entity users itself. This term does not evaluate any further. "[users]" is a reference to the variable defined upon the container users. The variable can be bound to any entity belonging to the underlying container. A variable can be bound in the context of a scope. Semantically, "[users]" can be interpreted as "the current user, when a decision is made to grant or deny access".

The above test definition compares the current binding of the variable of the container users with the fixed container including entity "Ann". As the third operator of a test θ is used as operator by default.

We define θ as a boolean function. Let C_1 and C_2 be containers. $\theta : C_1 \times C_2 \rightarrow \{true, false\}$

$$C_1 \theta C_2 = \begin{cases} true, & \text{if } C_1 \cap C_2 \neq \emptyset \\ false, & \text{otherwise} \end{cases}$$

The operator θ is a 2-ary boolean operator. It returns true, if both containers have in common at least one entity.

We see, the above test compares the current binding of the variable "[users]" with the fixed container including Ann, if they have in common at least one element. It is clear that this can be only the case if "[users]" includes Ann. In other words: "Does the container of the current user have at least one element in common with Ann?" Is the current user Ann?

```
CREATE TESTS user_is_owner: ([users],
 owner([files],.), theta);
CREATE TESTS user_proxy:
 (proxy([users],.), owner([files],.));
```

The first argument of the test "user_is_owner" is the current user. The second argument is a projection: The relation "owner" is projected towards its second argument. The relation "owner" has been defined by the Cartesian product of the containers $files \times users$. The point "." marks the target container of the projection while all other arguments have to be fixed entities (we refer to section 3). The container "files" is bound by the variable "[files]". Within a context – see later – a variable is always bound to certain entities. The result of this projection are all users who are linked to the current file (expressed by the variable file) by the relation owner. In other words, the relation returns all owners of the current file. The third argument is theta again. The test "user_is_owner" compares the current user with the owners of the current file. It returns true, if there is at least one match. In other words, the test checks whether the current user owns the current file.

The test "user_proxy" becomes true, if the current user is the proxy of the file owner. These tests can be used to allow the proxy of a user the same access rights as the file owner.

Let us come to policies. ADQL represents policies as sets (containers) of tests, subsequently applying the same syntax. Logically, a policy is an AND-combinations of all its tests. As each test can be evaluated true or false within a context, the same is true for a policy. It evaluates true if all tests evaluate true, it becomes false if at least one test becomes false.

```
CREATE POLICY proxy_can_write:
 {user_proxy, ([permissions],{write})};
```

The above policy becomes true if (1) the test user_proxy evaluates true, which is the case, if the current user is a proxy of the file owner. (2) the requested permission is write. The latter test is defined on-the-fly within the policy. In other words, write access is granted for the proxy of a file owner.

Let $P_1, P_2, ..., P_n$ all policies currently defined. Let P be the set of all policies. P is then the policy set.

Access Check Statements. Access statements are used to check whether access is granted or not. Access checks are performed on P, thus on all policies currently defined. If one policy is found allowing access, access is granted. If no policy returns true, access is denied.

Access checks need to be evaluated in the context of a scope. Within a scope, variables can be bound to fixed containers.

```
CHECK ACCESS: {[users]={Ann},
 [permissions]={read}, [files]={f1}};
```

The latter part within the curly brackets is the scope definition: The variable users is bound to the fixed single-element container holding the entity Ann. "[users]" is bound to Ann, "[permissions]" to read and "[files]" to f1. We assume, that all bindings are valid.

The inference engine checks if an access condition can be found, which evaluates to true. In our example there is only one policy "proxy_can_write". It consists of two tests. Both have to evaluate to true to allow access.

The BNF of ADQL and further details can be found on the project's homepage: http://iism.kit.edu/em/ref/adql.

5 EXAMPLE USE CASES OF ADQL

To show the flexibility of ADQL we provide some examples of standard access control models (Bell-LaPadula, SAP/R3) and how they are modeled in ADQL. We show the expressive power of ADQL with a third example related to an extended RBAC model.

5.1 Bell-LaPadula

The Bell and LaPadula model (Bell and LaPadula, 1975; McLean, 1988) assigns to principals (users) so-called security levels. The same happens to objects. A user may read (access) objects only, if his clearance level (his security level) is not smaller than the security level of the accessed object. This is called the read-down principle. On the other hand, a principal can write objects only, when assigning them a clearance level not smaller than his own security level. This rule is called write-up policy and prevents the leaking of confidential information.

```
CREATE CONTAINERS principals, objects,
    permissions, seclvls;
CREATE ENTITIES permissions:
 {read, write};
CREATE RELATION
 plevel(principals, seclvls),
 olevel(objects, seclvls);
CREATE POLICY read_down:
 {(plevel([principals],.),
   olevel([objects],.), >=),
 ([permissions],{read})};
CREATE POLICY write_up:
 {(plevel([principals],.),
   olevel([objects],.), <=),
 ([permissions],{write})};
```

The first command creates containers for “principals”, “objects”, “permissions”, and “security levels”. The principals will hold all entities representing users, the objects all protectable objects. The second command assumes two possible permissions, “read” and “write”. The third and fourth command say that security levels can be assigned to “principals” (“plevel”) and “objects” (“olevel”). We introduce two policies: First, we allow read down, that is, if the principals’ security level is larger or equal than the object’s security level and the requested permission is read, access is granted. Second, the write down permission is defined in a similar manner.

We did not yet provide a definition for the operators $<=$ and $>=$. Let $C_1 \in C$ be a decomposed ADQL container with entities $c_1, ..., c_n$, C being the set of all containers. Let $\mathcal{R}_\infty = \mathcal{R} \cup \{-\infty, +\infty\}$ with $\mathcal{R}$ being all real numbers in the usual way. It holds $\forall k \in \mathcal{R}$: $k > -\infty$ and $k < +\infty$.

Let $max : C \to \mathcal{R}_\infty$

$$max(C_1) = \begin{cases} -\infty, & \text{if } C_1 = \{\} \text{ or } \forall c_i \notin \mathcal{R}, \\ c_i, & \text{if } c_i, c_j \in \mathcal{R} \text{ and } \forall j : c_i \geq c_j \end{cases}$$

The *max* functions selects the entity with the maximal value of a container. If no or no entity as a real number is present, *max* returns $-\infty$. A *min* function is defined accordingly.

We define the ADQL operator $<=$: Let $C_i, C_j \in C$ be two ADQL containers. The test $(C_i, C_j, <=)$ evaluates the following way:

$$(C_i, C_j, <=) \mapsto \begin{cases} true, & \text{if } max(C_i) \leq min(C_j), \\ false, & \text{else} \end{cases}$$

An ADQL test with the $<=$ operator returns true, if the largest value of an entity of the first container is at most of the same value as the lowest value of an entity of the second container.

With these two policies, the whole Bell-LaPadula model is modeled in ADQL.

5.2 SAP R/3

Our second example is SAP’s R/3 system. The R/3 system[3] is an enterprise resource planning (ERP) system which is widely used by companies. The design of SAP R/3’s access control architecture is depicted in figure 2. We refer to (AG, 2008).

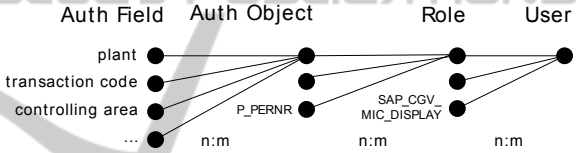

Figure 2: Structure of the SAP R/3 access control model. We provide some examples for possible object names (e.g. P_PERNR).

The related ADQL model is defined by the following commands:

```
CREATE CONTAINERS authfields,
    authobjects, roles, users;
CREATE RELATIONS
 obj_field (authobjects, authfields),
 role_obj (roles, authobjects),
 user_role (users, roles);
CREATE POLICY access: {([authfields],
  obj_field(role_obj(user_role([users
      ],.),.),.)
 )};
```

We immediately see, that SAP R/3 requires only a single policy. Containers are created for all entities of the access control model, thus “authfields”, “authobjects”, “roles” and “users”. The following commands link the container entities by relations and model the n:m relationships. Then, only a single access condition is required: If the required authfield value is available in the user’s roles, which consists of authobjects, that consist of authfields, access is granted. A

[3] http://www.sap.com/solutions/business-suite/erp/index.epx

SAP R/3 transaction – a transaction is a program in R/3 – has to check the access rights for all its necessary authorization fields. Please note the nested projections.

5.3 e-Science for Students' Thesis

Our next example shows the access control model for an e-Science environment which supports students in writing their Bachelor or Master Thesis. Universities organize the writing of a Bachelor or Master Thesis as a joint university-company project under the joint supervision of a professor, the professor's assistants and company employees. A project usually lasts 3 months. After a project's official end the state of the project (and all related documents) should be frozen. After a grace period of normally additional four weeks non-university members loose access to all project resources. For competitive reasons, project related documents should only be accessible to project-related members, and only project members may upload project documents. On the other hand, some of the company documents are restricted to company members and the student. With regard to his own documents, the student initially keeps his documents private. However, as the project progresses, he makes documents accessible either to all project members or to university or company members. In any case, the owner of a document can always read it and change the document's visibility.

The Model Layer. In figure 3 we define several containers to group the users: university staff, students, and company employees.

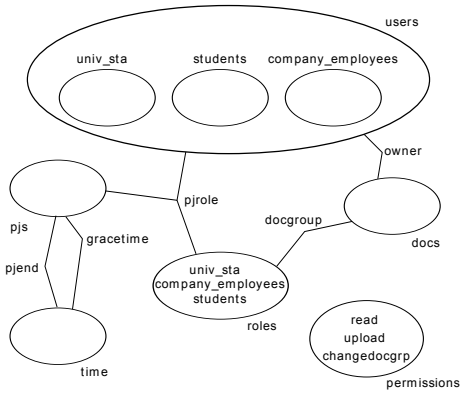

Figure 3: Graphical representation of the model layer for the e-Science scenario. Containers are depicted as ellipses, relations as lines. "pj" is an abbreviation for project.

Further, we need a container for projects, documents, and permissions. To model the end date and the grace period we need a time container and two relations. The grace time could also be expressed by adding four weeks to the project end date. Nevertheless, we model it explicitly which allows grace times to vary. As every document has an owner, we define such a relation. Because a user may have a role dependent on the project, we model a project role as 3-tuple. This allows us to assign users in different roles for each project. Alternatively, one could identify the role of a user in a project by the user's group membership, which would then be fixed for all projects. Documents can be visible to students, university staff, the company or the author only. We model a document group for this reason.

```
CREATE CONTAINERS users, univ_staff,
    students, company_employees;
CREATE CONTAINERS pjs, time, docs;
CREATE CONTAINERS permissions:{read,
    upload, changedocgrp};
CREATE CONTAINERS roles:{univ_staff,
    students, company_employees};
CREATE RELATIONS pjend(pjs, time),
 gracetime(pjs, time),
 owner(docs, users),
 docgroup(docs, roles),
 pjrole(users, pj, roles);
CREATE ASSIGNMENTS users:
 {(univ_staff),(students),
  (company_employees)};
```

All commands are self-explanatory, but the last: The container users gets assigned univ_staff, students and company_employees indirectly. When evaluating the entities in "users" all entities of the three subcontainers are returned.

The Policy Layer. We define the tests and policies.

```
CREATE TESTS perm_read:
 ([permissions],{read});
CREATE TESTS ingrace:
 ([time],gracetime([pj],.),<);
CREATE POLICY read_if_pjrole:
 {perm_read, ingrace,
   (pjrole([users],[pj],.),
    docgroup([docs],.), theta)
 };
```

The policy permits access, if (1) the current variable binding of permission has a non-emtpy intersection with $\{read\}$, in other words, the requested permission is read, (2) the current variable binding for time is less than the value of the relation *gracetime*, projected by the current project: $gracetime([pjs],.)$ is resolved to the assigned grace period(s) of the current project, (3) the assigned project role of the current project and user matches the document group of the current document. $pjrole([users],[pjs],.)$ is the

projection of the relation pjrole by the current project and user to the assigned roles. *docgroup*([*docs*],.) is the projection of the relation docgroup by the current document to the assigned role(s).

```
CREATE TESTS perm_up:
  ([permissions],{upload});
CREATE POLICY upload:
  {perm_up,
    ([time],endtime([pjs],.),<),
    (pjrole([users],[pjs],.),roles)};
```

Uploads are permitted, if (1) the requested permission is upload, (2) the project has not been finished, (3) the user in the project has at least one role. More in detail: If the projection of the relation pjrole by the current project and user (which is resolved to all roles the users has in the project), matches any element in the set role. Please note the missing square brackets with the expression role.

```
CREATE POLICY univ_gracetime:
  {perm_up, ingrace,
     (pjrole([users],[pjs],.),
     {univ_staff}, theta)
  };
```

The above access condition allows uploads within the grace period if the project role of the user is university staff.

```
CREATE POLICY owner_assign:
{([permissions],{changedocgrp,read}),
  ingrace, (owner([docs],.),[users]};
```

The last access conditions allows to change the visibility for the owner of a file as long as the project is in its grace period.

Some Example Facts for the Facts Layer. We provide some sample facts to be able to check access requests. There is one professor called Herb who has assistants Tom and Ulrick. The students Mark and Ann do their Bachelor thesis with Tom, respectively, Ulrick as their supervisor. Mark writes his thesis with the CRM Ltd. At the company his boss is Ben. Ann cooperates with the EM AG, where Jim is her boss. Mark creates two documents A and B. The first one is a working document and, therefore, private. The B document is visible for his complete project. Ann has uploaded one document C which shall be visible to her and the company only.

```
CREATE ENTITIES
  univ_staff:{Herb,Tom,Ulrick},
  students:{Mark,Ann},
  company_employees:{Ben,Jim},
  pjs:{CRM1,EM1};
CREATE LINKS pjrole:
  {(Mark,CRM1,students),
   (Ben,CRM1,company_employees),
   (Tom,CRM1,univ_staff),
   (Ann,EM1,students),
   (Ulrick,EM1,univ_staff),
   (Jim,EM1,company_employees)};
CREATE ENTITIES docs:{A,B,C};
CREATE LINKS
  owner:{(A,Mark),(B,Mark),(C,Ann)},
  docgroup:{(B,students),(B,univ_staff),
    (B,company_employees),
    (C,company_employees)};
```

Check Access Requests. With the above model, policies, and facts a check access requests can be performed:

```
CHECK ACCESS:
  ([users]={Tom}, [pjs]={CRM1},
   [docs]={B}, [permissions]={read},
   [time]={1300700213});
```

With this access request Tom is asking for document B by read access in the context of project CRM1 on the given date. We assume that this date is before the project's official end.

We evaluate the policy "read_if_pjrole": The test perm_read is *true*, ingrace is *true*. $pjrole([users],[pjs],.)$ evolves to $pjrole([Tom],[CRM1],.) = \{univ\_staff\}$. $docgroup([docs],.)$ is $docgroup([B],.) = \{univ\_staff, students, company\_employees\}$. The θ-operator evaluates to true.

As all tests of the access condition are true, access is granted. Further access conditions need not to be checked.

6 CONCLUSIONS AND FUTURE WORK

In this paper we suggested a formal language called Access Definition and Query Language. ADQL is located on the meta-model of an access control model: It allows to configure the model to behave like well-known access control models. We have shown this for Bell-LaPadula, SAP R/3, and an example of an extended RBAC-like model for e-science.

The model layer of ADQL distinguishes ADQL from other access control mechanisms: It provides the capability of freely defining the entities and their relations used by the access control mechanisms. We designed and implemented ADQL as a software service which allows to be used as external component for software engineers.

ADQL has been implemented as software service using Java v1.6. It can be managed by an ADQL web interface allowing to communicate with the ADQL

core. The ADQL core can currently be accessed by REST, SOAP, OSGi, Thrift, string literal IP interface, and a serialized objects IP interface. The network server layer sends all commands received to the parser. From there, the API layer is called. The API layer communicates with the ADQL core layer, where the logic and internal processes are done. Persistence is realized by standard relational database components. We currently use Hibernate, Hibernate Cache and Postgresql. Further details about ADQL can be found at http://iism.kit.edu/em/ref/adql.

Our future work will focus on extensions for ADQL and its exploitation.

REFERENCES

AG, S. (2008). *ADM940 - Berechtigungskonzept AS ABAP Schulungshandbuch*. SAP, Walldorf, Germany.

Ardagna, C. A., Cremonini, M., De Capitani di Vimercati, S., and Samarati, P. (2008). A privacy-aware access control system. *J. Comput. Secur.*, 16(4):369–397.

Barker, S. (2009). The next 700 access control models or a unifying meta-model? In *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, pages 187–196, New York, NY, USA. ACM.

Bell, D. E. and LaPadula, L. J. (1975). *Secure Computer Systems: Mathematical Foundations and Model*. M74-244. Mitre Corporation, Bedford, MA, USA.

Berners-Lee, T., Fielding, R., Irvine, U., and Masinter, L. (1998). Uniform resource identifiers (URI): generic syntax. http://www.ietf.org/rfc/rfc2396.txt. last accessed: 2011-02-26.

Crampton, J. and Huth, M. (2010). An authorization framework resilient to policy evaluation failures. In Gritzalis, D., Preneel, B., and Theoharidou, M., editors, *Computer Security ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 472–487. Springer Berlin / Heidelberg. 10.1007/978-3-642-15497-3_29.

Damiani, E., di Vimercati, S. D. C., Paraboschi, S., and Samarati, P. (2002). A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security (TISSEC)*, 5:169–202. ACM ID: 505590.

Geysin, S., Petrov, A., Charrue, P., Gajewski, W., Kain, V., Kostro, K., Kruk, G., Page, S., and Peryt, M. (2007). Role-Based access control for the accelerator control system at CERN. In *International Conference on Accelerator and Large Experimental Physics Control Systems*, pages 90–92, Knoxville, Tennessee, USA.

Gupta, R. and Bhide, M. (2005). A Generic XACML Based Declarative Authorization Scheme for Java. In di Vimercati, S. d. C., Syverson, P., and Gollmann, D., editors, *Computer Security ESORICS 2005*, volume 3679 of *Lecture Notes in Computer Science*, pages 44–63. Springer Berlin / Heidelberg.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Communications of the ACM*, 19(8):461–471.

Li, N., Wang, Q., Qardaji, W., Bertino, E., Rao, P., Lobo, J., and Lin, D. (2009). Access control policy combining: theory meets practice. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, pages 135–144, New York, NY, USA. ACM.

McLean, J. (1988). The algebra of security. In *IEEE Symposium on Security and Privacy*, Oakland, CA.

Ni, Q., Bertino, E., and Lobo, J. (2009). D-algebra for composing access control policy decisions. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 298–309, New York, NY, USA. ACM.

Notargiacomo, L. (1996). Role-based access control in ORACLE7 and trusted ORACLE7. In *Proceedings of the first ACM Workshop on Role-based access control*, RBAC '95, New York, NY, USA. ACM.

Rissanen, E. (2010). *eXtensible Access Control Markup Language (XACML) Version 3.0 Committee Draft 03*. OASIS eXtensible Access Control Markup Language (XACML) TC.

Samarati, P. and Vimercati, S. D. C. d. (2001). Access control: Policies, models, and mechanisms. In *Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures*, pages 137–196. Springer-Verlag.

Wang, Q., Jin, H., and Li, N. (2009). Usable access control in collaborative environments: authorization based on people-tagging. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 268–284, Berlin, Heidelberg. Springer-Verlag.

Yuan, E. and Tong, J. (2005). Attributed based access control (ABAC) for web services. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 569–578.