Fine-Granularity Access Control in 3-tier Laboratory Information Systems

Xueli Li* Nomair A. Naeem+ Bettina Kemme+

* Macromolecular Structure Group, Biotechn. Research Institute, National Research Council of Canada

+ School of Computer Science, McGill University, Montreal

Abstract

Laboratory information systems (LIMS) are used in life science research to manage complex experiments. Since LIMS systems are often shared by different research groups, powerful access control is needed to allow different access rights to different records of the same table. Traditional access control models that define a permission as the right of a user/role to perform a specific operation on a specific object cannot handle the enormous amount of objects and user/roles. In this paper we propose an enhancement to role-based access control by introducing conditions that can be added to the traditional concept of permissions in order to keep the number of permissions small. Furthermore, we present an implementation of our access control model at the application programming level. Although access control is performed for every single database access, our solution completely separates access control from the application logic by using aspect-oriented programming. With this, access control can be integrated into a legacy 3-tier information system without changing the application programs.

1. Introduction

Laboratory information systems (LIMS), e.g., [9, 11, 16, 10, 12], have become a crucial asset in life science research to manage the setup and execution of complex experiments, and to analyze the resulting data. Many recent systems are based on a three-tier architecture. Access is via the web, the application programs reside within an application server, and the data is stored within a database system.

With the fast development of such systems comes the need for equally sophisticated access control since research groups often collaborate in research projects. This requires to set up LIMS systems that are shared by different groups so that they all have access to common or project related data. But at the same time, each group wants to protect its private data from other groups. For instance, only project members might be allowed to access project related data before it becomes mature and stable. Furthermore, each individual member of a research group might have different responsibilities, and hence, access rights to the data. Although many powerful access control models exist, many of the current LIMS systems use quite simple access control. LIMS often start as small systems being used by a single group, and hence have not been developed with sophisticated access control in mind. As a result, integrating access control into such legacy systems is a challenge.

In this paper we propose a practical solution to integrate sophisticated access control into three-tier information systems. Although we focus on LIMS systems, we believe that the main concepts introduced in this paper can be used for other three-tier information systems. In particular, our paper makes two major contributions.

Firstly, our solution enhances the traditional role-based access control model (RBAC) [20] by introducing the concept of conditions. Conditions allow the specification of fine-granularity access control policies in a multi-level approach. The idea is to keep objects on a rather coarse level, i.e., an object is a database table, and a permission is the right of a role r to perform a certain operation op on a certain table t. Additionally, conditions are attached to this permission. If a user with role r wants to perform op on a data record of t, then the op is only allowed if the conditions attached to the permission are fulfilled. These conditions are checked at runtime, when the concrete records are accessed. This allows us to keep the number of permissions small (since they are on coarse objects) but still allows us to check access on the level of individual records.

Secondly, we present an elegant and modular way to integrate our access control module into an existing system. In principle, checking conditions can be performed in form of triggers within the database system. However, this requires all information needed to check conditions to reside in the database which might be difficult to achieve for session related data (e.g., user currently logged in). The second alternative is to perform access control within the application programs. For example, one can directly include access control measures into the interface provided to the user and/or the SQL methods sent to the database system. For instance, a student logged into a student database may only see a single link “my data”. At the same time the id of the student is maintained in a session variable sid, and all SQL statements accessing the student information contain a `WHERE student_id = sid` clause guaranteeing the student can only see his/her own data. This approach is quite inflexible since a change in access control requirements might require major changes to the application code.

In contrast, we propose an access control module implemented in the application layer but separated from the original application programs. This is achieved by using an aspect-oriented programming language like AspectJ [15]. In AspectJ, the developer of the access control module defines the routines to check data access, and specifies in a declarative manner at which points in the application programs these routines should be called. The application programmer, on the other hand, does not need to consider access control in his/her development. This allows a quite easy integration of access control into an existing system without changing the existing code. We are not aware of any other access control implementation that allows for such modular integration of access control into legacy systems.

In the rest of the paper, Section 2 provides an example LIMS application, and the need for access control. Section 3 describes our role based access control design using conditions. Section 4 presents our aspect-oriented implementation of access control. Section 5 discusses related work, and Section 6 concludes the paper.

2. Example Application

In order to illustrate our approach and show its feasibility in practice, we have taken a simple LIMS system, called Exp-DB [16], and extended it by an access control module. Exp-DB is an example application from the life sciences that keeps track of experimental data on protein expression, purification, crystallization and determination of three-dimensional structure of the protein. In this section, we outline the database design and the architecture of Exp-DB, and the tasks individual users perform. Then, we discuss how user access to the LIMS system should be restricted.

2.1. Data Model

Figure 1 shows a simplified version of the entity-relationship model (syntax taken from [18]) proposed by an ongoing standardization process among scientists conducting crystallography research [17]. The data model presented here is very simplified and leaves out many entitiy and relationship sets. All experiments are conducted in the context of projects. The `Project` table lists general information about all projects. The `Experiment` table contains

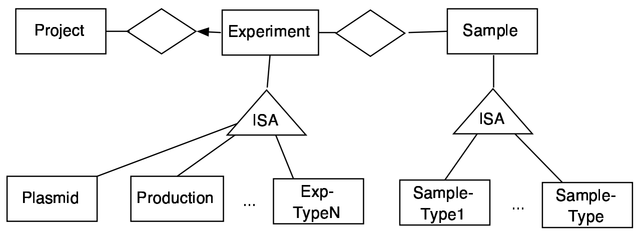


Figure 1. Data Model

common information about each individual experiment conducted. For instance, each experiment has a unique identifier, and start and end dates. Many different types of experiments are conducted, often in a certain sequence. For instance, in our example application the first two steps of a typical experiment workflow are to insert a DNA sequence into a plasmid (a DNA molecule of a bacteria), and to then import the plasmid into a host cell (often Ecoli) that produces many copies of the corresponding protein. Further steps produce protein crystals and analyze them via X-ray to determine the 3D structure. Other experiment types and experiment suites exist. For each experiment type (e.g. Plasmid, Production), a special entity set stores the information about experiments of that specific type. Input and output data of experiments is recorded in the `Sample` table and belongs to a sample type. A sample can be output of one and input of other experiments. There also exist tables not related to particular projects (e.g., general information about chemicals used in experiments). We defer the description of user related tables to Section 3.

2.2. Architecture

Figure 2 shows Exp-DB’s three-tier architecture. Users (client tier) interact with the system through a standard browser. The middle tier is responsible for the presentation and application semantics. The runtime environment is an Apache Tomcat 4 server using Java Technology [13]. The backend tier can be any relational database system (currently PostgreSQL). The internal architecture of the middle tier follows the well-known MVC (Model View Controller) pattern. The presentation logic generates the webpages (*view*) using JavaServer Pages (JSP). The application logic is implemented as a combination of Java Servlets and JavaBeans. The JavaBeans (*model*) encapsulate the data access to the database system and represent the data in an abstract interface to JSPs and Servlets. Finally, the Servlets *control* how the different components should be called.

The system consists of several modules implementing different functionality. Users can modify experiment or sample data, modify non-experiment tables, search the database for entries and submit queries, or ask for access to

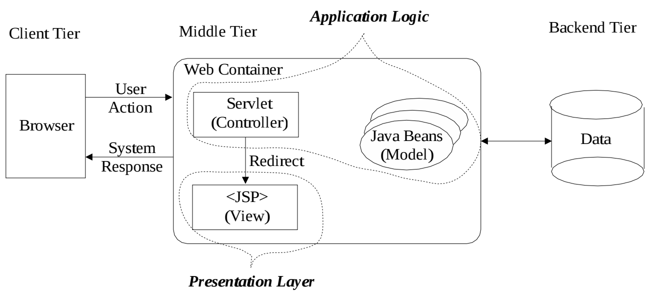


Figure 2. Architecture of Exp-DB

a specific database entry. All requests are filtered through a controller servlet that forwards it to the corresponding module. A special module is the login module. Only registered users can access the system. Otherwise, there was no access control within Exp-DB. Once a user has logged in, he/she is referred to an initial web-page that provides him/her with the different functionality choices.

2.3. Roles and Tasks

Our work has been motivated by the needs of the Maromolecular Structure Group of the Biotechnology Research Institute (BRI) at the National Research Council of Canada (NRC). This group works in protein crystallograhpy and and uses a LIMS system that follows a similar data model and architecture as described above. They want to open their system to external research groups with which they collaborate. As such, they are very concerned about protecting their data from illegal use. Below is the description of the different users, their roles, and how they should be restricted in their access to the data. We believe that these requirements are similar in nature for many different scientific systems, and we are not aware of any other publication that would present such requirements in such detail.

The basic operations on the data are the usual `read`, `insert`, `update`, and `delete`. Additionally, some specific operations exist. For instance, after a project is successfully completed, all project related data becomes mature, and should not be changed anymore. For this, the relevant tables have an additional attribute `status`. Upon insertion of a record, this attribute is set to “unfixed”, and can later be changed to “fixed” by the `fix` operator. As such, `fix` is an update operation with a special meaning.

The users are the subjects of the system and depending on their role (i.e., job), they have to perform certain job functions. For instance, initiating a new project is a job function of a group leader. Functions translate into permissions to perform operations on the data records and tables. For instance, project initiation requires to insert a data record into the project table. We have elaborated several roles that can be defined along two scopes. Most of the data in the system belongs to a project, and a user might be restricted to only access data of specific projects. Furthermore, users are divided into research groups. We might allow a group member to read data that was created by members of his/her group. Hence, this defines a project and a group scope. There might be roles that do not fit in any of the two categories, like a general system administrator.

Role	Job functions
Admin.	can do everything
Head	(i) insert new users and groups (ii) divide users into groups (ii) assign group leaders
Group Member	read all fixed data of projects his/her group is participating
Group Leader	(i) add new users to his/her own group (i) initiate projects (iii) assign groups to project (iv) decide on project leader (v) do all operations on initiated projects
Project Leader	(i) decide on which of his/her group members participate in project (ii) do all operations on specific project
Project technican	(i) insert data into project (ii) update/delete own data
Proj. reader	read all fixed data of project
Internet user	read public attributes of some data

Table 1. Roles

The roles we determined for the needs of the Macromolecular Structure Group, and their corresponding job functions are depicted in Table 1. Some functions translate to permissions that allow an operation on a table (e.g., insert a new project), others translate to permissions that allow an operation on specific data records within different tables (e.g., update all records associated with a specific project). Some job functions are of administrative nature. For instance, assigning a group member to a project assigns the role project technician to a user. Other tasks are on the standard experimental data. In our implementation, we have separated these two types of tasks as common for RBAC but for simplicity, we do not discuss this separation here.

We will only discuss some job functions in more detail. Within each group, there is usually one *group leader*. He/she can initiate a new project, and invite other groups to collaborate on the same project. Each group participating in a certain project has its own *project leader* assigned by the group leader of this group. There are several *project technicians* in a project responsible for the detailed technical work. A project technician can insert a project related record into an experiment table, but may only update and delete data inserted by him/herself, and may not fix a record.

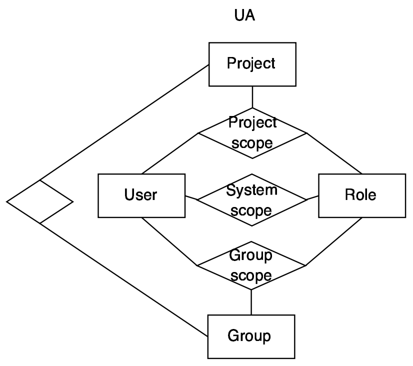


Figure 3. User/role assignment (UA)

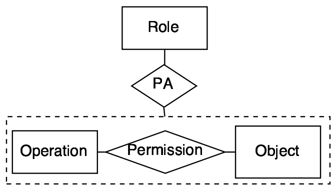
Some specific users might be given the right to access mature parts of given projects. We call this role *project reader*. After the project is completed, we would like to show the results to the public. For this, we introduce the *internet user*, who is only able to read public attributes of fixed data. A user may be entitled to more than one role. One can be group member of group g1 and project technician of project p2. In the current system we have not specified role hierarchies since the number of roles is relatively small.

Group leader and group member have *group scope*. Project leader, project technician, and project reader have *project scope*. Administrator, head and internet reader have general *system scope*. This is, however, only one option of partitioning roles. In some cases, a project scope might be too big. Different technicians might only perform very specific experiments, and hence, will only be allowed to change data related to these experiment types. Clearly, it is quite easy to define such experiment scope.

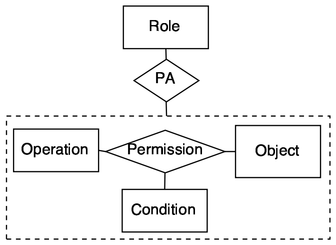
3. Access Control Design

Our approach is based on role based access control (RBAC) [6, 20, 8] which first assigns users to roles (UA assignment), and then assigns permissions to perform operations on objects to roles (PA assignment). We have extended the basic RBAC system in two ways. Firstly, we allow roles to have a group or project scope. Secondly, we indicate that permissions can be either on tables or data records within tables. We store most but not all data relevant for access control in the database in form of administrative tables. We describe our design using the entity-relationship model.

The UA assignment is depicted in Figure 3. There are not only users and roles but also projects and groups in order to capture the different role assignments. System level roles can be directly assigned to users. Group roles, however, are always associated with a group. Similar issues arise for project level roles. Therefore, group and project level roles are represented by a ternary relationship between user, role and project/group. For instance, user U1 is project technician in project P1, and project leader in project P2. Alternatively, we could have created a project technician and a project leader role for each project. With this, role would contain entries “project technician P1”, “project leader P1”, “project technician P2”, etc. Then, there would not be a project entity set, and users would only be assigned to roles. The disadvantage is that we would have many more roles leading to many more PA assignments. However, the PA assignments for all roles of the same type, e.g., all project technician roles, are very similar in nature. Hence, it would lead to redundancy if we indicated the PA assignments individually for roles of the same type.



(a) traditional



(b) adjusted

Figure 4. Role/Permission assignment (PA)

Figure 4(a) shows the traditional way to assign a permission (association of an operation and an object) to a role in form of an aggregation (see [18] for aggregation). If we choose this option we can use tables and/or data records as our granularity for an object. For instance, project initiation translates to the permission for a project leader to insert into the project table. For some of the permissions, such a table level assignment is enough. However, many other permissions have conditions associated with them. For example, a project leader can assign only users of his/her own group the role of project technician. Also, he/she can perform all operations on experiment related tables, however, only on data records that belong to the project for which he/she is the project leader. The complexity of these conditions can be arbitrary. Some of the conditions can be easily checked within the database system. For instance, data that is fixed cannot be updated anymore. To check this condition we only have to look at the status attribute. However, other conditions might require runtime dependent data that is not stored in the database, for instance, the user id of the user that attempts to perform the operation. The conditions might also require some complex joins on several tables. As such, we have decided to implement the programming logic that tests for such conditions outside the database system, that is, within the middle-tier of the 3-tier architecture. Within the database, however, we keep track of which conditions exist. For each condition, there will be one program that will test whether the condition is true. Figure 4(b) now shows the permission as a ternary relationship set connecting operation, object (a table), and condition. The role is again associated with the permission in form of an aggregation. Condition is simply a textual description. For instance, for a project leader to update an experiment related record, there are the conditions "data record may not be fixed", and "user must be involved in project data record belongs to". A project technician must additionally fulfill the condition "user must be the one that inserted the data record". Some permissions do not have any conditions associated with it. For instance, the head can enter users into the system. Such permissions have a dummy condition "none".

When a user with a specific role does a specific operation on a specific data record, the associated conditions can be easily found in the PA table that results when translating the model into relations[1]. Conditions allow us to specify access control on the level of a data record, to include context information, and to use few quite general roles (group member, project technician) instead of many specific roles (project technician P1, project technician P2, etc.).

4. Access Control Implementation

4.1. Overview

Our enhancement of Exp-DB performs access control in three steps. After successful login, the system checks in the database which scope specific roles the user has and lets the user choose one of the roles. For instance, if a user is member of group 1, and project technician of project 1 and 2, then he/she can choose one of these three roles. He/she can later explicitly change the role if necessary. We say access control reaches *role level*.

From there, the user is only presented with tables and operations according to the PA assignment (ignoring conditions) of his/her current role. For instance, a group member may not access any administrative tables like user, role, etc. Hence, the web-based interface does not even show these tables. If a user is only allowed to read data from a table, the table appears on the web-page but no links to modifying operations are provided. Access control reaches *table level*.

Finally, when a user with role r attempts to perform a certain operation op on a specific data record $dr1$ of a table t, the success of op depends on the specified conditions. The system scans the PA table for entries (r, op, t, $someCondition$), and checks the conditions one by one. If all conditions are true, the operation succeeds. Otherwise the user receives a message indicating the first unsatisfied condition. We say the access control reaches *record level*. This step is the soul of the access control module and described in more detail in the next section.

This 3-step approach has several advantages. Firstly, working under one role allows the system to only show those tables and operations that are relevant for the role. This helps the user in navigating through the system for the specific task to be performed. Secondly, it supports the principle of least privilege suggested for RBAC. A user should always invoke the role that is most suitable for a given task. Powerful roles should only be invoked when needed. Finally, fixing the role simplifies what data access the system is supposed to control. When the user attempts to access a specific data record, the system already knows that the user has, in principle, the right to do such an operation on the table. The only thing that remains to be done is to check the associated conditions. If access is denied, a clear and meaningful error message can be returned.

4.2. Aspect-oriented implementation

Integrating access control into a legacy system is a challenging task since each data access requires access control, and data access might be spread across many modules. Hence, altering the existing code to include access control would require to perform changes in all these modules.

Aspect-oriented programming (AOP) [5], as provided, e.g., by the AspectJ programming language [15], provides a means to implement such a cross-cutting concern in a more elegant way. The access control developer implements all methods needed to perform access control in a special module, called *aspect*. One of these methods, could be, for instance, an `update-check` method. It takes as input context information (e.g., user and role), and the record identifier of the record to be updated, and checks whether the user has the permission to update the record according to the conditions. This method should be executed every time before an update takes place. The interweaving of the application code performing the update, and the access control method performing the check is done in a declarative manner. In AspectJ, the access control developer must first decide which methods in the application programs require access control. These methods are defined as *join points*. Then, he/she has to group join points that require the same type of access control (e.g., `update-check`). These groups are called *pointcuts*. Finally, he/she has to indicate what access control actions have to be performed for the join points in the pointcut and when these actions should be performed (e.g., before,

[1]The PA table represents the PA relationship sets and contains four attributes referring to role, operation, table, and condition.

after or instead of the execution of the join point method). This is called an *advice*. An aspect is the combination of join point, pointcut, and advice declarations. A special AspectJ compiler compiles application code and aspects into one common executable with interweaved calling structure.

Read Operations For read operations (SQL select) only those data records that fulfill all conditions should be returned to the user. However, in order to check conditions we must retrieve information like the status of the record or the project the record is associated to. Hence, we must first execute the SQL select statement and then remove those records from the result set that do not fulfill the conditions. If necessary we have to perform additional SQL statements to retrieve further information from other tables or extend the original SQL statement to retrieve attributes that we need in order to check the conditions. Let's have a look at a simplified example. Assume the application code has a single reading method `Vector read(UInfo info, String table, Vector check, Vector crit)`
which internally simply performs the SQL statement
`SELECT * FROM table WHERE check[1] = crit[1] AND check[2] = crit[2] AND ...`
Then, we define a pointcut with one join point as:

```
pointcut read_chkp (UInfo info, String table,
                    Vector check, Vector crit):
// read is a join point
execution(public Vector read(UInfo,String,Vector,
  Vector) && args(info,table,check,crit));
```

We pass the context information (e.g, current user and role) in form of the `Uinfo` object, and the parameters of the method to the pointcut. This allows the advice to use this information for its internal processing. The advice is the actual access control execution at the pointcut. As described above, for read operations, we have to retrieve first the data records and then only return those that fulfill all conditions. For this we use an `around` advice. It replaces the original execution of the join point method with the advice.

```
Vector around(UInfo info, String table, Vector res,
         Vector check, Vector crit):
read_chkp(...) {
 Vector res = proceed(info, table, check, crit);
 for each record in res check conditions
    if at least one not fulfilled remove from res
    else keep
 return res;}
```

The `proceed` in the advice calls the original read method to retrieve all records. After that, conditions are checked on each returned data record. In case we only need to check whether a data record is fixed, it is enough to look at the data record itself. Otherwise the check might require to perform additional SQL statements. For instance, in order to check that a user's group is involved in the project the data record belongs to, we need a statement like `SELECT project_id from project_group where group_id is this user's group_id.` For these nested read operations access control should not be triggered. AspectJ allows us to specify such behavior but we have omitted that in above example for simplicity.

The `read` method above is quite simple. A more complicated read method could indicate a subset of attributes to be selected. In this case, if we needed additional attributes to check conditions (for instance, the status attribute), we could perform an additional SQL statement to retrieve the additional attributes within the check loop. Alternatively, we call `proceed` with the attributes needed for access control added as input parameters. Similar issues arise for joins.

Write Operations For write operations, access control should be performed before the execution of the operation to avoid undo in case access is denied. In order to have all data necessary to perform checks the access control method might have to perform additional read operations if the data is not yet available within the application logic. However, we can expect that the old and new values of the written data record to be available already before the write operation takes place. This is trivially true for most inserts. For other operations, the user often first views a data record before modifying it. Again, we use the `around` advice to intercept write operations. We perform the checks, and if access is allowed, we call `proceed` to execute the operation, otherwise we return an error message.

Access Control in Exp-DB In Exp-DB, database access was controlled by few classes leading to only few pointcuts.

5. Related Work

We are not aware of any access control implementation that uses aspect-oriented programming (AOP). [7] provides a basic RBAC implementation but does not describe how to integrate it into a legacy system. [19] implements RBAC administration in Oracle via stored procedures. We are not aware of any LIMS system that implements RBAC.

Many application and web servers provide what is called "filters" or "interceptors" that could be an alternative to using an AOP language like AspectJ. The server intercepts client requests before they are forwarded to the appropriate method and responses before they are returned to the client. It is possible to inject access control at these interceptor points. That is, at the time of interception, the server gives control to the access control module which can perform some tasks before forwarding the request to the called component and before returning the response to the client. The basic concepts are similar to AOP. In case of the webserver used, Apache Tomcat, the filter technology was less powerful than AspectJ, and hence, we chose AspectJ.

Previous enhancements to the basic RBAC model introduce role hierarchies (RBAC1) and constraints (RBAC2)

[20]. We can consider our newly introduced conditions as a new form of constraint on permissions to achieve record-level access control. Parameterised roles allow a role to adjust to the current context. For instance, [1] presents a health care environment where a user with the role doctor may only see the entire patient's record if he/she is one of the treating doctors of that patient. That is, the parameters included in an access request are used to determine whether the policies associated with a permission are fulfilled. Such policies have similarities to our conditions. The authors propose to present all policies in a controlled English and then translate them to higher order logic. We believe that such automatism can only be used in specific cases. Instead, we are more pragmatic, testing conditions by application programs that can be as complex as necessary to perform the checks. Other "context-aware" approaches look at temporal conditions [3], environment roles [4], and context-based policy classification [2]. Although they all increase the flexibility of expressing constraints on permissions, none is really focused on providing a more fine-granularity access to data. In contrast, our 3-level approach provides a simple way to express permissions on tables, and a constrained based way to express restrictions to access specific records. In [14], objects can be grouped to views which is similar to grouping all data relevant to a given project. The authors propose a first-order language to express all conditions but do not explain how the current values for the parameters given in the condition formulas are determined during runtime. Similar to us, they probably need to retrieve them by reading different records from the database.

6. Conclusion

This paper enhances traditional RBAC offering record-based access control without an explosion on the number of permissions. This is achieved by defining permissions on tables but adding conditions that restrict roles to a subset of the records in the table. The conditions can be arbitrary complex, and hence, are able to reflect real-life constraints. Additionally, our RBAC model distinguishes between different scopes of roles, namely system, group, and project roles since belonging to a group or participating in a project often determines the data that can be accessed. The introduction of scopes allows us to keep the number of roles, and hence, the number of role/permission assignments small. We believe such scoping can be useful not only for scientific research in general, but possibly also for other applications.

Furthermore, we have integrated the proposed RBAC model into an existing legacy 3-tier information system using aspect-oriented programming. Although the legacy code itself has not been changed, access control is enforced whenever this legacy code performs database access.

References

[1] J. Bacon, M. Lloyd, and K. Moody. Translating role-based access control policy within context. In *POLICY*, 2001.

[2] A. Belokosztolszki, D. M. Eyers, and K. Moody. Policy contexts: Controlling information flow in parameterised RBAC. In *POLICY*, 2003.

[3] E. Bertino, P. A. Bonatti, and E. Ferrari. TRBAC: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3), 2001.

[4] M. J. Covington, W. Long, S. Srinivasan, A. K. Dey, M. Ahamad, and G. D. Abowd. Securing context-aware applications using environment roles. In *SACMAT*, 2001.

[5] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming - introduction. *Comm. of the ACM*, 44(10), 2001.

[6] D. Ferraiolo and D. Kuhn. Role based access control. In *National Computer Security Conference*, 1992.

[7] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Trans. Inf. Syst. Secur.*, 2(1), 1999.

[8] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House Publishers, 2003.

[9] J. Frew and R. Bose. Earth system science workbench: A data management infrascrutcure for earth science products. In *Int. Conf. on Statist. and Scient. Database Mgmt.*, 2001.

[10] N. Goodman, S. Rozen, L. D. Stein, and A. Smith. The LabBase system for data management in large scale biology research laboratories. *Bioinformatics*, 14, 1998.

[11] P. W. Haebel, V. L. Arcus, E. N. Baker, and P. Metcalf. LISA: an intranet-based flexible database for protein crystallography project management. *Acta Crystallogr. D57*, 2001.

[12] Y. E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: a desktop experiment management environment. In *VLDB Conference*, 1996.

[13] Java technology, http://java.sun.com/products/.

[14] A. A. E. Kalam, S. Benferhat, A. Miège, R. E. Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin. Organization based access control. In *POLICY*, 2003.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10), 2001.

[16] N. A. Naeem, S. Raymond, A. Poupon, M. Cygler, and B. Kemme. Exp-DB: Fast development of information systems for experiment tracking. In *Caise (Short Paper)*, 2003.

[17] A. Pajon, J. Ionides, J. Diprose, J. Fillon, R. Fogh, A. Ashton, H. Berman, W. Boucher, M. Cygler, E. Deleury, R. Esnouf, J. Janin, R. Kim, I. Krimm, C. Lawson, E. Oeuillet, A. Poupon, S. Raymond, T. Stevens, H. van Tilbeurgh, J. Westbrook, P. Wood, E. Ulrich, W. Vranken, X. Li, E. Laue, D. Stuart, and K. Henrick. Design of a data model for developing laboratory information management and analysis systems for protein production. *Proteins*, 58(2):278–284, 2005.

[18] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGrawHill, 3 edition, 2003.

[19] R. S. Sandhu and V. Bhamidipati. Role-based administration of user-role assignment: The URA97 model and its Oracle implementation. *Journal of Computer Security*, 7(4), 1999.

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