Vers l’intégration automatique d’une politique de sécurité Or-BAC

Yliès Falcone, Mohamad Jaber

To cite this version:

Yliès Falcone, Mohamad Jaber. Vers l’intégration automatique d’une politique de sécurité Or-BAC. 2007. ⟨hal-05138468⟩

HAL Id: hal-05138468

https://hal.science/hal-05138468v1

Preprint submitted on 9 Jan 2026

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Schedae, 2007

Prépublication n° 48 | Fascicule n° 2

Vers l'intégration automatique d'une politique de sécurité Or-BAC

Yliès Falcone, Mohamad Jaber

Vérimag & Laboratoire d'Informatique de Grenoble

Ylies.Falcone@imag.fr, Mohamad.Jaber@imag.fr

Résumé :

Nous proposons une technique pour intégrer automatiquement dans une application une politique de sécurité définie en Or-BAC. L'application initiale est purement fonctionnelle, et la politique restreint le comportement en ajoutant des limitations guidées par des besoins de sécurité. Partant de règles décrivant des droits d'accès nous générons des aspects de sécurité. Ensuite, ces aspects sont utilisés pour construire un nouveau système en intégrant automatiquement la politique dans l'application originale. L'intégration est présentée pour le langage Java.

Mots-clés : sécurité, Or-BAC, programmation par aspects, politique, génération.

1 Introduction, préliminaires

Pour répondre aux besoins de sécurité croissants des logiciels et des systèmes d'information une approche largement utilisée aujourd'hui est celle basée sur les politiques de sécurité. La sécurité du système est assurée via l'application d'un ensemble de règles et de recommandations s'appliquant à divers niveaux tant physique que logique. Une des approches possibles est d'établir un contrôle d'accès qui régit les autorisations d'accès de sujets (des entités du systèmes) à des objets (les ressources du systèmes). Divers modèles ont été proposés comme les modèles d'accès discrétionnaires et les modèles de flux d'information, jusqu'au modèle RBAC [1] et ses dérivés. Un problème classique est alors d'assurer que le système est conforme à la politique exprimée.

Dans cet article nous utilisons Or-BAC (Organisation Based Access Control) comme politique de contrôle d'accès et la programmation par aspects pour sa mise en application.

Or-BAC – Le modèle Or-BAC [2, 3] généralise les modèles RBAC et ajoute une dimension organisationelle à la politique. Une *organisation* est une entité qui a la charge de gérer un ensemble de règles de sécurité (obligations, permissions, interdictions). Les opérations du système sont appelées *actions*. Un *sujet* est une entité active du système pouvant réaliser des actions au sein de celui-ci. Par opposition aux sujets, les *objets* sont les entités non

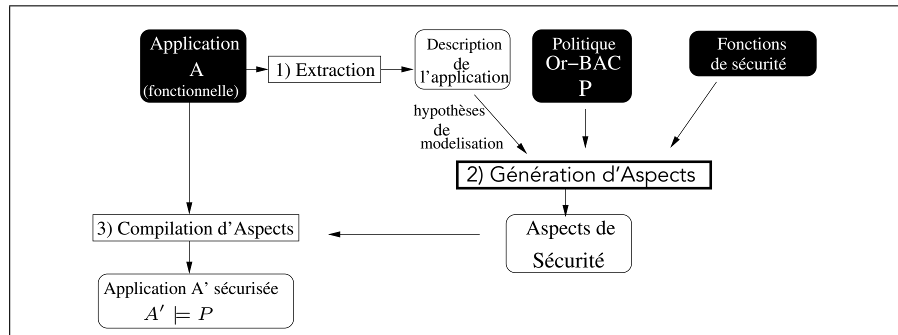

Fig. 1: Vue d'ensemble de notre approche.

actives du système (soumises aux opérations des sujets). L'organisation permet de structurer les entités du système. Les sujets (resp. objets, actions) sont abstraits en *rôles* (resp. *vues*, *activités*). Une notion de *contexte* [4] est ajoutée définissant les circonstances dans lesquelles une règle s'applique.

Programmation Orientée Aspect – Un *aspect* [5] regroupe un ensemble de jointpoints, pointcuts, et codes advices. Un *joinpoint* est un point dans le programme où peuvent agir différents aspects. Des exemples de jointpoints sont les méthodes, les constructeurs, les classes…Les *pointcuts* sont des éléments liés au flot d'exécution du programme autour desquels on peut greffer l'action des aspects. De manière abstraite, un pointcut définit un ensemble de jointpoints, une "coupe". Le *code advice* définit le code greffé par l'aspect dans l'application d'origine. Dans le cas de code advice de type *before* (resp. *after*, *around*), celui-ci sera intégré avant (resp. après, autour) le pointcut de l'aspect.

Approche proposée – L'approche que nous proposons dans cet article peut être vue de plusieurs manières. Tout d'abord il est possible de la considérer comme l'ajout automatique de sécurité à une implantation. Partant d'un système et sa politique Or-BAC, nous utilisons les règles de la politique pour modifier l'application initiale en utilisant les techniques de programmation par aspects. Pour cela notre approche propose une modélisation/intégration des concepts Or-BAC dans l'application initiale. À cet effet nous combinons des règles Or-BAC pour générer des aspects de sécurité. Notre approche est également une méthode de déploiement de politiques Or-BAC pour un type d'applications.

2 Génération d'aspects de sécurité à partir de règles Or-BAC

L'approche esquissée dans l'introduction repose sur les étapes suivantes (cf Figure 1) :

1. extraction des informations du système (classes, hiérarchie…) ;
2. génération d'aspects en utilisant la politique et les informations sur le système ;
3. intégration des aspects et du code non-fonctionnel dans l'application d'origine en utilisant le tissage d'aspects.

Le système initial est développé sans intégrer les règles de sécurité (tout est autorisé), *e.g.* pas de notion d'utilisateur ou de fonction d'authentification. Cela signifie que initialement seules les activités fonctionnelles du système sont considérées. Les activités non-fonctionnelles du système sont supposées être fournies séparément (*e.g.* mécanisme d'authentification, support des utilisateurs). La spécification d'une politique de sécurité Or-BAC est faite à un niveau organisationnel (abstrait) indépendamment de son déploiement. Il est donc nécessaire de lier les concepts Or-BAC avec l'architecture de l'application. La génération est présentée pour le langage Java. Dans notre approche, les phases 2 et 3 sont automatiques, la première requiert l'étude de l'architecture du système initial.

2.1 Extraction des informations du système

Mapping depuis les politiques Or-BAC vers Java – Nous lions les concepts de Or-BAC et de Java, les correspondances sont résumées dans la Tableau 1. Pour le système initial, représenté comme un programme orienté objet, nous considérons les notations suivantes. Nous dénotons par *Actions* (resp. *Activités*, *Vues*, *Classes*) l'ensemble des actions (resp. activités, vues, classes) adressées par la politique. Nous lions la notion de classe à la notion de vue en Or-BAC , la notion de méthode à la notion d'action, et la notion d'objet (instance de classe) à la notion d'objet Or-BAC. Parmi toutes les méthodes du programme, nous considérons un sous-ensemble $Methodes$ de toutes les méthodes du programme initial. Nous considérons les fonctions $MethodeAction : Actions \rightarrow Methodes$ (qui à chaque action de la politique fait correspondre une méthode du programme initial), et $MethodesActiviteSecurite$ (qui à chaque activité de sécurité associe la méthode correspondante). Nous définissons les fonctions $MethodeClass : Classes \rightarrow 2^{Methodes}$, $MethodeVue : Vues \rightarrow 2^{Methodes}$.

Pour déterminer quelles méthodes (correspondant aux actions d'une activité) s'appliquent sur les objets instances de la classe correspondant à une vue, il faut faire l'intersection entre les méthodes de l'activité et celles de la classe. Alors l'application d'une activité à une vue correspond aux actions de l'activité qui sont applicables sur la classe *v*.

Or-BAC (abstrait)	Implantation Orientée Objet (concret)
Action α	Méthode $ActionMethode(a)$
Objet o de la vue v	Objet instance de la classe $ClasseVue(v)$
Activité a sur vue v	($MethodeClasse(v) \cap MethodeActivite(a)$) sur les objets instances de la classe $ClasseVue(v)$
Contexte	Une méthode pouvant vérifier si l'état courant de l'application vérifie le contexte

Tab. 1: Correspondance entre concepts Or-BAC et OO.

Considérations de sécurité – Notre génération considère également l'utilisation d'activités de sécurité qui peuvent être absentes de l'implantation initiale. Ainsi toutes les actions des activités de sécurité adressées par la politique sont supposées être implémentées à part et disponibles en entrée pour notre génération.

Parmi les fonctions de sécurité utilisées dans la génération, des activités impliquées dans la politique peuvent être des activités de sécurité modifiant le rôle ou le contexte de l'utilisateur courant. Par exemple, l'utilisateur peut entrer dans le système avec un rôle quelconque puis dans le cas où il fait appel à une activité qui est interdite pour son rôle courant, on peut l'obliger à s'authentifier pour entrer avec un nouveau rôle dans lequel cette activité peut être autorisée.

2.2 Génération d'aspects

Les activités de sécurité ont pour but de modifier le rôle ou le contexte de l'utilisateur. Prenons l'exemple où l'on veut obliger un utilisateur à s'authentifier dans le cas où il applique une activité interdite pour son rôle courant. La génération d'aspect doit se faire en prenant en compte d'abord les règles comprenant des activités de sécurité. C'est pourquoi dans la génération d'aspects à partir des règles nous commençons par la génération des règles d'obligation qui contiennent une activité modifiant le rôle de l'utilisateur, puis on génère les autres obligations et enfin les interdictions. Nous présentons uniquement la génération pour les obligations avec une activité de sécurité.

Ces règles d'obligation sont de la forme $Obligation(org, r, a_1, _, cxt)$ (a_1 étant une activité de sécurité), où $cxt = input(a_2(v_2)) \wedge cxt_2$. Dans ce cas le $a_2(v_2)$ se trouve seulement

```
public aspect Obligation_Sécurité {
    pointcut a2() :
      execution (* Class-Vue(v2).*(..)) &&
      if (Appartient(thisJoinPoint.getSignature().toString()), Method-Activity(a2)) ;
      // Appartient : fonction qui vérifie l'appartenance à l'ensemble de méthodes
      Object around() throws Exception_Securite : a2(){
        if(RoleCourant()==r && Verification-Contexte(cxtclassical))
          {//RoleCourant fonction qui retourne le rôle courant de l'utilisateur
          MethodeActiviteSecurite(a1)() ;
          if(RoleCourant()==ri && Verification-Contexte(ci)) {
            throw new Exception_Securite() ;
          }
          else {
            Object x=proceed() ;
            return x ;
}}}}
```

Fig. 2: L'aspect généré pour une règle d'obligation avec activité de sécurité.

comme *input* dans le contexte puisque le but de cette règle est d'obliger l'utilisateur du rôle *r* d'appliquer un changement de rôle avant de pouvoir faire l'activité a_1.

On génère l'aspect (Figure 2) avec une coupe sur les méthodes ($MethodeClasse(v_2) \cap MethodeActivite(a_2)$) de la classe $ClasseVue(v_2)$, puis l'advice de type *around* dans lequel on vérifie la conformité avec la règle du rôle de l'utilisateur et du contexte restant cxt_2. Dans ce cas on fait appel à la méthode correspondant à l'action de l'activité a_1. Ensuite, on vérifie à nouveau le rôle et le contexte. On cherche s'il existe une règle d'interdiction contenant le même rôle et contexte : $Interdiction(org, r_i, a_2, v_2, c_i)$. Dans ce cas, on applique *Exception_Securite()* (pas d'appel à proceed), a_2 ne sera pas exécutée.

3 Conclusion

Cet article présente une technique de génération d'aspects à partir d'une politique de sécurité Or-BAC. Cette méthode permet le développement d'un système sans prendre en compte aucune notion de sécurité. Ensuite, à partir de la politique on génère un ensemble d'aspects de sécurité. L'ensemble des aspects sont ensuite intégrés dans l'application initiale en utilisant le compilateur d'aspects.

Ce travail ouvre plusieurs perspectives de recherche. Tout d'abord, il nous semble possible d'étendre la méthode présentée ici en limitant les hypothèses sur le système initial. Aussi, nous prévoyons d'intégrer cette méthode dans un outil de déploiement de politique Or-BAC s'intégrant dans MotOr-BAC [6]. Nous prévoyons également d'y intégrer une approche pour générer des tests [7] depuis une politique Or-BAC.

Références

[1] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. In *ACM Transactions on Information and System Security*, pages 4(3) :222–274, August 2001.

[2] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)*, June 2003.

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

[4] F. Cuppens and A. Miège. Modelling Contexts in the Or-BAC Model. In *19th Annual Computer Security Applications Conference (ACSAC '03)*, 2003.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, 1997.

[6] MotOr-BAC user manual, January 2007.

[7] K. Li, L. Mounier, and R. Groz. Test purpose generation from or-bac security rules. In *31st Annual IEEE International Computer Software and Applications Conference*, 2007.