Uniform Application-level Access Control Enforcement of Organizationwide Policies

Tine Verhanneman Frank Piessens Bart De Win Wouter Joosen
Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Leuven
{tine,frank,bartd,wouter}@cs.kuleuven.be

Abstract

Fine-grained and expressive access control policies on application resources need to be enforced in application-level code. Uniformly enforcing a single policy (referred to as the organizationwide policy) in diverse applications is challenging with current technologies. This is due to a poor delimitation of the responsibilities of application deployer and security officer, which hampers a centralized management of a policy and therefore compromises the uniformity of its enforcement.

To address this problem, the concept of an access interface *is introduced as a contract between an organization-wide authorization engine and the various applications that need its services. The access interface provides support for the central management of the policy by the security officer. By means of a* view connector*, the application deployer ensures that each application complies with this contract, so that the policy can be enforced.*

1. Introduction

Many applications require the enforcement of an expressive access control policy, which, for example, takes into consideration application state. In the context of an organization, where one single organizationwide policy needs to be enforced, support should be provided to administer the policy centrally and to enforce it uniformly in the various applications deployed within the organization.

Various stakeholders are involved in the enforcement of an access policy. The *security officer* manages the policy centrally. The *application deployer* tunes the access control enforcement for each application so that the policy can be enforced. The access control decision itself can be delegated to an organizationwide authorization engine, which may be developed independently of a particular application setting and provided by an *authorization engine provider*. The *application developer* provides the application logic.

In this paper, the observation is made that the delimitation of the responsibilities between the security officer and application deployer, is poorly supported by current technologies: In reality, the application deployer bears complete responsibility for the uniform enforcement of the policy. This renders it hard to manage the policy centrally, especially if this policy is liable to frequent changes.

The contribution of the paper consists in providing an abstraction layer, named *access interface*, which captures the requirements an application needs to fulfill so that the organizationwide policy can be enforced. This access interface, for example, includes explicitly the additional information that is needed to evaluate an access request. The access interface abstracts from application-specific details by including only information that is relevant for access control. It can therefore be specified by the security officer, who is responsible for the definition and centralized management of the policy.

The application deployer binds the access interface to each application by means of application-specific *view connectors*. A view connector specifies (1) how the application fulfills the requirements that are put forward in the access interface, and (2) how access requests within the application are translated to the access interface. A prototype has been implemented as an extension of an aspect-oriented application container, whereby the view connector acts as a deployment descriptor.

The remainder of the paper is organized as follows. Section 2 motivates the need for an intermediary abstraction layer, illustrated by a case in the health care application domain. In Section 3 the access interface approach is presented, followed by a discussion in Section 4. The prototype is discussed in Section 5. Section 6 gives an overview of related work and conclusions are drawn in Section 7.

2. Detailed motivation

In this section, the challenge of implementing an access control policy in an application with state-of-the-art technologies will be illustrated by means of a case in the health care application domain. After having identified the shortcomings of current technologies, we list the requirements our approach should meet.

2.1. High-level policy

Health care organizations must ensure that appropriate technical and organizational measures are in place to protect patient data: Based on the principles of *least privilege* and *minimum necessary* [24], the disclosure of health care information should be limited to the minimum necessary to accomplish the intended purpose.

We discuss a subset of the security policy of an academic hospital in Belgium [28, 18]. These rules are typical for access control policies in a medical context [24, 25, 3, 1]. Our setting is a hospital with a large number of physicians and associated general practitioners. The following rules deal with accesses to a contact, which is a logical unit of medical data.

Rule 1 *A physician will be granted access to a patient's data if a contact exists to which he was assigned. The access rights are only valid until 30 days after the contact was closed.*

The policy allows to overrule the access decision, for example for emergency access, provided that it is possible to hold physicians accountable for any access granted on the basis of this rule.

Rule 2 *The system provides the possibility to overrule the access decision, on condition that the user requesting access, specifies a reason. The reason, the requesting user's and the patient's name, along with some context information (time, place) are logged.*

To improve communication between the patient, his general practitioner (GP) and the team of caregivers, view access is granted to the patient's GP.

Rule 3 *The patient's general practitioner has view access to all the patient's contacts, whether these contacts have been closed or not.*

These three rules will serve as the basis for further discussion. In the following paragraphs, roles and permissions (objects and operations) are identified (conforming to RBAC [12]) as a first step towards an implementation of the policy.

Roles. Two roles can be distinguished: A *physician*, who is a staffmember and a licensed medical practitioner (e.g. a specialist), and the *general practitioner*, who maintains the overview of the patient's social background, medical history and current health condition and acts as a confidant for the patient.

Permissions. This policy only concerns objects which represent *identifiable medical data*. The status of medical data can be *open* or *closed*, depending on whether the contact, the data is part of, has been closed or not. The operations that can be carried out on a medical data object are restricted to *view*, *append* and *close*. The latter is invoked by the patient's responsible physician to close the contact.

Pure RBAC lacks granularity to enforce the rules mentioned earlier: For an access decision, the relationship between the user requesting access and the patient whose data is about to be accessed, should also be taken into account [3]. Table 1 summarizes the policy rules and illustrates that only the responsible physician is allowed to close the medical data of his patient.

2.2. Enforcing the policy in applications

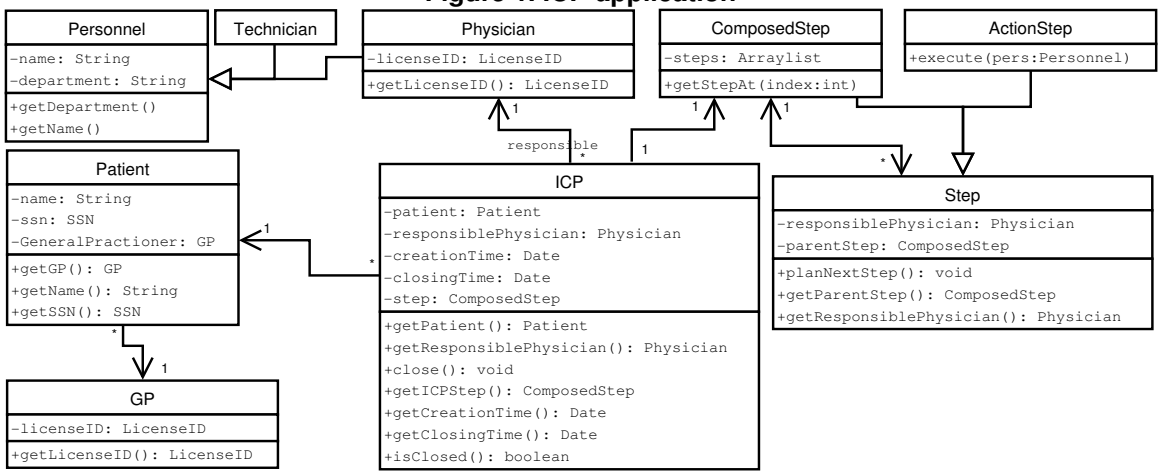
The organizationwide policy specified above must now be enforced in all applications running in the hospital, such as for example an appointment and prescription system [28]. Given the increased use of information technology in health care, this number of applications can be quite high.

We describe a simplified model of one example application: an Integrated Care Pathways (ICP) application. An Integrated Care Pathway [19] is a predefined plan for care relating to a certain diagnosis, which serves as a guideline to organize care more effectively and efficiently; e.g. to shorten hospital stays, to raise resource utilization and to reduce unnecessary variations in patient care and outcomes. In short, an ICP constitutes a workflow, which guides the health care provider through the different steps in the health care process by providing a template, indicating the health care services which should be provided at a certain point in the treatment. Upon commencing the treatment, the responsible physician instantiates an ICP for his patient, and plans and executes the steps as the treatment proceeds. These steps are, for example, examinations, medication prescriptions and notes. Figure 1 shows a simplified classdiagram for the ICP application. The medical data to protect is contained within the Integrated Care Pathway (`ICP`) and its associated steps (`Step`). The application keeps a reference to both the GP of the patient and the responsible physician.

Table 1. Medical data

Roles/status (Medical Data)		open	< 30 days closed	>30 days closed
Physician	***if** responsible*	view,append,close	view	-
Physician	***if** in overrule mode*	view	view	view
GP	***if** patient's GP*	view	view	view

Figure 1. ICP-application



2.3. Problems when implementing and managing the policy

Before elaborating on the challenges of the application-level enforcement of a policy, two important stakeholders are introduced:

- The *security officer* draws up and manages the policy without needing to have extensive knowledge of the internal operation of the different applications.
- The *application deployer* tunes the access control enforcement by the application, ensuring that it conforms to the policy.

How should an organizationwide policy, like the policy presented in Section 2.1, be enforced in the application? The deployer has to translate the high-level, organization-wide policy into application terms, by providing for example deployment descriptors, configuration files or code. This typically results in a series of lower-level rules, indicating for each type of object which methods may be invoked by whom. An example of such a low-level rule might be that a physician is allowed to invoke `getPatient()` on all objects of the class `ICP`. This means that once the high-level policy has been defined by the security officer, the burden is placed entirely on the application deployer to uniformly translate this policy into application terms for each of the applications deployed within the organization, which is a very intricate job.

This lack of an abstraction layer between an application and the security logic also becomes apparent if a common organizationwide authorization engine is used: Application-specific access requests need to be translated in terms understood by the engine. A rather ad hoc approach consists in conveying labels to the authorization engine, which abstract the action and/or object that is being accessed.

Uniformly enforcing an access policy tends to get harder as policy rules are frequently updated, added or removed. The application deployer has to translate the high-level policy once again and has to ensure that the access control enforcement points and the information passed to the authorization engine (if an engine is used) are adapted to reflect the updated policy. Consider, for illustration purposes, the following two additional rules:

Rule 4 *Each time the GP accesses his patient's medical data, the responsible physician is notified of this access.*

Rule 5 *Psychiatric - and human heredity records are classified as highly sensitive, and cannot be viewed by the GP [18].*

For Rule 4 the responsible physician and for Rule 5 the sensitivity level of the data need to be conveyed to the authorization engine. The deployer also faces similar problems when the application itself changes, e.g. due to code refactoring.

2.4. The requirements

We now define the major requirements that we took into account when developing our approach for the integration of access control enforcement in applications. These requirements are mainly based on [4].

1. The *expressiveness* of the policies that can be enforced, should not be constrained [3]. In practice, in order to enforce application-level security, the granularity of the policy that can be specified, should be small enough to encompass the application resources to be protected. Likewise, the variety (richness) and the amount of information serves as a criterion of the expressiveness of the supported policies. For example, the state a workflow process is in, the time or other contextual information may be relevant when making an access control decision.

2. *Separation of concerns* must be supported by clearly delimiting responsibilities of the stakeholders identified in Section 2.3. Separation of concerns is the key to support evolution, which encompasses both manageability and extensibility.

3. Multiple applications that obey the same security policy, must be treated and described *uniformly*. Uniformity requires support for the central management of an organizationwide policy, as well as the enforcement of a single policy in diverse applications. In short, what we aim for is to write the policy once and to enforce it everywhere.

Of course, any proposed design should also have no adverse effects on other important properties of an access control infrastructure (such as performance and scalability). We return to this point in the discussion section 4.

3. Proposed solution

In this section, our solution is described. In the overview shown in Figure 2, an organizationwide authorization engine is used to evaluate access requests.

Two new concepts are introduced as part of this solution. The *access interface* describes explicitly what the authorization engine expects from applications in order to make access decisions. Such an access interface should be relatively constant within one organization and its specification may be application domain specific, as it is driven by the high-level policy rules of the organization. For example, in a financial organization, the value of a transaction might be important information to decide about an access request. In a hospital, on the other hand, it is important to know whether this is an overrule access or not. Through this access interface, a centrally managed and configured authorization engine receives notifications of access attempts from applications, and can query applications for application state to decide whether these accesses should be allowed or not. The authorization engine can be configured by means of declarative policy rules that specify the access control policy in terms of the access interface.

View connectors realize the application-side of the contract. There is a separate view connector per application, mapping application-specific concepts to the concepts represented in the access interface. An important contribution of this paper is that we show that a view connector can be realized as a kind of deployment descriptor on an aspect-oriented application container. This allows to set the mapping declaratively, without needing to apply invasive changes to the application code. We elaborate on both concepts in the next sections.

3.1. Access interface

The access interface is an explicit representation of the contract between the authorization engine and applications. As such, it specifies the provided and expected functionality and data for both these parties. The authorization engine provides decisions on access requests and expects the applications to (1) notify the engine of relevant accesses, (2) provide the necessary information about these accesses, and (3) enforce the decision on the access request. So in particular, an access interface must specify what "relevant accesses" are, and what information must be provided for each of these accesses.

Any access request is a request by a subject to perform some action on an object. Our design starts from the assumption that objects are classified by the policy in *domains* (such as the medical data domain in our example policy), and that subjects are classified in *roles* (such as the physician role), and that the information needed to decide on an access request, can differ depending on the domain and role of respectively, the object and subject involved.

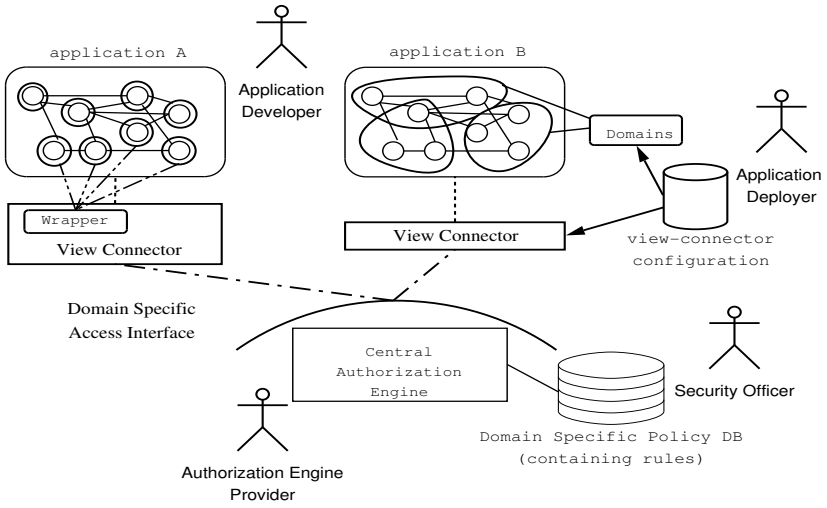
This leads to the following formalization: an access interface $\mathbb{A}$ consists of a set $\mathbb{O}$ of *object interfaces* (one per domain) and a set $\mathbb{S}$ of *subject interfaces* (one per role).

Object interfaces. An object interface $\mathcal{O}$ for a given domain is a pair $(attr, act)$, whereby:

1. $attr$ denotes a set of attribute names, specifying the information that the authorization engine needs about objects, and $Values(a)$ denotes the set of possible values for a given attribute $a \in \mathcal{O}.attr$

2. act denotes the set of relevant actions, about which the authorization engine expects to be notified.

Figure 2. Overview



An example of an object interface for the domain of medical data in our example policy is shown below. Note that the object interface is application-independent: actions and attributes are specified at an appropriate level of abstraction for making access control decisions. Binding these attributes and actions to actual application concepts, will be the task of the view connector.

```
ObjectInterface MedicalData{

   attribute:   {open, closed} status;
   attribute:   Date closingTime;
   attribute:   LicenseID ResponsiblePhysician;
   attribute:   LicenseID GP;

   action:      view;
   action:      append;
   action:      close;
}
```

Subject interfaces. A subject interface $\mathcal{S}$ for a given role specifies the information that the authorization engine needs about subjects in that role in the form of a set of attributes $attr$. As for the object interface, $Values(a)$ denotes the set of possible values for a given attribute $a \in \mathcal{S}.attr$

In our example policy, there is a rule that checks whether a physician is the *responsible physician* for a given piece of medical data. Hence, subjects of role physician need an attribute (licenseID) that can be matched with the corresponding attribute on the MedicalData object interface.

```
SubjectInterface Physician{

   attribute:   LicenseID licenseID;
   attribute:   {normal,overrule} accessmode;
}
```

The access control view on an application consists of a set O of security objects and a set S of security subjects. Each security object $o \in O$ has one associated object interface $objectinterface(o)$ and likewise, each security subject $s \in S$ has one associated subject interface $subjectinterface(s)$. As will be explained in the next section, the view connector defines the security state of each security object and subject. The security state of a security object or subject is determined by the values of the attributes, specified in their associated object interface and subject interface respectively. The security state of a security object o can be written as $\sigma(o) = (v_a)_{a \in objectinterface(o).attr}$, where each $v_a \in Values(a)$. Similarly, the security state of a security subject can be written as $\sigma(s) = (v_a)_{a \in subjectinterface(s).attr}$, where each $v_a \in Values(a)$.

Implementing the policy. An access request is a triple (s, o, a) consisting of a security subject s, a security object o and an action name a in the action name set act of $objectinterface(o)$. The access policy is the function that, given an access request (s, o, a) and the security states $\sigma(o)$ and $\sigma(s)$ returns whether the access is allowed or not.

The particular choice of authorization engine, which realizes this function, is irrelevant for the discussion in this paper, and many good designs of authorization engines are available (e.g. FAF [15]). Typically, such an authorization engine will interpret a set of declarative policy rules. In our example, we use the Ponder [10] policy language to formulate the rules in. The following rule states that the responsible physician for a piece of medical data can view, append or close that data, as long as its status is open.

```
inst
   auth+ openMedicalDataAccess {
      subject   <Physician> s=/Physician;
      target    <MedicalData> t=/MedicalData;
      action    view, append, close;
      when      t.status.equals(''open'') and
                t.ResponsiblePhysician.equals
                (s.licenseID);
   }
```

Ponder does not only provide a policy language, but also a deployment model for instantiating and distributing policies. The view connector concept, which is discussed next, is complementary to this model as it provides a means to integrate access control enforcement into each application.

3.2. View connectors

The access interface specifies access requests at an abstract level. At some point, this needs to be translated down to actual application concepts. This is the role of the view connector. We say that a view connector *binds* an application to the access interface. Each application will need its own view connector.

To implement a view connector one must:

- Decide how application objects map to security objects and subjects. In our example, we will have to identify all medical data that the application handles.
- Identify all operations on such data and map these operations to the corresponding actions in the object interface. This also determines all places where an access check needs to be performed. In our example, we will have to map all application operations on medical data to one of the three actions: view, append or close.
- Determine how to compute the necessary attributes for security objects and subjects.

An example view connector is shown in Table 2. We first discuss the computation of attributes.

Attribute computation. The view connector will need to specify how each of the access interface attributes is computed for the given application. For instance, an `ICP` object in our example application (Figure 1) is clearly medical data. The responsible physician for that object can be computed via a getter on the `ICP` class. The patient's GP must be computed by first getting the patient associated with the medical data, and then getting the GP of that patient.

The part of the view connector that computes these attributes, is very similar to Beznosov's attribute functions [5], or to the DynamicAttributeService in CORBA's Resource Access Decision (RAD) service [6].

```
view-connector
type                ICP
object-interface    MedicalData
```
(a) attribute computation

```
attributes
 ResponsiblePhysician → getResponsiblePhysician().get-
                        LicenseID()
 GP                   → getPatient().getGP().get-
                        LicenseID()
 status               → if(isClosed()) closed
                        else open
 closingTime          → getClosingTime()
```
(b) access enforcement points

```
actions
 view                 → get*
                      ...
```

Table 2. View connector for medical data

Access enforcement points. The access enforcement points in the applications are the points in the execution, where an access check needs to be done. The insertion of access checks at all these points, is technology dependent. This can be done, for example, by inserting the necessary calls to the authorization engine in the application code. Adding such calls to the application code during deployment/integration of an application, is not straightforward on typical application platforms such as J2EE, .NET or CORBA, as it requires access to the application source code. However, the technology of aspect-orientation makes it possible to provide an implementation of view connectors that can be configured at deployment time.

Our prototype is built on top of an aspect-oriented application container. An aspect-oriented application container offers the concept of *pointcuts*, expressions that denote sets of execution points in an application. The container also allows the injection of new code at each point identified by such a pointcut. The use of pointcuts by itself, does not improve the security of the overall system. However, due to a modular description of the access enforcement points it is easier to assess the security of the system than would be possible if these points were spread all over the code. Given such support, each action in an object interface can be mapped on such a pointcut, and the relevant access checks can be inserted at each point identified by that pointcut.

In our prototype, an extensible application container is extended with a so-called aspect component, which is configured with a view connector. An example of such a view connector is shown in Table 2. Part (a) configures how attribute computation should be done. Part (b) shows how actions are mapped to pointcuts. Based on this configuration file, the right access checks are injected at each of these points. In Section 5 our implementation is discussed in more detail.

4. Discussion

In this section, we discuss the presented approach by evaluating to what extent it fulfills the requirements mentioned in Section 2.4. In addition, we will show that our approach does not negatively influence other important properties of an access control infrastructure for distributed object systems. The discussion is based on the enumeration of typical requirements for these systems in [4].

Expressiveness. We found that for practical policies, the access interface does not impose restrictions on the expressiveness of the policies that can be enforced. A trade-off may need to be made between expressiveness and other requirements. E.g., defining a large number of object and subject interfaces should be avoided in order to keep the policy manageable.

The definition of the subject interface should be extended to support more complex principals, as for example presented in [17]. This would allow, for instance, to take into consideration the access path of a request.

The access interface is based on the high-level policy. Further research is required to develop techniques to determine an appropriate access interface for an application domain.

Separation of concerns and evolution. The responsibilities of security officer and application deployer are clearly separated. The former specifies the access interface, the latter the view connector.

Evolution of the policy is more straightforwardly supported. Consider for this purpose the additional rules in Section 2.3: Since Rule 4 fully complies to the access interface introduced in Section 3.1, adding the rule suffices to enforce it uniformly.

Rule 5 requires an extension of the medical data access interface to support a sensitivity attribute and the definition of the necessary attribute mappings in the corresponding view connectors. Our approach provides better support to apply this extension, as the view connector specifies explicitly which application objects represent medical data. A limited consistency check can be carried out to verify whether mappings have been defined for each attribute in the access interface.

If the policy changes more radically, new object and/or subject interfaces might need to be introduced with corresponding view connectors.

In case the application or its setting changes (e.g. due to code refactoring), only adapting the corresponding view connectors suffices.

Uniformity. The introduction of the access interface supports a central management of an organizationwide policy. View connectors support its enforcement in diverse applications.

Performance- and administration scalability. The performance overhead incurred by adding support for view connectors, depends on how they are implemented. An implementation with centralized management but distributed enforcement will likely achieve good performance.

The good support for separation of concerns between administrators and developers/deployers positively influences administration scalability.

Requirements that stem from the object paradigm. These requirements are subdivided in the following classes [4]:

1. Objects: *The access control technology should shield complex semantics of the diverse methods from the security officer.* This is one of the main goals of the access interface, so our approach fulfills this requirement.

   Secondly, *the technology should scale on large number of objects and methods*. As the access interface allows for grouping of objects (in policy domains with the same object interface) and methods (in actions), scalability is assured.

2. *Collections: Flexibility is required when grouping objects into collections for security purposes; solely providing grouping based on names or location is not sufficient. Moreover, collocation or similar names should not imply membership of the same security collections.*

   Our approach provides for grouping of objects independent of any existing structure on the application. Our current prototype implementation imposes the restriction that all objects of the same class should belong to the same object interface, but making this more flexible is just an implementation effort.

3. *Names: No human intervention should be required to enforce access control on transient objects.*

   Our approach is neutral with respect to this requirement: whether human intervention is required or not, depends on the implementation of the view connector.

   The security officer is not required to be aware of object names to roll out the policy.

   View connectors shield the security officer from application-specific object names.

5. Prototype

We have developed a prototype implementation for access interfaces and view connectors on top of the

aspect-oriented application container Java Aspect Components (JAC). First, Aspect-Oriented Software Development (AOSD) is shortly described in Section 5.1. Subsequently, in Section 5.2, the prototype itself is discussed.

5.1. AOSD

Aspect-orientation is based on the observation that current paradigms, such as for example object orientation, fall short in encapsulating so-called *crosscutting concerns* into separated modules and therefore provide poor support for the separation of concerns principle. An example of a crosscutting concern is application-level access control logic: it is spread all over the application and is often entangled with application logic [11].

The additional concept that aspect-oriented software development offers us to improve the modularization of the so-called aspects (concerns) is *quantification* [13]. Quantification enables us to formulate statements which have an impact on various points in the code. An example statement is "each time a method is invoked on an object, it should be verified whether the invoker has authorization to do so".

The second characteristic of AOSD (Aspect-Oriented Software Development) is *obliviousness* [13] of the application-logic developer regarding the applied aspect, resulting in a better separation of concerns between application deployer and application developer. The latter provides the application logic and, ideally, does not have to be aware of the security logic imposed on the application logic.

The construct with which this is realized, is called a *joinpoint*. A joinpoint is a place in the execution where the (in our case) access logic is superimposed on the application. Typical joinpoints are method invocations, exception handling, execution flows .... So-called *pointcuts* allow us to select a set of joinpoints based on one or more of their characteristics; e.g. the name of the method invoked or the target object. *Advice* is the logic injected into the application at the join point; in this case we would like to inject access control enforcement checks. For a good overview of AOSD, we refer the reader to [9, p 29-97].

5.2. Implementation

In this section, we describe an implementation based on aspect-oriented programming. We opted to implement the prototype on top of Java Aspect Components (JAC), which will be elaborated on in the next paragraph.

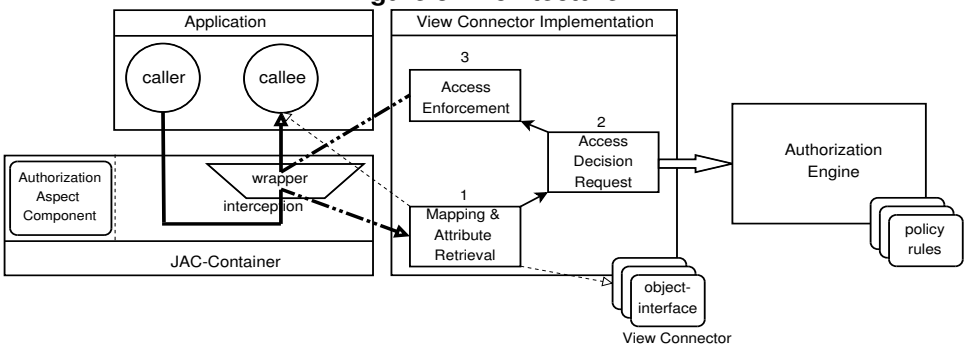
Java Aspect Components. JAC [23] is in essence an extensible application container. This platform provides an aspect-oriented middleware layer, which allows dynamic (un)loading of aspect components. These aspect components allow to weave (transparently to the application) *wrappers* around the target (java) objects, called *wrappees*. These wrappers intercept method invocations at execution points specified by pointcuts, and can add extra functionality by means of *role-methods*. JAC, moreover, provides support for contextual information by means of *collaboration flows*, which allow to attach attributes to an execution thread.

Architecture. In this paragraph, a broad overview is given of the architecture of the prototype (displayed in Figure 3). The ICP ***application*** runs on top of the aspect-oriented middleware layer provided by JAC, i.e. the ***JAC container***. The latter acts as a kind of reference monitor, as shown in Figure 3: The wrapper first intercepts the access request at the points in the execution conforming to the view connector configuration file, as explained in Section 3.2. The wrapper then literally connects the view to the application by mapping the access request onto the access interface. Hereto, the view connector executes the following actions:

1. *Mapping:* Based on the view connector the application-specific access request is projected onto the access interface by:
   - retrieving the object and subject interfaces applicable to respectively the subject and the callee
   - determining to which action in the object access interface, the access request corresponds
   - retrieving the attribute values needed by the authorization engine
2. *Access Decision:* The request is subsequently sent to the authorization engine, which is discussed below, for evaluation.
3. *Access Enforcement:* The access decision is enforced.

The ***Authorization Engine*** evaluates the access request based on the access rules. In the above mentioned approach, no knowledge of the internals of the specific application is required, since the access request is translated into terms of the access interface. In the prototype implementation, we opted to push the attributes to the authorization engine. The drawback is that more attributes are retrieved than may be necessary for the access decision. The advantage is that roundtrips are saved, if the attributes can be retrieved locally and the access decision function is deployed on a different node. Alternatively, a lazy evaluation strategy can be used, in which attributes are pulled by the authorization by means of callbacks. More experiments are needed to evaluate the performance of the prototype.

Figure 3. Architecture



6. Related work

The presented work is related with several research domains, which are discussed below:

Middleware infrastructures for application-level access control. Tivoli Access Manager [16] supports consistent and centralized management across heterogeneous systems by introducing a *hierarchical protected object namespace* to abstract resources. URL mappings specify how a dynamic URL should be resolved into a namespace object, and can be regarded as (limited) view connectors. The CORBA Resource Access Decision (RAD) service [6] abstracts respectively the asset and request, by conveying a protected resource name and access operation to the access decision function. It remains the responsibility of the application developer to apply these abstractions consistently. Additional attributes can be retrieved by means of DynamicAttributeServices. OSA [5], Object Security Attributes, are generic representations for application-specific factors. Java Authorization Contract for Containers [27] (JACC) specifies contracts between the application container (e.g. J2EE) and so-called policy providers. Policy Context Handlers allow providers to obtain additional context, such as for example the enterprise bean involved in the access request. These infrastructures do not capture explicitly the requirements of the authorization engine to enforce the access control policy, such as the necessary application-specific information.

View Policy Language [8] aggregates access rights in a type-safe manner into views, which can be assigned to a role. To construct these views, VPL starts from the application's use-cases, whereby the actors are directly mapped onto the roles. VPL aims at a better separation of concerns so that access control is manageable. VPL focusses on the design, specification and management of security policies rather than on the integration and uniform enforcement of access control.

The access interface groups methods, to which the same policy rule apply. In J2EE [7], methods are grouped according to the role, allowed to invoke that method.

Access control frameworks. The prototype (Figure 3) exhibits a similar architecture as the ISO/IEC 10181-3 Access Control Framework [14] and the XACML dataflow model [20]. The authorization engine is essentially the *Policy Decision Point* (PDP). The view connector acts as both *Policy Enforcement Point* (PEP) and *Policy Information Point* (PIP).

Policy languages and authorization engines. The access interface and view connector approach benefits from and complements the extensive research carried out in the field of policy languages, such as for example XACML [20] and Ponder [10], and authorization engines, such as for example the Flexible Authorization Framework (FAF) [15].

Model Driven Engineering. Our approach relates to Model Driven Architecture [21] (MDA) and SecureUML [2]. The difference is that MDA focusses at a (semi-)automatic translation of a high-level model into a platform dependent model and implementation for a specific application, whereas in the access interface approach, we aim at a uniform translation of requirements across the various applications.

Aspect-Oriented Software Development. The access interface approach is related to Multidimensional Separation of Concerns (MDSOC) [22] as it provides a view on the application from the viewpoint of access control. The use of Aspect-Oriented Software Development (AOSD) techniques has already been proven useful in the separation of the access control concern [11]. Song *et al.* [26] apply an Aspect-Oriented Modeling approach to compose the access control concern and the application in a verifiable manner.

7. Conclusion

Enforcing an expressive policy is hard due to a poor support of the separation of concerns principle. An *access interface*, makes explicit the contract between the authorization engine and the applications, for which the policy should be enforced. For each application, a *view connector* ensures that the contract is fulfilled by binding the particular application to the access interface. This approach naturally supports a centralized management of an expressive policy, as well as the enforcement of a single policy in diverse applications. Therefore it also enforces uniformity of access control enforcement in the applications, deployed within the organization. A prototype has been implemented on top of an aspect-oriented application platform.

Acknowledgements. This research is funded by a Ph.D grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). The authors would like to thank Professor Konstantin Beznosov and the anonymous reviewers for their helpful comments and suggestions.

References

[1] R. J. Anderson. A security policy model for clinical information systems. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 30, Washington, DC, USA, 1996. IEEE Computer Society.

[2] D. Basin, J. Doser, and T. Lodderstedt. Model driven security for process-oriented systems. In *SACMAT '03: Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 100–109, New York, NY, USA, 2003. ACM Press.

[3] K. Beznosov. *Engineering Access Control for Distributed Enterprise Applications*. PhD thesis, Florida International University, July 2000.

[4] K. Beznosov. Access Control Mechanisms in Commercial Middleware, June 2002. tutorial at SACMAT'02.

[5] K. Beznosov. Object Security Attributes: Enabling Application-Specific Access Control in Middleware. In *DOA'02: 4th International Symposium on Distributed Objects & Applications*, pages 693–710, London, UK, October 2002. Springer-Verlag.

[6] K. Beznosov, Y. Deng, B. Blakley, C. Burt, and J. Barkley. A Resource Access Decision Service for CORBA-based Distributed Systems. In *ACSAC '99: 15th Annual Computer Security Applications Conference*, pages 310–319, 1999.

[7] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, and B. Stearns. *The J2EE tutorial*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[8] G. Brose. Manageable access control for CORBA. *Journal of Computer Security*, 10(4):301–337, 2002.

[9] D. Crawford. *Communications of the ACM*, volume 44. ACM Press, New York, NY, USA.

[10] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. *LNCS*, 1995:18–28, 2001.

[11] B. De Win, W. Joosen, and F. Piessens. Developing secure applications through aspect-oriented programming. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 633–650. Addison-Wesley, Boston, 2005.

[12] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[13] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness, October 2000. Workshop on Advanced Separation of Concerns, OOPSLA 2000.

[14] ISO. Information technology - open systems interconnection - security framework for open systems: access control framework. ISO/IEC 10181-3 (ITU-T X.812).

[15] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[16] G. Karjoth. Access control with IBM Tivoli access manager. *ACM Trans. Inf. Syst. Secur.*, 6(2):232–257, 2003.

[17] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[18] U. Z. Leuven. Leuvense Internet Samenwerking Artsen (LISA). www.uzleuven.be/UZroot/content/Zorgverleners /login/lisa/ (dutch).

[19] S. Middleton, J. Barnett, and D. Reeves. What is an integrated care pathway? *What is ...? series*, 3(3), 2003. http://www.evidence-based-medicine.co.uk/What_is_series.html.

[20] OASIS. Core Specification: eXtensible Access Control Markup Language (XACML) Version 2.0.

[21] Object Management Group. OMG Model Driven Architecture. http://www.omg.org/mda/.

[22] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, 2001.

[23] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Framework for AOP in Java. In *Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, September 2001.

[24] Secretary of the Department of Health and Human Services. Final Privacy Rule, August 2002.

[25] Secretary of the Department of Health and Human Services. Final Security Rule, February 2002.

[26] E. Song, R. Reddy, R. France, I. Ray, G. Georg, and R. Alexander. Verifiable composition of access control and application features. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 120–129, New York, NY, USA, 2005. ACM Press.

[27] Sun Microsystems. Java Authorization Contract for Containers, final release, November 2003. JSR-115.

[28] B. Van den Bosch. *The design and the development of the hospital information system of the U.Z. Leuven*. PhD thesis, Katholieke Universiteit Leuven, 1996.