Authrule: A Generic Rule-Based Authorization Module

Sönke Busch[1], Björn Muschall[2], Günther Pernul[2], and Torsten Priebe[3]

[1] Booz Allen Hamilton GmbH, Zollhof 8, D-40221 Düsseldorf, Germany
[2] Department of Information Systems, University of Regensburg, D-93040 Regensburg, Germany
[3] Capgemini Consulting Österreich AG, Lassallestrae 9b, A-1020 Vienna, Austria

Abstract. As part of the access control process an authorization decision needs to be taken based on a certain authorization model. Depending on the environment different models are applicable (e.g., RBAC in organizations, MAC in the military field). An authorization model contains all necessary elements needed for the decision (e.g., subjects, objects, and roles) as well as their relations. As these elements are usually inherent in the software architecture of an access control module, such modules limit themselves to the use of a certain specific authorization model. A later change of the model consequently results in a substantial effort for revising the software architecture of the given module. Rule-based systems are well suited to represent authorization models by mapping them to facts and rules, which can be modified in a flexible manner. In this paper we present a generic authorization module, which can take authorization decisions on the basis of arbitrary models utilizing rule-based technology. The implementation of the popular RBAC and ABAC (attribute-based access control) models is demonstrated.

1 Introduction and Motivation

Depending on the environment of an IT system, there are varying requirements for its access control mechanism. These requirements determine which authorization model is adequate. For instance, for military purposes, the mandatory access control model (MAC) is favourable as it supports information flow control. This model is however inadequate for commercial purposes—in business IT systems, the mostly used authorization model is role-based access control (RBAC). For information services on the Internet, an attribute-based access control model (ABAC) might be the first choice due to the lack of stable role structures. It is important to note that there is no authorization model that is suitable for all different kinds of scenarios—the best suitable authorization model must be chosen depending on the requirements.

The usual approach for implementing an access control system is to first decide which authorization model matches the requirements and then develop a software module to implement this authorization model. This approach has a major drawback: The software is bound to a single authorization model. This

E. Damiani and P. Liu (Eds.): Data and Applications Security 2006, LNCS 4127, pp. 267–281, 2006.

means, that if the requirements change and if it is necessary to switch to a new authorization model, this would require substantial changes on the software.

This paper describes a generic approach that led to an authorization component that supports multiple authorization models and can easily be extended to support arbitrary authorization models. To accomplish this, a rule-based system is used to map authorization models to rule sets; an inference engine processes the authorization requests. Moving from one authorization model to another only requires minor changes on the software—different authorization models are represented as modules plus rule-base that can easily be replaced. To ensure that authorization models can be easily exchanged, the access control logic is kept in the separate authorization model and not interweaved with the business logic.

2 Fundamentals

2.1 Authorization Models

Authorization models are essential for access control. They represent all information that is needed to perform an authorization decision. This information consists of entities and their relation to each other. These entities and relations vary from model to model, but they have some commonality, like the subject (user) that requests the access, the object that is to be accessed and the operation that the subject requests to perform on the object. Operation and object together represent a permission. This paper focuses on two very popular and well elaborated authorization models, RBAC and ABAC.

The role based access control model (RBAC) is the de-facto standard for access control. The standardization process was initiated by the NIST [3]. It introduces an intermediation between user and permission, called role. Roles are assigned to users and permissions are assigned to roles. The NIST also defined two extensions of this basic Core RBAC model: Hierarchical RBAC and Constrained RBAC.

The attribute based access control model (ABAC) relies on attributes of the requesting subject as well as the object to perform the authorization decision. There is no common or standard model like for RBAC, but it can be found in many research works, e.g. DLAM [1,4] or UCON [8] as well as the XML access control language XACML [6]. An attempt to define a common ABAC reference model was made in [9][1],[10]. Permissions are assigned by defining which attributes a subject has to have in order to be able to access certain objects (with certain attributes). The ABAC model can be extended to take environment attributes into account or to directly compare the subject attributes with those of the object.

In general, an authorization decision is a simple yes/no query that can be described with a limited set of authorization rules. This characteristic allows representing authorization models in rule bases and performing authorization

[1] In this paper the authors refer to metadata-based access control (MBAC) instead of ABAC. However, the terminilogy has changed in more recent work.

decisions with inference engines. Such authorization module can therefore implement virtually any authorization model. In addition, the more complex an authorization model is structured, the more effectively rule based systems can perform their advantages over classical software architectures, that reflect the authorization model in their structure.

2.2 Rule-Based Systems

Rule-based systems are able to represent knowledge by storing structured information (called facts) and using rules to generate new facts from the already existing ones. Facts are represented by predicates, which—similar to tuples in relational databases—describe relations of some entities. A set of facts together with a set of corresponding rules is called a knowledge base. Queries can be run against this knowledge base by using an inference engine. When using a rule-based system for authorization decisions, the entities of the authorization model and their relations are represented by a set of facts. Additionally, rules are defined to state, under which circumstances access will be granted and when it will be denied. Authorization decisions are then performed by querying the knowledge base with a query like *granted(subject, object, operation)*.

Implementations of rule-based systems vary in their approach, flexibility and maturity. For the purpose of using a rule-based system as kernel for a Java-based authorization module, the Mandarax[2] distribution was chosen. Mandarax was found to fit best to our requirements. Mandarax is an open source Java library implementing a very flexible and extensible rule-based system. One of the main features of Mandarax is the possibility to load facts from any JDBC data source on demand (i.e. when they are needed to answer a query). This way, Mandarax can handle huge amounts of data as the data is not kept into main memory but rather read from the data source as needed. As an interface for communicating with the rule-based system, RuleML[3] was chosen, as this is a XML-based format capable of describing all important elements of a rule-based system.

3 Authrule Architecture and Design

3.1 Requirements

Authorization models represent the data and logic that are used to perform authorization decisions. As stated above, there is a vast variety of authorization models which differ quite significantly in the data and logic they use. The aim of this work is to design an authorization module that is able to execute any potential authorization model; this means that authorization models should be exchangeable and that new authorization models should be addable with only little effort. To make the implementation of a new authorization model as easy as possible, an easy format should be used for describing the elements of the

[2] http://www.mandarax.org
[3] http://www.ruleml.org

model. Furthermore, the system should be platform-independent and it should be able to support different data sources easily.

The choice of the authorization model of course in some way influences the functional requirements that can be divided into two groups: functionality necessary for performing authorization requests (called client functions) and functionality necessary to manage the authorization data (called administration functions). Additionally, the functional requirements can be devided into those that are specific to a certain authorization model (model-specific) and those that are not (model-unspecific). For the client functions for instance, the authorization request itself can be easily abstracted to a generic authorization request that has the same form regardless of the authorization model the software is currently applying. But there are also client functions that are model-specific and cannot easily be abstracted, e.g., for RBAC there must be a client function to create sessions and activate or deactivate roles. Some administration functions are model-unspecific, for instance creating or deleting users, objects or operations. Most of the administration functions however use specific characteristics of the authorization model and are therefore model-specific.

3.2 Realization of Authrule

As shown in Fig.1, the authorization module Authrule processes authorization requests from client business applications and additionally supplies a (separate) interface for administration. These services were implemented in an API consisting of Java interfaces. The interfaces are supplied by a class called *KnowledgeBaseManager*, which represents the software to the user.

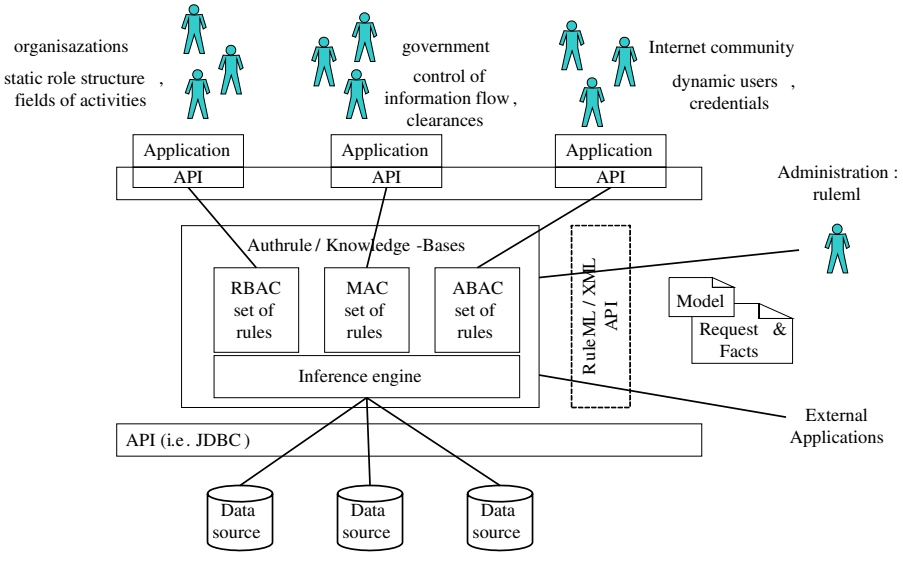

Fig. 1. Overall software architecture of Authrule

According to the requirements, these services are made available in a way that they are as generic as possible and at the same time offer all necessary functionality. Client functionality is provided by the Java interface *ClientI* (see Sec. 4) which forms the API for the applications in Fig.1. [4] For model-specific functionality, this interface was extended, e.g., the RBAC-specific functions (session management, etc.) are provided by the interface *RBACClientI*. Accordingly, the generic administration interface is *AdminI*, which is extended for model-specific functionality. For RBAC-specific administration (role management, etc.), the interface *RBACAdminI* was created. It is possible to use client functionality by using the generic *ClientI* interface—this allows a client business application to use Authrule for authorization requests without even knowing what authorization model is used. In contrast to the client side, administration of the Authrule can only be done properly, when using the model-specific administration interface, as the administration requires knowledge of the authorization model (e.g., for RBAC, the administrator must know that roles have to be used to assign permissions to users).

The API encapsulates the rule-based system, which actually processes the requests. A class derived from the abstract class *AuthorizationModel* transforms the method calls from the API to rule-based queries. This class also takes care of calls that require writing/modifying data, which cannot be done by simply querying the rule-based system. A helper class (called *PredicateDatasource*) was developed to provide an easy and abstracted way of writing and deleting data.

The representation of the authorization model as a set of facts and rules is described in a RuleML configuration file, which is loaded on startup. The RuleML format is human-readable an easy to understand and edit. This data format is mapped to the the internal, object-oriented and Mandarax-specific runtime representation of the rule-based system, which is not human-readable. The same format can also be used to directly query and/or as an intermediate step mapping calls from the Java API to query the the rule-based system. The core of Authrule is designed to be extensible in several aspects and therefore conforms to the above requirement. As mentioned above, the authorization model can be exchanged seamlessly. One class (derived from the class *AuthorizationModel*) includes all program code that is specific for this authorization model; by exchanging this class (and changing two configuration files), a new authorization model can be applied. Facts that have to be included in the knowledge base as they are necessary for the authorization decision (like user names, roles, etc.) can be added in two ways: If the amount of data for these facts is small, they can be manually added into the RuleML configuration file that is used to define a certain authorization model. This file should only comprise the set of rules (and some basic facts) that made up the model and are loaded into the knowledge base on startup. For larger amounts of data, it is more appropriate to map these predicates to a JDBC data source, this way the data does not reside in

[4] *Business applications* in Fig. 1 and the *SecurityProxy* in Fig. 3 are called *clients* from the perspective of Authrule. Not to be confused with *Client* in Fig. 3.

the knowledge base but is loaded into memory from the data source on demand. The mapping of predicates to data sources is configured in a different XML configuration file.

Administrative clients will use the administrative interfaces of Authrule. If they add new facts, this will internally result in adding a tuple to the database. If the access control model is modified, i.e. adding new authorization constraints, this will result in extending the authorization model with new rules.

4 Implementing Authorization Models

To implement an authorization model, only a Java class, two Java interfaces and two XML files have to be created, as described in Sec. 3.2. The elements of the authorization model are described as facts and rules in a RuleML file. Another XML file describes how the facts are mapped to data sources. Functionality for client use and administration is declared by extending the interfaces *ClientI* and *AdminI* (see Fig.2). An implementation of the abstract class *AuthorizationModel* is created to map the interfaces to requests to the rule-based system.

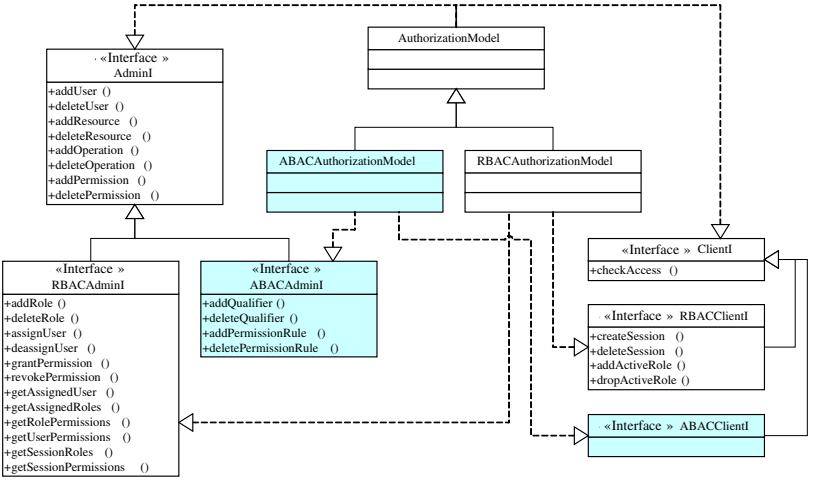

Fig. 2. Authrule application programming interface (with interfaces for RBAC and ABAC)

Each authorization model has to be assigned a unique ID, called *ModelId*. Each part of the authorization model implementation, i.e. the Java class derived from the class *AuthorizationModel* as well as the XML files for defining the rules and mappings, is tagged by this *ModelId*. Upon startup, Authrule loads the Java class and XML-files that are specified in the configuration file and checks if they all have the same *ModelId*. This ensures the integrity of the loaded authorization module.

4.1 Role-Based Access Control

The first step for implementing an authorization model is to describe the model itself, which is the static part of a RBAC policy, as a set of facts and rules. The following listing shows an excerpt of the RuleML file that forms the Core-RBAC model described in [3] translated to a rule-based representation.

```
...
<rulebase model_id="ifs.uni-regensburg.de/rbac-core/0.1">
<imp>
  <_head>
    <atom>
      <_opr><rel>granted</rel></_opr>
      <var>a User</var>
      <var>an Object</var>
      <var>an Operation</var>
    </atom>
  </_head>
  <_body>
    <and>
      <atom>
        <_opr><rel>hasRole</rel></_opr>
        <var>a User</var>
        <var>a Role</var>
      </atom>
      <atom>
        <_opr><rel>hasPermission</rel></_opr>
        <var>a Role</var>
        <var>a Permission</var>
      </atom>
      <atom>
        <_opr><rel>session</rel></_opr>
        <var>a Session</var>
        <var>a User</var>
      </atom>
      <atom>
        <_opr><rel>activeRole</rel></_opr>
        <var>a Session</var>
        <var>a Role</var>
      </atom>
... <!-- predicates to check existence of elements are omitted -->
   </and>
  </_body>
</imp>
</rulebase>
```

The RBAC model comprises just one rule for the predicate *granted* that represents the authorization decision. It is not necessary to declare the used predicates separately as they are declared implicitly when used in a rule definition. The use

of predicates that represent the existence of elements (like user, role, object, etc.) are omitted in this excerpt:

Since the facts according to the predicates tend to be numerous and change frequently, they can be mapped to an arbitrary number of JDBC data source and therefore separated from the definition of the model.

The Java interfaces *RBACClientI* and *RBACAdminI* were created to extend the interfaces *ClientI* and *AdminI* with additional, RBAC-specific functionality like handling sessions and defining roles. Fig.2 shows theses interfaces and their methods as well as the class *RBACAuthorizationModel*, which implements them. Requests can be passed to the rule based system in several ways, see Sec. 3.2. The call *checkAccess()* of the interface *ClientI* results in a RuleML query similar to that in Sec. 4.2 that is sent to the rule-based system to find out if a user is allowed to perform an operation on an object. In other words, the implementation of interface methods like *checkAccess()* use the provided RuleML interface to implement their functionality.

4.2 Attribute-Based Access Control

Implementing the ABAC model requires—similarly to the implementation of RBAC-a Java class, two Java interfaces and two XML files. A major difference compared to RBAC is that an ABAC policy cannot be so clearly distiguished into a static model and its instances. The ABAC model itself is translated into some static set of rules that can be separated from the dynamic authorizations of some instance of the ABAC model, also translated into rules. These instances of the model are formulated into a separated set of rules and facts that are temporarly combined for an authorization decision. A second major difference is, that in this ABAC implementation the requesting user and the user's attributes (also called credentials) are external to the rule base and handed over when *checkAccess()* is called. The attributes can, for example, originate from a X.509 attribute client certificate [5].This is in contrast to most RBAC implementations, where server-side roles are derived from the identity without additional information from the client. Therefore, the parameter user that is used with the API is an object that has methods to get and set its attributes. Consequently, the RuleML query for handing the authorization request to the rule-based system is extended to take the attributes of the user into account when processing the query. This is shown in the following listing:

```
<rulebase>
<!-- facts added temporarly to knowledgebase for executing the query -->
   <atom>
   <_opr><rel>user</rel></_opr>
      <ind>Bob</ind>
   </atom>
   <atom>
```

[5] X.509: Public Key and Attribute Certificate Frameworks. ITU-T Recommendation, 2000.

```
   <_opr><rel>hasAttribute</rel></_opr>
      <ind>Bob</ind>
      <ind>age</ind>
      <ind>23</ind>
   </atom>
<!-- the query, same for RBAC/ABAC -->
  <query>
    <_body>
      <atom>
        <_opr><rel>granted</rel></_opr>
        <ind>Bob</ind>
        <ind>DocumentA</ind>
        <ind>write</ind>
      </atom>
    </_body>
  </query>
</rulebase>
```

The rule base for the ABAC model is more complex. The following excerpt of the corresponding RuleML file show how ABAC can be formulated in rules:

```
<rulebase model_id="ifs.uni-regensburg.de/abac/0.1">
<!-- this rule maps the granted predicate to the hasPermission
  predicate and checks existence of the passed elements -->
<imp>
  <_head>
    <atom>
      <_opr><rel>granted</rel></_opr>
      <var>a User</var>
      <var>an Object</var>
      <var>an Operation</var>
    </atom>
  </_head>
  <_body>
    <and>
... <!-- predicates to check existence of elements are omitted -->
      <atom>
        <_opr><rel>hasPermission</rel></_opr>
        <var>a User</var>
        <var>an Object</var>
        <var>an Operation</var>
      </atom>
    </and>
  </_body>
</imp>
<!--  greater_equal (int) Operator -->
<imp>
  <_head>
    <atom>
      <_opr><rel>matchesQualifier</rel></_opr>
```

```
      <var>a User or Object</var>
      <var>a Qualifier</var>
    </atom>
  </_head>
  <_body>
    <and>
      <atom>
        <_opr><rel>qualifier</rel></_opr>
        <var>a Qualifier</var>
        <ind>greater_equal</ind>
        <var>an Attribute</var>
        <var type="Integer">IntValue 2</var>
      </atom>
      <atom>
        <_opr><rel>hasAttribute</rel></_opr>
        <var>a User or Object</var>
        <var>an Attribute</var>
        <var type="Integer">IntValue 1</var>
      </atom>
      <atom>
        <_opr><rel predefined="true">=&gt;</rel></_opr>
        <var type="Integer">IntValue 1</var>
        <var type="Integer">IntValue 2</var>
      </atom>
    </and>
  </_body>
</imp>
...
```

The *granted* predicate checks for the existence of the passed elements and then refers to the predicate *hasPermission*. The predicate *hasPermission* yields true, if one of the permission rules match the authorization query.

Permission rules use qualifiers [9,10] to describe what kind of attributes qualify users and objects for this rule. As qualifiers use operators (like *equal, greater than*, etc.), these operators must be first defined. The above excerpt shows the definition of the operator *greater_equal*[6] . Permissions are assigned by adding permission rules to the rule-based system. The following listing shows an example of a permission rule and the qualifier this permission rule uses:

```
<fact>
  <atom>
    <_opr><rel>qualifier</rel></_opr>
      <var>adult</var>
      <ind>greater_equal</ind>
```

[6] This is an example of a predefined predicate. It represents special functions that the Mandarax distribution supplies—in this example, it checks if the first term is greater than or equals the second term. To use predefined predicates with RuleML, the RuleML format was extended with a marker attribute predefined, as can be seen in the code example above.

```
      <var>age</var>
      <var type="Integer">18</var>
  </atom>
</fact>
<fact>
  <atom>
    <_opr><rel>qualifier</rel></_opr>
      <var>belongs_to_hemauer</var>
      <ind>equal</ind>
      <var>project_name</var>
      <var type=" String">Hemauer Project </var>
  </atom>
</fact>
<imp>
  <_head>
    <atom>
      <_opr><rel>hasPermission</rel></_opr>
        <var>a User</var>
        <var>an Object</var>
        <ind>read</ind>
    </atom>
  </_head>
  <_body>
    <and>
      <atom>
        <_opr><rel>matchesQualifier</rel></_opr>
        <var>a User</var>
        <ind>adult</ind>
      </atom>
      <atom>
        <_opr><rel>matchesQualifier</rel></_opr>
        <var>an Object</var>
        <ind>belongs_to_hemauer</ind>
      </atom>
    </and>
  </_body>
</imp>
```

Qualifiers are represented by a predicate called *qualifier* that defines the qualifier's name, an operation, an attribute type, and a value. In the above example, the qualifier *adult* is defined to match all users whose attribute age is grater or equals 18. The second qualifier *belongs_to_hemauer* matches all objects that have an attribute *project_name* indicating that they belong to the project "Hemauer Project". The permission rule that is listed afterwards uses these two qualifiers to specify that adult users are allowed to read objects that are associated to the project "Hemauer Project".

As shown in Fig.2 the Java interfaces for ABAC are quite straight forward. The API remains the same, however, the prerequisite must be met that the attributes are contained in the subject. Hence, only few additional functionality

is required on the client side. This can be accomplished in transparent way using general APIs for authentication (like JAAS[7] , PAM,.etc.). On the client side, the interface remains the same, which means that this ABAC model can be used by means of the generic interface *ClientI*. This is made possible as the parameters for the generic interface *ClientI* are based on instances of the interfaces *UserI*, *OperationI* and *ObjectI*. These interfaces supply functions to get and set attributes. For this reason, these methods can be used regardless of the authorization model that is actually used. On the administration side, the generic *AdminI* is extended by the interface *ABACAdminI* with methods to create and delete qualifiers and permission rules.

5 Usage Scenario

In order to evaluate our approach and the Authrule module, we deployed it for access control in a component-based business application. Over the last years J2EE (Java 2 Enterprise Edition) evolved as a major framework for enterprise application development. This framework comprises the software component standard EJB (Enterprise Java Beans) [5]. EJBs are well suited for constructing business applications as they come with out-of-the-box solutions and mechanisms for a set of non-functional requirements like security. EJBs can also be made externally available as web services using JAX-RPC[8]. Unfortunately the EJB standard has been designed to be tightly bound to RBAC. This fact inhibits principals like application field neutrality, transparency and flexibility. EJB-based applications can be deployed in a variety of application domains that have different requirements for authorization see Sec. 1. A further substantial inconvenience of the EJB security approach is, that roles are derived from the users' identity on the server side see Sec. 4.2. Consequently, we consider the existent role based access control in EJB systems as a form of basic security, which needs to be supplemented by additional security measures. In Fig.3 an interceptor (called a "security proxy") is placed in the communication path between the client and a component transparently to the application logic. Each software component can be preceded with its own interceptor that can contain authorization logic in which we integrate Authrule to protect several EJBs. Consequently each piece of application logic in form of an EJB can be supplied with attribute-based access control with arbitrary authorization rules. The non-functional requirements transparency, flexibility, and interoperability have been taken into special consideration. Transparency in this sense means, that security should not be part of the application logic. In this way security unaware applications can be secured without requiring to change their code and security mechanisms are interchangeable.

Fig.3 depicts our approach in detail. A user/principal uses a client, which is linked to the authentication interfaces. On startup of the client application, the

[7] Java Authentication and Authorization Service (JAAS).
http://java.sun.com/products/jaas/

[8] Java API for XML-Based RPC (JAX-RPC)
http://java.sun.com/webservices/jaxrpc/index.jsp

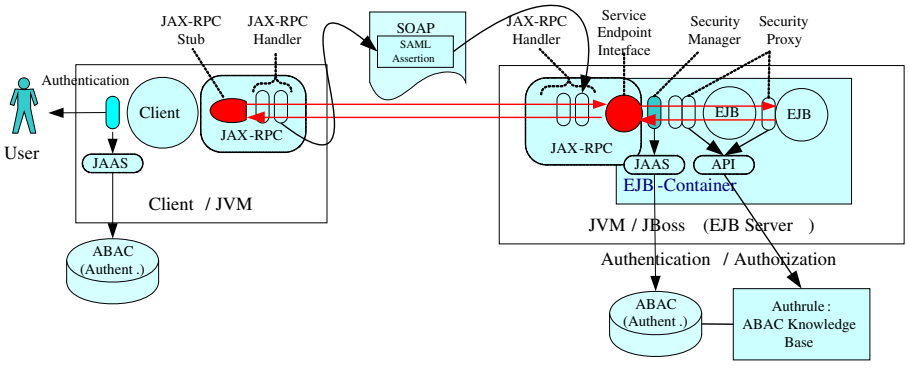

Fig. 3. Authrule for EJB access control

authentication process is being triggered. It calls the attribute-based authentication module, which is deployed in conformance to the server-side authorization module. The client is requested to provide the attributes, he wants to use and a *subject* instance containing these attributes is created. By invoking the web service methods the subject is transparently encapsulated. Server-side authentication is processed and a security context is established. Before invoking an EJB the interceptor forwards the attributes to Authrule in order to take the authorization decision. Depending on this decision the authorization enforcement within the interceptor/security proxy grants access to the bean or prevents the further invocation.

6 Related Work

Since it has been demonstrated that RBAC can be configured to also enforce DAC and MAC, RBAC has been considered to be a generic authorization model. The flexibility of RBAC and its ability to enforce MAC policies and a number of access constraints to realise the equivalent of Bell-LaPadula has been demonstrated in [13]. With the appearance of attribute-based access control models (XACML[6], UCON[8], DLAM[1]) authorization requirements have shown up, which eventually cannot be solved by RBAC. As a result, RBAC cannot longer be considered as "ultima ratio". A lot of RBAC implementations yet exist, one of it is described in [12]. It was implemented in a classical straightforward manner by mapping the static model to an equivalent software architecture. CSAP [12] afterwards has been extended with ABAC resulting in the above mentioned substantial changes on the software and runtime inefficiencies.

There have been other attempts to find a universal way of describing authorization models as set of rules and predicates. One of the broadest approaches is described by Bertino et al. [2]—however, the intention for describing authorization models in that work was to create a framework to compare them in respect to their expressiveness; it did not discuss how rule-based systems can be used to build a generic authorization module. Additionally, assumptions about

the authorization models (e.g. that users are organized in groups) were made, constraining the generality of their framework. Other research works that use logics-based languages and reasoners for access control can be found in the area of the Semantic Web. For example, the KAoS framework [11] provides a collection of services for distributed policy management and enforcement. It uses the description-logics-based Web Ontology Language (OWL) to specify the policies. Likewise, the policy engine in Rei [6] can handle policies specified in RDFS (a subset of OWL).

In this work, we decided to use RuleML as basis for the input and output format when communicating with the application core. One might argue that the XACML[6] could be used as it is also able to formulate authorization requests and express the permission data for several different authorization models. In fact, XACML is a very versatile standard that is capable of mapping many different authorization models. However, XACML has drawbacks that make it unsuitable when a very generic, but simple approach is desired. One reason is that the versatility of XACML resulted in a very complex format, which is not easy to handle and involves a lot of overhead. Another reason is that XACML, though very versatile, is not generic enough to cover all possible authorization requirements. For example, even though core RBAC requirements can be easily implemented using XACML, a full-featured constrained RBAC [3] is hard to achieve as XACML rules are not as expressible as the logics used by a rule-based system. Using a descriptive format like RuleML ensures that the approach is so generic that it can capture all authorization models that can be formulated as set of predicates and rules—and, as we argued in this paper, every authorization model can be formulated as set of predicates and rules.

7 Conclusions

This paper presented an approach that led to a generic authorization module that supports arbitrary authorization models and can be easily extended. To accomplish this, a rule-based system was used to map authorization models to rule sets and an inference engine processes the authorization requests. There are different authorization models for different application fields with different requirements. Usually, authorization modules limit themselves to the use of a certain specific authorization model and a later change or modification of the model consequently results in a substantial effort for revising the software architecture. Rule-based systems are well suited to represent authorization models by mapping their elements and relations to facts and rules, which can be modified in a flexible manner. The implementation of the popular RBAC and ABAC (attribute-based access control) models with our approach was demonstrated, giving the deployment in a J2EE/web service scenario as a usage scenario. This scenario was chosen, because it also demonstrates how flexibility and transparency can be reached in conjunction with other state-of-the-art mechanisms.

Future work will elaborate on more integrative tasks. We will investigate different models, additional constraints, delegation, and trust, as well as the environment as an attribute source. We will also examine the semantics of exchanged attributes in web service scenarios.

References

1. Adam, N.R., Atluri, V., Bertino, E., Ferrari, E.: A Content-based Authorization Model for Digital Libraries. IEEE Transactions on Knowledge and Data Engineering, Volume 14, Number 2, March/April 2002.
2. Bertino, E., Catania, B., Ferrari, E., Perlasca, P.: A Logical Framework for Reasoning about Access Control Models. In: ACM Transactions on Information and System Security, Volume 6, Number 1, pp. 71-127, Februar 2003.
3. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D., and Chandramouli, R.: Proposed NIST Standard for Role-based Access Control. In: ACM Transactions on Information and Systems Security, Volume 4, Number 3, August 2001.
4. Ferrari, E., Adam, N.R., Atluri, V., Bertino, E., Capuozzo, U.: An Authorization System for Digital Libraries. In: VLDB Journal, Volume 11, Number 1, 2002.
5. Enterprise JavaBeans 3.0. Java Specification Request 220 Proposed Final Draft, http://jcp.org/aboutJava/communityprocess/pfd/jsr220/index.html
6. Kagal, L., Finin, T., Joshi, A.: A Policy Based Approach to Security for the Semantic Web. In: Proc. 2nd International Semantic Web Conference (ISWC 2003), Sanibel Island, FL, October 2003.
7. OASIS eXtensible Access Control Markup Language v2.0 (XACML). http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf
8. Park, J., Sandhu, R.: The UCONABC Usage Control Model. In: ACM Transactions on Information Systems Security, Volume 7, Number 1, pp. 128-174, February 2004.
9. Priebe, T., Fernandez, E.B., Mehlau, J.I., Pernul, G.: A Pattern System for Access Control. In: Proc. 18th Annual IFIP WG 11.3 Working Conference on Data and Application Security, Sitges, Spain, July 2004.
10. Priebe, T., Dobmeier, W., Muschall, B., Pernul, G.: ABAC - Ein Referenzmodell für attributbasierte Zugriffskontrolle. In: Proc. 2. Jahrestagung Fachbereich Sicherheit der Gesellschaft für Informatik (Sicherheit 2005), Regensburg, Germany, April 2005.
11. Uszok, A. et. al.: KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction and Enforcement. In: Proc. 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), Comersee, Italy, June 2003.
12. Dridi, F., Fischer, M., Pernul, G.: CSAP – An Adaptable Security Module for the E-government System Webocrat. In: Proc. of the 18th IFIP International Information Security Conference (SEC 2003), Athens, Greece, May 2003.
13. Osborn, S., Sandhu, R., Munawar, Q.: Configuring Role-based Access Control to enforce Mandatory and Discretionary Access Control Policies In: ACM Transactions on Information and System Security (TISSEC), volume 3, pages 85-106, 2000