Etat de l'art sur les modèles dits "de workflow"

Liên Tran

ETAT DE L'ART SUR LES MODELES DITS « DE WORKFLOW »

LIÊN TRAN

Table des matières

Date: 3 décembre 2009.

1. Introduction

1.1. **Objectifs et terminologie de l'état de l'art.**

Notre objectif est d'étudier des politiques de sécurité tenant compte du temps.

Selon Fred B. Schneider [1], une **politique de sécurité** peut porter sur le contrôle d'accès (restreindre les opérations que des agents peuvent effectuer sur des objets), le flux d'information (restreindre les informations que peuvent obtenir des agents sur des objets, par l'observation du comportement du système) ou la disponibilité (restreindre la possibilité pour des agents de bloquer l'utilisation d'une ressource pour d'autre agents).

Les politiques de **contrôle d'accès** associent des droits d'accès à des ressources (fichiers, données en mémoire, réseau, imprimante, ...) uniquement à certaines entités (utilisateurs, processus, ...). Elles sont souvent formalisées dans des modèles.

Le modèle **RBAC**, par exemple, permet d'exprimer une politique de contrôle d'accès basée sur des rôles. Un **rôle** découle généralement de la structure d'une entreprise. Les **utilisateurs** exerçant des fonctions similaires sont regroupés sous un même rôle. Des **permissions** sont assignées aux rôles. Les utilisateurs obtiennent des permissions via leur rôle, au cours des différentes **sessions** qu'ils lancent. Ils peuvent alors réaliser des **opérations** (créer, lire, modifier, ...) sur des **objets** (fichiers, données en mémoire, ...) [2].

Cependant, un certains nombre d'applications nécessitent de mettre en oeuvre des politiques qui dépendent soit du temps (dates), soit de contraintes temporelles sur les activités d'un sujet ou sur le flux même, la succession de ces activités (workflow).

Un **workflow** représente un processus métier. Il se décompose en un ensemble de **tâches**, elles-mêmes découpées en **opérations**, que nous considérons comme des entités élémentaires, insécables. L'ordre d'exécution des tâches est pré-défini. Les ensembles d'opérations nécessaires à la réalisation de chacune des tâches sont pré-définis.

L'expression de contraintes temporelles sur l'exécution de tâches a souvent été étudiée en liaison avec les méthodes d'attribution de permissions et de contrôle d'accès. L'état de l'art fournit un aperçu des principales méthodes tenant compte de l'aspect temporel dans la spécification et l'implémentation de politiques de contrôle d'accès. Nous nous sommes interrogés sur les éléments de la politique de sécurité du système qui peuvent être décrits à l'aide de chacune de ces méthodes et nous avons identifié les formalismes utilisés pour les décrire. Chaque formalisme autorise des vérifications spécifiques, que nous avons également étudiées. Par exemple, le modèle de Krukow, Nielsen et Sassone s'appuie sur des spécifications de politiques d'interactions, écrites en logique temporelle, vérifiables par model-checking.

[1] Dans l'article *Enforceable Security Policies*, détaillé dans la suite de ce document

[2] Vocabulaire utilisé dans le modèle normalisé publié par la NIST (National Institute of Standards and Technology)

L'Atelier Focal fournit un cadre pour la formalisation et la preuve de correction de politiques de contrôle d'accès. Si nous réussissons à construire un nouveau modèle permettant l'expression de contraintes d'ordonnancement, indépendant de tout modèle de contrôle d'accès, et si nous réussissons à le formaliser dans l'Atelier Focal, nous disposerons alors d'outils pour prouver la correction de notre « politique d'ordonnancement ». Cette « politique d'ordonnancement » regroupe, pour nous, les contraintes de délai et d'enchaînement entre opérations et événements.

1.2. **Organisation du document.**

Nous avons sélectionné 8 textes portant sur 5 modèles intégrant l'aspect temporel à l'attribution de rôles et de tâches à des utilisateurs.

[1]	An authorization model for workflows	Atluri et Huang	1996
[2]	The specification and enforcement of authorization constraints in Workflow Management Systems	Bertino, Ferrari et Atluri	1999
[3]	Access control based on execution history	Abadi et Fournet	2003
[4]	A logical framework for reputation systems and history-based access control	Krukow, Nielsen et Sassone	2007
[5]	Temporal hierarchies and inheritance semantics for GTRBAC	Joshi, Bertino et Ghafoor	2002
[6]	Dependencies and separation of duty constraints in GTRBAC	Joshi, Bertino, Shafiq et Ghafoor	2003
[7]	A GTRBAC-based system for dynamic workflow composition and management	Shafiq, Samuel et Ghafoor	2005
[8]	Enforceable security policies	Schneider	2000

TAB. 1. Liste des articles étudiés

2. Etude détaillée des articles sélectionnés

2.1. *Workflow Authorization Model* (WAM).

Objectifs	S'assurer que les tâches constituant un workflow sont exécutées par les sujets autorisés. S'assurer que les sujets autorisés n'ont accès aux objets manipulés par une tâche que pendant l'exécution de cette tâche.
Spécification	Synchroniser l'attribution des accès aux ressources avec la progression du workflow.
Proposition	Nouveau modèle : *Workflow Authorization Model* (WAM). Nouvelle construction : *Authorization Template* (AT). Au moins une AT est associée à chaque tâche du workflow pour synchroniser l'attribution des accès à la date de début et à la durée d'une tâche. Modèle d'implémentation à l'aide de réseaux de Pétri colorés et temporisés.

Dans le modèle WAM, un workflow est un ensemble partiellement ordonné de tâches. Une **tâche** est elle-même un ensemble ordonné d'opérations. On se donne un ensemble de **sujets**, un ensemble d'**objets**, un ensemble de **types** et une fonction qui, à tout objet, associe un type. Un **workflow** est alors un ensemble de tâches coordonnées, constituées d'opérations, au cours desquelles des sujets manipulent des objets.

Dans ce modèle, les objets sont déplacés d'une tâche à une autre. L'arrivée d'un objet au niveau d'une tâche déclenche l'exécution de cette tâche. A la fin de la tâche, l'objet est envoyé ailleurs.

On définit ensuite un ensemble de **dates**, avec lesquelles on peut aussi définir des **intervalles de temps**.

Une **tâche** est définie par un quadruplet

$$w_i = (OP_i, \Gamma_{IN_i}, \Gamma_{OUT_i}, [\tau_{l_i}, \tau_{u_i}])$$

où OP_i est l'ensemble des opérations qui constituent la tâche w_i, ces opérations manipulent des objets dont le type appartient à Γ_{IN_i} et renvoient des objets dont le type appartient à Γ_{OUT_i}. La tâche w_i doit obligatoirement s'exécuter entre τ_{l_i} et τ_{u_i}.

A chaque fois qu'une tâche s'exécute, une **instance** de cette tâche est créée. Une instance de tâche est définie par un quadruplet

$$w - inst_i = (OPER_i, IN_i, OUT_i, [\tau_{s_i}, \tau_{f_i}])$$

où $OPER_i$ est l'ensemble des opérations effectuées pour réaliser l'instance de la tâche $w - inst_i$, IN_i est l'ensemble des objets effectivement manipulés par l'instance, et dont le type appartient à Γ_{IN_i}, OUT_i est l'ensemble des objets effectivement renvoyés par l'instance, et dont le type appartient à Γ_{OUT_i}. Enfin $[\tau_{s_i}, \tau_{f_i}]$ est l'intervalle de temps durant lequel l'instance a été réalisée. $[\tau_{s_i}, \tau_{f_i}]$ doit être inclus dans $[\tau_{l_i}, \tau_{u_i}]$.

Une **authorization template** est un triplet

$$AT(w_i) = (s_i, (\gamma_i, -), pr_i).$$

Une AT permet de spécifier que seul l'utilisateur s_i est autorisé à exécuter la tâche w_i, que seuls des objets de type γ_i peuvent être manipulés par la tâche w_i et que le sujet s_i possède le privilège pr_i sur ces objets. $(\gamma_i, -)$ est instancié à l'arrivée d'un objet de type γ_i au niveau de la tâche w_i. Toute tâche possède au moins une AT.

Une **autorisation** est un quadruplet

$$A = (s, o, pr, [\tau_b, \tau_e])$$

qui permet de spécifier que le sujet s possède un droit d'accès à l'objet o, avec le privilège pr. Ce droit lui est octroyé à la date τ_b et lui est retiré à la date τ_e.

Illustrons sur un exemple le calcul des autorisations à partir des AT.

Soient w_i et w_j deux tâches d'un workflow.
w_i doit obligatoirement s'exécuter entre τ_{l_i} et τ_{u_i}.
w_i doit manipuler un objet o de type γ_i.
Les AT associées à w_i et w_j sont :
$AT(w_i) = (s_i, (\gamma_i, -), pr_i)$
$AT(w_j) = (s_j, (\gamma_j, -), pr_j)$
(γ_i peut être égal à γ_j).

L'objet o arrive au niveau de la tâche w_i au temps τ_{a_i}.
Si $\tau_{a_i} \in [\tau_{l_i}, \tau_{u_i}]$ alors la tâche commence à s'exécuter et $AT(w_i) = (s_i, (\gamma_i, \mathbf{o}), pr_i)$.

On veut construire une autorisation $A_i = (s_i, o, pr_i, [\tau_b, \tau_e])$, connaissant $AT(w_i)$.
Si $\tau_{a_i} \geq \tau_{l_i}$, on assigne τ_{a_i} à τ_b et τ_{u_i} à τ_e. $A_i = (s_i, o, pr_i, [\tau_{\mathbf{a_i}}, \tau_{\mathbf{u_i}}])$. Ainsi, l'autorisation n'est accordée qu'au début de la tâche et, comme l'impose la spécification initiale, l'autorisation ne sera plus valide après τ_{u_i}, même si la tâche continue de s'exécuter.
Si $\tau_{a_i} < \tau_{l_i}$, c'est-à-dire que l'objet arrive avant la date de début imposé pour la tâche, on assigne τ_{l_i} à τ_b. $A_i = (s_i, o, pr_i, [\tau_{\mathbf{l_i}}, \tau_{u_i}])$. La tâche est mise en attente. Le sujet s_i doit attendre la date de début imposé pour la tâche, pour pouvoir acquérir le privilège pr_i sur l'objet o.
Si $\tau_{a_i} > \tau_{u_i}$, c'est-à-dire que l'objet arrive après la date de fin imposée pour la tâche, alors l'objet est rejeté par la tâche. L'autorisation n'est pas construite.

La tâche w_i termine au temps τ_{f_i} et l'objet o est envoyé à la tâche w_j.
On a alors $AT(w_j) = (s_j, (\gamma_j, \mathbf{o}), pr_j)$ et $AT(w_i)$ redevient vide ($AT(w_i) = (s_i, (\gamma_i, -), pr_i)$).

Si $\tau_{f_i} < \tau_{u_i}$ alors la valeur de τ_e dans A est remplacée par τ_{f_i}.
Sinon, la valeur de τ_e n'est pas modifiée.

Ainsi, l'autorisation A_i n'est valide que pendant l'exécution de la tâche. Le sujet s_i ne possède le privilège pr_i sur l'objet o que pendant l'exécution de la tâche.

Les auteurs utilisent des **réseaux de Pétri colorés et temporisés** pour représenter le modèle WAM. Les couleurs du réseau de Pétri sont les types des objets et les types de privilèges. L'intervalle de temps $[\tau_{l_i}, \tau_{u_i}]$, imposé par la spécification pour l'exécution de la tâche w_i, est associé aux transitions d'entrée et de sortie de la place représentant w_i. A chaque place représentant une tâche est associée une durée représentant la durée effective d'exécution de la tâche.

Il existe deux types de places : les places rondes, représentant les tâches, et les places carrées, représentant les sujets.

Illustrons le fonctionnement de ces réseaux de Pétri sur l'exemple des deux tâches w_i et w_j précédemment détaillé (cf. représentation graphique Fig 1).

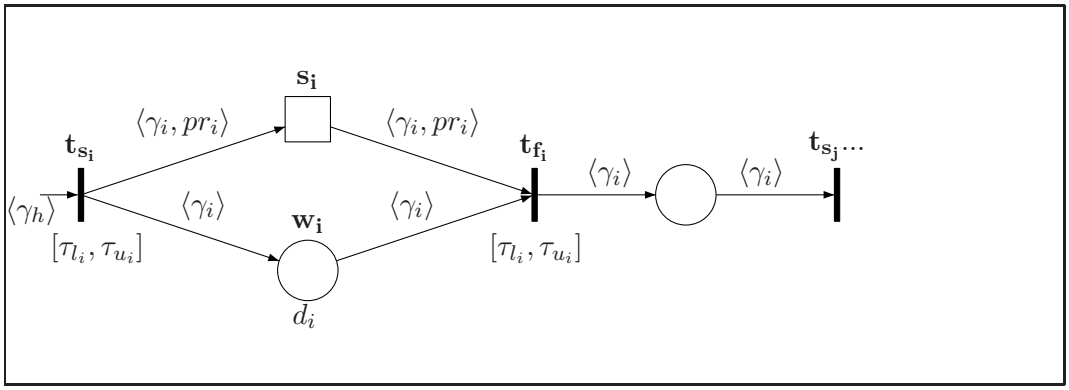

Fig. 1. Représentation d'une tâche dans le modèle WAM, à l'aide de Réseaux de Pétri colorés et temporisés

Pour pouvoir franchir une transition représentant le début d'une tâche, il faut qu'un token de la bonne couleur (une couleur est le type d'un objet) arrive au niveau de la transition, à un temps compris dans l'intervalle de temps associé à cette transition.

Un token coloré (par le type γ_i d'un objet manipulé par la tâche) est alors mis dans la place représentant la tâche et un token $\langle\ \gamma_i$, privilège accordé sur l'objet de ce type $\rangle$ est mis dans la place représentant le sujet autorisé à exécuter la tâche.

Au bout de la durée d_i d'exécution de la tâche, ces deux tokens sont envoyés vers la transition représentant la fin de la tâche.

Si la date de cette opération est comprise dans l'intervalle de temps associé à la transition de fin de tâche, alors un token coloré (par γ_i) est mis dans une place de sortie représentant un état particulier, où une tâche a fini et où une autre est sur le point de commencer.

Après le franchissement de la transition de fin de tâche, le sujet perd le privilège qu'il avait sur l'objet manipulé par la tâche.

Il y a donc une synchronisation entre l'exécution des tâches et l'attribution des autorisations aux sujets réalisant ces tâches.

Résoudre le problème de l'attribution des autorisations pour garantir la sûreté des données revient à résoudre un problème d'atteignabilité avec des réseaux de Pétri.

Pour finir, les auteurs expliquent brièvement comment intégrer la notion de rôle et comment spécifier des contraintes de séparation de fonctions dans le modèle WAM.

2.2. **Modèle d'Atluri et Bertino.**

Objectifs	Optimiser l'assignation de rôles et d'utilisateurs aux tâches d'un workflow en présence de contraintes de séparation de fonctions (*separation of duties*). Réduire la complexité et la durée de vérification pendant l'exécution du workflow, de la satisfaction de ces contraintes.
Spécification	Construire une base de contraintes et vérifier qu'une exécution donnée respecte bien cette base de contraintes.
Proposition	Langage permettant l'expression de contraintes logiques de séparation de fonctions. Typologie de ces contraintes (statiques, dynamiques ou « hybrides », c'est-à-dire vérifiables en partie statiquement et en partie dynamiquement). Une base de contraintes est associée à chaque workflow. Analyse de cohérence de cette base de contraintes et méthodologie (algorithmes) d'assignation des rôles et des utilisateurs aux tâches du workflow dans le respect de cette base de contraintes.

Dans ce modèle, un **workflow** est découpé en **tâches**, exécutées de manière séquentielle. Des **rôles** sont assignés à chacune des tâches. Des **utilisateurs** peuvent aussi être directement assignés aux tâches, s'ils possèdent le bon rôle.

Chaque tâche peut être activée plusieurs fois au cours d'un même workflow, par le même utilisateur ou par des utilisateurs différents. Si une tâche est activée plusieurs fois au même niveau du workflow par des utilisateurs différents, toutes les activations de cette tâche doivent être terminées avant le début de la tâche suivante.

De plus, les rôles sont hiérarchisés. Il existe un ordre partiel global entre les rôles (qui correspond généralement à la structure hiérarchique d'une organisation) et, pour les rôles qui ne sont pas en relation dans l'ordre global, il est possible de définir un ordre local. Une propriété exprime qu'un rôle « père » hérite des permissions assignées à tous ses rôles « fils ». Cependant, lors de l'assignation des rôles aux tâches, un rôle « fils » sera assigné en priorité. Un rôle « père » n'est assigné à une tâche que s'il n'y a plus aucun utilisateur ayant un rôle « fils » autorisé disponible.

L'assignation des rôles aux tâches est la définition d'une liste

$$[TRS_1, TRS_2, ..., TRS_n]$$

où chaque TRS_i (*Task Role Specification*) est un triplet $(T_i, (RS_i, >_i), act_i)$ avec T_i une tâche, RS_i l'ensemble des rôles autorisés à exécuter T_i, $>_i$ l'ordre local défini sur les rôles et act_i le nombre des activations possibles de T_i.

Les auteurs veulent exprimer des contraintes de séparation de fonctions entre utilisateurs et entre rôles.

Sur un exemple simple d'un workflow et d'une hiérarchie de rôles (représentés FIG. 2 et 3),

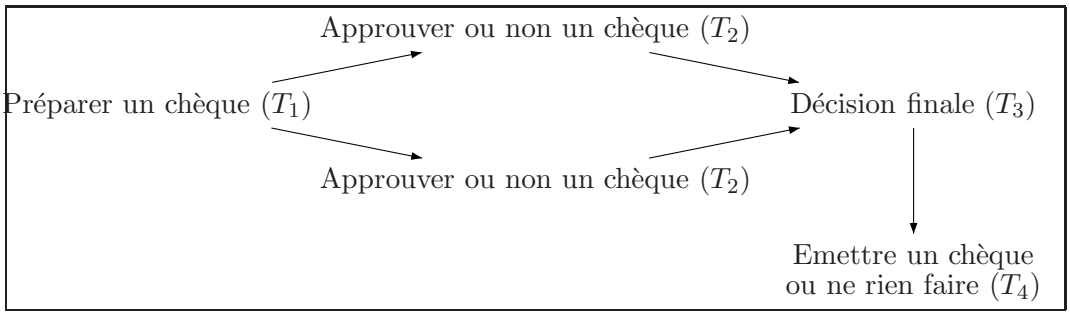

FIG. 2. Exemple : Workflow

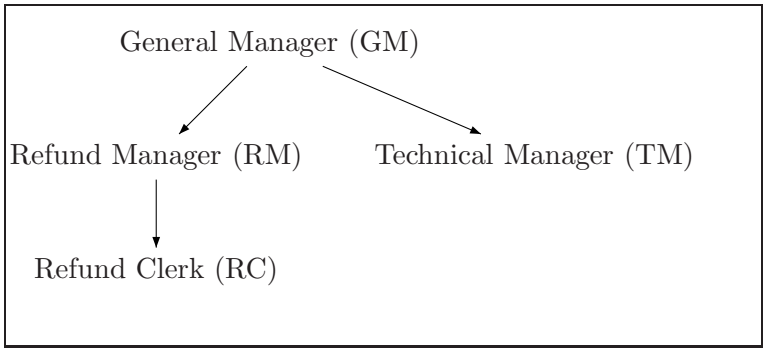

FIG. 3. Exemple : Hiérarchie des rôles

l'assignation des rôles aux tâches donne :

$$\begin{aligned} W = [\quad & (T_1, (\{RC\}, -), 1), \\ & (T_2, (\{RM, GM\}, -), 2), \\ & (T_3, (\{RM, GM\}, -), 1), \\ & (T_4, (\{RC\}, -), 1) \quad] \end{aligned}$$

Les contraintes de séparation de fonctions spécifiées sur ce cas d'application sont, par exemple :

C_1**:** Au moins 3 rôles doivent être appelés par le workflow.

C_2**:** T_2 doit être exécutée par un rôle hiérarchiquement supérieur aux rôles qui exécutent T_1 et T_4, sauf si T_2, T_1 et T_4 sont exécutés par le rôle GM.

C_3**:** Si un utilisateur a le rôle RC et a exécuté la tâche T_1 alors il ne peut pas exécuter T_4.

C_4**:** Si un utilisateur a exécuté T_2 alors il ne peut pas exécuter T_3.

C_5**:** Chaque activation de T_2 doit être exécutée par un utilisateur différent.

C_6**:** Si 4 activations de T_1 dans un même workflow par le même utilisateur échouent alors cet utilisateur ne peut plus exécuter T_1.

C_7**:** Si Bob exécute T_2 alors il ne peut pas exécuter T_4.

C_1 est une **contrainte statique** qui peut être vérifiée dès l'assignation des rôles aux tâches. C_6 est une contrainte **dynamique** qui ne peut être vérifiée qu'à l'exécution du workflow. C_2, C_3, C_4, C_5 et C_7 sont des contraintes **hybrides** qui peuvent être vérifiées en partie avant l'exécution du workflow. Par exemple, pour C_5, si un seul utilisateur est assigné à T_2 dans la spécification, C_5 ne pourra jamais être vérifiée pendant l'exécution du workflow ; si, au contraire, plusieurs utilisateurs sont assignés à T_2, il faut alors lancer l'exécution du workflow pour vérifier complètement que la contrainte est respectée.

Les auteurs définissent un langage permettant d'exprimer ces différents types de contraintes de séparation de fonctions sous la forme de clauses logiques. Plus précisément, ce langage permet l'expression de prédicats :

- d'assignation de rôles et utilisateurs à des tâches (prédicats `role` et `user`), d'assignation de rôles à des utilisateurs (`belong`),
- de définition de relations d'ordre global et local entre des rôles (`glb` (*greatest lower bound*), `lub` (*lowest upper bound*), $>$, $>_k$),
- de contrôle des utilisateurs et rôles qui exécutent une tâche (`execute`),
- de statut d'une activation (succès → prédicat `success` ou échec → prédicat `abort`),
- de contrôle des utilisateurs et rôles qui ne doivent pas être assignés à une tâche, ou, au contraire, qui doivent obligatoirement être assignés à une tâche (`cannot_do, must_execute`),
- de typage de contraintes (`statically_checked`),
- de statut du système (`panic`),
- de contrôle des résultats d'exécutions de workflows (`count, avg, min, max, sum`).

Ces prédicats sont utilisés pour exprimer des règles, qui correspondent aux contraintes de séparation de fonctions.

Ainsi, les contraintes de l'exemple développé précédemment se traduisent par les règles suivantes :

R_1: $\texttt{panic} \leftarrow \texttt{count}(\texttt{role}(R_i, T_j), n), n < 3$

$R_{2,1}$: $\texttt{cannot\_do}_r(R_i, T_2) \leftarrow \texttt{execute}_r(R_j, T_1, k), R_j \geq R_i, R_i \neq \texttt{GM}$

$R_{2,2}$: $\texttt{cannot\_do}_r(R_i, T_2) \leftarrow \texttt{execute}_r(R_j, T_1, k), R_j \not> R_i, R_i \not> R_j$

$R_{2,3}$: $\texttt{cannot\_do}_r(R_i, T_4) \leftarrow \texttt{execute}_r(R_j, T_2, k), R_i \geq R_j, R_i \neq \texttt{GM}$

$R_{2,4}$: $\texttt{cannot\_do}_r(R_i, T_4) \leftarrow \texttt{execute}_r(R_j, T_2, k), R_j \not> R_i, R_i \not> R_j$

R_3: $\texttt{cannot\_do}_u(U_i, T_4) \leftarrow \texttt{belong}(U_i, RC), \texttt{execute}_u(U_i, T_1, k)$

R_4: $\texttt{cannot\_do}_u(U_i, T_3) \leftarrow \texttt{execute}_u(U_i, T_2, k)$

R_5: $\texttt{cannot\_do}_u(U_i, T_2) \leftarrow \texttt{execute}_u(U_i, T_2, k)$

R_6: $\texttt{cannot\_do}_u(U_i, T_1) \leftarrow \texttt{count}((\texttt{abort}(T_1, k), \texttt{execute}_u(U_i, T_1, k)), n), n > 4$

R_7: $\texttt{cannot\_do}_u(Bob, T_4) \leftarrow \texttt{execute}_u(Bob, T_2, k)$

De plus, la hiérarchie des rôles et l'assignation des rôles aux tâches se traduisent aussi par un ensemble de règles (compte tenu de leur grand nombre, nous ne les détaillerons pas dans ce document).

La totalité de ces règles constitue une **base de contraintes** dont la cohérence est définie par un prédicat logique sur les ensembles $Denied\_Roles(T_i)$, $Obliged\_Roles(T_i)$, $Denied\_Users(T_i)$ et $Obliged\_Users(T_i)$, définis dans le même langage que les règles. Informellement, ce prédicat exprime qu'une base de contraintes est cohérente si :

- le prédicat `panic`, qui signale qu'une des contraintes du workflow n'est pas vérifiée, n'appartient pas au modèle considéré, construit à partir de la base de contraintes ;
- les faits appartenant au modèle considéré ne sont pas contradictoires :

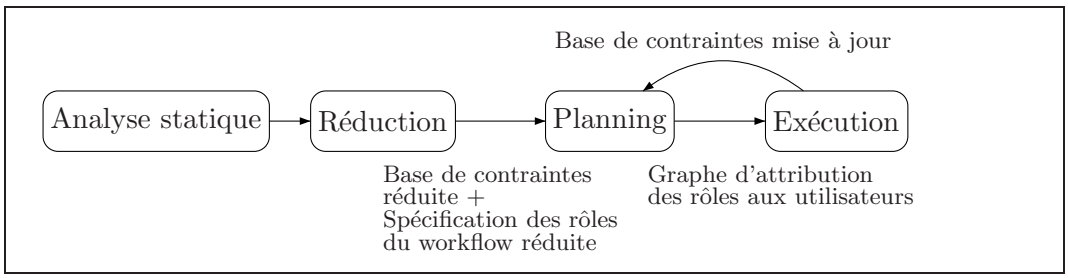

FIG. 4. Méthodologie d'attribution des rôles et des utilisateurs aux tâches

- pour toutes les tâches du workflow, les utilisateurs et les rôles qui doivent obligatoirement réaliser la tâche appartiennent à l'ensemble des utilisateurs et des rôles assignés à cette tâche ;
- l'intersection entre l'ensemble des utilisateurs et des rôles qui doivent obligatoirement réaliser la tâche et l'ensemble des utilisateurs et des rôles qui ne doivent pas être autorisés à réaliser la tâche est vide ;
- pour toute tâche sur laquelle porte une contrainte, il existe un utilisateur assigné à la tâche qui rend la contrainte vérifiable.

Les auteurs proposent ensuite une **méthodologie** pour attribuer les utilisateurs et les rôles aux tâches qui constituent le workflow. Cette méthodologie se décompose en quatre pas, chaque pas étant réalisé par un composant spécifique du système d'autorisation (cf. FIG 4).
Le premier pas est l'*analyse statique* des règles de la base de contraintes. Ce pas vérifie la cohérence de la base de contraintes et produit un modèle à partir des contraintes statiques de la base, il produit également les ensembles $Denied\_Roles(T_i)$, $Obliged\_Roles(T_i)$, $Denied\_Users(T_i)$ et $Obliged\_Users(T_i)$ pour toute tâche T_i du workflow.
Le deuxième pas est la réduction (*pruning phase*) des règles de la base de contraintes : les règles redondantes sont éliminées.
Le troisième pas est l'attribution des utilisateurs et des rôles aux tâches de manière à satisfaire toutes les contraintes (*planning phase*). Ce pas est réalisé avant toute exécution du workflow et produit un graphe d'attribution de tous les rôles possibles pour chaque tâche.
Le quatrième pas est l'exécution réelle du workflow (*runtime phase*). Ce pas est répété pour toute tâche du workflow et consiste en deux sous-phases : la sous-phase d'activation d'une tâche et la sous-phase de terminaison d'une tâche.

La sous-phase d'activation vérifie que l'utilisateur effectuant une requête possède un rôle autorisé pour la tâche demandée et que le nombre maximum d'activations de cette tâche n'est pas dépassé. Si ce nombre est dépassé, la phase de *planning* doit être relancée pour vérifier que l'activation courante de la tâche n'invalide pas le plan précédemment calculé.
Après cette nouvelle relance de la phase de *planning*, si l'utilisateur ayant lancé la requête courante fait toujours partie des utilisateurs autorisés, la tâche est effectivement activée.

Sinon, un autre utilisateur doit être sélectionné pour exécuter la tâche, et les vérifications précédemment décrites doivent être relancées.

La sous-phase de terminaison ajoute les règles de description de l'exécution courante à la base de contraintes (qui est donc l'équivalent d'un historique d'exécution).
Sur le graphe d'attribution des rôles aux tâches, les attributions futures rendues impossibles par l'activation courante (car elles entraîneraient une incohérence de la base de contraintes) sont supprimées. Au niveau de chaque noeud du graphe d'attribution des rôles, la liste des utilisateurs possédant le rôle et autorisés à exécuter la tâche associée, est ajoutée ou mise à jour.
Ce mécanisme permet de garantir le respect des contraintes dynamiques de séparation de fonctions.

Pour finir, les auteurs décrivent comment leur module, dont le fonctionnement a été précédemment décrit, peut constituer l'élément central d'un système de gestion de workflow, pour déterminer quel rôle ou quel utilisateur peut être assigné à quelle tâche.
Ainsi, ils décrivent une **architecture** centralisée, avec au centre un composant assurant l'ordonnancement et la synchronisation des tâches constituant un workflow et envoyant chaque tâche vers l'entité (utilisateur ou processus) qui pourra la traiter.
L'utilisateur entre sa spécification du workflow et sa base de contraintes par une interface graphique. Le workflow peut être spécifié comme un ensemble de tâches et un ensemble de dépendances entre ces tâches. Les dépendances entre tâches peuvent être exprimées sous la forme de tâches primitives (comme *begin (t)*, *abort (t)* ou *commit (t)* suivant le résultat d'une tâche ou selon des paramètres externes, comme le temps). Les contraintes sont exprimées dans le langage de spécification décrit dans l'article. Le système peut attribuer à l'avance les rôles aux tâches. Il peut attribuer partiellement les utilisateurs aux tâches et les attribuer définitivement au cours de l'exécution.

2.3. ***History-Based Access Control*** **(HBAC).**

Les deux textes présentés dans ce paragraphe présentent deux mécanismes de contrôle d'accès qui se basent sur des historiques d'opérations pour vérifier que le code exécuté et le comportement des utilisateurs du système respectent bien une politique de sécurité donnée.

Premier texte :

	Abadi et Fournet
Objectifs	S'assurer que les entités qui exécutent un code n'ont accès qu'aux ressources qu'elles sont autorisées à manipuler.
Spécification	Utiliser l'historique d'exécution du système pour déterminer en temps-réel les permissions que possède un code pour accéder à des ressources protégées.
Proposition	Mise à jour des permissions associées à un code pendant son exécution, en fonction du code qui a été précédemment exécuté, par analyse de l'historique d'exécution plutôt que par inspection des piles d'exécution. Conseils de codage exploitant les bibliothèques $C^\sharp$ existantes. Nouveaux éléments syntaxiques pour attribuer explicitement des permissions à du code en $C^\sharp$.

La sûreté du **typage** est une hypothèse essentielle mais non suffisante pour assurer la sécurité des environnements d'exécution.

Un exemple posant problème est celui du « *Confused Deputy* » : un code non fiable appelle un code fiable, ensuite ce même code fiable invoque des opérations sensibles. Ces opérations sensibles doivent être invoquées à un niveau « correct » de privilèges ; il faut tenir compte du fait que l'invocation de ces opérations est le résultat de l'action d'un code non fiable. De plus, on ne veut pas avoir à modifier le code fiable.

La technique d'**inspection des piles d'exécution** est une solution connue pour traiter ce problème. Elle consiste à associer statiquement, avant exécution, à tout morceau de code, la borne supérieure de l'ensemble de ses permissions. A l'exécution, les permissions associées à un morceau de code sont l'intersection de toutes les permissions statiques des morceaux de code présents dans la pile d'exécution. Avec cette technique, les permissions associées à une requête faite par un code fiable lorsque celui-ci a été appelé par un code non fiable, correspondent aux permissions statiques communes à ces deux codes.
Cette technique possède plusieurs défauts.
Dans le cas, par exemple, où A appelle B, attend la réponse de B, qui peut retourner un résultat non attendu ou quitter le système dans un état non attendu, puis appelle C. L'appel à C dépend alors du précédent appel à B, ce qui n'est plus visible dans les piles d'exécution. Les piles d'exécution ne fournissent qu'une vue partielle de l'exécution passée.
Le coût de l'inspection des piles d'exécution peut aussi pousser les programmeurs à ne pas utiliser les fonctionnalités de test des permissions.
Enfin, la manipulation de piles d'exécution est une technique très « bas niveau ». Il est difficile de coder et de contrôler cette opération avec des langages de haut-niveau.

Une alternative à l'inspection des piles d'exécution pour déterminer les permissions associées à un code consiste à **utiliser l'historique d'exécution**. Cet historique contient les informations des piles d'exécution mais aussi les traces des codes appelés et qui ont été supprimés des piles d'exécution après avoir terminé.

A chaque morceau de code sont associés des attributs : origine du code (disque local, code signé par un tiers de confiance, code téléchargé depuis un site non fiable, ...), et propriétés déterminées par une analyse de code automatique. Ces attributs sont utilisés pour calculer les permissions pendant l'exécution.
La démarche se décompose en sept étapes :

(1) **Déterminer les permissions statiques**, qui dépendent de l'origine et des propriétés du code. Les permissions statiques sont les permissions maximales pour ce code. Un code ne peut pas modifier ses permissions statiques mais peut les lire à tout moment. Les mécanismes de gestion des permissions sont standards et on peut facilement réutiliser l'existant ;
(2) **Déterminer les permissions courantes**. L'unité d'exécution est le thread ; lorsqu'un thread est lancé, il hérite, par défaut, des permissions de son thread parent ; lorsque deux threads se rejoignent, on calcule l'intersection de leurs ensembles de permissions ;
(3) **Vérification** : les permissions courantes sont examinées avant prises de décisions concernant la sécurité, appels de services et appels aux environnements d'exécution de machines distantes ; il n'est donc plus nécessaire de parcourir la pile d'exécution ;
(4) **Stockage** : les permissions courantes sont, en effet, conservées dans une variable globale, accessible à toutes les méthodes. Le flot de contrôle des activations de méthodes est indépendant des piles d'exécution, ce qui facilite les optimisations du compilateur et la compréhension et la modification du code par les programmeurs ;
(5) **Mises à jour automatiques des permissions** : l'intersection entre les permissions courantes et les permissions associées à tout code qui s'exécute est réalisée automatiquement ; (plusieurs optimisations permettant d'augmenter la vitesse des mises à jour, basées sur l'analyse statique du code, pour limiter le nombre de calculs d'intersections d'ensembles, sont proposées) ;
(6) En cas de **modifications explicites de permissions**, ces opérations doivent être réalisées par du code sécurisé et nécessitent un contrôle accru. Un code peut, au plus, acquérir ses permissions statiques ; une requête réclamant plus de permissions doit être refusée. Une politique de sécurité peut préciser les types de requêtes autorisées ;
(7) **Syntaxe** : les auteurs introduisent deux nouvelles constructions syntaxiques en $C^\sharp$ pour faciliter la gestion des permissions (*Grant(P)*$\{B\}$ permet d'ajouter l'ensemble des permissions P aux permissions courantes avant l'exécution du code B ; lorsque B termine, les permissions courantes sont réévaluées comme l'intersection des permissions courantes avant exécution de B et des permissions courantes après exécution de B. Et *Accept(P)*$\{B\}$ ajoute les permissions P aux permissions courantes si l'exécution du code B termine normalement, c'est-à-dire sans lever d'exception ; en cas de levée d'exception non traitée, les permissions courantes ne sont pas modifiées).

Lorsqu'un code termine, les permissions courantes sont plus restreintes, ou au mieux

égales, à la valeur qu'elles avaient avant l'exécution du code. Ce n'était pas le cas avec la technique d'inspection des piles d'exécution, où l'ensemble des permissions courantes était systématiquement restauré à sa valeur précédant l'exécution du code, ce qui pouvait agrandir cet ensemble. Pour un code donné, les permissions courantes déterminées par la technique de l'historique sont toujours incluses dans les permissions courantes déterminées par la technique des piles d'exécution.

L'historique enregistre les informations de contrôle d'appels de méthodes mais pas les dépendances entre les données.

Deuxième texte :

	Krukow, Nielsen et Sassone
Objectifs	Prendre une décision concernant la poursuite d'un protocole d'interaction en fonction du résultat de l'analyse des interactions passées.
Spécification	Analyser l'historique d'exécution du système pour vérifier que le comportement des utilisateurs respecte bien une politique d'interactions donnée.
Proposition	Le comportement des utilisateurs est traduit en termes d'événements. Nouveau cadre : *Event Structure* (ES) dans lequel sont définis des historiques d'interactions. Les politiques d'interactions sont traduites dans la logique temporelle linéaire *pure-past* (PPLTL). Vérification de la conformité des historiques d'interactions avec les politiques d'interactions, par « model-checking dynamique ».

Dans les systèmes distribués, les utilisateurs envoient des messages suivant des protocoles. On veut garder une trace des **comportements** des utilisateurs au cours de sessions de ces protocoles.
Les **algèbres de processus** peuvent décrire des protocoles d'interaction de manière très précise mais sont trop complexes pour les besoins des auteurs. Ceux-ci veulent simplement mémoriser, pour chaque événement, l'utilisateur qui l'a généré. Ils créent donc une nouvelle entité, appelée *event-structure*. Dans le cadre des *event-structures*, chaque utilisateur a accès à l'ensemble des sessions de protocoles auxquelles participent les autres utilisateurs.

Une **event-structure** est un triplet

$$ES = (E, \leq, \#)$$

où E est l'ensemble des événements, $\leq$ la relation de dépendance causale qui définit un ordre partiel sur E, $\#$ la relation de conflit, symétrique et non réflexive.
Les événements vérifient les deux axiomes suivants :

$$\{e' \in E \mid e' \leq e\} \text{ est fini } \text{ et } (e\#e' \text{ et } e' \leq e'') \Rightarrow e\#e''$$

Tous les sous-ensembles d'événements ne sont pas observables au cours d'une session d'un protocole. Les sous-ensembles observables sont appelés des **configurations** (notées $x \subseteq E$) et doivent respecter les propriétés d'absence de conflit (*conflict free*, C.F.) et de fermeture causale (*causally closed*, C.C.) :

$$\forall d, d' \in x \forall e \in E \ \ \text{(C.F.)} \ \neg(d \# d') \ \ \text{et (C.C.)} \ e \leq d \Rightarrow e \in x$$

Un **historique** est alors une séquence de configurations finies

$$h = x_1 x_2 ... x_n.$$

Les *event-structures* de Nielsen et Krukow manipulent donc des séquences d'ensembles d'événements. Le cadre proposé est une généralisation du modèle de contrôle d'accès à base d'historique, où les historiques sont des séquences d'événements uniques.

Le cadre des *event-structures* fournit les deux opérations suivantes : *new*, qui initie un nouveau protocole, et *update*, qui met à jour les ensembles d'événements de l'historique.

Le modèle permet de mémoriser les comportements des agents dans le but de guider des interactions futures : décider qui aura accès à quelle donnée et qui peut interagir avec qui. Ces décisions sont gouvernées par des politiques qui spécifient exactement les contraintes sur les historiques locaux d'interactions.
Les auteurs donnent la syntaxe et la sémantique d'un langage permettant de traduire formellement des **politiques de sécurité**.
Ils utilisent une variante de la logique temporelle linéaire qui permet d'exprimer des propriétés sur des comportements passés. (Ils n'ont pas besoin d'exprimer de propriétés sur le futur). Ils se restreignent à la **LTL *pure-past* (PPLTL)** car les algorithmes de model-checking de ce fragment de la LTL sont très efficaces, y compris sur l'aspect de vérification dynamique de politiques.

Ils définissent une relation de satisfaction, notée $\models$, entre des historiques et des politiques. Un **jugement**

$$h \models \psi$$

signifie que l'historique h satisfait les exigences de la politique ψ.

Dans une politique, on peut exprimer que l'événement e est observé dans une session (noté simplement e) ou qu'il pourra être observé dans le futur (noté $\diamond e$). On peut exprimer que deux politiques sont respectées en même temps ($\psi_0 \wedge \psi_1$), qu'une politique n'est pas respectée ($\neg\psi$), la dernière fois qu'une politique a été respectée (opérateur « *last time* », noté $\mathsf{X}^{-1}\ \psi$), une relation de précédence entre deux politiques (opérateur « *since* », noté $\psi_0\ \mathsf{S}\ \psi_1$). **Syntaxe** du langage, paramétré par les *event-structures* :

$$\psi ::= e \mid \ \diamond e \mid \psi_0 \wedge \psi_1 \mid \neg\psi \mid \mathsf{X}^{-1}\ \psi \mid \psi_0\ \mathsf{S}\ \psi_1$$

Pour la séquence de configurations finies $h = x_1 x_2 ... x_n$ et $i \in \mathbb{N}$, on définit la **sémantique** de $(h, i) \models \psi$ par induction structurelle :

$$\begin{array}{lcl}
(h,i) \models e & \Leftrightarrow & 1 \leq i \leq N \text{ et } e \in x_i \\
(h,i) \models \diamond e & \Leftrightarrow & 1 \leq i \leq N \Rightarrow \neg(e \# x_i) \\
(h,i) \models \psi_0 \wedge \psi_1 & \Leftrightarrow & (h,i) \models \psi_0 \text{ et } (h,i) \models \psi_1 \\
(h,i) \models \neg\psi & \Leftrightarrow & (h,i) \not\models \psi \\
(h,i) \models \mathsf{X}^{-1}\ \psi & \Leftrightarrow & i > 1 \text{ et } (h, i-1) \models \psi \\
(h,i) \models \psi_0\ \mathsf{S}\ \psi_1 & \Leftrightarrow & \exists j \leq i.[(h,j) \models \psi_1 \text{ et } \forall k.(j < k \leq i \Rightarrow (h,k) \models \psi_0)]
\end{array}$$

Le **model-checking** permet ensuite de vérifier qu'un historique respecte une certaine politique, mais il faut y ajouter l'aspect **dynamique** : le modèle h change suivant les observations comportementales des agents donc il ne suffit plus de mémoriser les réponses du système dans un cache.
La politique est connue à l'avance et fournie à l'algorithme de model-checking dès l'initialisation. On fournit aussi un historique de départ.
Le model-checker doit mettre à jour cet historique à chaque occurrence des opérations *new* et *update*, et pouvoir retourner à chaque instant la valeur du jugement $h \models \psi$.

Deux **implémentations** du model-checker sont possibles : l'une **à base de tableaux** et l'autre **à base d'automates**. Ces deux implémentations sont inspirées de l'algorithme de Havelund et Rosu [3] prenant en entrée une formule de PPLTL et retournant un algorithme de programmation efficace permettant de tester en temps linéaire que la formule entrée est satisfaite par une trace finie d'événements, également entrée en paramètres.

Cet algorithme s'appuie ici sur la définition inductive de $(h, m) \models \psi$. Vérifier $(h, m) \models \psi$ revient à vérifier $(h, m-1) \models \psi_j$ pour toute sous-formule ψ_j de ψ et $(h, m) \models \psi_i$ pour toute sous-formule ψ_i de ψ ($\psi_j \neq \psi$ et $\psi_i \neq \psi$). Par exemple : prouver $\psi_3 = \mathsf{X}^{-1}\psi_4 \wedge e$ revient à prouver $(h, m-1) \models \psi_4$ et $e \in h_m$.

Enfin, les auteurs présentent quelques extensions du langage PPLTL et implémentent un gestionnaire de sécurité en langage Java, capable de contrôler l'exécution d'un programme dans le respect d'une politique à base d'historique écrite dans le langage précédemment présenté.

[3] K. Havelund et G. Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference (TACAS'02)*, volume 2280 of *LNCS*, pages 342-356. Springer-Verlag, 2002

2.4. *Generalized Temporal Role-Based Access Control* (GTRBAC).

Le modèle GTRBAC est basé sur le modèle RBAC [4], auquel est ajoutée la possibilité de capturer en temps-réel des dépendances entre tâches réalisées par des utilisateurs travaillant en collaboration (c'est-à-dire que les utilisateurs peuvent travailler ensemble mais ne pas poursuivre les mêmes buts). Les trois textes étudiés détaillent trois spécificités du modèle GTRBAC :

- les effets sur le modèle de l'ajout de contraintes temporelles sur l'héritage de permissions entre rôles hiérarchisés ;
- l'expression de contraintes de dépendance et de séparation de fonctions ;
- la construction d'une architecture de création et de gestion dynamique de workflows.

Premier texte :

	Joshi, Bertino et Ghafoor
Objectifs	Décrire précisément l'héritage de permissions dans GTRBAC.
Spécification	-
Proposition	Distinction entre héritage d'assignations et héritage d'activations. Expression de contraintes plus strictes sur les rôles assignés ou activés à un moment donné, ou pour une période ou une durée données.

Dans la plupart des modèles RBAC, il existe une relation d'ordre entre les rôles. Lorsque deux rôles sont reliés, le rôle « senior » peut hériter des permissions assignées à ses rôles « juniors ». Cet héritage réduit le coût des opérations d'assignation de permissions aux rôles, puisqu'il suffit d'assigner des permissions à un rôle « junior » pour que le « senior » les possède aussi.

La possibilité d'exprimer des contraintes temporelles complique l'héritage des permissions. Il devient nécessaire de distinguer les actions d'autoriser un rôle (***role enabling***) : l'utilisateur *peut* acquérir la permission assignée à un rôle ; et d'activer un rôle (***role activation***) : l'utilisateur acquiert effectivement la permission assignée à un rôle, au cours d'une session. On peut ainsi mieux contrôler l'héritage de permissions : un rôle « senior » pourrait ne pas hériter d'une permission, pourtant assignée à l'un de ses rôles « juniors », si cette permission n'est pas encore activée.

Le modèle GTRBAC permet l'expression d'**événements** (un événement est, par exemple : « autoriser un rôle », « assigner une permission à un rôle », « désactiver temporairement une contrainte », « spécifier un événement déclencheur pour une action »...) et l'expression de **contraintes sur ces événements** (contraintes de durée, de périodicité, nombre maximum d'activations d'un rôle par différents utilisateurs...).

[4] Ce modèle a été décrit en 1.1

Pour cela, les auteurs utilisent des couples (X, E), où X est l'expression d'une période ou d'une durée et E est un événement, pour traduire des politiques de contrôle d'accès de systèmes d'information.
Ils se donnent une liste de **prédicats** :

Prédicat	Sémantique informelle
$enabled(r,t)$	le rôle r est autorisé au temps t
$u\_assigned(u,r,t)$	l'utilisateur u est assigné au rôle r au temps t
$p\_assigned(p,r,t)$	la permission p est assignée au rôle r au temps t
$can\_activate(u,r,t)$	l'utilisateur u peut activer le rôle r au temps t
$can\_acquire(u,p,t)$	l'utilisateur u peut acquérir la permission p au temps t
$can\_be\_acquired(p,r,t)$	la permission p peut être acquise par le rôle r au temps t
$active(u,r,s,t)$	le rôle r est actif dans la session s de l'utilisateur u au temps t
$acquires(u,p,s,t)$	l'utilisateur u acquiert la permission p dans la session s au temps t

Tab. 2. Liste de prédicats de GTRBAC

et des **axiomes** exprimant les relations entre ces prédicats :

1	$p\_assigned(p,r,t) \rightarrow can\_be\_acquired(p,r,t)$
2	$u\_assigned(u,r,t) \rightarrow can\_activate(u,r,t)$
3	$can\_activate(u,r,t) \wedge can\_be\_acquired(p,r,t) \rightarrow can\_acquire(u,p,t)$
4	$active(u,r,s,t) \wedge can\_be\_acquired(p,r,t) \rightarrow acquires(u,p,s,t)$

Tab. 3. Liste d'axiomes de GTRBAC

Un événement est associé à chaque prédicat. Par exemple, `enable` r est l'événement qui rend le prédicat $enabled(r,t)$ vrai. Mais la syntaxe formelle des événements n'est pas fournie dans ce texte.
Une contrainte est, par exemple : (*DayTime*, `enable` *DayDoctor*) qui exprime qu'il faut autoriser le rôle *DayDoctor* durant la journée, ou encore (`enable` *DayDoctor* $\rightarrow$ `enable` *DayNurse* `after` *10 min*) qui appartient à une catégorie particulière de contraintes, nommée ***triggers*** par les auteurs, et qui exprime que si le rôle *DayDoctor* est autorisé alors il faut autoriser le rôle *DayNurse* 10 minutes après. De même, la syntaxe formelle des contraintes n'est pas fournie dans ce texte.

Les auteurs distinguent ensuite **trois types d'héritage** : l'héritage d'assignations, l'héritage d'activation et un troisième type d'héritage permettant d'hériter à la fois d'assignations et d'activations. Une propriété de cohérence assure qu'une assignation ou une activation accordée par l'un des trois types d'héritage n'est pas invalidée par un autre de ces trois types d'héritage.
Les auteurs s'appuient sur ces trois types d'héritage pour restreindre plus strictement les

accès aux données : certains rôles qui auraient été autorisés à accéder à des données par héritage de permissions dans un modèle RBAC classique, pourront être interdits d'accès avec les trois types d'héritage précédemment décrits ; ou bien, il sera possible d'activer ou de désactiver un rôle « junior » à tout moment, tant que son rôle « senior » restera autorisé.
Nous renvoyons à la section 3.2 du texte pour un exemple détaillé illustrant, pour une même hiérarchie, toutes les combinaisons possibles de rôles pouvant être assignés ou activés, selon le type d'héritage choisi.

Les auteurs appliquent ensuite l'événement `enable/disable` directement sur les hiérarchies (pas seulement sur les rôles) pour pouvoir modifier dynamiquement la hiérarchie des rôles pour des durées données.
Par exemple, `enable` $r \rightarrow$ `enable`$(h = r \geq r_j)$ `after` *10 min* exprime que 10 minutes après l'autorisation de r, r devient un rôle « senior » de r_j. Une contrainte de durée sur une hiérarchie serait, par exemple, $(1h,$ `enable` $h)$ qui exprime que la hiérarchie h est appliquée pendant 1h.

Enfin, la possibilité de contrôler le nombre d'activations d'un rôle à l'aide de contraintes de cardinalité [5] permet implicitement de contrôler le nombre d'accès aux ressources.

Deuxième texte :

	Joshi, Bertino, Shafiq et Ghafoor
Objectifs	Exprimer des contraintes de cardinalité sur des accès dans le temps, des contraintes de dépendances dans les flux d'assignation et d'activation des rôles, et des contraintes de séparation de fonctions dans GTRBAC.
Spécification	-
Proposition	Extension de l'ensemble des prédicats de statut de GTRBAC et définition de nouvelles constructions pour exprimer les contraintes de cardinalité sur des accès dans le temps. Utilisation d'une notion de *trigger* étendu pour exprimer les dépendances entre des événements et des conditions de statut du système. Utilisation des prédicats de statut de GTRBAC pour exprimer les contraintes de séparation de fonctions.

La liste des **prédicats de statut** de GTRBAC est étendue :

[5] L'expression de contraintes de cardinalité dans GTRBAC est détaillée dans le texte suivant

Prédicat	Domaine d'évaluation	Sémantique
Soit P l'ensemble des permissions, R l'ensemble des rôles, U l'ensemble des utilisateurs, S l'ensemble des sessions, T l'ensemble des dates et $r \in R, p \in P, u \in U, s \in S, t \in T$		
$enabled(r,t)$	$R \times T$	r est autorisé au temps t
$u\_assigned(u,r,t)$	$U \times R \times T$	u est assigné à r à t
$p\_assigned(p,r,t)$	$P \times R \times T$	p est assignée à r à t
$can\_activate(u,r,t)$	$U \times R \times T$	u peut activer r à t
$can\_acquire(u,p,t)$	$U \times P \times T$	u peut acquérir p à t
$r\_can\_acquire(u,p,r,t)$	$U \times P \times R \times T$	u **peut acquérir** p **par** r **à** t
$can\_be\_acquired(p,r,t)$	$P \times R \times T$	p peut être acquise par r à t
$active(u,r,t)$	$U \times R \times T$	r **est actif dans la session de** u **à** t
$s\_active(u,r,s,t)$	$U \times R \times S \times T$	r est actif dans la session s de u à t
$acquires(u,p,t)$	$U \times P \times T$	u **acquiert** p **à** t
$r\_acquires(u,p,r,t)$	$U \times P \times R \times T$	u **acquiert** p **par** r **à** t
$s\_acquires(u,p,s,t)$	$U \times P \times S \times T$	u acquiert p dans la session s à t
$rs\_acquires(u,p,r,s,t)$	$U \times P \times R \times S \times T$	u **acquiert** p **par** r **dans la session** s **à** t

TAB. 4. Liste étendue des prédicats de GTRBAC

Les **relations sémantiques** entre les nouveaux prédicats de GTRBAC et les prédicats de base, présentés en 2.4, sont :

$$
\begin{array}{l}
can\_acquire(u,p,t) \leftrightarrow \exists r \in R, r\_can\_acquire(u,p,r,t) \\
active(u,r,t) \leftrightarrow \exists s \in S, active(u,r,s,t) \\
acquires(u,p,t) \leftrightarrow \exists r \in R, r\_acquires(u,p,r,t) \\
acquires(u,p,t) \leftrightarrow \exists s \in S, s\_acquires(u,p,s,t) \\
acquires(u,p,r,t) \leftrightarrow \exists s \in S, rs\_acquires(u,p,r,s,t) \\
acquires(u,p,s,t) \leftrightarrow \exists r \in R, rs\_acquires(u,p,r,s,t)
\end{array}
$$

TAB. 5. Relations sémantiques entre prédicats de GTRBAC

Les **axiomes** 1, 2 et 3 du modèle GTRBAC de base (présentés TAB. 3), exprimant les relations entre les prédicats, sont conservés. Le prédicat 4 devient, avec les nouvelles syntaxes introduites :

$$active(u,r,t) \wedge can\_be\_acquired(p,r,t) \rightarrow acquires(u,p,t).$$

Les auteurs définissent une fonction d'évaluation des prédicats de GTRBAC, notée `eval`, et un opérateur de projection sur les résultats de l'évaluation d'un prédicat, noté Π_i. Par exemple, `eval`(enabled(r, t)) renvoie l'ensemble des couples de $R \times T$ dont les rôles sont autorisés au temps t. Et Π_1`eval`$(enabled(r,t))$ renvoie uniquement les rôles autorisés au temps t, alors que Π_2`eval`$(enabled(r,t))$ renvoie tous les instants où le rôle r est autorisé.

Ces deux nouvelles constructions permettent d'exprimer des **contraintes de cardinalité**.
Par exemple, $\mid \Pi_1 \texttt{eval}(\mathit{enabled}(r,t)) \mid \geq n$ exprime que le nombre de rôles autorisés au temps t ne doit pas être inférieur à n, ou encore $\mid \Pi_1 \texttt{eval}(\neg \mathit{enabled}(r,t)) \mid \leq n$ exprime que le nombre de rôles non autorisés au temps t ne doit pas être supérieur à n.

Une notion de *trigger* étendu permet d'exprimer des **contraintes de dépendance** entre exécutions de tâches d'un workflow (*Control Flow Dependency*).
Les auteurs donnent les formes génériques des contraintes dites :

- de « pré-condition » (exemple : un rôle *Junior_Employee* ne peut être assigné *que si* un utilisateur a auparavant activé un rôle *Manager*),
- de « post-condition » (exemple : si le rôle *SysAdmin* est autorisé alors *il faut que* le rôle *SysAudit* soit également autorisé),
- et de « précédence » (exemple : si les deux rôles *SysAdmin* et *SysAudit* sont activés alors *SysAdmin* doit être activé avant *SysAudit*).

Ces formes génériques utilisent des *triggers* étendus.
Nous renvoyons à la section 4.2 du texte pour les définitions formelles du *trigger* étendu et des contraintes de pré-condition, post-condition et précédence.

Enfin, les auteurs expriment des **contraintes de séparation de fonctions** dans le temps :

- sur l'autorisation de rôles, en utilisant les prédicats *enabled/disabled* de GTRBAC. Par exemple, une contrainte de la forme
$$(I, P, EN, R) \wedge \mathit{enabled}(r_1, t) \rightarrow \neg \mathit{enabled}(r_2, t)$$
où (I, P) est un intervalle de temps, EN est un événement visant à autoriser plusieurs rôles et r_1 et r_2 sont deux rôles différents de R, exprime que deux rôles ne peuvent pas être autorisés au même moment pendant l'intervalle de temps (I, P),
- sur l'assignation des rôles et des permissions (prédicats *u_assigned* et *p_assigned*). Par exemple, une contrainte de la forme
$$(I, P, UAS_1, U, R) \wedge u\_\mathit{assigned}(u, r_1, t) \rightarrow \neg u\_\mathit{assigned}(u, r_2, t)$$
où (I, P) est un intervalle de temps, UAS_1 est un événement visant à assigner des rôles à un utilisateur et r_1 et r_2 sont deux rôles différents de R, exprime que deux rôles ne peuvent pas être assignés à un même utilisateur au même moment pendant l'intervalle de temps (I, P),
- sur l'activation des rôles (prédicat *active*),
- sur des possibilités d'activer des rôles (prédicats *can_activate, can_be_acquired*),
- et sur des possibilités d'acquisition de permissions (prédicats *can_acquire, can_be_acquired*).

Troisième texte :

	Shafiq, Samuel et Ghafoor
Objectifs	Proposer une architecture logicielle dans laquelle puisse être mis en oeuvre le modèle GTRBAC qui permet de construire dynamiquement et de gérer des workflows.
Spécification	-
Proposition	Spécification du workflow et des contraintes sur les tâches de ce workflow dans la syntaxe de GTRBAC. Description d'une architecture logicielle permettant de construire des workflows respectant cette spécification.

Le workflow et les permissions sont pré-définies. Mais, au moment de l'exécution du workflow, il est possible qu'un utilisateur ayant le rôle pré-assigné à une tâche ne soit pas disponible, qu'un changement dans l'environnement d'exécution ou qu'un événement imprévu vienne bloquer l'exécution du workflow (deadlock). Il faut alors trouver une **re-configuration** optimale qui permette de débloquer l'exécution ou d'arrêter le système en toute sécurité.

L'architecture logicielle suivante est proposée pour gérer la création et les éventuelles re-configurations de workflows :

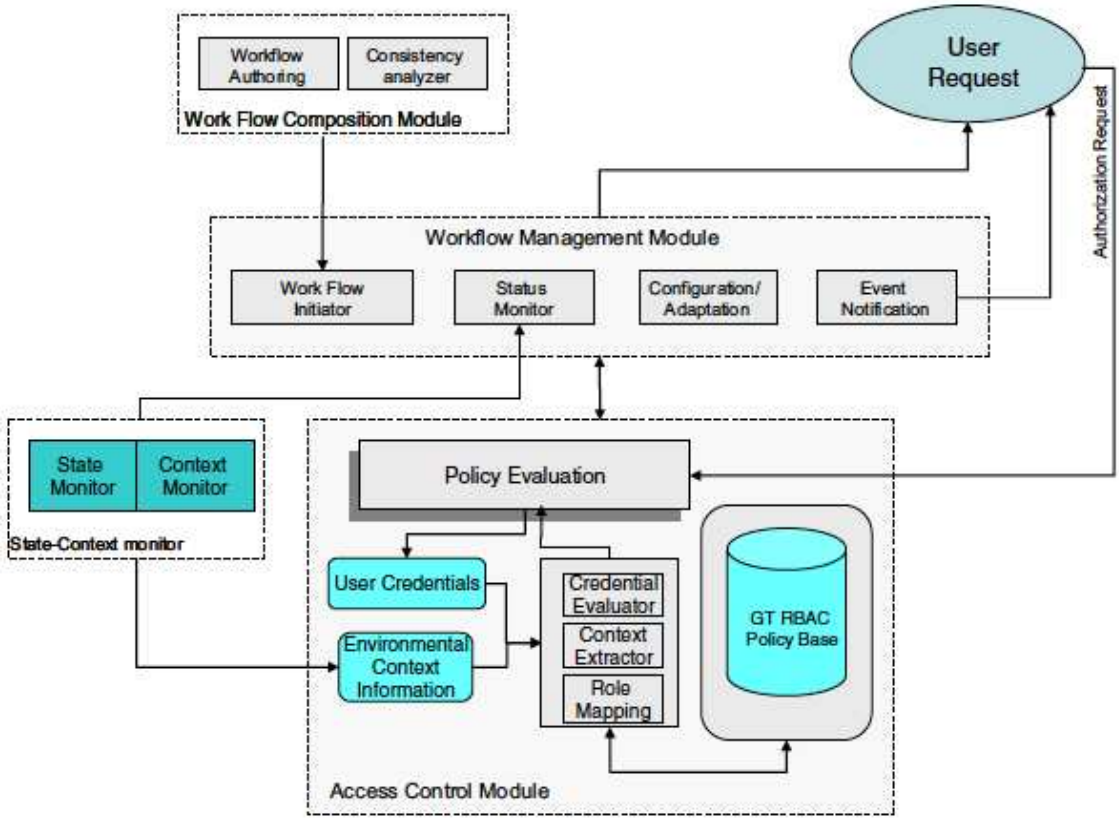

FIG. 5. Architecture logicielle pour la création et la gestion dynamique de workflows

Le ***Work Flow Composition Module*** sert à spécifier les tâches du workflow et les dépendances entre ces tâches. Il se décompose en un sous-module de création où l'utilisateur spécifie les contraintes dynamiques, les contraintes de séparation de fonctions et les contraintes sur le nombre d'exécution d'une tâche dans le formalisme de GTRBAC. Et en un sous-module d'analyse de cohérence qui vérifie la cohérence des contraintes, la correction du workflow, les dépendances entre tâches et le respect des deadlines et qui détecte les conflits dans la spécification.

Le ***Workflow Management Module*** est responsable de l'instanciation du workflow, il signale à l'utilisateur les événements système, il adapte le workflow et les invocations de ressources en fonction du contexte, de la disponibilité des données et des services. Il calcule la re-configuration optimale pour une instance du workflow : l'instance doit être la plus fidèle possible à la spécification des tâches, elle doit terminer (une exécution partielle peut être envisagée pour sortir d'une situation de blocage) et un maximum de contraintes doivent être respectées.

L'***Access Control Module*** détermine les autorisations accordées aux utilisateurs pour

l'exécution des tâches du workflow. Il contient la base des utilisateurs et des rôles spécifiés par la politique de sécurité. Il réalise l'assignation des utilisateurs aux rôles et des tâches aux rôles en se basant sur les caractéristiques de l'utilisateur et du contexte.

Les moniteurs (***State-Context monitor***) enregistrent les changements de contexte (temps, conditions ambiantes...), les changements dans l'environnement de l'utilisateur (localisation de l'utilisateur, niveau de qualification de l'utilisateur...) et les événements imprévus.

2.5. **Automates de sécurité.**

Objectifs	Vérifier qu'une politique de sécurité est bien appliquée.
Spécification	Vérifier une politique de sécurité en contrôlant le flux d'exécution du système et terminer l'exécution lorsque le système est sur le point de violer la politique de sécurité.
Proposition	Caractérisation du type de politiques de sécurité dont l'application peut être contrôlée par analyse du flux d'exécution (par un mécanisme d'*Execution Monitoring*). Ce type de politiques peut être représenté par des automates de sécurité, qui sont des automates de Büchi que l'on fait tourner en parallèle avec le programme pour vérifier que chaque étape de son exécution respecte la politique de sécurité.

Une **exécution** est une séquence finie ou infinie. L'auteur ne se préoccupe pas de la façon dont est représentée une exécution : par des actions atomiques, des pas du système à un plus haut niveau, des états du programme ou des paires « état-action »...

Un **pas d'exécution** est, par exemple, une action grain-fin (comme un accès mémoire), une opération de plus haut niveau (comme un appel de méthode) ou une opération qui modifie la configuration de sécurité et, par conséquent, restreint l'exécution future.

Un **mécanisme d'EM** (*Execution Monitoring*) est, par exemple, un noyau, un moniteur de référence, un firewall. Un compilateur ou un *theorem-prover* ne sont pas des mécanismes d'EM car ils utilisent plus d'informations que la simple observation des pas d'exécution du système, pour calculer toutes les exécutions possibles du système. Des mécanismes qui modifient le système avant de l'exécuter n'appartiennent pas non plus aux mécanismes d'EM.

L'auteur introduit une nouvelle notion de « cible ». Une **cible** est, par exemple, un objet, un module, un processus, un sous-système ou un système entier.

La caractérisation des politiques de sécurité pouvant être traitées par les mécanismes d'EM doit être suffisamment large pour inclure toutes les politiques de sécurité existantes, tout en étant indépendante de la définition des mécanismes d'EM.

Une **politique de sécurité** est un prédicat sur des ensembles d'exécutions. Une cible satisfait une politique de sécurité si et seulement si toutes les exécutions de la cible respectent la politique de sécurité.

Les **automates de sécurité** (*security automata*) sont des automates de Büchi qui acceptent des propriétés de sûreté. Ce sont des automates à états finis non-déterministes, acceptant des séquences finies et infinies.
L'automate entier est connu initialement. Les états de l'automate peuvent être étiquetés ; les transitions sont étiquetées par des fonctions totales de type $I \rightarrow bool$. I est l'ensemble des symboles d'entrée de l'automate ; ces éléments peuvent correspondre à des états du système, des actions atomiques, des actions de haut-niveau ou des paires « état-action ».
Ces automates sont difficiles à représenter dès que le nombre d'états devient important ou que la fonction de transition est complexe. On peut alors encoder les états par des variables multiples et utiliser des commandes gardées (*guarded commands*, de la forme $B \rightarrow S$ où B est la garde et c'est un prédicat sur les symboles d'entrée courants et les variables encodant l'état courant de l'automate, S est la commande et c'est une mise à jour des variables de l'état courant de l'automate) pour décrire la fonction de transition.

La cible est exécutée en tandem avec une simulation de l'automate de sécurité. La création ou l'initialisation d'une cible se traduit par la création et l'initialisation d'une instance de simulation de l'automate de sécurité. La cible et l'automate de sécurité partagent les mêmes symboles d'entrée. Si un symbole d'entrée est accepté par l'automate de sécurité, alors la cible est autorisée à réaliser la transition étiquetée par ce symbole d'entrée. L'automate de sécurité passe alors dans l'état suivant, calculé par sa fonction de transition. Si un symbole d'entrée n'est pas accepté par l'automate de sécurité, alors la cible est arrêtée pour tentative de violation de la politique de sécurité.

Ce modèle à base d'automates de sécurité fait plusieurs hypothèses implicites :
- Bien que la simulation de l'automate de sécurité génère des ensembles d'états de plus en plus gros pour pouvoir mémoriser les informations concernant les exécutions passées, l'expérience sur des systèmes réels montre que la quantité de mémoire requise reste toujours raisonnable.
- Le mécanisme d'EM est toujours capable d'arrêter la cible, si un symbole d'entrée est refusé par l'automate de sécurité.
- Le mécanisme d'EM est intègre. Les entrées du mécanisme doivent correspondre à l'exécution de la cible et les sorties du mécanisme doivent être prises en compte dans l'exécution de la cible (le mécanisme est correct et complet). Mécanisme d'EM et cible doivent être bien isolés l'un de l'autre, les variables d'état et les transitions de l'automate de sécurité ne doivent pas être modifiables par la cible.

3. Conclusions de l'état de l'art

3.1. **Notre tentative de modélisation.**

Nous avons cherché à modéliser la politique d'ordonnancement des tâches réalisées par un composant de chiffrement réel.

Pour ce type de composants, il existe des **normes de sécurité** qui spécifient les caractéristiques du composant et les propriétés qu'il doit respecter pour garantir un fonctionnement sûr. La norme FIPS 140-3 en est un exemple.

Nous nous sommes basés sur les recommandations de cette norme pour construire un modèle reposant sur la logique du premier ordre. Car notre objectif initial était de **composer** ce modèle avec d'autres modèles de politiques de sécurité, formalisées, elles aussi, en logique du premier ordre, dans l'Atelier Focal.

Nous avons décomposé la spécification du fonctionnement du composant cryptographique en **trois politiques indépendantes** :
- une politique de contrôle d'accès « classique », basée sur des rôles,
- une politique sur les interfaces du composant cryptographique,
- et une politique d'ordonnancement.

La politique de contrôle d'accès permet de vérifier que seuls les rôles autorisés sont à l'origine de l'exécution d'une tâche. La politique sur les interfaces contrôle l'activation des points d'entrée et de sortie du composant. Et la politique d'ordonnancement spécifie, pour chaque tâche, les passés et/ou les futurs obligatoires et/ou interdits, selon le résultat de l'exécution de la tâche considérée (succès, échec ou refus). Nous voudrions associer à chacune de ces politiques un module logiciel bien délimité dans l'architecture globale du composant cryptographique.

Pour modéliser la **politique d'ordonnancement des tâches** du composant, nous avons adopté une approche à base de propriétés, qui consiste à caractériser tous les états sûrs du système. Chaque état du système est caractérisé, dans notre modèle, par la suite des événements observés dans son passé. Un événement étant soit la réussite, soit l'échec, soit le refus d'une action requise par un utilisateur. Ces événements doivent donc vérifier une propriété, que nous avons nommée « prédicat de sécurité ».

Une action requise par un utilisateur entraîne l'exécution de tâches par le composant cryptographique. Toute exécution d'une tâche peut supposer que certains événements ont nécessairement eu lieu dans le passé, que d'autres, au contraire, n'ont jamais eu lieu dans le passé proche, et toute exécution d'une tâche peut imposer que certains événements doivent survenir dans le futur et que d'autres soient interdits dans un futur proche. Le prédicat de sécurité sert à caractériser ces ensembles d'événements pré-requis et post-requis à l'exécution d'une tâche.

Cependant, une particularité de notre système est difficile à formuler dans ce prédicat de logique du premier ordre. La survenue d'un événement imprévu peut modifier voire

annuler les futurs imposés par les événements précédents. Il faut donc trouver une structure de données ou un formalisme capable de retrouver rétroactivement dans le passé les actions dont le futur imposé est incomplet au moment de la survenue de l'événement imprévu, et capable de remplacer ces futurs imposés par le futur dicté par l'événement imprévu (les contre-mesures).

De plus, pour pouvoir prouver que notre système est sûr, nous devrons exprimer, par exemple, des propriétés de sûreté (« une tâche ou un événement dangereux pour la sûreté du système ne doit jamais survenir »), des propriétés de vivacité (« une tâche ou un événement finit toujours par survenir »), des propriétés d'atteignabilité (« un état du système peut être atteint ») et des propriétés d'arbitrage (« si deux événements peuvent survenir au même moment, le système doit décider lequel est exécuté en priorité »). Ces propriétés nous semblent plus facilement formalisables à l'aide d'opérateurs de logique temporelle, qui peuvent être ajoutés à l'Atelier Focal.

Deux familles de logique temporelle permettent classiquement de formaliser nos besoins : la logique temporelle linéaire (*Linear Temporal Logic*, LTL), permettant d'exprimer des propriétés portant sur des chemins individuels (issus de l'état initial) du programme, et la logique à embranchements (*Computation Tree Logic*, CTL), permettant d'exprimer des propriétés portant sur les arbres d'exécution (issus de l'état initial) du programme. Nous pourrions enrichir les bibliothèques de l'Atelier Focal pour permettre la spécification de ces propriétés de logique temporelle. Nous disposerions alors de deux types d'outils pour vérifier ces propriétés : des outils de preuve automatisée, qui sont proposés par l'Atelier Focal, et des outils de model-checking fondés sur des automates et des systèmes de transitions entre états, qui sont classiquement associés à des spécifications écrites en logique temporelle.