Model-Driven Adaptive Delegation*

Phu H. Nguyen, Gregory Nain, Jacques Klein, Tejeddine Mouelhi, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg
4, rue Alphonse Weicker, L-2721 Luxembourg
{phuhong.nguyen, gregory.nain, jacques.klein, tejeddine.mouelhi,
yves.letraon}@uni.lu

ABSTRACT

Model-Driven Security is a specialization of Model-Driven Engineering (MDE) that focuses on making security models productive, i.e., enforceable in the final deployment. Among the variety of models that have been studied in a MDE perspective, one can mention access control models that specify the access rights. So far, these models mainly focus on static definitions of access control policies, without taking into account the more complex, but essential, delegation of rights mechanism. User delegation is a meta-level mechanism for administrating access rights, which allows a user without any specific administrative privileges to delegate his/her access rights to another user. This paper analyses the main hard-points for introducing various delegation semantics in model-driven security and proposes a model-driven framework for 1) specifying access control, delegation and the business logic as separate concerns; 2) dynamically enforcing/weaving access control policies with various delegation features into security-critical systems; and 3) providing a flexibly dynamic adaptation strategy. We demonstrate the feasibility and effectiveness of our proposed solution through the proof-of-concept implementations of different systems.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; K.6.m [**Management of Computing and Information Systems**]: Miscellaneous—*Security*

General Terms

Design, Security

Keywords

Model-driven security, model-driven engineering, model composition, delegation, access control, dynamic adaptation

*This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, under the MITER project C10/IS/783852.

AOSD'13, March 24-29, 2013, Fukuoka, Japan.

1. INTRODUCTION

Software security is a polymorphic concept that encompasses different viewpoints (hacker, security officer, end-user) and raises complex management issues when considering the ever increasing complexity and dynamism of modern software. In this perspective, designing, implementing and testing software for security is a hard task, especially because security is dynamic, meaning that a security policy can be updated at any time and that it must be kept aligned with the software evolution.

Managing access control to critical resources requires the dynamic enforcement of access control policies. Access control policies stipulate actors access rights to internal resources and ensure that users can only access the resources they are allowed to in a given context. A sound methodology supporting such security-critical systems development is extremely necessary because access control mechanisms cannot be "blindly" inserted into a system, but the overall system development must take access control aspects into account. Critical resources could be accessible to wrong (or even malicious) users just because of a small error in the specification or in the implementation of the access control policy.

Several design approaches like [20] [4] have been proposed to enable the enforcement of classical security models, such as Role-Based Access Control (RBAC) [25]. These approaches bridge the gap from the high-level definition of an access control policy to its enforcement in the running software, automating the dynamic deployment of a given access control policy. Although such a bridge is a prerequisite for the dynamic administration of a given access control policy, it is not sufficient to offer the advanced administration instruments that are necessary to efficiently manage access control. In particular, delegation of rights is a complex dimension of access control that has not yet been addressed by the adaptive access control mechanisms. User delegation is necessary for assigning permissions from one user to another user. An expressive design of access control must take into account all delegation requirements.

Delegation models based on RBAC management have been known as *secure*, *flexible* and *efficient* access management for resources sharing, especially on distributed environment. Flexible means that different subjects for delegation should be supported, i.e. delegation of roles, specific permissions or obligations. Also, different features of delegation should be supported, like temporary and recurrent delegations, transfer of role or permissions, delegation to multiple users, multi-step delegation, revocation, etc. However, the addition of

flexibility for delegation must come with mechanisms to make sure that the security policy of the system is securely consistent. And last but not least, the administration of delegations must remain simple to be efficient.

Delegation is a complex problem to solve and to our best knowledge, there has been no complete approach for both specifying and dynamically enforcing access control policies by taking into account various characteristics of delegation. Having such an expressive security model is crucial in order to simplify the administrator task and to manage collaborative work securely, especially with the increase in shared information and distributed systems.

Based on previous work [20], in this paper we propose a new Modular Model-Driven Security solution to easily and separately specify 1) the business logic of the system without any security concern using a Domain Specific Modeling Language (DSML) for describing the architecture of a system in terms of components and bindings; 2) the "traditional" access control policy using a DSML based on a RBAC-based metamodel; 3) an advanced delegation policy based on a DSML dedicated to the delegation management. In this third DSML, delegation can be seen as a "meta-level" mechanism which impacts the existing access control policies similarly as an aspect can impact a base program. The security enforcement is enabled by leveraging automated model transformation/composition (from security model to architecture model). Consequently, in addition to [20], an advanced model composition is required to correctly handle the new delegation features.

To be more specific, only basic delegation features have been considered in [20]. Moreover, these delegation features have been handled as traditional access control rules. In this paper, we claim that delegation needs to be clearly separated from access control since a delegation impacts access control rules. Therefore, delegation and access control are not at the same level and should be separated. This separation involves an advanced model composition approach to dynamically know, at any time, which are the set of new access controls that have to be considered, i.e., the "normal" access control rules as well as the access control rules modified by the delegations. From a more technical point of view, the security enforcement is dynamically done by leveraging automated model transformation/composition (from security model to architecture model) and dynamic reconfiguration ability of modern adaptive execution platforms.

The contributions of this paper are the followings: 1) A metamodel/DSML dedicated to the delegation management, for specifying RBAC and RBAC-based delegation features. OCL constraints are used to check the consistency of the security policy (access control + delegation); 2) A model-driven framework for dynamically enforcing access control and delegation mechanisms specified with our DSMLs. In this framework, newly defined model transformation rules play an important role in the dynamic enforcement of security policies. We claim that to handle advanced delegation features, an ideal solution is to separate the delegation rules from the access control policy, each being specified in isolation, and then compose/weave them together to obtain a new access control policy reflecting the delegation-driven policy; 3) A flexibly dynamic adaptation strategy with better support for delegation, and the co-evolution of the security policy and the security-critical system.

The rest of this paper is organized as follows. Section 2 briefly presents the background on RBAC, delegation, and the security-driven model-based dynamic adaptation. Next, a running example is given in Section 3. It will be used throughout the paper to describe the diverse characteristics of delegation and illustrate the various aspects of our approach. In Section 4, we first give an overview of our approach. Then, we formalize our delegation mechanism based on RBAC and show how our delegation metamodel can be used to specify expressive access control policies that take into account various features of delegation. Based on the delegation metamodel, we describe our model transformation/composition rules used for transforming and weaving security policy into the architecture model. This section ends with a discussion of several security policy dynamic adaptation and evolution strategies. Section 5 describes three case studies that have been used for evaluating our approach. It is followed by Section 6 which presents related work. Section 7 concludes the paper and discusses the future work.

2. BACKGROUND

2.1 Access Control

Access Control [11] is known as one of the most important security mechanisms. It enables the regulation of user access to system resources by enforcing access control policies. A policy defines a set of access control rules which expresses: who has the right to access a given resource or not, and the way to access it, i.e. which actions a user can access under which conditions or contexts.

2.2 Delegation

In the field of access control, delegation is a very complex but important aspect that plays a key role in the administration mechanism [5]. A software system, which supports delegation, should allow its users without any specific administrative privileges to grant some authorizations. Delegation of rights allows a user, called the delegator, to delegate his/her access rights to another user, called the delegatee. By this delegation, the delegatee is allowed to perform the delegated roles/permissions on behalf of the delegator [9]. The delegator has full responsibility and accountability of the delegated accesses since he/she provides the accesses to the resources to other users, who are not initially authorized by the access control rules to access these resources.

Delegation is a powerful and very useful way to perform policy administration. On one hand, it allows users to temporarily modify the access control policy by delegating access rights. By delegation, a delegatee can perform the delegated job, without requiring the intervention of the security officer. On the other hand, the delegator and/or some specific authorized users should be supported to revoke the delegation either manually or automatically. In this way, the administrative task can be simplified and collaborative work can be managed securely, especially with the increase in shared information and distributed systems [1]. However, the simpler the administrative task is, the more complex features of delegation have to be properly specified and enforced in the software system. To the best of our knowledge, there is no approach for both specifying and dynamically enforcing access control policies taking into account all delegation features like temporary delegation, transfer delegation, multiple delegation, multi-step delegation, etc.

2.3 Security-Driven Model-Based Dynamic Adaptation

In [20], the authors propose to leverage MDE techniques to provide a very flexible approach for managing access control. On one hand, access control policies are defined by security experts, using a DSML, which describes the concepts of access control, as well as their relationships. On the other hand, the application is designed using another DSML for describing the architecture of a system in terms of components and bindings. This component-based software architecture only contains the business components of the application, which encapsulate the functionalities of the system, without any security concern. Then, they define mappings between both DSMLs describing how security concepts are mapped to architectural concepts. They use these mappings to fully generate an architecture that enforces the security rules. When the security policy is updated, the architecture is also updated. Finally, the proposed technique leverages the notion of models@runtime [17] in order to keep the architectural model (itself synchronized with the access control model) synchronized with the running system. This way, the running system can be dynamically updated in order to reflect changes in the security policy. Only users who have the right to access a resource can actually access this resource. The different steps of this approach are summed up in Fig. 1.

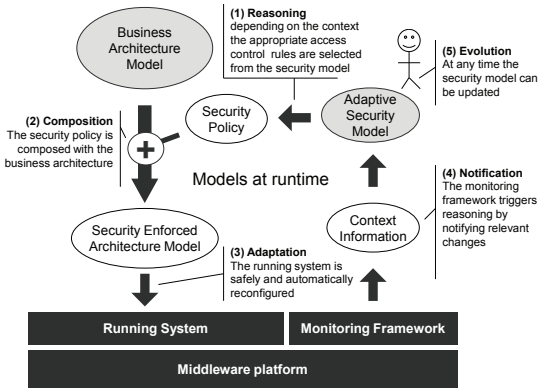

Figure 1: Overview of the Model-Driven Security Approach of [20]

3. A RUNNING EXAMPLE

In this section, we give a motivating example which will be used throughout the paper for describing the diverse characteristics of delegation and illustrating the various aspects of our approach.

Let us consider a library management system (LMS) providing library services with security concerns like access control and delegation management. There are two types of user account: personnel accounts (director, secretary, administrator and librarian) are managed by administrator; and borrower accounts (lecturer and student) are managed by secretary. The director of the library has the same accesses as the secretary, but additionally, he can also consult the personnel accounts. The librarian can consult the borrower accounts. A secretary can add new books in the LMS when they are delivered. Lecturers and students can borrow, reserve and return books, etc. In general, the library is organized with the following entities and security rules.

Roles (users): access rights (e.g. working days)

Director (*Bill*): consult personnel account, consult, create, update, and delete borrower account.
Secretary (*Bob* and *Alice*): consult, create, update, and delete borrower account, deliver book.
Administrator (*Sam* and *Tom*): consult, create, update, and delete personnel account.
Librarian (*Jane* and *John*): consult borrower account, find book by state, find book by keyword, report a book damaged, report a book repaired, fix a book.
Lecturer (*Paul*) and **Student** (*Mary*): find book by keyword, reserve, borrow and return book.

Resources and actions to be protected

Personnel Account: consult, create, update, and delete personnel account.
Borrower Account: consult, create, update, and delete borrower account.
Book: report a book damaged, report a book repaired, borrow a book, deliver a book, find book by keyword, find book by state, fix a book, reserve a book, return a book

In this organization, users may need to delegate some of their authorities to other users. For instance, the director may need the help of a secretary to replace him during his absence. A librarian may delegate his/her authorities to an administrator during a maintenance day.

It is possible to only specify role or action delegations by using the DSML described in [20]. For instance, a role delegation rule can be created to specify that *Bill*, the director (prior to his vacation) delegates his role to *Bob*, one of his secretaries. But it is impossible for *Bill* to define whether or not *Bob* may re-delegate the *director* role to someone else (in case *Bob* is also absent for some reason). The role delegation of *Bill* to *Bob* is also handled manually: it is enforced when *Bill* creates the delegation rule and only revoked when *Bill* deletes this rule. There is no way for *Bill* to define a temporary delegation, i.e. its active duration is automatically handled. Obviously the DSML described in [20] is not expressive enough to specify complex characteristics of delegation.

There are many delegation situations that motivate our work. We consider in the following some delegation situations:

1. The director (*Bill*) delegates his role to a secretary (*Bob*) during his vacation (the delegation is automatically activated at the start of his vacation and revoked at the end of his vacation).
2. A secretary (*Alice*) delegates her task/action of create borrower account to a librarian (Jane).
3. A secretary (*Bob*) transfers his role to an administrator (*Sam*) during maintenance day. In case of a transfer delegation, the delegator temporarily looses his/her rights during the time of delegation.
4. The role administrator is not delegable.
5. The permission of deleting borrower account is not delegable.
6. The director can delegate, on behalf of a secretary, the secretary's role (or some his/her permitted actions) to a librarian (e.g. during the secretary's absence).
7. If a librarian empowered in role *secretary* by delegation is no longer able to perform this task, then he/she can delegate, again, this role to another librarian.

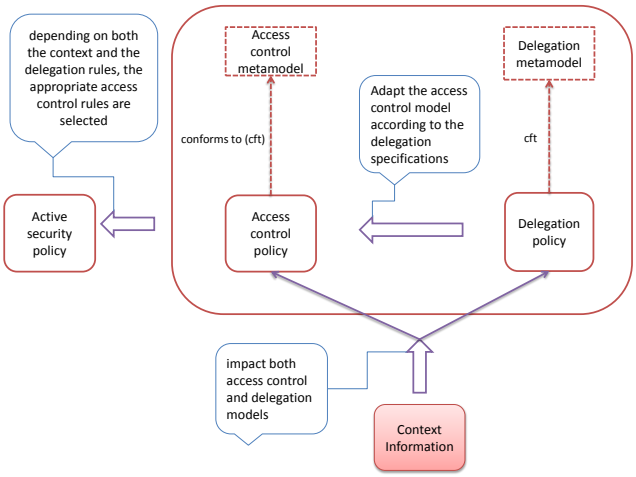

Figure 2: Delegation impacting Access Control

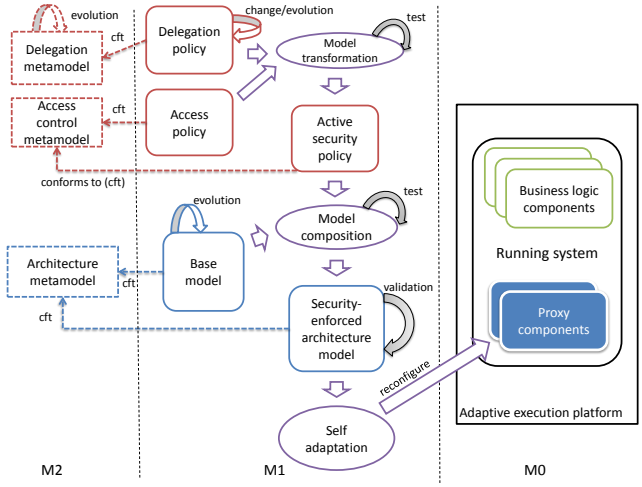

Figure 3: Overview of our approach

8. The secretary empowered in role *director* by delegation is not allowed to delegate/transfer, again, this role to another secretary.
9. A secretary is allowed to delegate his/her role to a librarian only and to one librarian at a given time.
10. A secretary is allowed to delegate his/her task of book delivery to a librarian only and scheduled on every Monday.
11. *Bill* can delegate his role and permitted actions only to *Bob*
12. *Bob* is not allowed to delegate his role.
13. *Alice* is not allowed to delegate her permitted action of book delivery.
14. Users can always revoke their own delegations.
15. The director can revoke users from their delegated roles.
16. A secretary can revoke librarians empowered in *secretary* role by delegation, even if he/she is not the grantor of this delegation (e.g. the grantor is the director or another librarian).

It can be seen that there are two levels of delegation rules: user-level (rules defined by a user: e.g. situations 1, 2, 3) and master-level (rules defined by a security officer: e.g. 4, 5, 6). Normally, delegations at user-level have to conform to rules at master-level. For example, the security officer can define that users of role *director* are able to delegate on behalf of users of role *secretary*. Then at user-level, *Bill* (director) can create a delegation rule to delegate, on behalf of *Alice*, her role (secretary) to *Jane* (librarian).

4. A MODEL-DRIVEN APPROACH FOR ADAPTIVE DELEGATION

4.1 Overview of our approach

In our approach, delegation is considered as a "meta-level" mechanism which impacts the existing access control policies, like an aspect can impact a base program. We claim that to handle advanced delegation rules, an ideal solution is to separate the delegation rules from the access control policy, each being specified in isolation, and then compose/weave them together to obtain a new access control policy (called active security policy) reflecting the delegation-driven policy (Fig. 2). We present our metamodel (DSML) for specifying delegation based on RBAC in Section 4.2.

The separation of concerns is not only between delegation and access control, but also with the business logic of the system. Fig. 3 presents a wider view of the overall approach. In order to enforce security policy to the system, the core business architecture model of the system is composed with the active security policy previously obtained. The architecture model is expressed in another DSML, called architecture metamodel (an architecture modeling language described in [20]). The idea is to reflect security policy into the system at the architecture level. Section 4.3 defines transformation rules to show how security concepts are mapped into architectural concepts.

The security-enforced architecture model obtained above is a pure architecture model which by itself reflects how the security policy is enforced in the system. It is important to note that the security-enforced architecture model is not used for generating the whole system but only the proxy components. These proxy components can be adapted and integrated with the running system at runtime to physically enforce the security policy. The adaptation and integration can be done by leveraging the runtime adaptation mechanisms provided by modern adaptive execution middleware platforms. The approach of generating proxy components overcomes some main limitations of [20]. Section 4.4 is dedicated to discuss our strategy for adaptation and evolution.

4.2 Delegation metamodel

Our metamodel displayed in Fig. 4 defines the conceptual elements and their relationships that can be used to specify access control and delegation policies. Because delegation mechanism is based on RBAC, we first explain the main conceptual elements of role-based access control. Then, we show how our conceptual elements of delegation, based on the RBAC conceptual elements, can be used to specify various delegation features.

As shown in Fig. 4, the root element of our metamodel is the *Policy*. It contains *Users*, *Roles*, *Resources*, *Rules*, and *Contexts*. Each user has one role. A security officer can specify all the roles in the system, e.g. *admin*, *director*, etc., via the *Role* element. In order to specify an access control policy, the security officer should have defined in advance the *resources* that should be protected from unauthorized access. Each resource contains some *actions* which are only

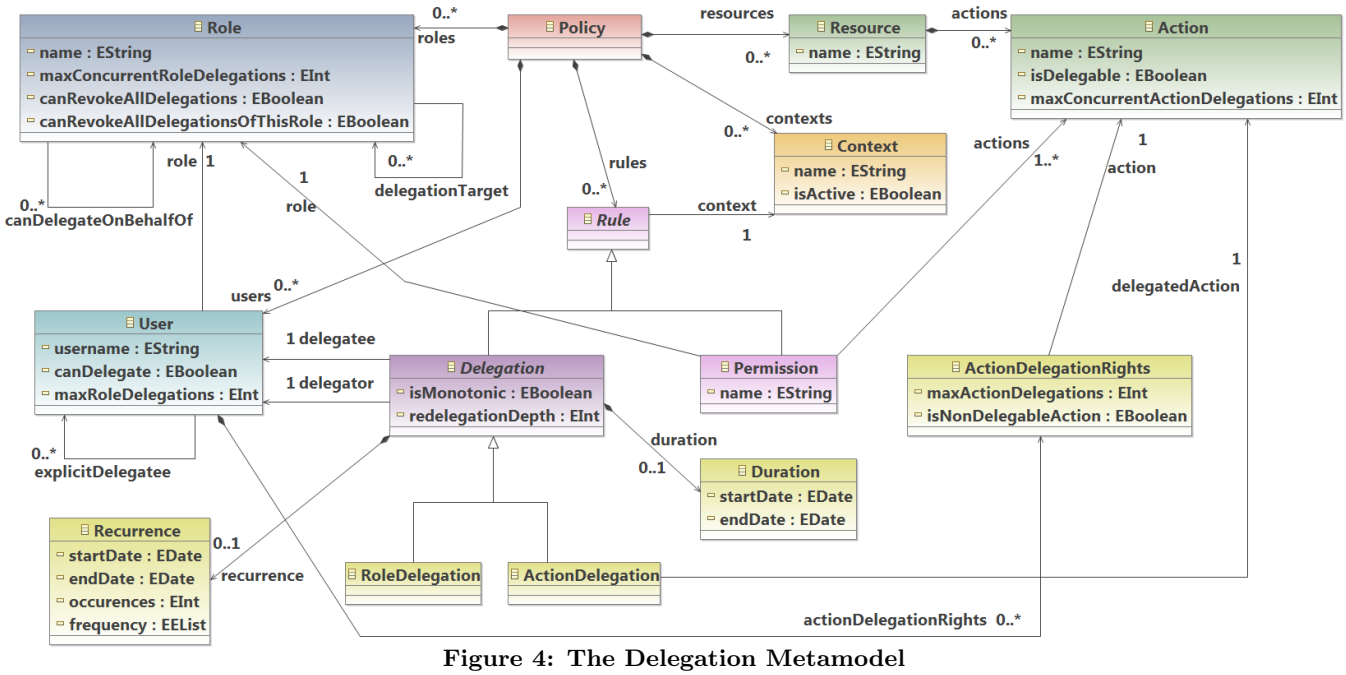

Figure 4: The Delegation Metamodel

accessible to authorized users. These protections are defined in rules: permission rules and delegation rules. Permission rules are used to specify which actions are accessible to users based on their *roles*. That means, without delegation rules or user-specific access control rules, every user is able to access the actions associated with his/her role only. Delegation rules are used to specify which actions are accessible to users by delegation. There are two basic types of delegation:
- **Role delegation**: When users empowered in role(s) delegated by other user(s), they are allowed to access not only actions associated with their roles but also actions associated with the delegated role(s).
- **Action delegation**: Instead of delegating their roles, users may want to delegate only some specific actions associated with their roles.
Another important aspect of our access control and delegation framework is the notion of *context*. It can be seen from our metamodel that every permission/delegation rule is associated with a context. A rule is only active within its context. The concept of context actually provides our model with high flexibility. Security policies can be easily adapted according to different contexts.

The full metamodel for specifying delegation is displayed in Fig. 4. It depicts the features that are supported by our delegation framework. All delegation management features are developed based on two basic types of delegation mentioned above. In the following, we present the delegation features and show how they can be specified, w.r.t. our metamodel.
- **Temporary delegation**: This is one of the most common types of delegation used by users. It describes when the delegation *starts* to be active and when it *ends*. The grantor can specify that the delegated role/action is authorized only during a given time interval, e.g. situation 1 of the running example in Section 3. Actually, this can be specified using the recurrence of delegation described below, but we want to define it separately because of its common use.
- **Monotonicity (Transfer of role or permissions)**: A property *isMonotonic* can be used to specify if a delegation is monotonic or non-monotonic. The former (*isMonotonic* = *true*) specifies that the delegated access right is available to both the delegator and delegatee after enforcing this delegation. The latter (*isMonotonic* = *false*) means the delegated role/action is transferred to the delegatee, and the delegator temporarily loses his rights while delegating, e.g. situation 3. In this case, the delegation is called a transfer.
- **Recurrence**: It refers to the repetition of the delegation. A user may want to delegate his role to someone else for instance every week on Monday. Recurrence defines how the delegation is repeated over time. It is similar to what is implemented in calendar system and more precisely the icalendar standard (RFC2445[1]). It has several properties; the *startDate* and *endDate* are the starting and ending dates of the recurrence. In addition, the *startDate* defines the first occurrence of the delegation. The *frequency* indicates one of the three predefined types of frequency, daily, weekly or monthly. The *occurrences* is the number of times to repeat the delegation. If the *occurrences* is for instance equals to 2 it means that it should only be repeated twice even when the *endDate* is not reached. An example of this delegation is situation 10.
- **Delegable roles and delegable actions**: These kinds of delegation define which roles and actions can be delegated and how. A policy officer can specify that a role can only be delegated/transferred to specific role(s), e.g. situation 9. If no *delegationTarget* is defined for a role, this role cannot be delegated/transferred, e.g. situation 4. If a role or action (*isDelegable* = *false*) is not delegable, it should never be included in a delegation rule. Moreover, a role can also be delegated by a user not having this role but his/her own role is specified as can delegate on behalf of a user in this role (*canDelegateOnBehalfOf* = *true*), e.g. situation 6.
- **Multiple delegations**: It should be possible to define the

[1]http://www.rfc-editor.org/info/rfc2445

max number of concurrent delegations in which the same role or action can be delegated at a given time. The properties *maxConcurrentRoleDelegations* and *maxConcurrentActionDelegations* define how many concurrent delegations of the same role/action can be granted, e.g. situation 9. Moreover, it is possible to define for each specific user a specific maximum number of concurrent delegations of the same role/action: *maxRoleDelegations* and *maxActionDelegations*.
- **User specific delegation rights**: All user-specific elements are used to define more strict rules for a specific user rather for his/her role. There are other user-specific delegations than *maxRoleDelegations* and *maxActionDelegations*. It is possible to define that a specific user is allowed to delegate his role/permitted action(s) or not (*canDelegate* = *true* or *false*), e.g. situation 12. The property *isNonDelegableAction* specifies an action that a specific user cannot delegate, e.g. situation 13. Moreover, the security officer can define to which explicit user(s) only (*explicitDelegatee*) a user can delegate/transfer his role to, e.g. situation 11.
- **Multi-step delegation**: It provides flexibility in authority management, e.g. situations 7, 8. The property *redelegationDepth* is used to define whether or not the role/action of a delegation can be delegated again. When a grantor creates a new delegation, he/she can specify how many times the delegated role/action can be re-delegated. If the *redelegationDepth* = 0, it means that the role/action cannot be delegated anymore, e.g. situation 8. If the *redelegationDepth* > 0, that means the role/action can be delegated again and each time it is re-delegated, the *redelegationDepth* is decreased by 1.
- **Revocations**: All users can revoke their own delegations, e.g. situation 14. Security officer may set *canRevokeAllDelegations* = *true* for a role with a super revocation power in such a way that a user empowered in this role can revoke all delegations, e.g. situation 15. Moreover, a role can also be defined such that every user empowered in this role can revoke any delegation from this role (*canRevokeAllDelegationsOfThisRole* = *true*), even he/she is not the grantor of the delegation, e.g. situation 16.

Moreover, each possible instance of the security policy has to satisfy all necessary validation condition expressed as OCL invariants. For example[2], we can make sure that no delegation is out of target, meaning that delegatee's role has to be a delegation target of delegator's role:

context Delegation **inv** NoDelegationOutOfTarget: self.delegator.role.delegationTarget ->exists (t | t = self.delegatee.role)

Or to check that for every user, the number of concurrent role delegations cannot be over its thresholds:

context User **inv** NoRoleDelegationOverMax: RoleDelegation.allInstances ->select (d | d.delegator = self) ->size() ≤ self.role.maxConcurrentRoleDelegations and RoleDelegation.allInstances ->select (d | d.delegator = self) ->size() ≤ self.maxRoleDelegations

Other examples are to restrict the value of the *redelegationDepth* must not be negative, or *startDate* cannot be later than *endDate*:

context Delegation **inv** NonNegativeDeleDepth: self.redelegationDepth ≥ 0

context Duration **inv** ValidDates: self.startDate ≤ self.endDate

[2]Due to space restrictions, the OCL expressions presented here are not exhaustive.

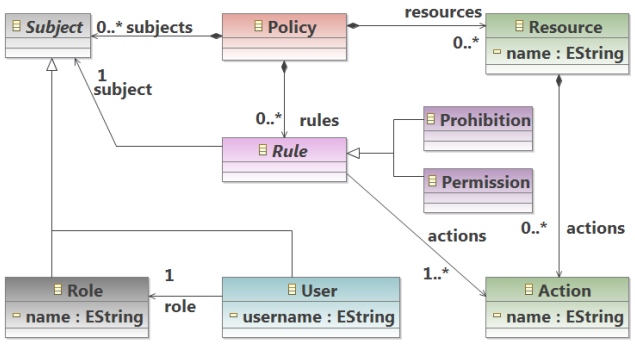

Figure 5: A pure RBAC metamodel

4.3 Transformations/Compositions

After specifying security policies by the DSML described in Section 4.2, it is crucial to dynamically enforce these policies into the running system. Transformations play an important role in the dynamic enforcement process. Via model transformations, security models containing delegation rules and access control rules are automatically transformed into component-based architecture models. Note that instances of security models and architecture models are checked before and after model transformations, using predefined OCL constraints.

The model transformation is executed according to a set of transformation rules. The purpose of defining transformation rules is to correctly reflect security policy at the architectural level. Based on transformation rules, security policy is automatically transformed to proxy components, which are then integrated to the business logic components of the system in order to enforce the security rules. The metamodel of component-based architecture can be found in [20] and an instance of it can be seen in Fig. 7. We first describe the transformation that derives an access control model according to delegation rules (Section 4.3.1), and then describe another transformation to show how security policy can be reflected at the architecture level (Section 4.3.2). Moreover, we also show an alternative way of transformation that combines two steps into one step.

4.3.1 Adapting RBAC policy model to reflect delegation

Within the security model shown in Fig. 2, delegation rules are considered as "meta-level" mechanisms that impact the access control rules. The appropriate access control rules and delegation rules are selected depending on the context information and/or the request of changing security rules coming from the system at runtime. According to the currently active context (e.g. WorkingDays), only **in-context** delegation rules and **in-context** access control rules of the security model (e.g. rules that are defined with context = WorkingDays) are taken into account to derive the active security policy model (Fig. 2). Theoretically, we could say that delegation rules impact the core RBAC elements in the security model in order to derive a pure RBAC model (without any delegation and context elements) which conforms to a "pure" metamodel of RBAC (Fig. 5). Delegation elements of a security policy model are transformed as follows:

A.1: Each **action delegation** is transformed into a new permission rule. The *subject* of the permission is *user* (dele-

gatee) object. The set of *actions* of the permission contains the delegated action.

A.2: Each **role delegation** is transformed as follows. First, a set of actions associated to a role is identified from the permissions of this role. Then, each action is transformed into a permission like transforming an **action delegation** described above.

A.3: A **temporary delegation** is only taken into account in the transformation if it is in active duration defined by the **start** and **end** properties. In fact, when its active duration starts the (temporary) action/role delegation is transformed into permission rule(s) as described above. When its active duration ends the temporary delegation is removed from the policy model.

A.4: If an action delegation is of type transfer delegation (**monotonic**), then it is transformed into a permission rule and a prohibition rule. The *subject* of the permission is the *user*-delegatee object. The set of *actions* of the permission contains the delegated action. The *subject* of the prohibition is the *user*-delegator object. The set of *actions* of the prohibition contains the delegated action.

A.5: If a role delegation is of type transfer delegation, then it is also transformed into a permission rule and a prohibition rule. The *subject* of the permission is the *user*-delegatee object. The set of *actions* of the permission contains the delegated actions. The delegated actions here are the actions associated with this role. The *subject* of the prohibition is the *user*-delegator object. The set of *actions* of the prohibition also contains the delegated actions.

A.6: If a delegation rule is defined with a **recurrence**, based on the values set to the recurrence, the delegation rule is only taken into account in the transformation within its *fromDate* and *untilDate*, repeated by *frequency* and limited by *occurrences*. In other words, only active (during recurrence) delegation rules are transformed.

A.7: (User-specific) If a user is associated with any **non-delegable action**, the action delegation containing this action and this user (as delegator) is not transformed into a permission rule. Similarly, if a user is specified as he/she **cannot delegate** his/her role/action, no role/action delegation involving this user is transformed.

A.8: (Role/action-specific) Any delegation rule with a **non-delegable** role/action will not be transformed. In fact, a delegation rule is only transformed if it satisfies (at least) both user-specific and role/action-specific requirements.

A.9: Only a role delegation to a user (delegatee) whose role is in the set of **delegationTarget** will be considered in the transformation.

A.10: Before any delegation is taken into account in the transformation, it has to satisfy the requirements of **max concurrent action/role delegations**. Note that the user-specific values have higher priorities than the role-specific values.

A.11: A delegation is only transformed if its **redelegationDepth** > 0. Whenever a user empowered in a role/an action by delegation re-delegates this role/action, the newly created delegation is assigned a **redelegationDepth** = the previous **redelegationDepth** - 1.

After transforming all delegation rules, we obtain a pure RBAC model which reflects both the delegation model and access control model. This pure RBAC model is then transformed into a security-enforced architecture model as described next.

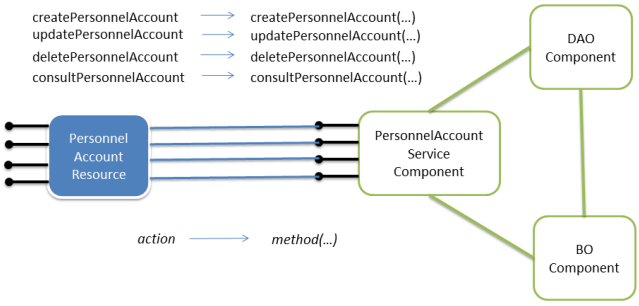

Figure 6: Mapping Resources to Business Logic Components

4.3.2 *Transformation of Security Policy to Component-based Architecture*

The transformation rules are defined below. The goal is to transform every security policy model (pure RBAC model obtained in step 1) which conforms to the metamodel shown in Fig. 5 to a component-based architecture model which conforms to the metamodel described in [20]. However, both the security policy model and the *base model* provided by a system designer are used as inputs for the model transformation/composition. Via a graphical editor, the security designer must define in advance how the resource elements in the policy model are related to the business components in the *base model*. Fig. 6 shows how each action in the policy can be mapped to the Java method in the business logic.

Because the *base model* already conforms to the architecture metamodel, we now only focus on transforming the security policy model into the security-reflected architecture model. As we know, this transformation/composition process will also weave the security-reflected elements into the base model in order to obtain the security-enforced architecture model.

The core elements of RBAC like *resource*, *role*, and *user* are transformed following these transformation rules. All the transformation rules make sure that the security policy is reflected at the architectural level.

R-A.1: Each *resource* is transformed into a component *instance*, called a *resource* proxy component. According to the relationship between the resource elements in the policy model and the business components in the *base model*, each *resource* proxy component is connected to a set of business components via bindings. To be more specific, each action of a resource element is linked to an operation of a business component (Fig. 6). By connecting to business components, a *resource* proxy component provides and requires all the services (actions) offered by the resource.

R-A.2: Each *role* is also transformed into a *role* proxy component. According to the granted accesses (permission rules associated with this role) to the services provided by the resources, the corresponding *role* proxy component is connected to some *resource* proxy component(s) (Fig. 7). A *role* proxy component is connected to a *resource* proxy component by transforming granted accesses into ports and bindings. Each (active) access granted to a role is transformed into a pair of ports: a client port associated with the *role* proxy component, a server port associated with the *resource* proxy component, and a binding linking these ports.

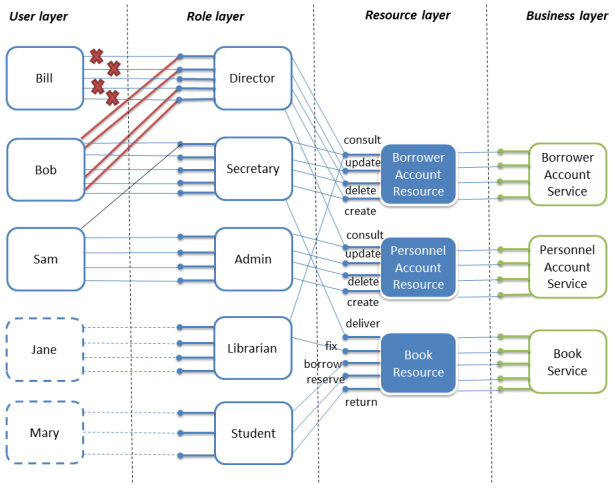

Figure 7: Architecture reflecting security policy before and after adding a delegation rule (bold lines)

R-A.3: Each *user* element defined in the policy model is also transformed into a *user* proxy component. Because each user must have one role, each *user* proxy component is connected to the corresponding *role* proxy component. However, each user may have access to actions associated to not only his/her role but also to actions associated to other roles by delegation. Thus, each *user* proxy component may connect to several *role* proxy components. The connection is established by transforming each access granted to a user into a pair of ports: a client port associated with the *user* proxy component, a server port associated with the corresponding *role* proxy component (providing the access/port), and a binding linking these ports (Fig. 7). Actually, the granted accesses are calculated not only from *permission* rules but also from *prohibition* rules. Simply, the granted accesses that equal permissions exclude prohibitions.

In our approach, revocation of a delegation simply consists in deleting the corresponding delegation rule. In this way, the revocation is reflected at the architectural level and physically enforced in the running system. Moreover, both the delegator and delegatee elements will be removed if these users are not involved in any delegation rules. As described above, user elements are transformed into proxy components. However, it is important to stress that only users involved in delegation rules (e.g. Bill, Bob and Sam in Fig. 7) are created in the security policy model and transformed into proxy components. Users who are not involved in any delegation rules (e.g. Jane and Mary in Fig. 7), are manipulated as session objects which directly access the services offered by the corresponding role proxy components.

Two steps described above are two separate model transformations that mainly used to explain how delegation can be considered as a "meta-level" mechanism for administrating access rights. The first model transformation is to transform a delegation-driven security model into a pure RBAC model. The second model transformation is to transform the RBAC model into an architecture model. In fact, these two steps could be done in only one model transformation that directly transforms the delegations, the access control policy and the business logic model into an architecture model reflecting the security policy. However, this alternative way (described in the following) has the disadvantage of losing the intermediate security model (the active security policy) that could be useful for traceability purpose.

4.3.3 *An alternative way: using only one transformation*

In this way, we have to define other transformation rules to transform directly every security policy model which conforms to the metamodel, shown in Fig. 4, to a component-based architecture model which conforms to the architecture metamodel described in [20].

Core elements of RBAC like *resources*, *roles*, and *users* are transformed following these transformation rules:

R-B.1: Each *resource* is transformed into a component *instance*, called a *resource* proxy component (already presented).

R-B.2: Each *role* also is transformed into a *role* proxy component (already presented). The only difference is the *context* has to be taken into account (in Section 4.3.2, no *context* existed because *context* already dealt with in Section 4.3.1). Because every permission is associated with a *context*, we only transform permissions with the *context* that is active at the moment.

R-B.3: Each *user* element defined in the policy model is also transformed into a *user* proxy component. However, the connection (via bindings) from a *user* proxy component to the *role* proxy component(s) is not only depended on the user's role but also delegation rules that the corresponding user involved in. The transformation of delegation rules is presented below.

All the transformation rules above make sure that access control rules are reflected at the architecture level. However, the delegation rules will impact this transformation process in order to derive the security-enforced architecture model reflecting both access control and delegation policy. Delegation elements of a policy model are transformed as follows:

R-B.4: Each action involved in an **action delegation** is transformed into a pair of ports and a binding. A client port (representing the required action) is associated with the user (*delegatee*) proxy component. The binding links the client port to the corresponding server port (representing the same action provided) that associated with the *role* proxy component reflecting the role of the *delegator*.

R-B.5: Each **role delegation** is transformed in a similar way as **action delegation**. First, a set of actions associated to a role can be identified from the permissions of this role. Then, each action in the set is transformed into a pair of ports and a binding as transforming an action delegation.

R-B.6: A **temporary delegation** is only transformed into bindings if it is still in active duration defined by **start** and **end** properties.

R-B.7: If a delegation is of type **transfer delegation**, then both user elements (delegator and delegatee) are transformed into delegator and delegatee proxy components as described above. The delegator proxy component is not connected to the corresponding role proxy component because he/she already transfered his/her access rights to the delegatee. Fig. 7 shows a change in the architecture when *Bill* transfers his role to *Bob*.

R-B.8: If a delegation is defined with a **recurrence**, based on the values set to recurrence, the delegation rule is only active during the recurrence (similar to A.6).

R-B.9: If a user is associated with any **non-delegable ac-**

tion, the delegation of this action is not taken into account while doing the transformation. Similarly, if a user is specified as he/she **can not delegate** his/her role/action, no delegation requested by this user will be transformed.
R-B.10: Only a role delegation to a user (delegatee) whose role is in the set of **delegationTarget** will be consider in the transformation.
R-B.11: Before any delegation is taken into account in the transformation, it has to satisfy the requirements of **max concurrent action/role delegations**. Note that the user-specific values have higher priorities than the role-specific values.
R-B.12: A delegation is only transformed if its **redelegationDepth** > 0. Whenever a user empowered in a role/an action by delegation re-delegates this role/action, the newly created delegation is assigned a **redelegationDepth** = the previous **redelegationDepth** - 1.

By taking into account delegation rules while transforming access control rules of policy model into security-enforced architecture model, both delegation and access control rules are reflected at the architecture level.

4.4 Adaptation and Evolution strategies

The model transformation/composition presented in Section 4.3 ensures that the security policies are correctly and automatically reflected in an architectural model of the system. The key steps to support delegation (i.e. specifications and transformations) are already presented in Sections 4.2 and 4.3. The last step consists in a physical enforcement of the security policy by means of a dynamic adaptation of the running system. In this section, our adaptation and evolution strategy is discussed.

4.4.1 Adaptation

The input for the adaptation process is a newly created security-enforced architecture model (Fig. 8). First, this new architecture model is validated using simulation or invariant checking [18]. This valid architectural model actually represents the new system state the runtime must reach to enforce the new security policy of the system. According to the classical MAPE control loop of self-adaptive applications, our reasoning process performs a comparison (using EMFCommpare) between the new architecture model (target configuration) and the current architecture model (kept synchronized with the running system) [19]. This process triggers a code generation/compilation process, and also generates a safe sequence of reconfiguration commands [18]. Actually, the code generation/compilation process is only triggered if there are new proxy components, e.g. new user proxy components involved in delegation, that need to be introduced into the running system. The dynamic adaptation of the running system is possible thanks to modern adaptive execution platforms like OSGi [26] or Fractal [8] which provide low-level APIs to reconfigure a system at runtime. The running system is then reconfigured by executing the safe sequence of commands, compliant to the platform API, issued by the reasoning process. In this process, the generation/compilation phase is time consuming. However, this phase has no impact on the running system, which remains stable until being adapted by executing the reconfiguration script. Thus, the actual adaptation phase only lasts for several milliseconds, during which the system is not available.

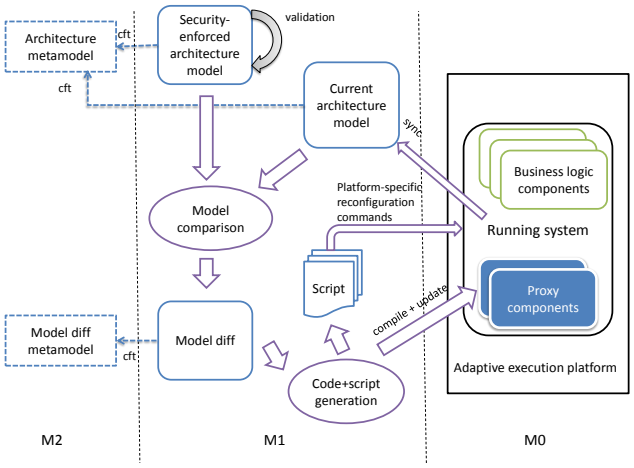

Figure 8: Overview of our adaptation strategy

In [20], the adaptation is entirely based on executing platform - specific reconfiguration scripts specifying which components have to be stopped, which components and/or bindings should be added and/or removed. This results in several limitations regarding delegation mechanisms:
L.1: Using reconfiguration scripts only implies to create all the potentially needed ports (used for bindings between *user* proxy components) beforehand. But all the combinations of users, roles, resources, actions could lead to a combinatorial explosion and make it infeasible for implementation.
L.2: In [20], the delegation between users are reflected using bindings connecting one *user* proxy component to another. But this approach is not suitable for supporting complex delegation features. For example, a transfer delegation will be reflected such as adding bindings between the *delegator* and *delegatee* but removing bindings between *delegator* and the corresponding *role* proxy component. Consequently both *delegator* and *delegatee* cannot access the resource, that does not correctly reflect a transfer delegation.

L.1 can be solved by the automatic re-generation of proxy components and bindings between them according to changes in the architectural model. Moreover, as mentioned in Section 4.3.2, only users involved in a delegation are transformed into *user* proxy components with necessary ports and bindings. In this way, only required ports and bindings are created dynamically. **L.2** is solved by our model transformation approach. All complex delegation features are considered as "meta-level" mechanisms that impact access control rules. In this way, a transfer delegation will be reflected such as adding bindings between the *delegatee* and the corresponding delegated *role* proxy component, but removing bindings between *delegator* and the corresponding *role* proxy component.

Our adaptation strategy could take more time than simply running a reconfiguration script because of the generation and compilation time of newly generated proxy components. But the process of generating and compiling new proxy components could not harm the performance because each proxy component is very light-weight and only necessary proxy components are generated (see Section 5).

4.4.2 Evolution

In [20], the evolution of the security policy is not totally

dealt with. It is possible to run a reconfiguration script to reflect the changes like adding, removing and updating rules. But adding a new user, role or resource requires the generation and compilation of new proxy components, which is impossible using only reconfiguration scripts. Thus, our strategy of automatically generating and compiling proxy components (see Section 5) is more practical w.r.t evolution.

Another important aspect of evolution relates to the addition, removal or update of resources and actions in the business logic. The base architecture model can be updated with the changes in the business logic, e.g. a new resource is added. On the other side, security officers can manually update the mappings (Fig. 6) following the changes of resources/actions in the base architecture model. By composing the security model with the base architecture model as described earlier, the security policy is evolved together with the business logic of the system.

5. EVALUATION

To evaluate the feasibility of our approach, we have applied it on three different Java-based case studies, which have also been used in our previous research work on access control testing [21]:

1) LMS: already described in our running example.

2) VMS[3]: The Virtual Meeting System offers simplified web conference services. The virtual meeting server allows the organization of work meetings on a distributed platform. When connected to the server, a user can enter (exit) a meeting, ask to speak, eventually speak, or plan new meetings. There are three resources (Meeting, Personnel Account, User Account) and six roles (Administrator, Webmaster, Owner, Moderator, Attendee, and Non-attendee) defined for this system with many access control rules, and delegation situations between the users of each role.

3) ASMS: The Auction Sale Management System allows users to buy and sell products online. Each user in the system has a profile including some personal information. Users wanting to sell a product (sellers) are able to start a new auction by submitting a description of the product, the starting and ending date of the auction. There are five resources (Sale, Bid, Comment, Personnel Account, User Account) and five roles (Administrator, Moderator, Seller, Senior Buyer, and Junior Buyer) defined for this system, also with many access control rules, and delegation situations between users of each role.

Table 1: Size of each system in terms of source code.

	# Classes	# Methods	# LOC
LMS	62	335	3204
VMS	134	581	6077
ASMS	122	797	10703

We applied our approach to enable dynamic security enforcement for these systems, and examined how effective our approach is. Table 1 provides some information about the size of these three applications (the number of classes, methods and lines of code). In terms of security policies, Table 2 shows the number of access control (AC) rules and delegation rules defined for each system, used in our experiments.

[3]For more information about VMS (server side), please refer to http://franck.fleurey.free.fr/VirtualMeeting.

Table 2: Security rules defined for each system.

	# AC rules	# Delegations	Total
LMS	23	4	27
VMS	36	8	44
ASMS	89	8	97

All these systems are component-based systems. The business components of each system contain the business logic, e.g. Book Service component, Personnel Account component, Meeting, Sale, Authenticate component, Data Access Object components, etc. To enable dynamic security enforcement for a system, the resources (components that have to be controlled) are specified in the base model, and mapped to the resources of the security policies. Our metamodels are applicable for different systems without any modification or adaptation. The structure of delegation and access control policies for all case studies is the same, only roles, users, resources, actions are specific to each case study. The proxy components are automatically generated and synchronized with the security policy model via model transformations and reconfiguration at runtime. The model-to-model transformation and model-to-text transformation (code generation) can be implemented using any transformation engines like Kermeta [22] (or ATL [4]) and Xpand [14]. The security policy models that are stored in eXistDB [16], a native XML database, can be easily modified by using XPath, XQuery, and XUpdate. For experimenting with performance of adapting the running system, we have implemented the model transformation/composition rules using not only Kermeta but also ATL.

There are two kinds of response time we would like to measure in our case studies: the authorization mechanism and the dynamic adaptation according to changing security policies. The experiments were performed on Intel Core i7 CPU 2.20 GHz with 2.91 GB usable RAM running on Windows 7 and Equinox [5]. Because all our access control and delegation rules are transformed to proxy components reflecting our security policy, response times to an access request only depends on method calls between these proxy components and business components (Fig. 7). Unsurprisingly, response time to every resource access is a constant, only about 1 millisecond, because the access is already possible or not by construction. In other words, our 3-layered architecture reflecting security policy enables very quick response, independently from the number of access control and delegation rules.

Table 3: Performance of weaving Security Policies using Kermeta and ATL.

	# Rules	Kermeta 1.4.1	ATL 3.2.1
LMS	27	4s	0.048s
VMS	44	7s	0.055s
ASMS	97	18s	0.140s

Regarding the adaptation process, Table 3 shows results of each case study for performing the model transformations of security policies mentioned in Table 2, using Kermeta 1.4.1 and ATL 3.2.1 correspondingly. At first, we used Kermeta 1.4.1 to implement our model transformations. However, the results shown in Table 3 have been disappointing. We have

[4]http://www.eclipse.org/atl/

[5]http://www.eclipse.org/equinox/

tried to use Kermeta 2.0.4 (the latest version of Kermeta at this moment, compiled to byte code, which means much better performances), but the tool is not mature yet. To know if this performance problem is inherently linked to our approach or simply linked to the use of Kermeta 1.4.1, we decided to also implement our model transformations using ATL 3.2.1. Our experiments show that the implementation using ATL 3.2.1 is much more efficient. We can conclude that the initial performance issue was due to Kermeta 1.4.1.

Note that the transformation, code generation and compilation are performed "offline" meaning that the running system is not yet adapted. The actual adaptation happens when the newly compiled proxy components are integrated into the running system to replace the current proxy components. This actual adaptation process takes only some milliseconds by using the low-level APIs to reconfigure a system at runtime provided by the modern adaptive execution platforms, e.g. OSGi [26]. Right after the new proxy components are up and running, the new security policy is really enforced in the running system.

6. RELATED WORK

There is a substantial work related to delegation as extension of existing access control models. Most researchers focused on proposing models solely relying on the RBAC formalism [25], which is not expressive enough to deal with all delegation requirements. Therefore, some other researchers extended the RBAC model by adding new components, such as new types of roles, permissions and relationships [2, 27, 1, 9, 23]. In [5], the authors proposed yet another delegation approach for role-based access control (more precisely for OrBAC model) which is more flexible and comprehensive. However, no related work has provided a model-driven approach for both specifying and dynamically enforcing access control policies with various delegation requirements. Compared to [20], we extend the model-based dynamic adaptation approach of [20] with some key improvements. More specifically, we propose a new DSML for delegation management, but also new composition rules to weave delegation in a RBAC-based access control policy. In addition, we present a new way (by generating proxy) to implement the adaptation of the security-enforced architecture of the system. Indeed, we provide an extensive support for delegation as well as co-evolution of security policy and security-critical system. That means our approach makes it possible to deeply modify the security policy (e.g. according to evolution of the security-critical system) and dynamically adapt the running system, which is often infeasible using the other approaches mentioned above.

In addition, several researchers proposed new flexible access control models that may not include delegation, but allow to have a flexible and easy to update policy. For instance, Bertino *et al.* [6] proposed a new access control model that allows expressing flexible policies that can be easily modified and updated by users to be adapted to specific contexts. The advantage of their model resides in the ability to change the access control rules by granting or revoking the access based on specific exceptions. Their model provides a wide range of interesting features that increase the flexibility of the access control policy. It allows advanced administrative functions for regulating the specification of access controls rules. More importantly, their model supports delegation, enabling users to temporarily grant other users some of their permissions. Furthermore, Bertolissi *et al.* proposed DEBAC [7] a new access control model based on the notion of event that allows the policy to be adapted to distributed and changing environments. Their model is represented as a term rewriting system [3], which allows specifying changing and dynamic access control policies. This allows having a dynamic policy that is easy to change and update.

As far as we know, no previous work tackled the issue of enforcing adaptive delegation. Some previous approaches were proposed to help modeling more general access control formalisms using UML diagrams (focusing on models like RBAC or MAC). RBAC was modeled using a dedicated UML diagram template [13], while Doan *et al.* proposed a methodology [10] to incorporate MAC in UML diagrams during the design process. All these approaches allow access control formalisms to be expressed during the design. They do not provide a specific framework to enable adaptive delegation at runtime. Concerning the approaches related to applying MDE for security, we can cite UMLsec [12], which is an extension of UML that allows security properties to be expressed in UML diagrams. In addition, Lodderstedt *et al.* [15] propose SecureUML which provides a methodology for generating security components from specific models. The approach proposes a security modeling language to define the access control model. The resulting security model is combined with the UML business model in order to automatically produce the access control infrastructure. More precisely, they use the Meta-Object facility to create a new modeling language to define RBAC policies (extended to include constraints on rules). They apply their technique in different examples of distributed system architectures including Enterprise Java Beans and Microsoft Enterprise Services for .NET. Their approach provides a tool for specifying the access control rules along with the model-driven development process and then automatically exporting these rules to generate the access control infrastructure. However, they do not directly support delegation. Delegation rules should be taken into account early and the whole system should be generated again to enforce the new rules. Our approach enables supporting directly the delegation rules and dynamically enforcing them by reconfiguring the system at runtime.

7. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an extensive model-driven approach for RBAC-based delegation. It has been shown that various delegation requirements can be specified using our delegation DSML. Our DSML supports complex delegation characteristics like temporary, recurrence delegation, transfer delegation, multiple and multi-step delegation, etc. We have shown that we can deal with revocation in a simple manner. Another main contribution of this paper is to provide adaptive delegation enforcement in which delegation is considered as a "meta-level" mechanism that impacts the access control rules. A complete model-driven framework has been proposed to enable dynamic enforcement of delegation and access control policies that allows the automatic configuration of the system according to the changes in delegation/access control rules. Moreover, our framework also enables an adaptation strategy that better supports co-evolution of security policy and the security-critical system. Our approach has been validated via three different case studies with consideration of performance and extensibility issues.

Our approach could be better supported using an optimized models@runtime framework such as Kevoree[6] instead of Equinox. We have not dealt with this idea yet in this paper, but keep it for our future work. Moreover, revocation mechanism in our current approach has not been completely taken into account, i.e. without options of strong/weak revocation. So far, we only focused on the delegation of rights, further work will also be dedicated to the delegation of obligations and the support for usage control [24].

8. REFERENCES

[1] G.-J. Ahn, B. Mohan, and S.-P. Hong. Towards secure information sharing using role-based delegation. *J. Netw. Comput. Appl.*, 30(1):42–59, Jan. 2007.

[2] E. Barka and R. Sandhu. Role-based delegation model/ hierarchical roles (rbdm1). In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 396–404, Washington, DC, USA, 2004. IEEE Computer Society.

[3] S. Barker and M. Fernández. Term rewriting for access control. In *DBSec*, pages 179–193, 2006.

[4] D. Basin and J. Doser. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software*, (1945):353–398, 2006.

[5] M. Ben-Ghorbel-Talbi, F. Cuppens, N. Cuppens-Boulahia, and A. Bouhoula. A delegation model for extended rbac. *Int. J. Inf. Secur.*, 9(3):209–236, June 2010.

[6] E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Trans. Inf. Syst.*, 17(2):101–140, 1999.

[7] C. Bertolissi, M. Fernández, and S. Barker. Dynamic event-based access control as term rewriting. In *DBSec*, pages 195–210, 2007.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL Component Model and its Support in Java. *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

[9] J. Crampton and H. Khambhammettu. Delegation in role-based access control. *Int. J. Inf. Sec.*, 7(2):123–136, 2008.

[10] T. Doan, S. Demurjian, T. C. Ting, and A. Ketterl. Mac and uml for secure software design. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 75–85, New York, NY, USA, 2004. ACM.

[11] 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.

[12] J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *UML'02: 5th International Conference on The UML*, pages 412–425, Dresden, Germany, 2002. Springer-Verlag.

[13] D.-K. Kim, I. Ray, R. B. France, and N. Li. Modeling role-based access control using parameterized uml models. In *FASE*, pages 180–193, 2004.

[6] http://www.kevoree.org/, last access March 2012

[14] B. Klatt. Xpand: A closer look at the model2text transformation language. *Language*, (10/16/2008), 2007.

[15] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *UML'02: 5th International Conference on The UML*, pages 426–441, Dresden, Germany, 2002. Springer-Verlag.

[16] W. Meier. exist: An open source native xml database. In *Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops*, pages 169–183. Springer, 2002.

[17] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, Oct. 2009.

[18] B. Morin, O. Barais, G. Nain, and J. Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *ICSE'09: 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[19] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 782–796, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] B. Morin, T. Mouelhi, F. Fleurey, Y. Le Traon, O. Barais, and J.-M. Jézéquel. Security-driven model-based dynamic adaptation. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 205–214, New York, NY, USA, 2010. ACM.

[21] T. Mouelhi, Y. L. Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 171–180, Washington, DC, USA, 2009. IEEE Computer Society.

[22] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713*, pages 264–278. Springer, 2005.

[23] S. Na and S. Cheon. Role delegation in role-based access control. In *Proceedings of the fifth ACM workshop on Role-based access control*, RBAC '00, pages 39–44, New York, NY, USA, 2000. ACM.

[24] R. Sandhu and J. Park. Usage control: A vision for next generation access control. *V. Gorodetsky et al. (Eds.): MMM-ACNS 2003, LNCS 2776*, 1:17–31, 2003.

[25] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb. 1996.

[26] O. The OSGi Alliance. Osgi service platform core specification, release 4.1. 2007.

[27] X. Zhang, S. Oh, and R. Sandhu. Pbdm: a flexible delegation model in rbac. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, SACMAT '03, pages 149–157, New York, NY, USA, 2003. ACM.