Enforcing Complex Access Control Policies in Rich Domain Applications

Gonçalo Mascarenhas Dumiense

Extended Abstract:

Introduction

Successful applications’ development is stressed by an increasing demand for more functionality. As their functionality grows, so does their complexity. Handling the complexity of applications is one of the goals of many software engineering approaches.

One of those approaches, is called domain-driven design. This approach states that the development of complex applications, should be centered around the entities that compose the problem being solved by the application - its domain. These entities, should be structured by what is called a domain model. A domain model can be seen as a set of concepts that translate the abstraction of the underlying reality of a given problem.

The domain-driven design approach defends that the business logic of an application should result from a web of interconnected objects that hold both the data and the behavior of the entities in the domain of the application. Applications with a complex domain model, with rich behavior are said to be Rich Domain Model (RDM) application.

Although RDM applications can be very useful for dealing with complex domain problems, they present some challenges to the enforcement of access control policies. In particular, I have identified four types of problems that I describe next.

Problem definition

The first of these problems relates to code scattering and tangling. Although these problems are not exclusive of RDM applications, they are particularly difficult to tackle in this type of application due to their ever growing complexity. Code scattering, refers to problems related to code of a specific concern of an application, like access control, transactionality, or persistence, that becomes scattered through all the application, making it hard to manage the correspondent concern. The code tangling refers to the problem that arises when code from these type of concerns, that are orthogonal to the application logic, become embedded in that logic creating a dependency between the code of the logic and that concern.

The second is related with the need of enforcing access control policies with complex constraints. By complex constraints, I mean constraints that can require a high level of granularity. For instance, an access control decision can depend on the verification of a property of an instance of a domain's object. In this case it is needed a granularity of object's instances. This fine grained access control can be hard to achieve with normal access control approaches.
The fourth problem is related with a typical situation that occurs when enforcing access control in object oriented applications, as RDM applications. It manifests itself when a given method calls another method that has access control constraints attached to it. Consider two methods \( m_1 \) and \( m_2 \), and a user \( u \). Method \( m_1 \) invokes \( m_2 \). User \( u \) does not have permission to access \( m_2 \) but it has permission to access \( m_1 \). How should the access control decision go in this situation? There are possible approaches to deal with this problem, all valid, but the one that I consider most suitable for RDM applications is based on amplification of privileges. This amplification of privileges should be done in a controlled and compositional way so that, it eases the task of handling the application’s complexity.

The last problem is the support of delegation of rights, as it can provide the application with a flexibility that the users are used to have in performing their tasks without the aid of computer applications. If this kind of flexibility is not supported by the application, it may lead to more serious security problems like password sharing. The delegation of rights from one subject to another has two components: a static, and a dynamic one. The static component of delegation in an access control system expresses the rules that control the delegation flow. This component however, is not enough by itself. The actual delegation only occurs when a subject delegates the right to do something, to another subject, which only happens in run time. Due to this run time nature of delegation, in order for a system to support delegation it must allow its representation of the access control policy to change in run time. So, delegation comprises the alteration of the policy representation in run time. This run time requirement render many traditional approaches to access control in applications useless, because most of them use as a base for their policy representation static development artifacts.

These are the problems that I address in this dissertation and for which I propose a solution.

Solution Overview

The proposed solution consists in a mix of a new access control model, a Domain Specific Language (DSL) named Domain Model Authorization Policy Language (DMAPL), and an access control engine, all tailored specifically for RDM applications.

The access control model gives developers an abstract framework with which they can conceive and reason about an access control policy. Problems posed by expressiveness issues when dealing with complex constraints are addressed by this model. Apart from that, the model supports the features needed to deal with the requisites of RDM applications, like delegation and amplification of privileges.

The DMAPL provides a way to aid in the task of specifying and maintaining an auditable access control policy, based on the access control model. This DSL have the same expressiveness of the model when dealing with complex constraints.

With the policy expressed in a formal language, specifically tailored to the domain of RDM applications, developers can use and create tools that automate most of the work involved with the enforcement of the policy in the application. One of this tasks is the placement of the access control verifications, within the code of the application, through code injection, which can be automatically done. For all this, the DMAPL aids in the mitigation of the code scattering and tangling problem.

In order to support some run-time features, as delegations and amplifications of privileges, access control
verifications cannot be static. They should consist on the invocation of some run time entity that keeps track of the global state of the policy. This is where the third part of the solution enters, the engine. The engine is the run time entity responsible for the enforcement of the policy described with the DMAPL. For that, the engine loads the policy and creates a run-time representation of that policy, using the model’s concepts.

In this dissertation I formally define the proposed access control model, I introduce the syntax and semantic for the DMAPL, and I provide implementation detail of the run-time engine. I shall now briefly introduce the concepts of the model in order to provide a general notion of the underlying ideas of the proposed solution.

The model is based on the notion of access control rules, i.e., access control decision are made based on the existence of rules. An access control rule is an entity that states if the access to a given method’s execution is to be permitted or denied, for a given user. The resources that are due to be protected by an access control policy in a RDM application, are its methods’ executions. There are different types of rules that address different types of permission or prohibition. The rules considered in the proposed model are: positive and negative authorization rules, amplification rules, positive and negative delegation rules, and delegated rights. These rules are grouped in policies.

I shall now explain each type of rules and what their purpose is in a given policy.

Authorization

Generally speaking, authorization defines which subjects have access to a given resources when certain circumstances are met. An authorization rule is the simplest type of rule, as it directly states if a given user can access a given resource under certain circumstances.

To express authorization rules, I introduce the definition of two entities: subjects and targets. A subject is an entity that denotes a set of users of a given application. A target is an entity that denotes a set of methods of the same application.

With the concepts of subject and target I can define authorization rule as a tuple

\[(\text{SUBJECTS}, \text{TARGETS}, \alpha)\]

where \text{SUBJECTS} is a set of subjects, and \text{TARGETS} is set of targets, and \(\alpha\) is a predicate that takes as input a user and a method execution.

Authorization rules express which subjects have access to which targets under the conditions expressed by the predicate \(\alpha\). As mentioned before, a policy can have both positive, and negative authorization rules. Positive authorization rules are the ones that express that authorization can be granted, whereas negative authorization rules forbid that authorization to be granted.

The utilization of \text{SUBJECT} and \text{TARGET} entities to define an authorization rule creates a decoupling between the rule and an application, which brings a number of advantages.

The \text{SUBJECT} creates a separation between the rules and the users to which it applies that greatly eases the management of a policy, because it allows the users to vary without the need to be constantly modifying its rules.

As for the \text{TARGET}, it also creates a separation between the rules and the methods of an application. This also allows the application to evolve and grow without the need of redefining the existent rules. The use
of a TARGET as a building block to the construction of an application’s access control policy contributes to the compositionality of the model.

The predicate \( \alpha \) of an authorization rule makes an authorization rule to be valid only under circumstances that may be not directly related to a user or a method’s execution. This permits the expressive power of a rule to be greater than normal access control models that only take into account the user and the resource being accessed.

To accomplish the goal of minimizing accidental errors that can lead to an access control breach, I opted to follow an approach where, whenever in doubt, forbid. For that reason, if there is a positive authorization rule that allows a subject to access a target, but there is a negative authorization rule that forbids, the access should be denied. For the same reason, if no valid positive rule is found, although there is no valid negative rule to forbid, the access is also denied. This is how access control decisions are made in the proposed model.

**Amplification of Privileges**

To support a controlled amplification of privileges I opted for an approach based on authorization tickets. In real life, tickets provide their owners the access to something that without them, they probably would not have the right to access. This is the same objective of the tickets in the proposed solution.

In the model defined in this dissertation, there are three concepts associated with amplification of privileges: ticket type, ticket, and amplification rule.

A ticket type is the entity that is responsible to describe a generic right that a user may possess in the course of the an application’s execution, although it may not have been stated previously with authorization rules. The ticket type is what defines the resources that a user who possess a ticket of that type may access. Consider the analogy in real life with tickets for a movie session or for a football match. As they are different types of tickets, they provide access to different type of resources.

The specific instances of a given ticket type are called tickets. A ticket is the association of a ticket type with a state. This state, that is set during the creation of a ticket, enables access control decision to be made on base of information relating ticket creation’s circumstances. This way, tickets created at different times, for different users, in different circumstances can provide different rights. It is like the difference of a football match ticket won in a radio contest that give its owner the right of handshaking the players after the match, or the tickets that are bought at the entrance which do not provide this right.

For the purpose of solving the method dependency problem, when a user is about to execute a method \( m_1 \) that depends on a method \( m_2 \), that have a certain access control constraint, for which the user does not have permission, a ticket that allow the execution of \( m_2 \) should be granted to the user. Amplification rules are a special type of policy rules that state this type of behavior, i.e., they express when does a user should receive a ticket of a specific ticket type. After the execution of \( m_1 \) the ticket becomes invalid and the user looses access to \( m_2 \).

As it can be seen, with ticket and amplification rules, the access control decision no longer depends solely on the existence of authorization rules. In case of the absence of an authorization rule that allows of forbids the execution of a given method, it is also verified if the user possesses a ticket that grants her with the access
to execute the method in question.

This amplification of privileges mechanism help developers to correctly enforce an access control policy in a compositional manner. Amplification rules and tickets allow the definition of a self contained access control policy for a given method \( m \). This way, if method \( m \) depends on method \( m_1 \) to complete its execution, and method \( m_1 \) is protected by some authorization rule, it is possible to specify that during the execution of method \( m, m_1 \) should be accessible through an amplification rule with a corresponding ticket. When developing other methods that use method \( m \), the policy does not need to be altered. This greatly simplifies the development of features in RDM applications.

**Delegation of Rights**

As mentioned before delegation of rights is a two dimensional problem. One is the problems related to the specification of the delegation policy, i.e., what rights can be delegated to whom, under what circumstances. The other one, is how to represent a right that has been delegated, and how does this delegated right influences the access control decision process.

To solve the first part of the problem I introduce in the model concept of delegation rule. A delegation rule is a special type of rule that states what can be delegated in a system. A positive delegation rule is a tuple:

\[
(AUTH, GRANTOR, GRANTEE, TARGETS, \chi, \lambda)
\]

where \( AUTH \) denotes a positive authorization rule of the policy, \( GRANTOR \) and \( GRANTEE \) are subjects, \( TARGETS \) is set of targets, and \( \chi \) and \( \lambda \) are predicates.

A positive delegation rule states who can delegate what to whom, under what circumstances and for how long. The right which a delegation rule refers to, is stated by the \( AUTH \) element of a delegation rule. Note that a right cannot be represented by a negative authorization rule, because those do not express rights, but prohibitions. \( GRANTOR \) represents a subject that expresses who can delegate the right \( AUTH \).

On the other hand, the \( GRANTEE \) component denotes the subject that the delegation rule expresses, as being entitled to be granted the right denoted by \( AUTH \). The \( TARGET \) of a delegation rule states what subset of targets, from the \( AUTH \)'s targets, can be delegated.

The predicate \( \chi \) is a predicate that expresses under what circumstances can a delegation be performed. The circumstance can involve some relation between the user who is delegating, the grantor, and the grantee. So, this predicate receives two users as arguments, that denote the grantor and grantee users, respectively.

After a right has been delegated, it may be useful to control until when is the delegated right valid. A grantor may just want to delegate a right for one hour, one day, during his vacations, until the grantee turn 18 years old, etc. The predicate \( \lambda \) is intended to capture this idea, i.e., it expresses what are the conditions needed to be met for the delegated right to still be valid, after the delegation has occurred. The arguments for the \( \lambda \) predicate, besides the grantor and grantee users, also contain a method execution. This method execution corresponds to the execution of the method denoted by the \( TARGET \) of the delegation rule. This allows the validity of a delegated right to depend also on the state of the execution of the target.
The difference in structure of positive and negative delegation rules is that the latter do not possess the \( \lambda \) predicate. This is because as a negative delegation rules forbid delegation, they do not need to specify this type of predicate, as it is only used after the delegation has happened.

When a user is allowed to delegate something to another user, i.e, there is a delegation rule that allows it, a new authorization should be created. This new authorization is what I call a delegated right and it is how I address the second part of the delegation problem. A delegated right is what represents, in run time, that a certain user, the grantee, was granted a certain right by another user, the grantor.

A delegated right is a tuple:

\[
(\text{GRANTOR}_u, \text{GRANTEE}_u, \text{TARGETS}, \alpha, \lambda)
\]  

A delegated right states that a user \( \text{GRANTOR}_u \) delegated the right to execute the methods denoted by \( \text{TARGETS} \) to user \( \text{GRANTEE}_u \).

The users \( \text{GRANTOR}_u \) and \( \text{GRANTEE}_u \) must be users that were denoted by the originating delegation rule’s \( \text{GRANTOR} \) and \( \text{GRANTEE} \) correspondently. The \( \text{TARGETS} \) components of a delegated right, as well as the \( \lambda \) predicate are the same as the ones of the originating authorization rule.

The predicate \( \alpha \) of a delegated right serves the same purpose of the predicate of an authorization rule: it states the circumstances under which \( \text{GRANTEE}_u \) should be allowed to access \( \text{TARGETS} \).

With delegation rules and delegated rights it is possible to devise a mechanism that addresses both parts of the delegation of rights problem.

**Validation**

To validate the proposed solution, I present a case study of a real RDM application called Fénix. Fénix is an open source web application aimed at integrating all the aspects linked with the management of a university campus: from administrative tasks such as room management, to students tasks such as students’ course enrollment, and even to teachers tasks as course’s webpage creation. Due to its enormous set of functionalities and importance, I consider that Fénix is an application that can be used to study how the proposals in this work can affect this and other real-life applications.

Fénix uses various mechanisms to address the problem of access control. From those it is worth mentioning:

- **Service Level Access Control**: In a previous, service oriented architecture of Fénix, all its operations were encapsulated by services. Those services were supported by the Berserk framework which, with an implementation of the Intercepting Filter Pattern, provides a way to wrap a service invocation with filters that can express pre and post conditions for the service to finish correctly. This allows developers to create filters that checked if the user in the current session had the right privileges to invoke a given service. As most of the Fénix functionalities were implemented in these services, enforcing access control at the service filter level was a good way to cover almost all the actions that users could do on the system.
• **Method Level Access Control**: Using this mechanism developers declare which conditions are needed to be met for a given method to execute in predicates that implement a given interface. Then they annotate the desired methods with a predetermined annotation, and a code weaving tool injects the code necessary to invoke the predicate in the correspondent methods marked with annotations. This way every time that method is executed it performs the desired access control verification.

• **Functionality Level Access Control**: The evolution of Fénix led to the appearance of a functionality model, which pretended to represent all the functionalities available in the system and organize them in a tree structure. This model tried to address access control with the concept of availability policy. This policy states which users can access a given functionality. Being more related to the presentation layer, this concept of functionality availability permits the creation of a user interface, that depends on authorization, i.e., it is displayed to the user only what she has permission to do. The tree structure of functionalities provides this model with a compositionality that is desired in RDM applications like Fénix.

With this broad range of mechanisms, Fénix suffers from a code scattering problem that results from the lack of a solution that is systematically applied through all the application. The proposals in this dissertation provide a way to specify complex access control policy using a single development artifact, the DMAPL specification file, which could help solving these problems.

One reason for the use of so many different access control approaches is the granularity that is possible to achieve with each one of them. The proposed solution can work on all types of the granularities that the above mechanism provide, so they can safely be replaced by my proposals.

Besides the functionalities availability, all the other mechanisms do not allow the compositionality needed for RDM applications. In the proposed solution, the amplification of rights mechanisms enables the compositional development of the RDM while maintaining a correct access control enforcement. The main advantage over the functionalities availability is that the proposed solution allows the specification of access control rules at a domain level, not directly related with users’ operations, but with the domain behavior.

All the discussed mechanisms involved some sort of programatic effort to implement a given access control verification. Most of the time, developers had to declare a new class with a new filter or a new predicate. With the introduction of a declarative language like the DMAPL, this development effort is reduced, because this language’s constructs are specialized for the declaration of normal access control verifications, whereas the Java programming language is not. This relieves programmers from repetitive work and let them concentrate on what is more important.

None of the existent mechanisms provide a way to delegate rights between users. With the proposed solution, Fénix would have, almost automatically, delegation support which would enhance users’ flexibility.

Another important aspect that the proposed solution enables, and that is still not present in Fénix, is the possibility to dynamically change the access control policy without re-deployment. All the existent mechanisms are static and do not provide this type of feature.