Approches formelles de mise en oeuvre de politiques de contrôle d'accès pour des applications basées sur une architecture orientée services

Michel Embe Jiague

MISE EN ŒUVRE DE POLITIQUES DE CONTRÔLE D'ACCÈS FORMELLES POUR DES APPLICATIONS BASÉES SUR UNE ARCHITECTURE ORIENTÉE SERVICES

par

Michel Embe Jiague

Thèse en cotutelle présentée

au Département d'informatique en vue
de l'obtention du grade de Philosophiæ doctor (Ph.D.)
FACULTÉ DES SCIENCES, UNIVERSITÉ DE SHERBROOKE

à l'École Doctorale MSTIC en vue
de l'obtention du grade de Docteur
UNIVERSITÉ PARIS-EST

19 décembre 2012

<dédicaces>

<grand-mère-bien-aimée>Mami Oli</grand-mère-bien-aimée>

<fils-adoré>Natou</fils-adoré>

</dédicaces>

Sommaire

La sécurité des systèmes d'information devient un enjeu préoccupant pour les organisations tant publiques que privées, car de tels systèmes sont pour la plupart universellement accessibles à partir de navigateurs Web. Parmi tous les aspects liés à la sécurité des systèmes d'information, c'est celui de la sécurité fonctionnelle qui est étudié dans cette thèse sous l'angle de la mise en œuvre de politiques de contrôle d'accès dans une architecture orientée services. L'élément de base de la solution proposée est un modèle générique qui introduit les concepts essentiels pour la conception de gestionnaires d'exécution de politiques de contrôle d'accès et qui établit une séparation nette entre le système d'information et les mécanismes de contrôle d'accès. L'instanciation de ce modèle conduit à un cadre d'applications qui comporte, entre autres, un filtre de contrôle d'accès dynamique. Cette thèse présente également deux méthodes systématiques d'implémentation de ce filtre à partir de politiques écrites en ASTD, une notation graphique formelle basée sur les statecharts augmentés d'opérateurs d'une algèbre de processus. La notation ASTD est plus expressive que la norme RBAC et ses extensions. La première méthode repose sur une transformation de politiques de contrôle d'accès, instanciées à partir de patrons de base exprimés en ASTD, en des processus BPEL. La deuxième méthode est basée sur une interprétation de spécifications ASTD par des processus BPEL. Dans les deux cas, les processus BPEL s'exécutent dans un moteur d'exécution BPEL et interagissent avec le système d'information. Ces deux méthodes permettent une implémentation automatique d'un cadre d'applications à partir de la spécification de départ. Finalement, un prototype a été réalisé pour chacune des deux méthodes afin de montrer leur faisabilité au niveau fonctionnel et de comparer leurs performances au niveau système.

Mots-clés: système d'information, sécurité fonctionnelle, contrôle d'accès, architecture orientée services, méthode de spécification formelle, service Web, transformation de spécifications, interprétation de spécifications.

Remerciements

Ce travail est le résultat d'un long parcours. Je veux profiter de ces quelques mots pour exprimer ma gratitude à toutes les personnes exceptionnelles qui ont contribué de près ou de loin à son aboutissement. Je pense bien sûr tout d'abord à des enseignants exceptionnels. Richard St-Denis et Marc Frappier de l'Université de Sherbrooke, Régine Laleau et Frédéric Gervais de l'Université Paris-Est, merci ! Merci pour vos conseils éclairés. En effet, je ne saurais dénombrer le nombre de fois où vos indications m'ont permis de trouver des solutions à des problèmes rencontrés. Merci pour votre implication et votre soutien financier, car sans ceux-ci ce travail de recherche ne serait pas achevé. Enfin, merci pour votre patience et l'opportunité que vous m'avez offertes. Je tiens aussi à exprimer ma gratitude à mes collègues de doctorat, Jérémy et Pierre, dont la coopération a été un élément indispensable à cette thèse.

Je tiens ensuite à remercier particulièrement ma famille. Papa, maman, mes frères et sœurs, Gwladys l'élue de mon cœur, m'ont accompagné pendant mes études doctorales par leurs encouragements incessants, leurs vœux bienveillants et leurs prières. Je remercie également mes amis, d'ici et d'ailleurs, pour le soutien moral et la patience qu'ils ont manifestés à mon égard. Enfin, à ceux qui ne sont pas mentionnés mais qui m'ont soutenu durant cette thèse, merci.

Abréviations

ACIT	Availability, Confidentiality, Integrity and Traceability
ACL	Access Control List
ASM	Abstract-State Machine
ASTD	Algebraic State Transition Diagram
BPEL	Business Process Execution Language
CSP	Communicating Sequential Processes
EB^3	Entity-Based Black-Box
EB^{3}SEC	EB^{3}SECure
ESB	Enterprise Service Bus
DAC	Discretionary Access Control
DOM	Document Object Model
GTRBAC	Generalized Temporal Role-Based Access Control
GUID	Globally Unique IDentifier
HTTP	HyperText Transfer Protocol
LOTOS	Language Of Temporal Ordering Specification
MAC	Mandatory Access Control
MDE	Model-Driven Engineering
NIST	National Institute of Standards and Technology
OASIS	Organization for the Advancement of Structured Information Standards
ODE	Orchestration Director Engine

PAP	Policy Administration Point
PDP	Policy Decision Point
PEM	Policy Enforcement Manager
PEP	Policy Enforcement Point
PIP	Policy Information Point
POSIX	Portable Operating System Interface
RBAC	Role-Based Access Control
SELKIS	SEcure heaLth care networKs Information Systems
SGBD	système de gestion de base de données
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SoD	Separation of Duty
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
W-RBAC	Workflow Role-Based Access Control
W3C	World Wide Web Consortium
WfMS	Workflow Management System
WSDL	Web Service Definition Language
X-GTRBAC	XML Generalized Temporal Role-Based Access Control
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language
XPath	XML Path Language
XSD	XML Schema Document
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation

Table des matières

Table des matières

Liste des figures

Liste des tableaux

Liste des programmes

Introduction

La décennie des années 1980 ont vu un accroissement significatif du nombre de systèmes d'information [51]. Ceux-ci offrent une solution à la prolifération des données d'entreprise. Ils permettent en effet d'*acquérir* des données au moyen d'une interface utilisateur sur le bureau ou à travers le réseau, ou encore au moyen d'une interface de type service. Ces données sont *stockées* dans des bases de données ou des dépôts de données. Les systèmes d'information peuvent ensuite *traiter* ces données suivant les besoins de leurs utilisateurs et permettre de *visualiser* des données de natures diverses. Un autre type de systèmes, très orienté vers les utilisateurs, a pris de l'expansion ces dernières années. Il s'agit des réseaux sociaux, qui tout comme les systèmes d'information, collectent un volume important de données soumises à de fortes contraintes de respect de la vie privée. En effet, leur principale fonction est le partage d'informations — de nature privée très souvent — entre des groupes d'utilisateurs formés autour d'un intérêt ou lien commun. Cet intérêt ou lien peut être la famille, l'amitié, un sport, une institution d'enseignement ou le même lieu d'habitation.

Contexte

De nos jours les utilisateurs des systèmes d'information y accèdent de plus en plus de n'importe où. Dans les entreprises, les utilisateurs accèdent aux systèmes par des terminaux qui peuvent être des clients matériels légers ou encore des clients logiciels (applications Web ou Java par exemple). Il existe aussi des clients embarqués dans des terminaux, tels que les téléphones intelligents, qui sont continuellement connectés ou qui utilisent un mode différé de synchronisation aux systèmes centraux. Il se pose donc de nombreux problèmes de sécurité des systèmes et de leurs données. Ces

problèmes sont d'autant plus nombreux que les moyens d'accès aux systèmes d'information le sont, car ceux-ci constituent les vecteurs d'attaque les plus courants. Les ingénieurs doivent donc protéger ces systèmes et leurs données contre les attaques physiques au niveau du matériel et les attaques au niveau du logiciel. La redondance de certains équipements réseau permet dans certains cas de contrecarrer des attaques de type déni de service. Dans [35] par exemple, elle est utilisée pour offrir une certaine protection des données. Cette même technique peut être utilisée pour lutter contre les pannes matérielles des systèmes. Les attaques de type « Man-in-the-Middle » peuvent être bloquées par des techniques qui utilisent de manière appropriée un protocole de communication adéquat et la cryptographie. Schneier [46] passe en revue de nombreux protocoles et algorithmes de cryptographie. Il explore aussi leur utilisation dans le but d'obtenir la confidentialité des données. Dans les systèmes d'information, les mesures de sécurité mises en place doivent assurer les propriétés ACIT : disponibilité (Availability en anglais), confidentialité (Confidentiality en anglais), intégrité (Integrity en anglais) et traçabilité (Traceability en anglais). La *disponibilité* est la capacité du système à rendre le service pour lequel il est déployé conformément à ses spécifications techniques non fonctionnelles. La *confidentialité* est la propriété d'un système qui n'autorise l'accès à ses ressources qu'aux utilisateurs accrédités. L'*intégrité* est l'aptitude du système à pouvoir maintenir les données dans un état cohérent et à éventuellement reprendre son fonctionnement normal en cas de panne, c'est-à-dire sans que les données soient corrompues. La *traçabilité* renvoie à la capacité d'un système à fournir des informations sur les événements survenus (modification ou suppression de données, accès à une donnée ou à un traitement particulier par exemple). Ces informations peuvent permettre l'audit du système ou même la récupération en cas de panne.

La plupart des systèmes mettent en place un protocole d'authentification. Celui-ci a pour but de ne permettre l'utilisation du système que par des utilisateurs légitimes. Toutefois, une fois un utilisateur authentifié, les menaces de sécurité restent toujours importantes pour les systèmes et leurs données. Une solution est d'appliquer un *contrôle d'accès* adéquat aux actions effectuées et aux données accédées par l'utilisateur. Le contrôle d'accès a d'ailleurs toujours été utilisé dans une certaine me-

sure pour limiter la vue d'un utilisateur du système (données et traitements) vis-à-vis du système dans son ensemble. En particulier, par rapport aux propriétés ACIT, le contrôle d'accès garantit en partie les propriétés de *disponibilité* et de *confidentialité*.

Notre thèse fait partie de deux projets de recherche sur la sécurité fonctionnelle des systèmes d'information qui assure, par des moyens utilisés au niveau application, l'accès aux ressources des systèmes selon les conditions prévues. Le projet canadien EB³SECure (EB³SEC) cible la sécurité fonctionnelle des systèmes d'information d'institutions financières. Le projet français SELKIS, acronyme pour « SEcure heaLth care networKs Information Systems », cible aussi la sécurité fonctionnelle, mais celle des systèmes d'information des établissements de santé. Ils ont pour but l'élaboration de langages, méthodes et procédés permettant d'obtenir des systèmes d'information dont l'aspect sécuritaire intègre les caractéristiques *disponibilité* et *confidentialité* de ACIT et dont le développement s'effectue de manière indépendante du système visé [13, 14].

Problématique

Dans de nombreux secteurs de la société tels que la santé et la finance, les activités sont régies par des réglementations gouvernementales strictes. À celles-ci peuvent aussi s'ajouter des règles internes de l'entreprise qu'elle s'impose pour mener à bien ses affaires dans le cadre de ce que la loi permet. Les entreprises, qui doivent se conformer à ces deux catégories de règles de nature légale, mettent en place des systèmes d'information qui intègrent de telles règles d'une façon ou d'une autre. Mais généralement dans les implémentations des systèmes d'information, le code des programmes associé aux règles légales est entrelacé avec celui associé aux règles d'affaires. Les changements apportés aux programmes deviennent des opérations coûteuses, voire risquées puisqu'elles entraînent la dégradation des systèmes [7, 48]. Ces changements interviennent d'autant plus que les programmes auxquels ils s'appliquent sont issus de réglementations qui sont sujettes à des modifications fréquentes. En effet, les lois sont définies par des instances qui les mettent à jour suivant les besoins de la société, les avancées technologiques ou encore les changements de gouvernement. De même, les

politiques internes des entreprises sont influencées par les changements de direction à la tête de celles-ci.

Un autre problème est l'inverse de celui décrit précédemment. En effet, il peut être requis pour des entreprises de prouver la conformité de leurs systèmes d'information aux lois en vigueur. C'est-à-dire que les systèmes d'information traitent les données et respectent les niveaux d'accès à l'information prévus par les réglementations. Dans les deux cas, une solution consiste en la formulation des règles de contrôle d'accès dans une notation particulière. Cette notation se doit d'avoir une sémantique formelle. Cette caractéristique de la notation offre des pistes de solutions intéressantes, entre autres, à ce dernier problème.

En pratique, dans l'industrie du logiciel, la solution d'implémentation privilégiée est le contrôle d'accès basé sur les rôles (Role-Based Access Control (RBAC) en anglais). Cette méthode largement répandue a d'ailleurs donné lieu à une norme [32]. L'idée sous-jacente de cette technique est d'associer les ressources dont on veut contrôler l'accès (cela peut être des *actions/fonctions* du système, des *données*) à des rôles. Ces rôles permettent ainsi de définir des niveaux d'accès pour des parties du système. Les utilisateurs sont donc ensuite liés aux rôles et la sémantique de cette liaison est qu'ils ne pourront accéder qu'aux ressources du système rattachées à leurs rôles. Le système doit donc être pourvu de mécanismes nécessaires à la mise en œuvre d'une politique de contrôle d'accès. Pour un système d'information, la politique de contrôle d'accès est l'ensemble des règles qui précisent les privilèges des utilisateurs pour l'accès aux ressources du système. Dans le cadre du contrôle d'accès basé sur les rôles, la politique correspond, en plus des associations abstraites entre les ressources, les rôles et les utilisateurs, aux valeurs concrètes pour ces entités. RBAC a donné lieu à de nombreuses évolutions pour prendre en compte de nouveaux besoins, notamment l'ajout de la séparation des devoirs, qui est une contrainte qui empêche qu'un utilisateur d'un système initie un processus et le valide. Cependant, RBAC ne dispose pas de constructions adaptées pour exprimer de façon native les contraintes de type séparation des devoirs, ou plus généralement des contraintes dynamiques ou statiques qui n'entrent pas dans le schéma usuel des politiques RBAC. Une contrainte est consi-

dérée comme *dynamique* lorsque son évaluation tient compte, en plus de l'état actuel du système, de l'historique d'événements survenus dans le système. Cette faiblesse du pouvoir expressif de RBAC peut être comblée par l'utilisation d'une notation basée sur les traces d'actions. C'est ce qui a été proposé dans le cadre des projets EB^{3}SEC et SELKIS.

Avec des politiques de contrôle d'accès qui utilisent cette nouvelle notation, de nouveaux mécanismes de mise en œuvre doivent être trouvés. Le travail de cette thèse s'attaque donc à la problématique de ces moyens de mise en œuvre, particulièrement dans le cas des applications ayant une architecture orientée services (Service Oriented Architecture (SOA) en anglais). Ces mécanismes de mise en œuvre vont se traduire dans la pratique par un cadre d'applications qui comporte un filtre de contrôle d'accès dynamique, composant indispensable pour mettre en vigueur les politiques de contrôle d'accès ayant des aspects dynamiques. Ce cadre d'applications doit être dérivé, automatiquement autant que possible, de la politique de contrôle d'accès. Bien qu'obtenu de façon automatique, un cadre d'applications doit être viable et opérationnel pour le système d'information auquel il est destiné en production. L'automatisation dans ce cas doit assurer que le cadre d'applications est conforme à toute politique de contrôle d'accès et exempt de défauts. Cette automatisation vise aussi à simplifier la maintenance des programmes de mise en application des règles de sécurité.

Objectifs

Le principal objectif de cette thèse est de valider l'approche globale proposée dans les projets EB^{3}SEC et SELKIS qui s'inscrivent dans le domaine de la sécurité fonctionnelle des systèmes d'information, plus particulièrement celui du contrôle d'accès. En effet, cette approche consiste, d'une part, en l'utilisation de méthodes formelles pour la spécification de politiques de contrôle d'accès, et d'autre part, en l'implémentation correcte de ces politiques dans les applications cibles. La finalité de notre travail consiste à vérifier que les implémentations des solutions proposées dans cette thèse sont réalisables et viables dans le contexte de systèmes d'information de type SOA. Cet objectif se décline en trois sous-objectifs précis.

Premièrement, il s'agit de définir un modèle générique qui spécifie les concepts nécessaires pour la mise en œuvre du contrôle d'accès dans les applications de type SOA. Ce modèle est représenté par un métamodèle qui décrit les principaux concepts d'un « Policy Enforcement Manager » (PEM). Cette description doit comprendre, entre autres, une vue du fonctionnement de chacun des éléments du métamodèle, ainsi que leurs relations les uns par rapport aux autres. Les instances de ce métamodèle doivent pouvoir mettre en œuvre non pas une seule politique, mais plusieurs politiques de contrôle d'accès de façon simultanée et flexible. En effet, dans le milieu de la santé par exemple, il arrive des situations d'urgence pendant lesquelles les politiques usuelles des établissements de santé sont suspendues et des règles particulières s'appliquent. Tout comme en cas d'une catastrophe, les organisations impliquées disposent de procédures d'accès à l'information qui sont différentes de celles utilisées en situation normale. Le métamodèle doit prendre en compte cette particularité. Il doit aussi clairement dégager les responsabilités de chaque composant. En particulier, il doit permettre une séparation nette entre l'application/les données contrôlées et les mécanismes de contrôle. Cette séparation doit être telle que les changements au niveau de l'application contrôlée entraînent peu ou pas — dans l'idéal — de modifications aux mécanismes de mise en œuvre du contrôle d'accès. Une telle séparation doit aussi exister entre les composants du PEM. En effet, la modification d'une politique de contrôle d'accès ne doit pas entraîner des modifications profondes sur les composants du PEM.

Deuxièmement, il s'agit d'identifier et de réaliser de possibles implémentations du PEM basées sur des technologies qui s'intègrent le mieux à des applications SOA. Ces implémentations doivent permettre la gestion aisée de politiques de contrôle d'accès. Elles doivent aussi requérir un minimum d'effort pour passer de l'étape de spécification à l'étape de déploiement sans toutefois compromettre les performances du cadre d'applications de contrôle d'accès. Deux méthodes d'implémentation ont été retenues, chacune avec ses propres avantages et inconvénients. Les politiques de contrôle d'accès considérées sont exprimées avec des notations formelles, notamment la notation « Algebraic State Transition Diagram » (ASTD) [21]. Cette notation graphique est dotée d'une sémantique formelle. Ses éléments sont hérités des statecharts [30] et

sont fortement influencés par les opérateurs du langage d'expressions de processus EB³ [24]. La première implémentation visée considère la *transformation*, la plus automatique possible, de politiques de contrôle d'accès vers un composant autonome dans le cadre d'applications de contrôle d'accès. La seconde implémentation consiste en l'*interprétation* directe de politiques de contrôle d'accès, c'est-à-dire l'exécution de celles-ci par le PEM sans transformation dans une représentation intermédiaire. Un composant particulier du PEM a la responsabilité d'utiliser cette interprétation pour effectuer la mise en œuvre de politiques de contrôle d'accès. Les deux implémentations doivent permettre une réutilisation des règles qui composent les politiques de contrôle d'accès.

Troisièmement, il s'agit d'évaluer, à partir d'une étude expérimentale, l'exactitude et la viabilité de chacune des méthodes d'implémentation proposées dans un environnement semblable à celui d'un environnement de production en entreprise. Cette évaluation s'effectue à l'aide d'un banc d'essai. De plus, les implémentations sont comparées l'une à l'autre. Cette comparaison se fait par la mise en opposition des résultats de tests de performance effectués pour les deux implémentations sous des conditions similaires voire identiques.

Méthodologie

La méthodologie adoptée pour s'attaquer à notre problème comporte trois phases : conceptuelle, algorithmique et expérimentale. Durant la phase conceptuelle, un cadre d'applications pour le développement du contrôle d'accès dans des applications SOA est élaboré. Il se dérive à partir d'un modèle générique qui définit tous les composants nécessaires au contrôle d'accès ainsi que leurs interactions, et cela dans l'optique où les politiques de contrôle d'accès peuvent être hiérarchiques. Ce modèle générique est décrit par un métamodèle en utilisant la notation UML [41, 42], notamment à travers des diagrammes de classes et des diagrammes de séquences. UML est l'une des notations graphiques les plus utilisées en ingénierie du logiciel. Elle permet non seulement d'analyser et de concevoir des applications (orientées objet pour la plupart), mais aussi de représenter des métamodèles de telles applications. Dans le cadre de

cette thèse, ce modèle générique est basé sur les concepts proposés par la norme d'un langage XML défini par l'« Organization for the Advancement of Structured Information Standards » (OASIS) [1] et nommé « eXtensible Access Control Markup Language » (XACML) [39]. Ce langage permet d'exprimer des politiques de contrôle d'accès. Cette norme propose aussi une architecture cadre pour la mise en œuvre du contrôle d'accès. Ce sont les principaux éléments de cette architecture qui sont repris et constituent les pivots du modèle générique proposé. Ces principaux éléments sont le « Policy Enforcement Point » (PEP) et le « Policy Decision Point » (PDP). Le PEP est le point focal du mécanisme de mise en œuvre de politiques de contrôle d'accès. Il reçoit les requêtes d'accès aux ressources ou services d'une application et, suivant les politiques en vigueur, autorise l'exécution de la requête. Le composant PDP est responsable du calcul des décisions d'autorisation pour les requêtes reçues au niveau du composant PEP.

La phase algorithmique, basée sur une instance du métamodèle, donne une forme concrète au traitement effectué par le *filtre de contrôle d'accès dynamique*, un des composants du métamodèle. Les données d'entrée à ce traitement sont des politiques de contrôle d'accès exprimées à l'aide de la notation formelle ASTD, car leur mise en application dans le cadre du métamodèle se fait à travers le filtre de contrôle d'accès dynamique qui calcule au niveau de granularité le plus fin, pour une requête donnée, une décision d'autorisation. Deux méthodes sont examinées tour à tour.

D'abord le mécanisme de la *transformation* est détaillé sous la forme d'un algorithme qui traduit des politiques de contrôle d'accès exprimées en ASTD en un ensemble de processus écrits dans le langage BPEL [40] et qui s'exécutent dans un moteur d'exécution BPEL, une autre norme de l'OASIS. Il s'agit là aussi d'un langage XML pour l'implémentation de processus qui vont interagir avec leur environnement par le mécanisme de services Web. La transformation proposée prend la forme d'un algorithme qui met en correspondance les éléments de la notation ASTD et des constructions du langage BPEL. Les limitations de cet algorithme sont considérées, et en particulier les restrictions sur les politiques de contrôle d'accès acceptées par

1. Site Web de OASIS : www.oasis-open.org/org

l'algorithme ainsi que des pistes de solutions de contournement sont envisagées. L'algorithme génère aussi des documents auxiliaires selon les formats « Web Service Definition Language » (WSDL) [2] et « XML Schema Document » (XSD) [3] [4] qui contiennent respectivement la définition des interfaces et la déclaration des types nécessaires au bon fonctionnement des processus BPEL.

Ensuite, l'*interprétation* est utilisée pour implémenter une autre approche du filtre de contrôle d'accès dynamique. À cet égard, l'état initial d'une politique de contrôle d'accès, toujours exprimée avec la notation ASTD, est calculé. Cet état évolue ensuite en fonction des requêtes, aux ressources contrôlées, acceptées par la spécification. Dans ce cas de figure, une requête est vue comme une action de l'environnement et la décision d'autorisation rendue est positive si la requête est acceptée dans l'état courant de la spécification. L'interpréteur est composé d'un ensemble de processus BPEL et de quelques documents XML tels qu'un document d'interfaces WSDL, un document de types XSD et des documents en format « Extensible Stylesheet Language Transformation » (XSLT) [5]. Il comprend notamment un processus BPEL pour chaque type de structure ASTD et chacun de ces processus, en tant que service, offre les opérations `Execute`, `IsFinal` et `IS`. L'opération `Execute` calcule un nouvel état pour une spécification, un état et une requête reçus en paramètres. L'opération `IsFinal` détermine si l'état d'une spécification est *final* et l'opération `IS` « initial state » calcule l'état initial d'une spécification ASTD. Les documents XSLT réalisent des calculs sur des données XML qu'il n'est pas possible d'effectuer directement avec les éléments du langage BPEL, essentiellement à cause des limites de celui-ci.

Enfin, la phase expérimentale compare les deux méthodes d'implémentation du filtre de contrôle d'accès dynamique. À cet égard, les deux solutions retenues du filtre sont développées et déployées dans le moteur d'exécution « Orchestration Director Engine » (ODE) [6]. Ce moteur d'exécution est un projet de la fondation Apache du

2. Standard WSDL du W3C : www.w3.org/TR/wsdl
3. Structures de données de la norme XSD : www.w3.org/TR/xmlschema-1
4. Types de données de la norme XSD : www.w3.org/TR/xmlschema-2
5. Standard XSLT du W3C : www.w3.org/TR/xslt
6. Site Web du moteur ODE : ode.apache.org

logiciel (Apache Software Foundation en anglais) [7]. Il peut être aisément installé dans un conteneur d'applications tel que Apache Tomcat [8]. Le déploiement des processus BPEL se fait à chaud — sans redémarrage de l'application Web ODE ou du conteneur Tomcat — par une simple copie de fichiers et nécessite uniquement une déclaration des processus (puisqu'ils sont exposés comme des services Web) dans un fichier XML. Deux types de test sont effectués pour chacun des filtres déployés dans ODE. Les tests de fonctionnalité valident les implémentations de ces filtres tandis que les tests de performance donnent une vue précise du comportement de celles-ci sous certaines conditions. Durant ces tests, des métriques relatives aux temps d'exécution et à la taille des données échangées sont évaluées. Les valeurs de ces métriques sont interprétées pour comparer les implémentations de façon objective.

Contributions

Les résultats de notre thèse s'inscrivent dans le cadre des projets EB^{3}SEC et SELKIS sous deux hypothèses principales. La première soutient que la spécification des politiques de contrôle d'accès aux ressources d'un système d'information dans un langage d'expressions de processus comme ASTD contribue à pallier les faiblesses des méthodes actuelles dans ce domaine. La seconde prétend qu'une séparation nette entre le système d'information et les mécanismes de contrôle d'accès constitue une solution qui facilite la maintenance des systèmes d'information par rapport à leurs politiques de contrôle d'accès, réduisant ainsi l'entropie de ces systèmes. Une telle solution est aussi viable non seulement dans l'écriture de politiques de contrôle d'accès, mais aussi dans leur déploiement, leur exécution et leur maintenance.

Deux solutions originales de filtre de contrôle d'accès dynamique constituent l'apport principal de cette thèse. Bien que les deux solutions suggèrent d'implémenter un filtre de contrôle d'accès dynamique avec un ensemble de processus BPEL, elles diffèrent par les méthodes utilisées : la transformation de spécifications ASTD et l'interprétation de spécifications ASTD. Dans le premier cas, comme il n'a pas été

7. Site Web de la fondation Apache : apache.org
8. Site Web du conteneur d'applications Tomcat : tomcat.apache.org

possible d'associer tous les éléments syntaxiques de la notation ASTD à ceux du langage BPEL à cause des limitations de ce langage, quatre patrons de politiques de contrôle d'accès ont été considérés. Bien que cette restriction puisse paraître sévère, les quatre patrons retenus permettent d'exprimer les contraintes d'accès usuellement rencontrées dans la pratique. La solution proposée est complète par rapport à ces patrons. Elle inclut aussi la génération de documents auxiliaires nécessaires au bon fonctionnement des processus BPEL. Dans le second cas, un processus BPEL, avec ses documents auxiliaires, est associé à chaque type de structures ASTD. Un processus BPEL permet donc l'interprétation de toute structure du type correspondant. Au total douze processus BPEL ont été ainsi développés afin de constituer une machine virtuelle complète d'exécution de spécifications ASTD.

Un autre apport de cette thèse est la définition d'un modèle générique d'un gestionnaire d'exécution de politiques de contrôle d'accès ou PEM qui, une fois instancié, a permis de bien situer le contexte dans lequel les deux solutions de filtre de contrôle d'accès dynamique ont été élaborées, celui d'applications dont l'architecture est orientée services. Ainsi, dans ce contexte, le filtre de contrôle d'accès dynamique est vu comme un service, tout comme les processus BPEL qui le composent, et cela dans les deux méthodes retenues. Une des caractéristiques de ce modèle est qu'il est suffisamment général pour servir de base à d'autres cadres d'applications différents de celui proposé dans cette thèse, à savoir les applications de type SOA.

Le dernier apport de cette thèse est un banc d'essai complet qui permet de corroborer dans une certaine mesure les hypothèses de départ. Il comporte l'implémentation de deux prototypes, un pour chaque solution du filtre de contrôle d'accès dynamique. Il inclut aussi un ensemble exhaustif de jeux d'essai de différentes grandeurs et de nombreux scripts qui permettent le déploiement de chaque solution, l'exécution de jeux d'essai et la génération automatique de tableaux ou graphiques en format Latex à partir des statistiques recueillies. Les conclusions tirées, ainsi que les statistiques obtenues, de cette étude expérimentale constituent des résultats intéressants, puisqu'elles pourront servir à évaluer la pertinence d'approches similaires considérant les limites actuelles des technologies Web, en particulier BPEL, et des méthodes formelles dans la mise en œuvre de solutions logicielles.

Organisation de la thèse

Le reste de la thèse est organisé comme suit. Le chapitre 1 présente les notions nécessaires à la lecture de cette thèse. Entre autres, la notation ASTD, les systèmes d'information dans un environnement SOA et le langage BPEL sont abordés. Le chapitre 2 décrit les différentes techniques de contrôle d'accès utilisées dans le contexte des systèmes d'information. Le chapitre 3 situe notre solution par rapport aux travaux du domaine. Le chapitre 4 introduit le modèle générique qui détaille les composants de l'architecture d'un PEM dans le cas de systèmes d'information de type SOA. Le chapitre 5 donne les détails sur l'implémentation d'un PEM qui met en œuvre des politiques de contrôle d'accès formelles. Il présente l'essence d'un algorithme de transformation de politiques de contrôle d'accès. Le chapitre 6 présente un filtre de contrôle d'accès dynamique qui fait usage de l'interprétation de spécifications formelles. Enfin, le chapitre 7 décrit les prototypes de filtre de contrôle d'accès dynamique réalisés à partir des algorithmes des deux chapitres précédents. Des mesures effectuées pendant leur exécution sont aussi détaillées et commentées. Une conclusion termine cette thèse.

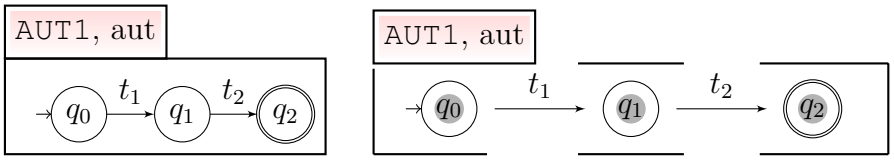
Chapitre 1

Concepts de base

La revue des éléments nécessaires à la compréhension de cette thèse inclut une description de la notation formelle ASTD utilisée pour spécifier des politiques de contrôle d'accès, les applications SOA comme architecture des systèmes d'information, le protocole « Simple Object Access Protocol » (SOAP) pour les échanges de messages entre les services Web, le langage « Web Service Definition Language » (WSDL) pour spécifier les interfaces des processus et le langage de processus « Business Process Execution Language » (BPEL) pour l'implémentation des solutions proposées dans cette thèse.

1.1 Le langage ASTD

Le langage ASTD [23] est une notation formelle qui combine une représentation graphique et les opérateurs des notations telles que CSP [31] et LOTOS [8]. La définition complète des structures ASTD, incluant la représentation mathématique et graphique des structures et des états ASTD ainsi que la sémantique formelle, est disponible dans le rapport [21]. Une spécification ASTD est un ensemble fini de structures ASTD dont l'une, nommée *main*, est marquée comme la structure ASTD principale. L'exécution d'une spécification ASTD est intuitive. L'*état* initial de la structure ASTD principale est calculé et les événements reçus de l'environnement font évoluer l'état courant de la



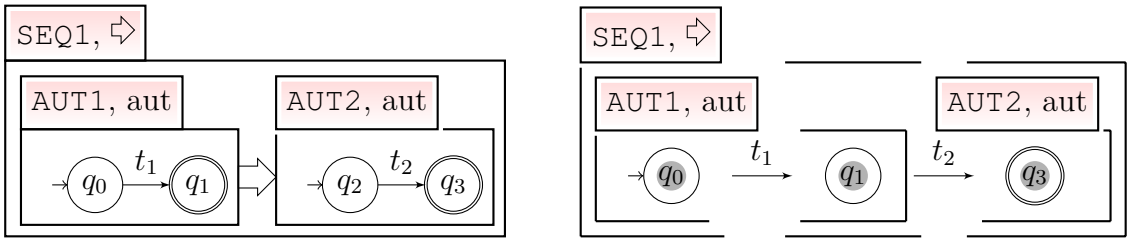
(a) Structure *automate* (b) États successifs de l'automate

Figure 1.1 – Exemple d'une structure ASTD *automate*

spécification vers un nouvel état en fonction de la spécification elle-même (l'ensemble des structures ASTD qui la composent). Il existe neuf structures ASTD.

Une structure ASTD peut être de type *automate* (aut). Dans ce cas, l'automate contient des structures *élémentaires* ou encore d'autres structures ASTD qui sont liées par des transitions $t(\overrightarrow{x})[\phi]$, où t est l'événement ou l'action, $\overrightarrow{x}$ est l'ensemble des paramètres de l'événement et ϕ est un prédicat optionnel qui doit être évalué à vrai pour déclencher la transition. Une structure ASTD *automate* est, dans sa version simplifiée, un système d'*états-transitions* avec la possibilité de paramétrer les transitions par des variables et d'ajouter des conditions d'exécution à celles-ci. La figure 1.1a illustre une structure ASTD *automate* nommée `AUT1`. Elle a deux transitions sans paramètres t_1 et t_2 qui doivent être exécutées en séquence. La figure 1.1b montre l'évolution de l'état de cette structure ASTD depuis l'état initial après les occurrences des événements t_1 et t_2. L'état initial de la structure ASTD *automate* de l'exemple est q_0 comme le montre le jeton présent dans cet état dans la partie gauche de la figure 1.1b. D'après la spécification à la figure 1.1a, seule la transition t_1 est possible dans cet état. Après l'occurrence de l'événement t_1, il y a un changement d'état de q_0 à q_1 comme le montre le jeton dans l'état q_1 (voir la partie médiane de la figure 1.1b). De même, après l'occurrence de l'événement t_2, l'état courant de la structure ASTD *automate* passe de l'état q_1 à l'état q_2 et le jeton se retrouve dans ce dernier état comme l'illustre la partie droite de la figure 1.1b.

Une structure ASTD *séquence* (⇨) permet l'exécution l'une après l'autre de deux structures ASTD qui en sont membres. L'état de la structure ASTD *séquence* est celui de la sous-structure en cours d'exécution. Quant cet état est celui de la première sous-



(a) Structure *séquence* (b) États successifs de la séquence

Figure 1.2 – Exemple d'une structure ASTD *séquence*

structure, le passage à l'état initial de la seconde sous-structure est possible si l'état de la première sous-structure est *final*. Dans l'exemple présenté par la figure 1.2a, la structure ASTD principale est la séquence `SEQ1` et elle comporte deux sous-structures ASTD *automate* qui seront exécutées en séquence. La figure 1.2b illustre l'évolution de l'état de la structure ASTD *séquence* depuis son état initial jusqu'à son état final. Cette évolution se fait à la suite des occurrences successives des événements t_1 et t_2.

Une structure ASTD *choix* ($|$) permet l'exécution au choix d'une structure parmi plusieurs sous-structures ASTD membres. Une telle structure ASTD offre la possibilité d'exécuter les transitions de toutes les sous-structures et, une fois le choix de la sous-structure à exécuter effectué, les autres sous-structures sont ignorées lors d'occurrences futures d'événements. Le choix de la sous-structure est déterminé par la première occurrence d'un événement acceptable.

Une structure ASTD *fermeture de Kleene* ($\star$) spécifie l'exécution répétitive de sa sous-structure ASTD. Son état est celui de la sous-structure. Pour que l'exécution puisse repartir depuis son état initial, il faut que l'état courant soit final.

Une structure ASTD *garde* ($\Rightarrow$) est composée d'une sous-structure ASTD et d'un prédicat. Ce dernier doit être évalué à vrai pour que la sous-structure puisse s'exécuter.

La structure ASTD *appel* ($\langle\rangle$) est l'équivalent de l'appel d'une fonction d'un langage de programmation. En effet, celle-ci permet d'exécuter une structure ASTD définie dans la spécification englobante, en lui passant les paramètres requis.

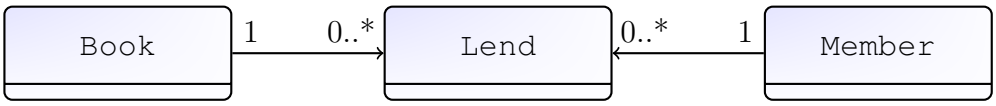
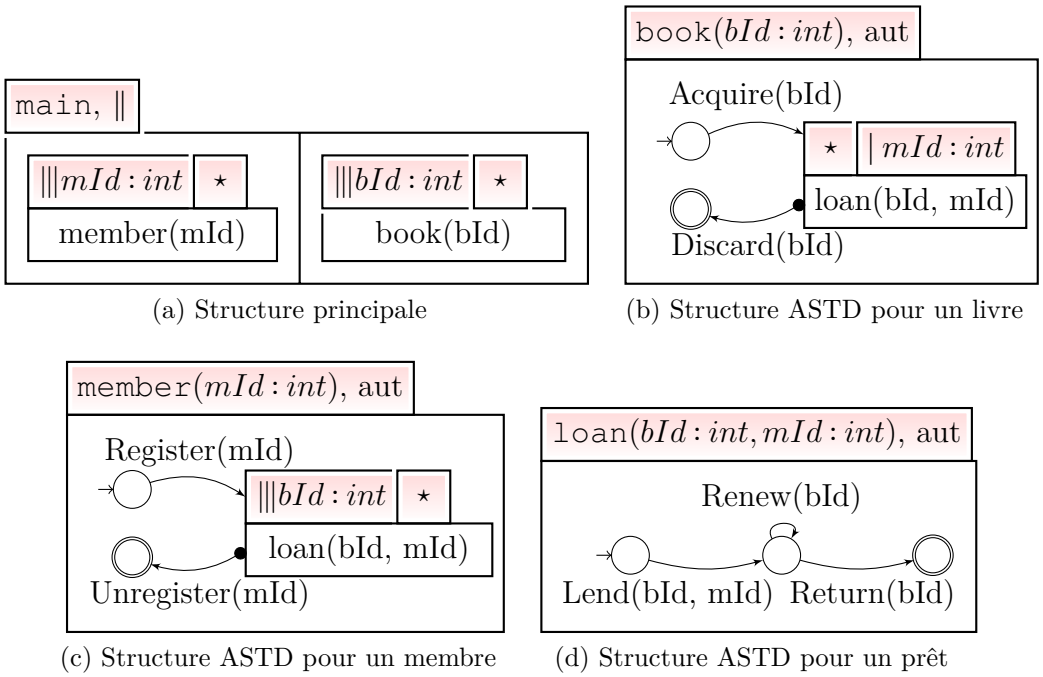


Figure 1.3 – Diagramme de classes d'un système d'information

Une structure ASTD *synchronisation paramétrée* ($|\,[\{\cdots\}]\,|$) permet l'exécution de plusieurs sous-structures ASTD tout en synchronisant cette exécution sur les événements spécifiés dans un ensemble d'événements synchrones Δ. Cette structure a une version *quantifiée* ($|\,[\{\cdots\}]\mid x:T$) qui comporte une unique sous-structure ASTD, c'est-à-dire qu'elle permet d'exécuter, en parallèle ou de manière entrelacée ($|||\ x:T$), la sous-structure pour différentes valeurs de la variable de quantification.

La dernière structure ASTD est le *choix quantifié* ($|\ x:T$) qui permet l'exécution de la sous-structure pour une unique valeur de la variable de quantification (d'où l'appellation *choix quantifié*).

La méthode de spécification ASTD, qui inclut le langage ASTD, a été proposée dans la perspective d'être utilisée pour modéliser des systèmes d'information. La spécification complète d'un système comprend en plus de la description des ASTD, un modèle UML des *entités* du système et la définition des *attributs* de ces entités. Dans cet ensemble de modèles, la spécification ASTD vient préciser le comportement des différentes entités du système et chaque transition de la spécification modifie éventuellement des attributs de ces entités. Le diagramme de classes de la figure 1.3 montre la relation entre les entités `Book` et `Member` du système d'information d'une bibliothèque. Pour modéliser le comportement du processus de prêt de livres de cette bibliothèque, on peut lui ajouter la spécification ASTD de la figure 1.4. La structure ASTD *automate* de la figure 1.4b montre le comportement des instances de l'entité `Book` : un livre est acquis (événement `Acquire`), ensuite il peut être emprunté un certain nombre de fois (structure ASTD `loan`) et enfin il est retiré de la bibliothèque (événement `Discard`). La spécification ASTD *automate* de la figure 1.4d modélise le cycle d'un prêt. Celui-ci débute par l'emprunt d'un livre par un membre (événement `Lend`). Le prêt peut être ensuite renouvelé un certain nombre de fois (événement



(a) Structure principale

(b) Structure ASTD pour un livre

(c) Structure ASTD pour un membre

(d) Structure ASTD pour un prêt

Figure 1.4 – Spécification ASTD d'une bibliothèque [21]

`Renew`) et, enfin, le livre est retourné à la bibliothèque et le prêt se termine (événement `Return`). La spécification du comportement d'un membre de la bibliothèque (voir la figure 1.4c) est semblable à celle du comportement d'un livre. La structure `main` est une structure ASTD *synchronisation paramétrée* (voir la figure 1.4a) qui met en parallèle les structures ASTD des entités précédemment mentionnées. Dans les spécifications de la figure 1.4 il faut noter l'utilisation des variables identifiantes `bId` et `mId` pour différencier les instances des entités `Book` et `Member`.

Le langage ASTD présente une structure hiérarchique qui se prête bien au format de données XML. Une spécification ASTD peut aisément être écrite dans un document XML. La structure ASTD *automate* `AUT1` de la figure 1.1 correspond au document XML du programme 1.1. Le programme liste les états q_0, q_1 et q_2 de l'automate aux lignes 3 à 13. La liste des transitions t_1 et t_2 de l'automate complète la spécification de l'automate aux lignes 14 à 33. L'attribut `N0` à la ligne 2 spécifie quel est l'état initial de l'automate.

```
1  <Specification>
2   <Automaton Name="AUT1" N0="q0">
3    <States>
4     <State Name="q0">
5      <Elementary Final="false"/>
6     </State>
7     <State Name="q1">
8      <Elementary Final="false"/>
9     </State>
10    <State Name="q2">
11     <Elementary Final="true"/>
12    </State>
13   </States>
14   <Transitions>
15    <Transition Final="false">
16     <Phi>
17      <Predicate>
18       <Boolean>true</Boolean>
19      </Predicate>
20     </Phi>
21     <LocalArrow N1="q0" N2="q1"/>
22     <Event Name="t1"/>
23    </Transition>
24    <Transition Final="false">
25     <Phi>
26      <Predicate>
27       <Boolean>true</Boolean>
28      </Predicate>
29     </Phi>
30     <LocalArrow N1="q1" N2="q2"/>
31     <Event Name="t2"/>
32    </Transition>
33   </Transitions>
34  </Automaton>
35 </Specification>
```

Programme 1.1 – Document XML d'une structure ASTD *automate*

Pour pouvoir valider le document XML d'une spécification ASTD, il est important d'avoir un modèle de référence, c'est-à-dire un métamodèle du langage ASTD. Il décrit la forme de chacune des constructions de ce langage. Le programme 1.2 illustre une partie de ce métamodèle. La version complète du document XSD est disponible à l'annexe A dans le programme A.1. Il inclut une spécification d'éléments XML pour chacune des structures ASTD et chacun des états qui leur sont associés. Il inclut également une spécification des variables et de leurs types. Développer ce métamodèle dans le cadre de cette thèse a permis de valider que chaque spécification ASTD est syntaxiquement bien formée. Aux lignes 6 à 24 du programme 1.2, le métamodèle précise qu'un automate (élément `Automaton`) contient un élément `States` qui à son tour contient une liste, possiblement vide, d'éléments `State`. Le contenu de l'élément `State` peut être un élément `Elementary` ou un élément ASTD. De même que pour les états, l'élément `Automaton` contient un élément `Transitions` qui à son tour contient une liste, possiblement vide, d'éléments `Transition` (lignes 25 à 31). Le document XSD dans le programme 1.2 contient aussi la définition d'un métamodèle pour les états d'une spécification ASTD. Ce schéma de données est discuté plus en détails à la section 6.2.

```
1  <element name="Automaton"
   type="tAutomaton"/>
2  <complexType name="tAutomaton">
3   <complexContent>
4    <extension base="tASTDBase">
5     <sequence>
6      <element name="States">
7       <complexType>
8        <sequence>
9         <element name="State"
           minOccurs="0"
           maxOccurs="unbounded">
10         <complexType>
11          <choice>
12           <element name="Elementary">
13            <complexType>
14             <attribute name="Final"
                type="boolean"
                default="false"/>
15            </complexType>
16           </element>
17           <group ref="gASTD"/>
18          </choice>
19          <attribute ref="Name"
             use="required"/>
20         </complexType>
21        </element>
22       </sequence>
23      </complexType>
24     </element>
25     <element name="Transitions">
26      <complexType>
27       <sequence>
28        <element ref="Transition"
           maxOccurs="unbounded"/>
29       </sequence>
30      </complexType>
31     </element>
32    </sequence>
33    <attribute name="N0"
       use="required"/>
34   </extension>
35  </complexContent>
36 </complexType>
```

Programme 1.2 – Métamodèle d'une structure ASTD *automate*

Le programme 1.3 est la représentation XML d'un état de l'automate ASTD de la figure 1.1. Cet état correspond à l'état courant de l'automate après l'occurrence de l'événement t_1, c'est-à-dire l'état représenté dans la partie médiane de la figure 1.1b. À la ligne 1 du programme 1.3, l'attribut `N` spécifie que l'automate se trouve dans l'état q_1. Les lignes 3 à 5 indiquent que l'état courant de l'automate est un état élémentaire (élément `ElementaryState`).

1.2 Applications SOA

Le paradigme SOA est une technologie née dans les entreprises comme une solution pratique au développement d'applications dont le bloc de construction principal est le *service*. Un service représente l'encapsulation de fonctions reliées. Il est le plus souvent

```
1  <AutomatonState N="q1">
2   <Histories/>
3    <S>
4     <ElementaryState/>
5    </S>
6  </AutomatonState>
```

Programme 1.3 – Document XML de l'état d'une structure ASTD *automate*

accédé par des protocoles normalisés, requis pour réaliser un couplage faible entre le service et ses clients. Cette caractéristique a pour conséquence que les services sont le plus souvent utilisés par des systèmes distribués.

Selon Erl [19], le succès de la mise en œuvre d'un projet d'applications SOA nécessite à un certain degré l'emploi des principes suivants :

- les contrats de service à partir desquels les clients et les fournisseurs s'accordent, entre autres, sur les protocoles de communication, la qualité du service rendu et le coût de celui-ci ;
- le couplage de services ;
- la réutilisabilité des services ;
- l'autonomie des services ;
- l'absence d'état des services, c'est-à-dire que l'échange qui se produit entre le client et le service doit comporter toute l'information requise pour traiter la requête du client et une fois cet échange terminé le service ne conserve pas de données sur celui-ci ;
- les répertoires de services, c'est-à-dire que ceux-ci permettent aux clients de pouvoir *découvrir* des services pouvant répondre à leurs besoins ;
- la composition des services, c'est-à-dire que les services peuvent être composés pour offrir de nouveaux comportements plus complets ou satisfaire de nouveaux besoins précis.

L'implémentation la plus courante des services sont les *services Web*. Ceux-ci utilisent des technologies répandues du Web pour réaliser le *couplage faible* entre services. De nombreuses trousses de développement peuvent être utilisées pour leur réalisation. Les échanges entre les services et leurs clients sont souvent réalisés avec des protocoles basés sur des formats XML. Le plus utilisé de ces protocoles est le protocole SOAP pour le formatage des messages échangés. Leurs interfaces sont décrites avec le langage WSDL.

Le langage WSDL est le langage XML de description des services Web. Cette description peut être *abstraite*. Dans ce cas, seuls les types de port, leurs opérations

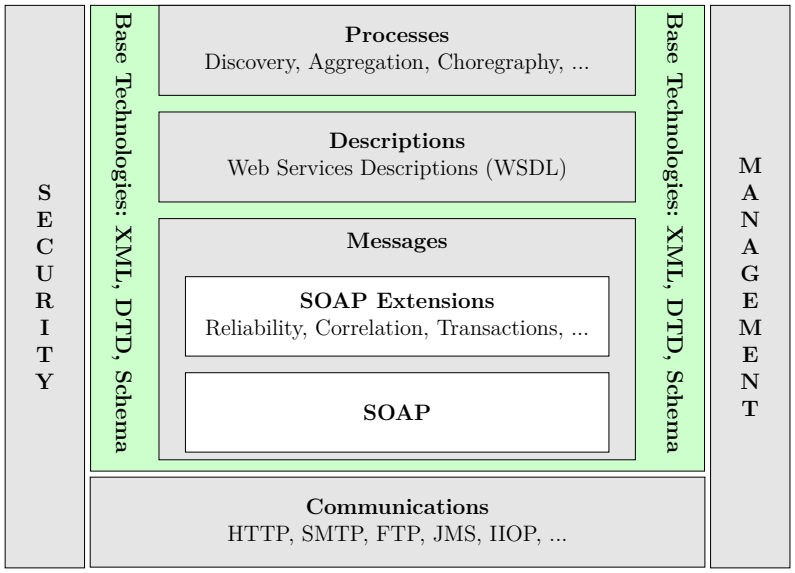


Figure 1.5 – Technologies des services Web

et leurs paramètres incluant les types sont spécifiés. La description des services Web peut être aussi *concrète* et inclure en plus la spécification des protocoles d'encodage des messages ainsi que les protocoles de transport de ces messages.

1.2.1 Services Web

L'entité W3C définit les services Web comme des systèmes logiciels conçus pour effectuer des interactions de machine à machine interopérables sur un réseau. L'interface d'un service Web est décrite par un document WSDL. Le client et le fournisseur qui s'accordent sur ce contrat peuvent interagir. Ces interactions sont réalisées par des messages conformes au protocole SOAP, généralement à travers des canaux HTTP. Un service Web supporte deux modes de communication. Le premier, *simple appel* (one-way call en anglais), permet au client d'appeler un service Web de façon asynchrone et de ne pas attendre de réponse. Le second mode est le mode *requête-réponse* (request-response en anglais). Il permet aux clients d'un service Web d'envoyer une requête de façon synchrone et d'attendre une réponse de celui-ci. La figure 1.5 (figure

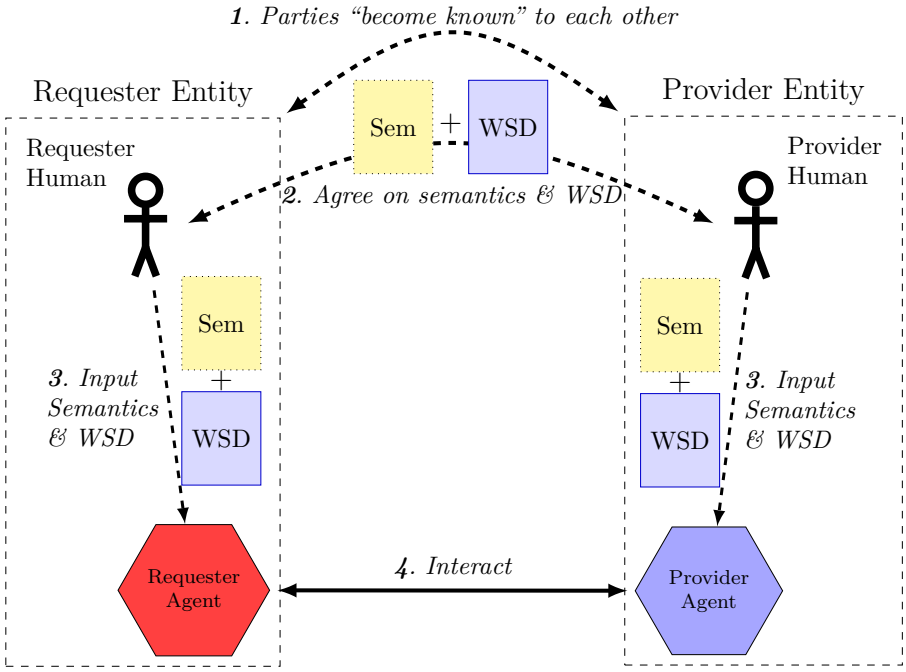


Figure 1.6 – Interactions avec un service Web

3.1 du document Web Service Architecture [1]) montre différentes technologies autour desquelles les services Web sont bâtis.

Une interaction avec un service Web implique un client et un fournisseur du service. Elle se déroule en plusieurs étapes résumées par la figure 1.6 (figure 1.1 du document Web Service Architecture [2]) :

- le client et le fournisseur se découvrent, ceci est souvent réalisé à travers la consultation d'un répertoire par le client pour chercher un fournisseur pour le service dont il a besoin ;
- le client et le fournisseur s'accordent sur le contrat qui va régir l'interaction, le plus souvent le client respecte la description WSDL du service fournie par le fournisseur de celui-ci ;
- le client du service et celui-ci interagissent par l'échange de messages.

1. Web Service Architecture Stack : www.w3.org/TR/ws-arch
2. ib.

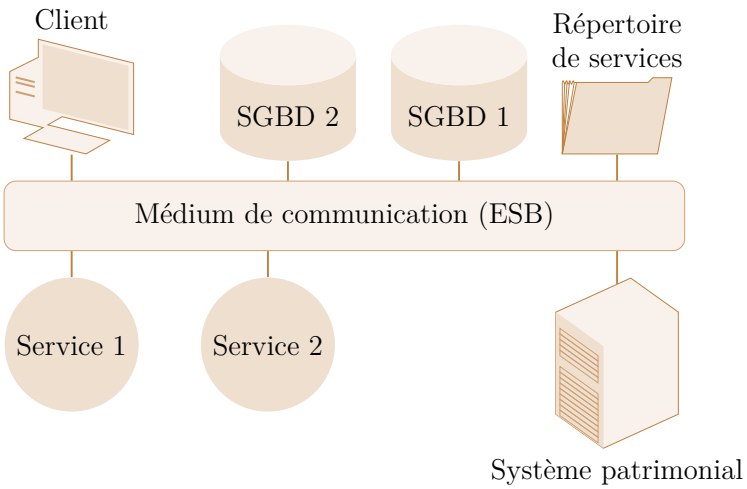


Figure 1.7 – Implémentation d'une application SOA

1.2.2 Implémentation d'une application SOA

La figure 1.7 montre une implémentation courante d'une application SOA. En particulier les relations entre les différents éléments d'un environnement SOA. Le composant ESB joue le rôle de composant « médium de communication » pour les services de l'application. Il fournit de nombreuses fonctionnalités à l'environnement. Par exemple, il effectue le routage des requêtes aux services visées par celles-ci. Il peut jouer le rôle de répertoire pour les services de l'application. Le composant ESB fait aussi le lien avec des services ou des composants particuliers de l'environnement qui offrent par exemple des fonctionnalités de sécurité (par exemple pare-feu, authentification).

1.3 Le protocole SOAP

Le protocole SOAP [3] est un format de données XML utilisé dans l'industrie pour sérialiser des objets. Ces objets peuvent être des messages ou des appels de méthodes. Cette sérialisation se fait le plus souvent dans le but de faire communiquer des processus distincts. Par exemple, un programme suspendu peut sérialiser l'état de sa mémoire pour reprendre son exécution de façon transparente. Dans le cadre des

3. Standard SOAP : www.w3.org/TR/soap12-part1

```
1  <soap:Envelope
2   xmlns:soap="http://.../envelope/"
3   xmlns:fil="http://.../Filter"
4   xmlns:com="http://.../Common">
5   <soap:Header/>
6   <soap:Body>
7    <fil:ARRequestType>
8     <com:PId>001</com:PId>
9     <fil:Event com:Name="T1"/>
10   </fil:ARRequestType>
11  </soap:Body>
12 </soap:Envelope>
```

Programme 1.4 – Requête SOAP

```
1  <soap:Envelope
2   xmlns:soap="http://.../envelope/"
3   xmlns:fil="http://.../Filter">
4   <soap:Header/>
5   <soap:Body>
6    <fil:ARResponseType>
7     <fil:Access>
8      granted
9     </fil:Access>
10    <fil:Debug/>
11   </fil:ARResponseType>
12  </soap:Body>
13 </soap:Envelope>
```

Programme 1.5 – Réponse SOAP

services Web, le protocole est utilisé pour formater les requêtes et les réponses des services.

Un document SOAP est constitué de deux parties. La première partie est l'*en-tête* (élément **`Header`**). Elle est optionnelle et contient des métadonnées sur les données métier du document. Elle peut aussi contenir des données provenant d'extensions au protocole SOAP. Ce moyen est utilisé pour sécuriser le contenu des messages par des mécanismes de signature digitale ou de cryptographie sur tout ou partie des données métier du document. Les informations relatives à la signature/cryptographie sont incluses dans l'en-tête. Dans le programme 1.4 (respectivement le programme 1.5), la ligne 5 (respectivement la ligne 4) indique que la partie en-tête de la requête (respectivement la réponse) SOAP est vide. La seconde partie du document SOAP est le *corps* (élément **`Body`**) et contient les données métier du document. Le corps d'un document SOAP pour une requête à un service Web contient le nom du service souhaité ainsi que les paramètres passés à celui-ci. En fonction de la version du protocole utilisé, le corps du document peut ne contenir qu'une instance du message de l'opération. Dans le programme 1.4 (respectivement le programme 1.5), le corps de la requête (respectivement la réponse) se trouve entre les lignes 6 à 11 (respectivement les lignes 5 à 12).

Les implémentations les plus courantes des serveurs de services Web possèdent un mécanisme de gestionnaire (handler en anglais) de messages SOAP. Ces gestionnaires

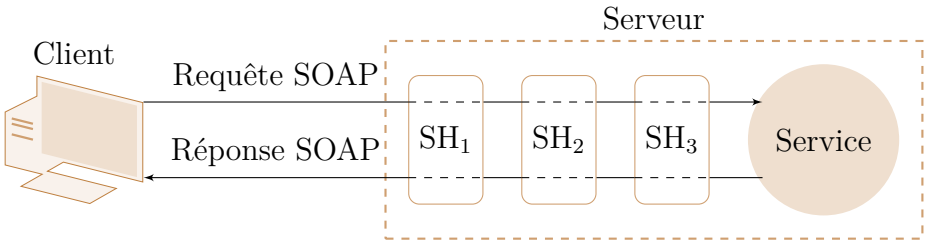


Figure 1.8 – Exemple de gestionnaires SOAP

sont interposés entre les services et leurs clients et ils sont transparents vis-à-vis à la fois des services et des clients. Leur rôle est très souvent de fournir une fonctionnalité supplémentaire et basée sur le contenu des données présentes dans l'en-tête du message traité. Par exemple, un gestionnaire peut effectuer le décryptage de données cryptées dans le corps du message SOAP à partir de l'information sur l'algorithme à utiliser dans l'en-tête. Un service Web peut avoir plusieurs gestionnaires de messages installés. Dans l'exemple de la figure 1.8, trois gestionnaires SOAP ($\texttt{SH}_1$, $\texttt{SH}_2$ et $\texttt{SH}_3$) sont déployés pour le service Web `Service`. Un gestionnaire installé pour un service Web reçoit les messages SOAP de toutes les requêtes adressées à ce service. Il effectue les traitements désirés sur ces messages et « pousse » les messages SOAP résultants vers le gestionnaire suivant de la chaîne de gestionnaires, ou vers le service cible s'il est le dernier de la chaîne. Dans cette direction, les messages SOAP sont dits *entrants* (inbound en anglais). De même, toutes les réponses envoyées par un service Web à un client passent par ses gestionnaires SOAP, mais dans le sens inverse des messages entrants. Les messages SOAP sont dits *sortants* (outbond en anglais). Le mécanisme des gestionnaires SOAP sont transparents pour les services et leurs clients.

1.4 Le langage WSDL

Le langage WSDL est une norme du W3C. Il en existe deux versions[4] [5] et leurs différences sont données dans la figure 1.9 (figure d'un document du wiki PyWPS[6]).

4. Standard WSDL 1.1 : www.w3.org/TR/wsdl
5. Standard WSDL 2.0 : www.w3.org/TR/wsdl20
6. WSDL : wiki.rsg.pml.ac.uk/pywpswiki/index.php?title=WSDL&oldid=746

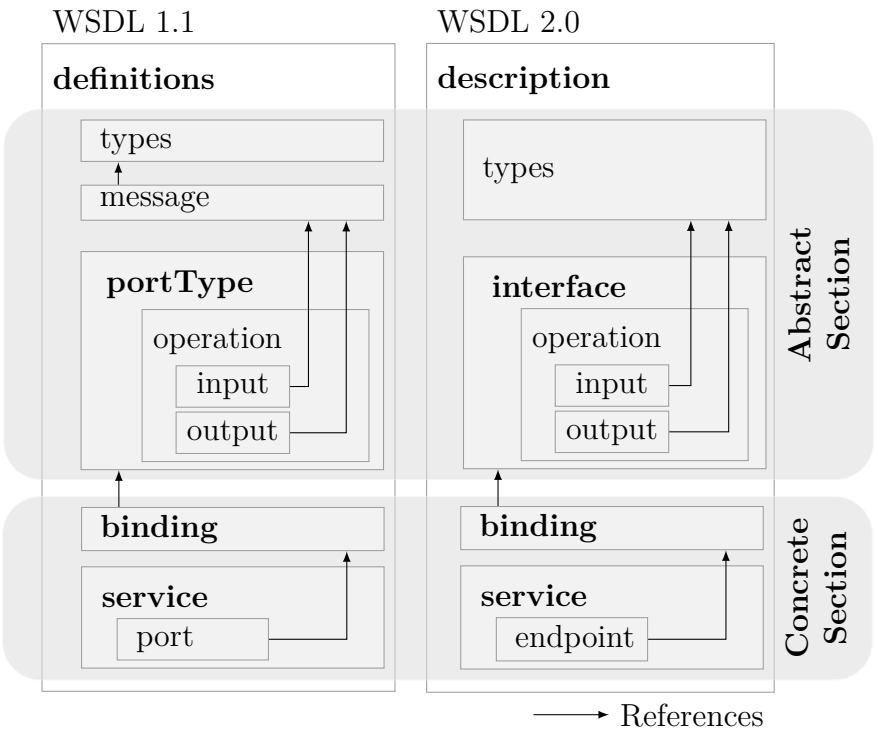


Figure 1.9 – Structure d'un document WSDL selon deux versions

Une spécification WSDL est un document XML qui décrit en détail l'interface d'un ou plusieurs services Web. Un service Web est décrit comme un ensemble d'opérations liées et accessibles sur le réseau. Cette description inclut les types de données échangées avec les clients des services ainsi que le protocole d'encodage de ces données.

Un document WSDL distingue les aspects *abstrait* et *concret* de la description des services Web. La partie abstraite de la définition (version 1.1 de la norme) comprend la définition de types (simples, complexes ou éléments XML), des messages construits sur ces types et des opérations utilisant ces messages pour les types d'information reçue ou envoyée. Pour compléter la description abstraite des services, ce langage permet la définition de *types de port* (port types en anglais) qui rassemblent une ou plusieurs opérations précédemment définies. Cette définition abstraite correspond à l'interface d'une classe dans un langage comme C++. La définition concrète d'un

```
1  <definitions name="Filter" xmlns=".../wsdl/"
    xmlns:xsd=".../XMLSchema" xmlns:soap=".../soap/" xmlns:fx=".../Filter">
2
3   <types>
4    <xsd:schema targetNamespace="http://gril.udes.ca/astd/wsdl/Filter">
5     <xsd:import schemaLocation="Filter.xsd" namespace="..."/>
6    </xsd:schema>
7   </types>
8   <message name="ARRequest">
9    <part name="payload" element="fx:ARRequestType" />
10  </message>
11  <message name="ARResponse">
12   <part name="result" element="fx:ARResponseType" />
13  </message>
14  <portType name="FilterPortType">
15   <operation name="AuthorizationRequest">
16    <input name="inputAR" message="ARRequest"/>
17    <output name="outputAR" message="ARResponse"/>
18   </operation>
19  </portType>
20
21  <binding name="FilterBinding" type="FilterPortType">
22   <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
23   <operation name="AuthorizationRequest">
24    <soap:operation
       soapAction="http://.../Filter_AuthorizationRequest"/>
25    <input name="inputAR"><soap:body use="literal"/></input>
26    <output name="outputAR"><soap:body use="literal"/></output>
27   </operation>
28  </binding>
29  <service name="FilterService">
30   <port name="FilterPort" binding="FilterBinding">
31    <soap:address
       location="http://192.168.1.1:8080/ode/processes/Filter_Service"/>
32   </port>
33  </service>
34 </definitions>
```

Programme 1.6 – Extrait d'un document WSDL

service introduit deux éléments concrets. Elle permet de définir pour chaque type de port, l'encodage des messages reçus ou envoyés pour les opérations de ce type de port ; cette définition utilise l'élément **`binding`**. La définition concrète se termine avec la définition des points d'accès sur le réseau au service. L'élément **`service`** est utilisé et précise à la fois l'adresse et le protocole de transport sur le réseau.

Dans le programme 1.6, les lignes 3 à 19 constituent la définition abstraite, dans la version WSDL 1.1 de la norme, du service `FilterService` (ligne 29). À ces lignes, les messages `ARRequest` et `ARResponse` sont définis chacun par un seul élément

part, mais il est possible d'en avoir plus d'un. Les types des parties (élément **part**) des messages sont définis entre les éléments **types**. Dans le cas de l'exemple du programme 1.6, les types sont définis dans un document XSD importé à la ligne 5. Le service décrit par le document WSDL comporte l'opération `AuthorizationRequest` définie aux lignes 15 à 18. La définition concrète du service intervient aux lignes 21 à 33. La ligne 22 indique que les échanges avec le service peuvent se faire par des messages SOAP et la ligne 31 précise l'adresse pour accéder au service ainsi que le protocole HTTP comme protocole de transport réseau.

1.5 Le langage de processus BPEL

Le langage BPEL est un langage XML pour implémenter les processus d'affaires. Les processus ainsi implémentés sont exécutés dans des moteurs d'exécution BPEL et exposés comme des services Web. Dans ce cas, le processus est dit *exécutable*. Il est aussi possible d'écrire un processus BPEL *abstrait*, c'est-à-dire un processus qui a des activités dont les détails d'implémentation sont omis.

Un processus BPEL est constitué d'*activités* qui sont l'unité élémentaire d'exécution. Les activités de base sont les opérations pour communiquer avec les services Web : l'activité **receive** qui permet de recevoir le message d'un service, l'activité **reply** qui permet de répondre à une activité **receive** précédemment exécutée et l'activité **invoke** qui permet de faire un appel à un service Web. Avec ces activités, un processus BPEL peut être à la fois client et fournisseur de services Web. Le langage fournit aussi des constructions plus complexes telles les boucles et les activités conditionnelles. Il comporte aussi l'instruction **flow** qui permet l'exécution de plusieurs activités en parallèle tout en autorisant des points de synchronisation.

Le langage BPEL résulte des travaux de l'OASIS [7], une organisation rassemblant de grands constructeurs logiciels tels que IBM, Microsoft et SAP. Il existe de nombreux moteurs d'exécution du langage. Chacun d'entre eux couvre tout ou partie

7. Site Web de l'OASIS : www.oasis-open.org/org

de la norme du langage et les implémentations peuvent varier [8] [28, 29]. De plus, ils proposent des extensions telles que l'utilisation de langages tiers (par exemple Java, Javascript) ou encore la génération d'identifiants uniques globaux (Globally Unique IDentifier (GUID) en anglais). Le moteur ODE d'exécution BPEL est implémenté par la fondation Apache. Il est conforme à la version 2.0 du langage BPEL [9]. Il peut exécuter des processus BPEL en mémoire ou encore les sauvegarder dans un système de gestion de base de données avant de les exécuter. Il permet le développement flexible des processus en autorisant, entre autres, le déploiement et la désinstallation des paquetages de processus BPEL à chaud.

1.5.1 Interface d'un processus BPEL

Le langage BPEL repose sur la norme WSDL pour les contrats. Cela signifie qu'un processus déployé est exposé comme un service Web classique. BPEL utilise la notion de *lien de partenaire* pour représenter, à l'intérieur du processus, soit le point d'accès d'un client au processus, soit le client d'un autre service. La figure 1.10 représente l'interface d'un processus BPEL qui s'occupe de vente de biens multimédia. À gauche, `selling` est le lien de partenaire qui représente le client qui utilise les opérations `GetCatalog`, `Order` et `Pay` du processus. Ce lien de partenaire est de type `Library PT` (type de port Library). À droite dans la figure 1.10, le processus BPEL est le client de services Web pour l'expédition (`Shipper`) et pour les opérations d'une banque (`Bank`). Le processus peut ainsi utiliser des opérations disponibles à travers les types de port des liens partenaire ainsi déclarés.

1.5.2 Activités du langage BPEL

Comme le langage BPEL a été créé pour la définition de processus d'affaires, ses constructions sont des activités, plutôt que des instructions. L'activité **`receive`** (respectivement **`reply`** et **`invoke`**) permet de recevoir un message d'un client (respectivement répondre à une requête d'un client et appeler en tant que client un service Web). Chacune de ces activités de communication précise dans ses paramètres quel

8. Comparaison de moteurs BPEL : code.google.com/p/bpel-g/wiki/BPELComparison
9. Version 2.0 de la norme BPEL : docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

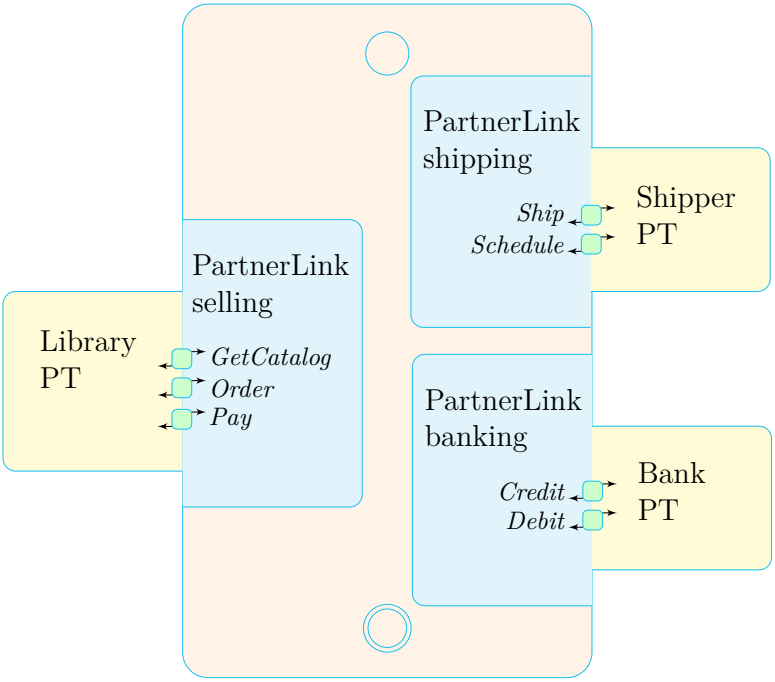


Figure 1.10 – Exemple d'un processus BPEL

est le partenaire client ou fournisseur, quelle opération l'initie et quelle variable est lue ou modifiée. Il existe deux versions plus complexes de l'activité **`receive`** : les activités **`onMessage`** et **`onEvent`**. L'activité **`onMessage`** permet à la fois de recevoir une requête et de choisir une branche de traitement appropriée à la requête reçue. Elle fait partie en réalité de l'activité **`pick`** qui offre la possibilité de choisir parmi plusieurs possibilités une branche de traitement en fonction de la requête reçue. L'activité **`onEvent`** permet de traiter des requêtes en parallèle de l'exécution d'une activité principale. Elle est souvent employée pour obtenir de l'information sur l'état du déroulement d'un traitement en cours ou même demander son annulation. Dans le programme 1.7, l'instruction **`onMessage`** à la ligne 2 (respectivement 11) active l'attente des requêtes `AuthorizationRequest` (respectivement des requêtes `Rollback`). La ligne 2 (respectivement 11) spécifie aussi que les requêtes sont attendues du lien de partenaire `FilterPtrLnk` et, une fois celles-ci reçues, leur contenu est stocké dans la variable `ARIn` (respectivement `RollbackIn`). En cas d'occurrence d'une requête `AuthorizationRequest` (respectivement `Rollback`), le traitement du processus

```
1  <pick name="PickShopIsOpen">
2   <onMessage partnerLink="FilterPtrLnk" operation="AuthorizationRequest"
      portType="fw:FilterPortType" variable="ARIn">
3    <correlations><correlation set="CorrSetPId" initiate="no"/></correlations>
4    <sequence name="SequenceAR">
5     ...
6     <invoke name="InvokeE" partnerLink="ASTDPtrLnk" operation="Execute"
       portType="aw:ASTDPortType" inputVariable="EIn" outputVariable="EOut"/>
7     ...
8     <reply name="ReplyAR" partnerLink="FilterPtrLnk"
       operation="AuthorizationRequest"
       portType="fw:FilterPortType" variable="AROut"/>
9    </sequence>
10  </onMessage>
11  <onMessage partnerLink="FilterPtrLnk" operation="Rollback"
      portType="fw:FilterPortType" variable="RollbackIn">
12   <correlations><correlation set="CorrSetPId" initiate="no"/></correlations>
13   <sequence>
14    ...
15    <reply name="ReplyRollback" partnerLink="FilterPtrLnk" operation="Rollback"
        portType="fw:FilterPortType" variable="RollbackOut"/>
16   </sequence>
17  </onMessage>
18 </pick>
```

Programme 1.7 – Activités de communication BPEL

se poursuit à l'activité **sequence** des lignes 4 à 9 (respectivement des lignes 13 à 16). Dans le programme 1.7, les activités **receive**, **reply** et **invoke** sont illustrées avec des valeurs pour l'attribut **partnerLink**. Par exemple à la ligne 6, l'activité **invoke** appelle à son exécution l'opération `Execute` du service Web accessible par le lien de partenaire `ASTDPtrLnk` avec la requête SOAP `EIn` (attribut **inputVariable**) et la réponse SOAP `EOut` (attribut **outputVariable**).

Le langage BPEL offre des activités complexes de contrôle du fil d'exécution. L'activité **if** permet l'exécution conditionnelle d'une sous-activité. Sa version étendue permet de conditionner l'exécution de plusieurs activités mutuellement exclusives à la *if ... then ... else if ... else ...*. Il est possible de répéter l'exécution d'une activité avec les activités de boucle telles que **while**, **repeatUntil** et **forEach**. Les activités **while** et **repeatUntil** exécutent leur sous-activité en boucle — de façon séquentielle — suivant la valeur d'une expression booléenne. L'activité **forEach** opère de façon différente en permettant l'exécution en parallèle de plusieurs branches de la même sous-activité suivant les valeurs d'un compteur entier.

BPEL emprunte des fonctionnalités à des langages modernes comme Java. La principale est la gestion des exceptions avec les activités **`throw`** et **`rethrow`** pour respectivement déclencher et renvoyer le traitement d'une exception. Puisque le langage peut considérer des processus de longue durée, il offre un mécanisme complexe pour la gestion des erreurs appelé *compensation*. Ce dernier peut être considéré comme un moyen d'annuler l'effet d'activités faisant partie d'un processus qui échoue en cours de traitement. Les activités qui permettent de mettre en œuvre la compensation sont **`compensate`** et **`compensateScope`**. Elles déclenchent le mécanisme de compensation pour des activités qui implémentent un gestionnaire de compensation.

Le langage BPEL est orienté vers la réception et l'envoi de messages et l'expression de structures de contrôle du flot de traitement, plutôt que vers la manipulation des données. Les variables et les messages reçus ou échangés ont des types de base (par exemple entier, chaîne de caractères) ou des éléments XML. L'activité **`assign`** du langage permet la modification des variables. Cependant cette activité est limitée dans ses possibilités. Par exemple, il n'est pas possible d'ajouter un élément XML à un document XML qui est une valeur d'une variable. Concrètement considérons l'exemple d'une commande de produits écrite en XML. L'ajout d'une ligne de produit à cette commande est impossible avec seulement des éléments du langage BPEL. En pratique, la norme BPEL offre la possibilité d'utiliser un processeur XSLT qui effectue la transformation de documents XML en fonction des paramètres d'entrée (document à transformer et feuille de style XSL). Pour utiliser cette fonctionnalité du langage, il faut avoir recours à la fonction `doXslTransform`, comme le montre le programme 1.8 aux lignes 6 à 11. La feuille de style XSL `GetSpecification.xsl` et la variable XML `SpecificationRequest` sont passées en paramètres au processeur XSLT utilisé par le moteur d'exécution du processus. Le résultat de la transformation du document en format XML par la feuille de style est stocké dans la variable `Specification` à la ligne 10. Le programme 1.8 montre aussi à la ligne 3 l'utilisation d'une fonction du langage XPath pour extraire une chaîne de caractères du contenu XML de la variable `InitIn`.

```
<assign>
 <copy>
  <from>substring($InitIn.payload/cx:PId, 1, 4)</from>
  <to variable="SpecificationRequest"/>
 </copy>
 <copy>
  <from>
   bpws:doXslTransform("GetSpecification.xsl", $SpecificationRequest)
  </from>
  <to variable="Specification"/>
 </copy>
 <copy>
  <from>
   <literal>
    <tx:ExecuteRequestType>
     ...
    </tx:ExecuteRequestType>
   </literal>
  </from>
  <to variable="EIn" part="payload"/>
 </copy>
 <copy>
  <from>false()</from>
  <to variable="RepeatAR"/>
 </copy>
</assign>
```

Programme 1.8 – Exemple de l'activité **`assign`**

1.6 Conclusion

L'intégration d'un gestionnaire de contrôle d'accès, comme un PEM de la norme XACML, à des applications de type SOA requiert nécessairement une compréhension des principales technologies couramment utilisées dans la réalisation de services Web. En particulier, le langage BPEL constitue un outil intéressant pour l'implémentation de filtres de contrôle d'accès déployés comme des services dans une architecture SOA. En effet, il semble raisonnable d'exprimer les politiques de contrôle d'accès dans une notation d'expression de processus comme ASTD et de concrétiser de tels processus abstraits par des processus BPEL.

Chapitre 2

Contrôle d'accès dans les systèmes d'information

Le contrôle d'accès, dans un système d'information, est l'ensemble des mesures en place pour restreindre l'accès aux ressources du système suivant des contraintes préétablies. Il existe de nombreux modèles de contrôle d'accès. Une instance d'un tel modèle représente une *politique de contrôle d'accès*. Cette dernière définit donc les accès aux ressources d'un système.

2.1 Politique de contrôle d'accès

Les contraintes qui régissent les accès aux ressources d'un système peuvent être de nature statique ou dynamique. On distingue ainsi deux types de politiques : les politiques de contrôle d'accès statique et les politiques de contrôle d'accès dynamique.

2.1.1 Politique de contrôle d'accès statique

Pour un système d'information donné, une politique de contrôle d'accès statique est caractérisée par le fait que son état ne change pas par rapport à l'évolution dynamique du système, car elle comporte seulement des contraintes statiques. Celle-ci peut être mise à jour pour refléter divers changements dans l'organisation (par exemple,

un changement d'affectation dans une organisation qui entraîne une augmentation des privilèges d'un utilisateur). L'initiation d'une action par un utilisateur déclenche l'évaluation de l'état de la politique. En fonction des autorisations accordées par la politique dans son état courant, l'exécution de l'action est permise ou pas. Dans la plupart des implémentations, une politique de contrôle d'accès statique associe les utilisateurs du système à leurs privilèges.

2.1.2 Politique de contrôle d'accès dynamique

Pour un système d'information donné, une politique de contrôle d'accès dynamique possède plusieurs états, car elle est associée à l'évolution du système. L'autorisation de l'exécution d'une action est basée sur l'évaluation de l'état courant du système et sur la définition même de la politique. L'état courant est mis à jour lors de chaque exécution d'une action contrôlée. La version élémentaire de ce type de politiques de contrôle d'accès utilise un historique des actions exécutées par le système d'information qui est mise à jour par le gestionnaire de mise en œuvre de la politique. Les langages formels qui supportent les traces d'événements, comme les langages basés sur une algèbre de processus, se prêtent bien à l'expression de contraintes dynamiques.

2.2 Formalisation du contrôle d'accès

Le contrôle d'accès dans un système d'information permet de contraindre l'accès à ses ressources. Une façon d'interpréter une politique est de considérer un système d'information composé d'un ensemble d'états Q et d'un ensemble de transitions entre ces états, où chaque transition représente un accès à ses ressources. Ainsi une politique de contrôle d'accès partitionne l'ensemble des états Q en un ensemble d'états autorisés $Q_{aut} \subseteq Q$ et un ensemble d'états interdits. La mise en œuvre des politiques de contrôle d'accès est réalisé par un *mécanisme de sécurité* qui empêche le système de se retrouver dans un état interdit. Désignons Q_{acc} l'ensemble des états accessibles par le mécanisme de sécurité. Le mécanisme est dit *sûr* lorsque $Q_{acc} \subseteq Q_{aut}$. Dans ce cas, les états accessibles par le mécanisme de sécurité sont tous des états autorisés par la politique de contrôle d'accès. Lorsque $Q_{acc} = Q_{aut}$, le mécanisme de sécurité

```
1  mon-texte.txt : {(Alain,rwxo); (Cédric,rx)}
2  ton-archive.tar.gz : {(Bob,rw); (Daniel,rwo)}
```

Programme 2.1 – Exemple d'ACL pour des fichiers

est dit *précis* et tous les états autorisés par la politique sont accessibles par le mécanisme de sécurité et vice-versa. Dans le cas général où $Q_{acc} \cap Q_{aut} \neq \varnothing$, le mécanisme de sécurité est dit *large*, car le système peut se retrouver dans un état interdit. Les modèles présentés dans la suite sont des moyens qui précisent comment les différents accès au système sont réalisés de telle sorte que son état courant soit toujours dans l'ensemble Q_{aut}.

2.3 Listes de contrôle d'accès

Les *listes de contrôle d'accès* [26] (Access Control List (ACL) en anglais) sont un moyen d'exprimer une politique de contrôle d'accès. Dans cette approche les privilèges des objets actifs, c'est-à-dire des utilisateurs, sont listés de façon décentralisée pour chaque objet du système. Lors d'une demande d'accès pour effectuer une opération, le mécanisme de contrôle d'accès vérifie que l'objet actif qui initie l'opération possède le privilège de la faire sur l'objet cible à travers l'ACL de cet objet. Le programme 2.1 montre deux fichiers et les entrées de leur liste de contrôle d'accès. Dans cet exemple, l'utilisateur `Alain` possède tous les droits (lecture `r`, écriture `w`, exécution `x` et propriété `o`) sur le fichier `mon-texte.txt` et l'utilisateur `Bob` peut lire et modifier le fichier `ton-archive.tar.gz`.

L'implémentation la plus répandue des ACL se retrouve dans les systèmes POSIX [1] et Linux. Dans ces systèmes, la technique des ACL vient compléter le système de gestion des droits existant. Les ACL deviennent indispensables lorsque l'on souhaite définir de façon fine l'accès à un objet (fichier ou répertoire). Par exemple, étant donné un fichier `file` qui appartient à un groupe `group` qui possède uniquement le droit de lecture sur celui-ci, est-il possible d'attribuer le droit de modification à un utilisateur spécifique qui fait partie du groupe `group` ? On ne peut pas modifier les droits d'accès

1. Norme POSIX : standards.ieee.org/findstds/standard/1003.1-2008.html

du groupe au fichier car les privilèges de tous les autres utilisateurs du groupe vis-à-vis du fichier seraient alors affectés. Créer un groupe avec uniquement l'utilisateur visé par le droit de modification, définir ce groupe comme le groupe propriétaire du fichier et lui attribuer le droit du fichier visé ne constitue pas une solution acceptable. Grâce à l'ACL du fichier, il suffit d'attribuer le droit visé à l'utilisateur cible.

Pour manipuler l'ACL d'un fichier, la commande `setfacl` est utilisée, tandis que la lecture est effectuée avec la commande `getfacl`. La commande `setfacl` prend en paramètre :

- le type d'opération, c'est-à-dire l'ajout ou le retrait à l'ACL ;
- le type d'objet actif visé par la modification, c'est-à-dire un utilisateur, un groupe ou le masque de l'ACL ;
- l'objet actif, c'est-à-dire l'utilisateur ou le groupe ;
- les droits d'accès accordés à l'objet actif, c'est-à-dire l'autorisation ou l'interdiction de lire, modifier ou exécuter ;
- l'objet dont l'ACL est modifiée.

Le masque de l'ACL (du point 2 précédent) indique les droits de base qui peuvent être attribués à toutes les entrées spécifiées par le système ACL. Par exemple, si les droits de lecture et d'écriture sont spécifiés comme masque et que les droits de lecture et d'exécution sont spécifiés pour un utilisateur dans l'ACL d'un fichier, alors cet utilisateur possède les *droits effectifs* de lecture sur le fichier en question car le droit d'exécution est exclu par le masque et le droit de modification pour l'utilisateur. La combinaison de la gestion classique des droits du système de fichiers et des ACL est très flexible et permet d'accommoder les besoins les plus importants pour les fichiers et dossiers.

2.4 Matrices de contrôle d'accès

Cette méthode de gestion du contrôle d'accès est réalisée à l'aide d'une matrice qui lie les objets accédés — fonctions ou données — aux objets actifs qui y accèdent. Elle a été développée par Lampson [36] et les mécanismes de protection ont été précisés

Tableau 2.1 – Exemple d'une matrice de contrôle d'accès

	fichier-1	fichier-2	fichier-3	processus-1	processus-2	peripherique-1	peripherique-2
utilisateur-1	r*	r, w, o	r, w, o	o		r	w*
utilisateur-2	r, w, o	r, w*	r, w*	r, w		r	w
utilisateur-3	r	r, w	r, w	r, w		r	
processus-1	r	r, w	r	c	o	r	w
processus-2		r, w			c	r	

par Graham et Denning [27]. Avec cette méthode, une politique de contrôle d'accès est définie par une *matrice de contrôle d'accès*. Désignons par O l'ensemble des objets du système dont l'accès doit être contrôlé, par R les droits d'accès considérés et par $S \subset O$ les objets actifs, c'est-à-dire ceux qui accèdent aux objets. La matrice de contrôle d'accès est définie par :

$$A = (r_{so})_{s \in S,\ o \in O}, \text{où } r_{so} \subseteq R.$$

Dans un système de fichiers où les droits sont

$$R = \{lire(r), écrire(w), propriétaire(o), contrôle(c)\},$$

le tableau 2.1 représente une matrice de contrôle d'accès. Dans cette matrice, l'utilisateur `utilisateur-1` est propriétaire des fichiers `fichier-2` et `fichier-3`, et il est aussi le propriétaire du processus `processus-1` qu'il a démarré. Il possède aussi le droit de lecture sur le fichier `fichier-1` avec le témoin (*) qui signifie qu'il peut transmettre ce droit à un autre sujet du système. Cet utilisateur a aussi les droits d'utiliser les périphériques `peripherique-1` et `peripherique-2`. La matrice montre également que le processus `processus-1` a démarré le processus `processus-2`.

Tout comme l'utilisateur `utilisateur-1`, le processus `processus-1` peut utiliser les périphériques. Il ne peut cependant pas transmettre le droit d'écriture sur le périphérique `peripherique-2`.

La méthode requiert un *moniteur* pour chaque type d'objets à protéger du système. Chaque opération sur un objet du système doit être approuvée par le moniteur associé au type de l'objet cible avant d'être exécutée. Le mécanisme de protection, évident, est le suivant :

1. un objet actif s initie l'opération r sur l'objet o ;
2. la requête est transmise au moniteur associé au type de l'objet o ;
3. le moniteur vérifie dans la matrice A si $r \in r_{so}$, et si le résultat est positif, l'opération peut s'exécuter (dans le cas contraire elle n'est pas autorisée).

Pour compléter le mécanisme de protection, celui-ci possède aussi un moniteur spécial pour la matrice elle-même ainsi qu'un ensemble d'opérations pour manipuler celle-ci. D'autres opérations qui ont un effet indirect sur la matrice, comme par exemple la création d'objets ou de sujets, sont prises en compte par le mécanisme de protection. Toutes ces opérations requièrent des droits spécifiques pour être exécutées et causent une mise à jour de la matrice lors de leur exécution. Les plus importantes de ces opérations, vis-à-vis des objets y compris les sujets, sont les opérations de création et de suppression de ceux-ci. La création ne requiert aucun privilège et attribue les droits appropriés au sujet qui crée : le droit `proprietaire` pour la création d'un objet et les droits `controle` et `proprietaire` pour un sujet. La suppression requiert le droit `proprietaire` sur l'objet supprimé. Après la suppression d'un objet (respectivement d'un sujet), la colonne (respectivement la ligne) correspondante est retirée de la matrice. Les autres opérations du mécanisme de protection sont les opérations de transfert, attribution ou suppression d'un droit, ou lecture des droits d'un objet ou d'un sujet. Le moniteur spécial utilise les règles du tableau 2.2 pour autoriser les opérations mentionnées.

Tableau 2.2 – Opérations de modification de matrices de contrôle d'accès [27]

Règle	Opération (par s_0)	Droit	Exécution
Règle 1	`transferer` $\left\{ \begin{matrix} \alpha^* \\ \alpha \end{matrix} \right\}$ pour o à s	$\alpha^* \in A[s_0, o]$	mettre $\left\{ \begin{matrix} \alpha^* \\ \alpha \end{matrix} \right\}$ dans $A[s, o]$
Règle 2	`accorder` $\left\{ \begin{matrix} \alpha^* \\ \alpha \end{matrix} \right\}$ pour o à s	`proprietaire` $\in A[s_0, o]$	mettre $\left\{ \begin{matrix} \alpha^* \\ \alpha \end{matrix} \right\}$ dans $A[s, o]$
Règle 3	`supprimer` α pour o à s	`controle` $\in A[s_0, s]$ ou `proprietaire` $\in A[s_0, o]$	supprimer α de $A[s, o]$
Règle 4	`lire` les droits de s pour o	`controle` $\in A[s_0, s]$ ou `proprietaire` $\in A[s_0, o]$	lire $A[s, o]$
Règle 5	`creer` objet o	—	ajouter colonne o dans A ; placer `proprietaire` dans $A[s_0, o]$
Règle 6	`supprimer` objet o	`proprietaire` $\in A[s_0, o]$	supprimer colonne o dans A
Règle 7	`creer` sujet s	—	ajouter ligne pour s dans A ; exécuter `creer` objet s ; placer `controle` dans $A[s, s]$
Règle 8	`supprimer` sujet s	`proprietaire` $\in A[s_0, s]$	supprimer ligne s dans A ; exécuter `supprimer` objet s

2.5 Modèle de Bell-LaPadula

Ce modèle de contrôle d'accès a été développé pour le département de la défense américain dans les années 70 par Bell et LaPadula [4]. Il s'inspire de la vue militaire de la sécurité qui veut que les utilisateurs d'un système n'ont strictement accès qu'aux informations dont ils ont besoin — « on a need to know basis ». Pour ce faire, ce modèle utilise à la fois le contrôle d'accès obligatoire (Mandatory Access Control (MAC) en anglais) [12] et le contrôle d'accès discrétionnaire (Discretionary Access Control (DAC) en anglais) [12]. Du point de vue DAC, les sujets peuvent définir les

droits des autres sujets pour les objets dont ils sont propriétaires. Du point de vue MAC, la politique de contrôle d'accès attribue des *niveaux de sécurité* aux objets, objets inactifs et sujets confondus. Chaque niveau de sécurité est une combinaison entre un degré d'habilitation des sujets et un ensemble de catégories des objets.

Le degré d'habilitation d'un sujet correspond au niveau de *confidentialité* auquel le sujet est autorisé. Des exemples d'habilitation sont : *très secret*, *secret* et *public*. L'ensemble des degrés d'habilitation est un ensemble non vide et totalement ordonné. La relation d'ordre total sur cet ensemble signifie qu'un sujet d'un niveau d'habilitation plus grand a potentiellement accès à toutes les informations d'un niveau d'habilitation plus bas. Les niveaux de sécurité comportent chacun un ensemble de catégories. Les ensembles {SCRS, CIA, FBI}, {CIA, FBI} et {SCRS} constituent des exemples pour des ensembles de catégories. L'ensemble des ensembles des catégories, muni de la relation d'inclusion, est un ensemble partiellement ordonné.

Notons $S = \{(h, c) \mid h \in H \wedge c \in C\}$ l'ensemble des niveaux de sécurité, où H est l'ensemble des niveaux d'habilitation et C est l'ensemble des ensembles de catégories. Notons aussi par « $\preceq$ » la relation d'ordre total sur l'ensemble des habilitations H. Sur l'ensemble S on définit la relation d'ordre *dom* (domine) par :

$$(h, c)\ dom\ (h', c') \Leftrightarrow h' \preceq h \wedge c' \subseteq c$$

où $(h, c), (h', c') \in S$. La relation ainsi définie est une relation d'ordre partiel et (S, dom) est un treillis, c'est-à-dire que chaque paire d'éléments de S admet une borne inférieure et une borne supérieure. Soit un sujet s de niveau de sécurité (h, c) et un objet o de niveau de sécurité (h', c'). Le sujet s a accès à l'objet o (pour les attributs DAC qui lui sont attribués) si et seulement si $(h, c)\ dom\ (h', c')$. Quand cette relation est vraie, l'aspect MAC de la politique définie autorise l'accès à l'objet o par le sujet s. L'opération désirée par le sujet s n'est exécutée que si les privilèges définis par le propriétaire de l'objet o (aspect DAC du modèle) le permettent.

Considérons les objets du tableau 2.3. Le sujet Alain a l'habilitation *secret* par son niveau de sécurité. Il a accès au fichier `fichier-1` car celui-ci requiert un niveau

Tableau 2.3 – Niveaux de sécurité pour des objets

Objet	Type	Niveau de sécurité
`Alain`	sujet	(*secret*, {SCRS, FBI})
`Bob`	sujet	(*très secret*, {CIA, FBI})
`fichier-1`	objet	(*secret*, {FBI})
`fichier-2`	objet	(*public*, {CIA, FBI})
`fichier-3`	objet	(*très secret*, {CIA})

d'habilitation égal à celui du sujet et de plus {FBI}⊆{SCRS, FBI}, c'est-à-dire que les catégories de l'objet sont autorisées pour le sujet. On remarque aussi que Alain n'est pas autorisé à accéder au fichier `fichier-2` car bien qu'il dispose de l'habilitation nécessaire, son niveau de sécurité ne lui autorise pas les objets de la catégorie CIA. Alain n'a pas accès au fichier `fichier-3`, mais dans ce cas il ne dispose tout simplement pas de l'habilitation nécessaire (*très secret*).

Pour compléter le modèle, deux règles importantes sont ajoutées :

No read up — un sujet ne peut *lire* un objet que si son niveau de sécurité domine celui de l'objet et qu'il dispose du droit de lecture sur cet objet ;

No write down — un sujet ne peut *écrire* sur un objet que si le niveau de sécurité de l'objet domine celui du sujet et que le sujet a le droit d'écriture sur l'objet.

Le modèle, utilisé avec ces deux règles, implique qu'un sujet ne peut créer des documents qu'à un niveau de sécurité supérieur ou égal au sien et, ne peut lire des documents qu'à un niveau de sécurité inférieur au sien. Bell et LaPadula ont prouvé qu'avec un tel modèle, un système sécurisé dès son initialisation le reste, tant que les règles *no read up* et *no write down* sont respectées.

2.6 Contrôle d'accès basé sur les rôles

Ce type de contrôle d'accès est utilisé dans la plupart des SGBD modernes. Les *droits* sont attribués à des *rôles* plutôt qu'aux sujets directement. À ces derniers, l'on attribue un ou plusieurs rôles et les droits d'un sujet sont les droits combinés des

rôles qui lui sont associés. Dans la gestion courante du contrôle d'accès, l'attribution des droits aux rôles est faite suivant les responsabilités d'une fonction au sein d'une organisation. Cette attribution des droits apporte une certaine flexibilité puisque les rôles, bien qu'ils ne soient pas figés, évoluent très peu. Ce modèle a été normalisé par le « National Institute of Standards and Technology » (NIST) [32].

La méthode repose sur la définition d'un ensemble de relations qui lient, d'une part, les sujets aux rôles et, d'autre part, les rôles aux permissions. Une permission dans ce cas signifie l'autorisation d'exécuter une fonction ou encore l'accès (lecture, modification) à une donnée. Soit S, R et P respectivement l'ensemble des sujets, rôles et permissions. On note par $a\_le\_rôle$ (respectivement $a\_la\_permission$) la relation entre les sujets et les rôles (respectivement entre les rôles et les permissions). À partir de ces ensembles et relations, on définit les fonctions $rôle\_actif$, $rôles\_autorisés$ et $a\_le\_droit\_dexécuter$:

- $rôle\_actif : S \to R$ qui retourne le rôle actif d'un utilisateur à un instant donné de l'exécution du système ;
- $rôles\_autorisés : S \to \wp(R)$ qui retourne l'ensemble de rôles attribués à un sujet, elle peut être définie formellement à partir de la relation $a\_le\_rôle$

$$rôles\_autorisés(s) = \{r \in R \mid a\_le\_rôle(s, r)\};$$

- $a\_le\_droit\_dexécuter : S \times P \to \{vrai, faux\}$ qui indique qu'un sujet a une permission à un instant donné de l'exécution du système.

Cette dernière peut s'exprimer aisément en fonction des autres :

$$\begin{aligned} \forall s \in S\ \forall p \in P, \quad & a\_le\_droit\_dexécuter(s, p) = vrai \\ & \Leftrightarrow \\ & a\_la\_permission(rôle\_actif(s), p). \end{aligned}$$

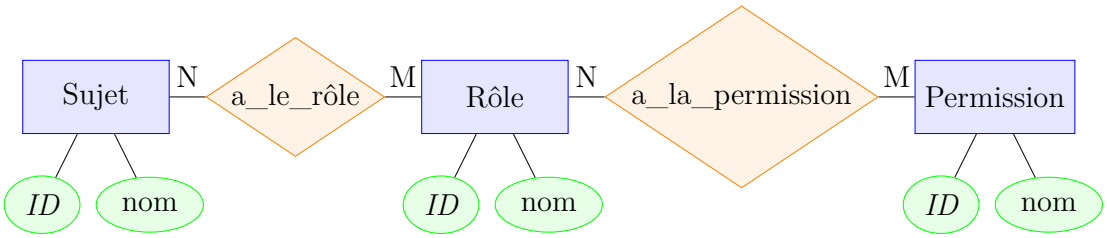


Figure 2.1 – Diagramme entité-relation d'un modèle à la RBAC

Dans un système d'information classique, les informations de contrôle d'accès sont stockées dans une base de données. Les permissions sont les droits d'exécuter les fonctions du système d'information et constituent les éléments de l'ensemble P. La figure 2.1 illustre le modèle de contrôle d'accès pour une telle application. Un échantillon des données contenues dans ces tables sont présentées dans le tableau 2.4. L'utilisateur `Alain` a le rôle `Etudiant` et peut donc exécuter l'action `Emprunter`, car le sujet `Etudiant` a la permission `Emprunter`. Les utilisateurs `Bob` et `Cedric` sont autorisés à exécuter les actions `Acheter` et `Retourner` car ils ont le rôle `Bibliothecaire`.

La méthode permet d'avoir une hiérarchie de rôles. Cette hiérarchie peut être définie comme une relation d'ordre partielle sur l'ensemble des rôles. Le sens donné à cette relation d'ordre est qu'un rôle r *supérieur à* un rôle r' contient tous les droits de ce dernier. La hiérarchisation permet à un utilisateur, lorsqu'un rôle est activé, de posséder en plus des droits du rôle même, les droits de tous les rôles inférieurs dans

Tableau 2.4 – Données d'une politique de contrôle d'accès RBAC

Sujet		*a_le_rôle*		Rôle		*a_la_permission*		Permission	
ID	Nom	Sujet	Rôle	ID	Nom	Rôle	Permission	ID	Nom
S1	`Alain`	*S1*	*R1*	*R1*	`Etudiant`	*R1*	*P2*	*P1*	`Acheter`
S2	`Bob`	*S4*	*R1*	*R2*	`Bibliothecaire`	*R2*	*P1*	*P2*	`Emprunter`
S3	`Cedric`	*S2*	*R2*			*R2*	*P3*	*P3*	`Retourner`
S4	`Daniel`	*S3*	*R2*						

la hiérarchie des rôles. La norme RBAC offre la possibilité de définir des contraintes pour compléter le modèle. Ainsi des contraintes de séparations des devoirs statiques ou dynamiques peuvent être spécifiées. Une séparation des devoirs statique stipule qu'un utilisateur ne peut se voir attribuer des rôles particuliers en même temps. De façon formelle les séparations des devoirs statiques d'une politique RBAC forment un ensemble $SSD \subseteq \wp(R) \times (\mathbb{N} - \{0, 1\})$. La politique de contrôle d'accès est telle que :

$$\forall\ (rs, n) \in\ SSD,\ \forall\ t\ \subseteq\ rs,\ |\,t\,| \geq\ n \implies \bigcap_{r \in t} \{s \in\ S \mid a\_le\_rôle(s, r)\} = \varnothing.$$

Cette relation signifie que si $(rs, n) \in SSD$, alors n rôles ou plus de l'ensemble rs ne peuvent être attribués simultanément à un utilisateur. Les séparations des devoirs dynamiques sont définies de façon similaire mais limitent les rôles qu'un utilisateur peut activer simultanément lors d'une session. La norme RBAC fait une distinction entre les permissions en séparant les notions d'objet et de fonction ou action accédant à ces objets.

2.7 La méthode EB³SEC

La méthode « Entity-Based Black-Box » (EB³) [24] est une méthode formelle de développement de systèmes d'information. Tout comme le langage ASTD, elle associe au diagramme de classes d'un système d'information une expression de processus qui décrit le comportement acceptable du système. Dans les faits, la méthode EB³ est précurseur de la méthode ASTD. Elle a été créée comme langage pivot du cadre d'applications formel « Automated Production of Information Systems » [22] pour le développement automatique des systèmes d'information.

Konopacki [34] a proposé une extension du langage EB³ (le langage EB³ est l'algèbre de processus de la méthode EB³) nommée EB³SEC. Tout comme EB³, EB³SEC est à la fois une méthode et un langage pour la spécification de politiques de contrôle d'accès. La méthode combine une politique de contrôle d'accès statique et une politique de contrôle d'accès dynamique pour établir une politique générale de contrôle d'accès. La politique de contrôle d'accès statique est basée sur la norme RBAC. Elle

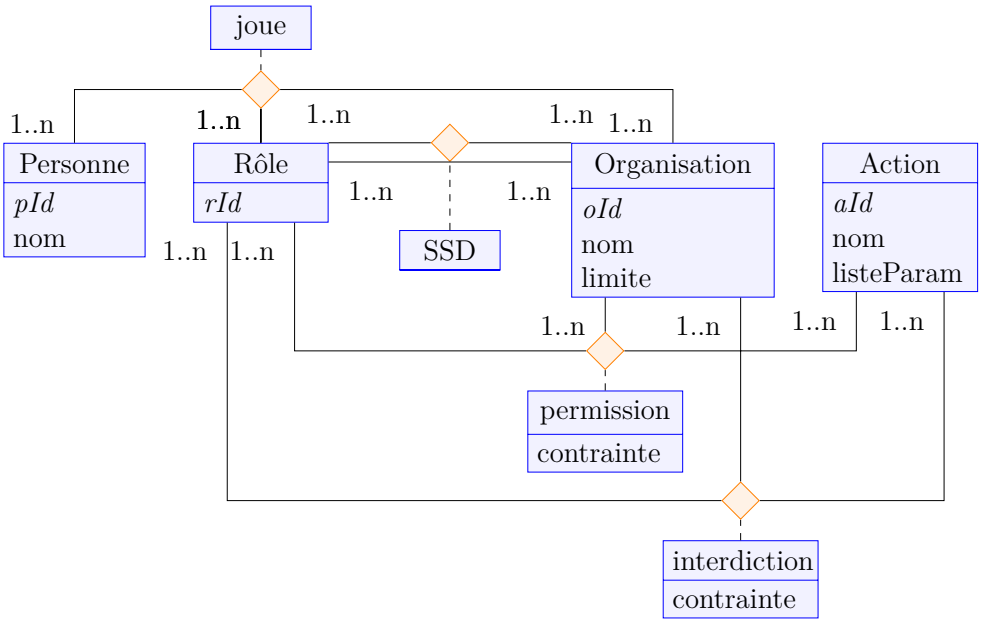


Figure 2.2 – Diagramme de classes d'un modèle EB³SEC [34]

permet de définir à la fois des permissions et des interdictions, avec d'éventuelles contraintes. De plus, l'assignation des actions aux permissions ou aux interdictions met en relation, non seulement des rôles et des actions, mais aussi des *organisations* (voir la figure 2.2). Cette nouvelle entité est introduite dans la politique de contrôle d'accès pour contraindre l'accès des utilisateurs en fonction du lieu où la session est initiée. Dans la figure 2.2, les entités *Personne*, *Rôle* et *Organisation* sont liées par la relation *joue*. Une personne peut ainsi jouer différents rôles dans différentes organisations. Par exemple, un employé d'une entreprise dans une branche de celle-ci peut être client dans une autre branche. Les *permissions* et les *interdictions* sont définies comme des associations des entités *Rôle*, *Organisation* et *Action*. Ainsi un rôle peut détenir le droit d'exécuter une action dans une organisation mais pas dans une autre. La méthode EB³SEC précise aussi le prédicat utilisé par le mécanisme de mise en œuvre de la politique pour autoriser un accès aux ressources.

La partie dynamique de la politique de contrôle d'accès est basée sur l'algèbre de processus du langage EB³. Pour spécifier le comportement attendu, EB³SEC utilise

des *événements sécurisés*. Ceux-ci sont définis à partir des événements du langage EB^3 et des *paramètres de contrôle d'accès*. Ces derniers se définissent comme les entités du système d'information sur lesquelles s'expriment les contraintes du contrôle d'accès. Par exemple

$\langle$ `Bob`, `Bibliothecaire`, `Sherbrooke`, `Emprunter`(Alain, L'art de la guerre)$\rangle$

est un événement sécurisé : l'utilisateur `Bob` assumant le rôle de `Bibliothecaire` à `Sherbrooke` veut exécuter l'action `Emprunter` avec les paramètres « Alain » et « L'art de la guerre ». La spécification d'une politique de contrôle d'accès dynamique utilise des événements sécurisés et les opérateurs de l'algèbre de processus EB^3 pour exprimer le comportement attendu du système.

Le mécanisme de contrôle d'accès effectue une double autorisation des demandes d'accès aux ressources du système d'information. La première autorisation repose sur le prédicat statique exprimé sur la partie statique de le politique de contrôle d'accès. Ce prédicat est *sp* défini par :

$$
\begin{aligned}
sp(\langle p, r, o, e\rangle) \triangleq\ & \langle p, r, o\rangle \in joue \\
\wedge\ & (\langle r, o, idEvt(e)\rangle \in permission && (2.1)\\
& \wedge permission(r, o, idEvt(e)).contrainte) && (2.2)\\
\wedge\ & ((interdiction(r, o, idEvt(e)).contrainte && (2.3)\\
& \Longrightarrow \langle r, o, idEvt(e)\rangle \notin interdiction).) && (2.4)
\end{aligned}
$$

Pour que le mécanisme accepte l'exécution de l'événement e demandé par l'utilisateur p assumant le rôle r dans l'organisation o, il faut que le rôle r soit assigné à l'utilisateur p dans l'organisation o, que l'action $idEvt(e)$ soit assignée au rôle r dans l'organisation o (ligne 2.1) et que la contrainte imposée sur la permission s'évalue à la valeur vrai (ligne 2.2), et que si une interdiction pour l'action $idEvt(e)$ existe pour le rôle r dans l'organisation o alors la contrainte sur cette interdiction s'évalue à la valeur faux (lignes 2.3 et 2.4). La seconde autorisation repose sur l'acceptation, par l'expression de processus qui définit la partie dynamique de la politique de contrôle d'accès, de l'événement sécurisé correspondant à la demande d'exécution de l'utilisa-

teur. Cette exécution n'est autorisée que si ces deux niveaux d'autorisation s'évaluent positivement.

Dans cette thèse, la méthodologie décrite par Konopacki est reprise. Notre démarche cependant utilise le langage ASTD plutôt qu'une algèbre de processus pour la partie dynamique de la politique de contrôle d'accès. Ce langage présente l'avantage, contrairement à l'algèbre de processus EB³, de posséder une notation graphique. Cette caractéristique en fait un langage mieux adapté à la conception de politiques de contrôle d'accès. En termes de pouvoir d'expressivité, le langage ASTD est plus puissant que l'algèbre de processus EB³, avec notamment des éléments syntaxiques tels que l'état élémentaire *historique* dans une structure ASTD *automate* qui n'a pas d'équivalent en EB³. La section suivante présente des guides recommandés pour la spécification de politiques de contrôle d'accès dynamique. Ces guides sont inspirés aussi des travaux de Konopacki [34].

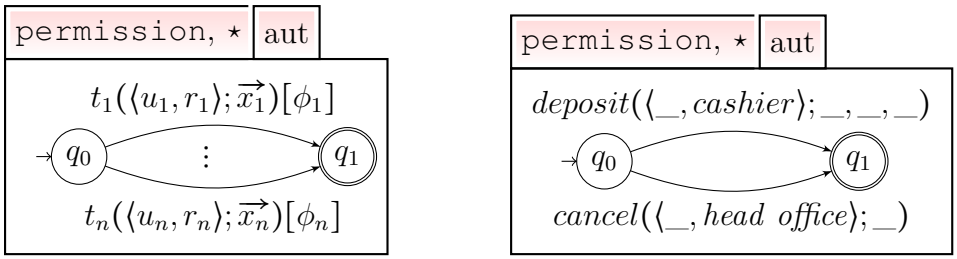
Une politique de contrôle d'accès exprimée avec le langage ASTD présente deux principaux avantages par rapport aux méthodes précédemment proposées. Milhau et al. [38] ont proposé une traduction de modèles ASTD vers des spécifications Event-B [1]. Cette traduction ouvre la voie à la vérification de propriétés et la validation des spécifications avant toute phase d'implémentation par l'utilisation d'outils appropriés [2, 5]. Avec les ACL, les matrices de contrôle d'accès, Bell-LaPadula et RBAC, il n'est pas possible de spécifier des contraintes qui portent sur l'ordonnancement des actions ou événements du système. De même, les propriétés sur l'ordre d'exécution par le système des événements ne peuvent pas être vérifiées. Cette limitation est inhérente à ces modèles. En effet, un autre inconvénient de ces solutions par rapport à celle proposée dans cette thèse est la faiblesse du pouvoir expressif. Les ACL, les matrices de contrôle d'accès et le modèle de Bell-LaPadula ne permettent pas d'exprimer des contraintes sur l'accès en fonction, par exemple, de la ressource accédée ou de l'environnement. Ces méthodes supportent encore moins l'expression de contraintes sur l'ordonnancement des requêtes d'accès aux ressources précédemment reçues. Or c'est justement cette faille que vient combler un langage comme ASTD qui utilise des opérateurs d'une algèbre de processus.

2.8 Patrons de contrôle d'accès

Dans le cadre du travail de cette thèse, le langage ASTD est utilisé non pas pour formaliser le comportement d'un système d'information mais pour modéliser des politiques de contrôle d'accès. Dans ce cas, une spécification précise le comportement autorisé pour les utilisateurs du système suivant les privilèges de ceux-ci. Pour y arriver toutes les transitions de la spécification sont pourvues, en plus de leurs paramètres métiers, de paramètres de contrôle d'accès. Ces paramètres peuvent inclure l'utilisateur qui initie l'action, son rôle dans le système ou d'autres attributs de celui-ci. Les paramètres de contrôle d'accès sont utilisés pour exprimer des contraintes limitant l'accès des utilisateurs, par exemple dans les prédicats associés aux transitions. Les patrons sont un moyen de catégoriser les règles de contrôle d'accès les plus fréquemment utilisées. Il faut noter que le langage ASTD permet, de façon libre, l'expression de contraintes plus générales. Dans les exemples qui suivent, on considère un système d'information simplifié d'une banque qui comporte les actions : `deposit` (dépôt d'un chèque), `validate` (validation d'un dépôt de chèque), `cancel` (annulation d'un dépôt de chèque), `register` (enregistrement d'un client avec un caissier). Les paramètres de contrôle d'accès considérés sont l'identifiant de l'utilisateur ainsi que le rôle de celui-ci. Les événements sécurisés sont donc notés $t(\langle u, r\rangle, \overrightarrow{x})[\phi]$, où t est l'événement à exécuter, u et r respectivement l'utilisateur qui initie l'événement et son rôle actif, $\overrightarrow{x}$ les paramètres métiers de l'événement t et ϕ une garde optionnelle de cet événement.

Une *permission* permet d'autoriser l'exécution d'une action. La forme d'une spécification ASTD pour une permission est présentée dans la figure 2.3a. L'exécution d'une transition t_i $(1 \leq i \leq n)$ est permise si le prédicat ϕ_i s'évalue à la valeur vrai. La figure 2.3b illustre une instance de ce patron. Dans ce cas, l'action `deposit` est autorisée pour les caissiers (`cashier`) et l'action `cancel` pour les chefs d'agence (`head office`). Dans cette règle, la valeur spéciale « _ » est utilisée pour signifier que n'importe quelle valeur est acceptée pour le paramètre correspondant.

Une *interdiction* empêche l'exécution d'une action. Le patron ASTD pour cette catégorie de règles utilise une structure ASTD *garde*. Cette garde comporte un prédi-

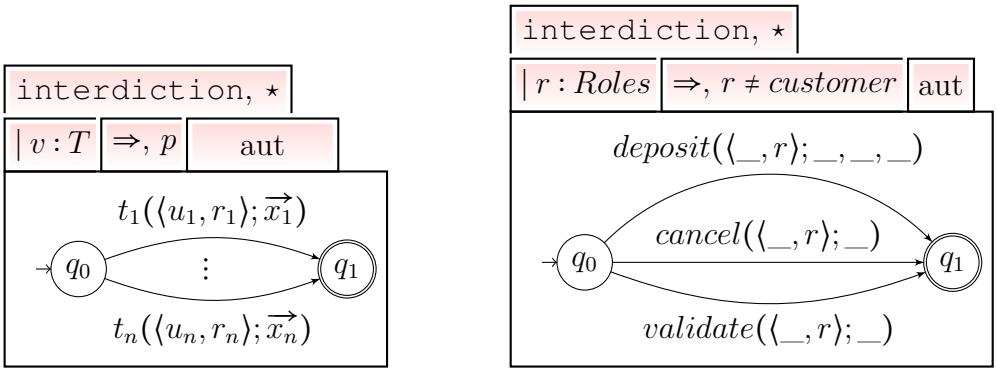


(a) Patron pour la permission (b) Instance du patron permission

Figure 2.3 – Patron ASTD pour la permission

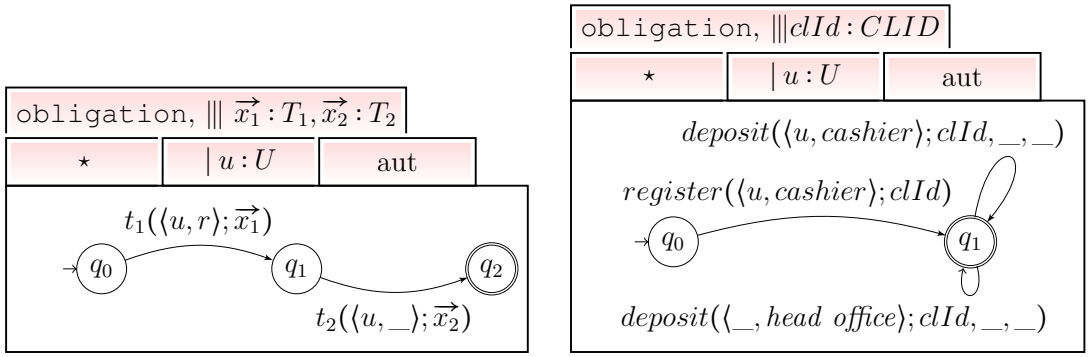
cat qui permet de spécifier la prohibition sous une certaine condition portant sur les paramètres de contrôle d'accès ou métiers. La figure 2.4a illustre le patron de l'interdiction. L'action t_i $(1 \leq i \leq n)$ n'est exécutée que si le prédicat p s'évalue à la valeur *vrai*. La variable v peut être n'importe laquelle des variables apparaissant dans les transitions ou dans le prédicat p. Une instance du patron interdiction est représenté à la figure 2.4b. Elle spécifie la règle de contrôle d'accès : les actions `deposit`, `cancel` et `validate` sont interdites pour les utilisateurs ayant le rôle client (`customer`).

Les *obligations* constituent une autre catégorie de règle de contrôle d'accès. Une obligation représente la contrainte de l'exécution d'une action — possiblement non immédiatement — après l'exécution d'une première action. Les deux actions sont



(a) Patron pour l'interdiction (b) Instance du patron interdiction

Figure 2.4 – Patron ASTD pour l'interdiction

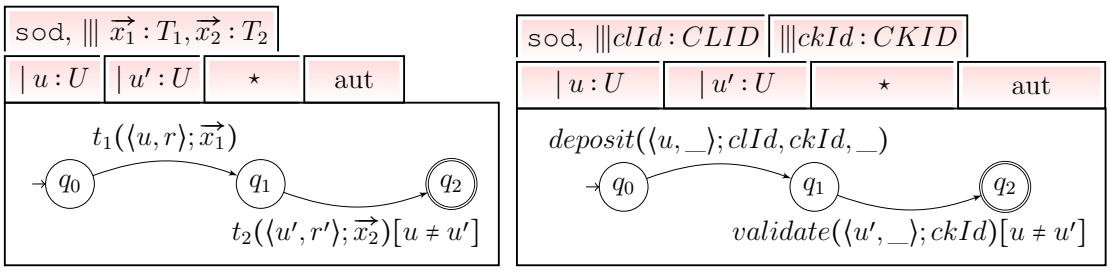


(a) Patron pour l'obligation (b) Instance du patron obligation

Figure 2.5 – Patron ASTD pour l'obligation

souvent liées par les valeurs de leurs paramètres. La figure 2.5a montre le patron ASTD pour les obligations. L'action t_2 doit être exécutée après l'action t_1 pour le même utilisateur u. Les paramètres métiers respectifs $\overrightarrow{x_1}$ et $\overrightarrow{x_2}$ des événements t_1 et t_2 sont généralement liés par une relation implicite ou explicite. Dans l'exemple de la banque, pour des raisons de discrétion un client doit être enregistré avec un caissier et seul ce caissier ou un chef d'agence peut effectuer des dépôts de chèque pour ce client. Cette règle est spécifiée par la structure ASTD de la figure 2.5b. Dans cette figure, l'obligation apparaît par l'utilisation des paramètres u pour l'utilisateur et $clId$ pour l'identifiant du client. La relation entre les paramètres métiers des événements `register` et `deposit` est explicite : c'est l'égalité sur l'identifiant du client de la banque concerné par les événements.

La dernière catégorie de règles est la *séparation des devoirs* (SoD). Elle est couramment utilisée pour attribuer à des utilisateurs distincts la responsabilité de l'exécution des parties d'un processus vulnérable pour le système d'information. La figure 2.6a illustre le patron ASTD pour les SoD d'un processus composé des actions t_1 et t_2. Le prédicat $u \neq u'$ associé à l'action t_2 impose l'exécution de celle-ci par un utilisateur u' différent de l'utilisateur u qui exécute l'action t_1. Le patron ici porte sur les utilisateurs, mais le même principe peut être appliqué sur un autre paramètre de contrôle d'accès. Par exemple, l'exécution des différentes parties du processus vulnérable peut être confiée à des rôles distincts. Comme dans le cas des obligations, les paramètres



(a) Patron pour SoD (b) Instance du patron SoD

Figure 2.6 – Patron ASTD pour la séparation des devoirs

métiers $\overrightarrow{x_1}$ et $\overrightarrow{x_2}$ sont généralement liés de manière implicite ou explicite. La figure 2.6b présente un exemple de ce patron. Cette spécification formalise une règle de contrôle d'accès de la banque qui interdit le dépôt d'un chèque et sa validation par le même utilisateur.

Pour obtenir la politique complète de contrôle d'accès d'un système d'information, il faut combiner les différentes règles. Cela se fait avec l'utilisation de certaines structures ASTD telles que la *synchronisation paramétrée* et la *synchronisation quantifiée*.

2.9 Conclusion

Cette présentation des modèles de contrôle d'accès aux ressources d'un système les plus fréquemment rencontrés dans la littérature scientifique a permis, d'une part, de positionner le modèle EB³SEC et, d'autre part, de retenir qu'une combinaison de deux modèles, soit un modèle à la RBAC pour des politiques de contrôle d'accès statique et un modèle à la EB³SEC, (mais reformulé dans la notation ASTD) pour les politiques de contrôle d'accès dynamique, constitue un modèle suffisamment puissant pour atteindre les objectifs des projets EB³SEC et SELKIS.

Chapitre 3

État de l'art

Afin de situer l'originalité de la solution présentée dans cette thèse, en particulier le modèle générique de PEM et les méthodes qui permettent l'implémentation des filtres de contrôle d'accès d'un PEM donné, les travaux similaires ou connexes sont décrits dans la suite.

3.1 La norme XACML

XACML [39] est une norme de l'OASIS. Elle consiste essentiellement en un langage XML qui permet d'encoder, d'une part, des politiques de contrôle d'accès de type RBAC étendu et, d'autre part, des requêtes/réponses aux décisions d'autorisation. La norme décrit aussi un modèle de flux de données (figure 1 de [39]) pour une application et un gestionnaire de mise en œuvre d'une politique XACML. Notre travail s'inspire en partie de ce diagramme de flux (voir la section 3.1.3) pour développer le modèle générique décrit au chapitre 4.

3.1.1 Structure d'un document XACML

La figure 3.1 présente le diagramme de classes d'un modèle XACML qui illustre les concepts du langage utiles pour la description d'un document XACML. Un document XACML définit un ensemble de politiques (`PolicySet`). Chaque politique

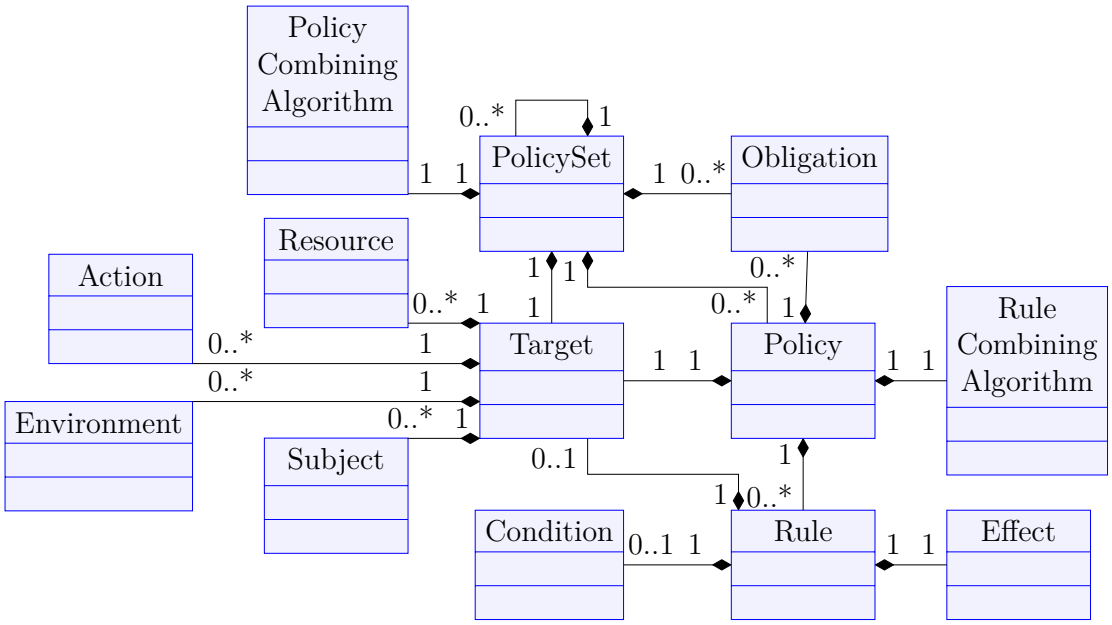


Figure 3.1 – Modèle d'une politique XACML [39]

(`Policy`) vise une cible (`Target`) et comporte des règles (`Rule`) qui sont évaluées dans le contexte courant de la requête d'autorisation reçue de l'environnement. Les règles d'une politique sont évaluées seulement lorsque la cible de celle-ci correspond à la cible de la requête d'autorisation et, les résultats de cette évaluation sont combinés pour fournir un résultat unique pour la politique. La combinaison de ces résultats est faite suivant un algorithme de combinaison (`Rule Combining Algorithm`) précisé par la politique. La politique comporte également des obligations (`Obligation`). XACML définit les obligations comme des actions à exécuter par le gestionnaire de mise en œuvre lorsque la requête d'autorisation traitée est autorisée. Cette fonctionnalité est utilisée par les règles, les politiques et les ensembles de politiques.

Une règle XACML comporte une cible, une condition (`Condition`) et un effet (`Effect`). Tout comme dans le cas des ensembles de politiques et des politiques, la cible indique pour quelles ressources (`Resource`) du système la règle doit être évaluée. Dans le cas d'une requête d'autorisation pour laquelle la cible correspond à la cible précisée par la règle (une requête vise une ou plusieurs ressources du système),

la condition est évaluée. La condition d'une règle est un prédicat qui porte sur les attributs des ressources de la requête et aussi de l'environnement (`Environment`). Les valeurs de retour possibles lors de l'évaluation d'une condition de règle sont *vrai* (*True*) et *faux* (*False*). Suivant la valeur de l'évaluation de la condition de la règle, cette dernière retourne un effet. L'effet correspond à un résultat partiel de la politique qui contient la règle. Les effets possibles sont, entre autres, *permis* (*Permit*) et *non applicable* (*NotApplicable*). Au niveau de la politique, les effets de toutes les règles sont combinés par un algorithme. Par exemple, la combinaison des effets *Permit* et *NotApplicable* par l'algorithme `permit-overrides` retourne l'effet *Permit*.

3.1.2 Requête et réponse d'autorisation

La norme XACML précise un format XML de données pour les requêtes d'autorisation. Un exemple de requête est donné au programme 3.1. Chaque requête précise tous les éléments requis pour que le mécanisme de mise en œuvre puisse calculer un effet. Le sujet (`Subject`) qui est à l'origine de la requête initiant la demande d'autorisation est précisé en premier. Son identifiant est donné par la valeur de l'attribut `subject-id`. Les autres attributs présents dans la requête d'autorisation explicitent des valeurs pour identifier respectivement la ressource qui fait l'objet de l'accès par le sujet, l'action que ce dernier veut exécuter sur la ressource et l'environnement d'exécution de l'action. L'environnement contient des informations pertinentes à l'évaluation de la décision d'autorisation. Un exemple d'une telle information est le lieu, géographique ou institutionnel, à partir duquel le sujet initie l'action. Les réponses à ces requêtes utilisent elles aussi un format XML. Une réponse typique indique essentiellement l'effet de la décision calculée, comme l'illustre le programme 3.2. Elle peut cependant être accompagnée d'obligations à considérer par le gestionnaire de mise en œuvre. Dans les deux programmes, les préfixes et les attributs d'espaces de noms sont omis pour alléger le texte.

3.1.3 Composants d'un PEM

La norme XACML décrit un flux de données dans une application. Ce flux de données ne fait pas partie intégrante de la norme. Cependant le modèle de flux de données

```
<Request>
 <Subject>
  <Attribute
   AttributeId="subject-id"
   DataType="rfc822Name">
   <AttributeValue>
    john@doe.com
   </AttributeValue>
  </Attribute>
 </Subject>
 <Resource>
  <Attribute
   AttributeId="resource-id"
   DataType="http://...#anyURI">
   <AttributeValue>
    file://.../special-file
   </AttributeValue>
  </Attribute>
 </Resource>
 <Action>
  <Attribute
   AttributeId="action-id"
   DataType="http://...#string">
   <AttributeValue>
    read
   </AttributeValue>
  </Attribute>
 </Action>
 <Environment/>
</Request>
```

Programme 3.1 – Requête XACML

```
<Response>
 <Result>
  <Decision>
   NotApplicable
  </Decision>
 </Result>
</Response>
```

Programme 3.2 – Réponse XACML

(voir la figure 3.2) fait ressortir divers composants nécessaires à la spécification d'une politique XACML et à sa mise en œuvre dans une application.

Le premier composant qui intervient dans le flux de données est le « Policy Administration Point » (PAP). Il est le composant responsable de la spécification des politiques XACML. Il fournit les fonctions d'édition et de mise à jour des politiques. La norme suggère que plusieurs composants PAP peuvent coopérer pour la spécification de règles ou de politiques pour une cible. Le PAP doit aussi, lorsque la nature des données référencées dans la politique sous édition le requiert, appliquer des mesures de contrôle d'accès et assurer la confidentialité d'informations sensibles.

Les politiques spécifiées avec le PAP sont accédées par le PDP. Cet accès peut être réalisé au moyen d'un dépôt sécurisé. Le PDP calcule des décisions d'autorisation

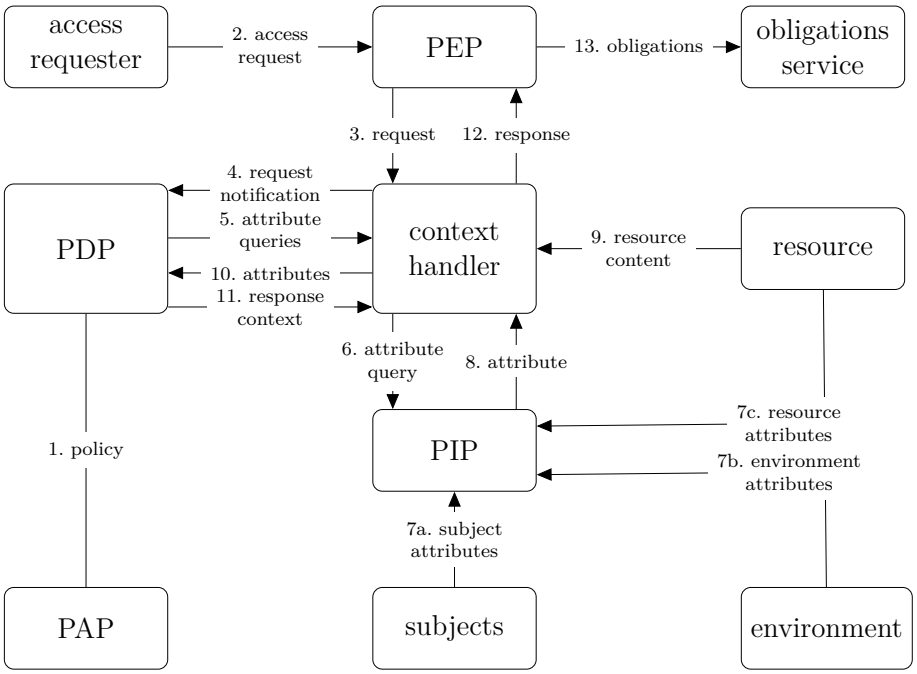


Figure 3.2 – Diagramme de flux de données XACML [39]

pour les politiques déployées par le PAP. La décision d'autorisation est calculée à la suite d'une requête d'autorisation. Les politiques qui correspondent à la cible de la requête d'autorisation sont utilisées pour faire le calcul. En plus des informations de la requête d'autorisation, le composant PDP peut nécessiter d'autres informations. Celles-ci, suivant la norme XACML, sont des attributs des objets du système. Ainsi, des attributs de la ressource accédée ou encore du sujet de la requête peuvent être demandées par le PDP pour arriver à une décision d'autorisation. Le composant responsable de la collecte des valeurs de ces attributs est le « Policy Information Point » (PIP). Il a la responsabilité de fournir au PDP, de façon indirecte, les valeurs des attributs sur les sujets, l'environnement et les ressources du système.

Du point de vue de la norme XACML, le lien entre le système à sécuriser et les autres composants du mécanisme de contrôle d'accès se fait à travers le PEP. Les requêtes aux ressources du système passent par le PEP. À la suite de la décision d'autorisation calculée par le composant PDP, le PEP autorise ou pas l'accès à la

ressource visée. Dans le cas où la réponse permet l'accès demandé à la ressource, le PEP exécute non seulement l'action spécifiée par la requête originale, mais aussi les éventuelles obligations attachées à la décision d'autorisation rendue par le PDP. L'exécution des obligations est la responsabilité d'un *service des obligations*. Entre le PEP et le PDP, le *gestionnaire de contexte* transmet les requêtes en effectuant la conversion nécessaire entre le format de la requête initiale du sujet et le format de requête XACML. Le gestionnaire de contexte achemine aussi les requêtes et réponses des valeurs des attributs entre le PDP et le PIP.

3.1.4 Discussion

La solution proposée dans cette thèse met d'avantage l'emphase sur les composants PEP, PDP et PIP, plus particulièrement sur leurs interactions qui sont différentes de celles présentes dans la partie non normalisée du document [39]. Contrairement au modèle XACML, dans lequel un seul type de politiques est défini de manière générale, notre modèle de politiques de contrôle d'accès comporte plusieurs spécialisations de politiques, ce qui entraîne une différentiation importante au niveau des interactions des composants. Notre solution permet aussi le déploiement de PDP hiérarchiques de façon à prendre en compte la priorité des politiques, ce qui n'est pas possible avec l'architecture XACML. Des aspects comme les algorithmes de combinaison sont aussi inclus dans notre solution, bien qu'ils constituent des éléments de moindre importance.

3.2 Implémentations du contrôle d'accès

Dans cette section, deux approches du contrôle d'accès basées sur RBAC sont présentées. Celles-ci sont complètes car elles comprennent à la fois un langage ou une notation pour spécifier les politiques de contrôle d'accès et un mécanisme de mise en œuvre adapté à la notation. Les implémentations des autres modèles de contrôles d'accès, comme les ACL, les matrices de contrôle d'accès et le modèle de Bell-LaPadula, sont omis puisqu'elles sont généralement intégrées dans des systèmes d'exploitation. Dans tous ces derniers cas, ces méthodes sont hors de notre champ d'application.

3.2.1 Le cadre d'applications X-GTRBAC

Bhatti et al. [6] ont développé « XML Generalized Temporal Role-Based Access Control » (X-GTRBAC), un langage pour l'expression de politiques de contrôle d'accès. Il est basé sur le modèle « Generalized Temporal Role-Based Access Control » (GTRBAC) [33] qui lui-même est fondé sur la norme RBAC. GTRBAC ajoute des contraintes temporelles généralisées à RBAC décrit à la section 2.6. X-GTRBAC est essentiellement une grammaire hors-contexte qui décrit le format de documents XML pour l'écriture de politiques de contrôle d'accès GTRBAC.

GTRBAC reprend en effet tous les éléments d'une politique de contrôle d'accès RBAC. Cela inclut les utilisateurs, les rôles, les permissions et les relations entre, d'une part, les utilisateurs et les rôles et, d'autre part, les rôles et les permissions. Il supporte aussi les contraintes de type *séparation des devoirs* (statique et dynamique) de la norme. GTRBAC ajoute des contraintes de temps à tous ces éléments. Une permission peut être assignée à un rôle pour une période de temps précise. Cette contrainte de temps peut inclure aussi d'autres conditions en fonction du contexte. Une telle contrainte signifie que le rôle possède la permission en question seulement pour l'intervalle de temps spécifié et si seulement l'éventuelle condition additionnelle s'évalue à vrai. De même GTRBAC permet de définir un intervalle de temps pendant lequel un rôle est assigné à un utilisateur, ainsi qu'une contrainte additionnelle qui doit s'évaluer à vrai pour que l'assignation se fasse.

Les rôles, avec la notation GTRBAC, peuvent aussi être spécifiés pour être autorisés ou pas à certaines périodes de temps. Lorsqu'un rôle n'est pas autorisé pendant un intervalle de temps, cela signifie que ce rôle existe bien mais qu'il ne peut être activé par aucun utilisateur durant cet intervalle de temps. Tout comme les relations, l'intervalle de temps utilisé pour les rôles peut être associé à des contraintes exprimées en fonction du contexte d'exécution. La notation GTRBAC permet la spécification d'événements pendant la phase d'exécution des systèmes. L'*activation* ou la *désactivation* d'un rôle par un utilisateur au cours d'une session sont des exemples de tels événements. Il est possible d'exprimer des relations de cause à effet entre les événements. GTRBAC permet ainsi de spécifier des événements futurs en fonction

d'événements antérieurs : ce sont des *déclencheurs*, avec l'exception que l'activation d'un rôle — qui est un événement — ne peut être réalisée que par un utilisateur.

X-GTRBAC permet de représenter tous les concepts définis avec GTRBAC. Ainsi il est possible avec X-GTRBAC de spécifier dans un format XML les rôles (avec leurs justificatifs d'identité et les ensembles de séparations des devoirs auxquels ces rôles appartiennent) et les permissions (comportant chacune l'objet et l'opération pour lesquels elle symbolise un privilège). Il permet également de spécifier les relations entre, d'une part, les rôles et les permissions et, d'autre part, les utilisateurs et les rôles. La spécification des instances de chacune de ces entités se conforme au format défini par X-GTRBAC et les instances sont regroupées par entités dans des *feuilles*. Le format prend aussi en charge la spécification des contraintes temporelles qui peuvent être attachées aux rôles (pour leur autorisation ou leur activation) et aux relations précédemment mentionnées. Enfin, le format permet aussi de spécifier les déclencheurs qui peuvent être activés pendant l'exécution.

X-GTRBAC fournit non seulement un format de document XML pour représenter les politiques de contrôle d'accès, mais aussi une architecture de composants pour le calcul de décisions d'autorisation. Le système ainsi modélisé prend en paramètres une politique de contrôle X-GTRBAC et une requête d'accès, et calcule une décision d'autorisation qui sera utilisée par le mécanisme de mise en œuvre pour autoriser ou pas l'accès à la ressource visée. L'architecture de composants est illustrée à la figure 3.3. Les administrateurs du système utilisent le module de composition de documents pour spécifier les politiques de contrôle d'accès X-GTRBAC. Les politiques ainsi spécifiées sont ensuite chargées dans le module X-GTRBAC pour le calcul de décision. Ce module est constitué d'un processeur XML et d'un processeur GTRBAC. Le processeur XML analyse des politiques de contrôle d'accès et construit un arbre « Document Object Model » (DOM) correspondant à celles-ci. Avant le traitement par le processeur XML, les documents des politiques sont chargés et validés respectivement par un module de chargement et un module de validation. La validation vérifie par exemple que les utilisateurs référencés dans la feuille de la relation entre les utilisateurs et les rôles existent dans la feuille des utilisateurs du système.

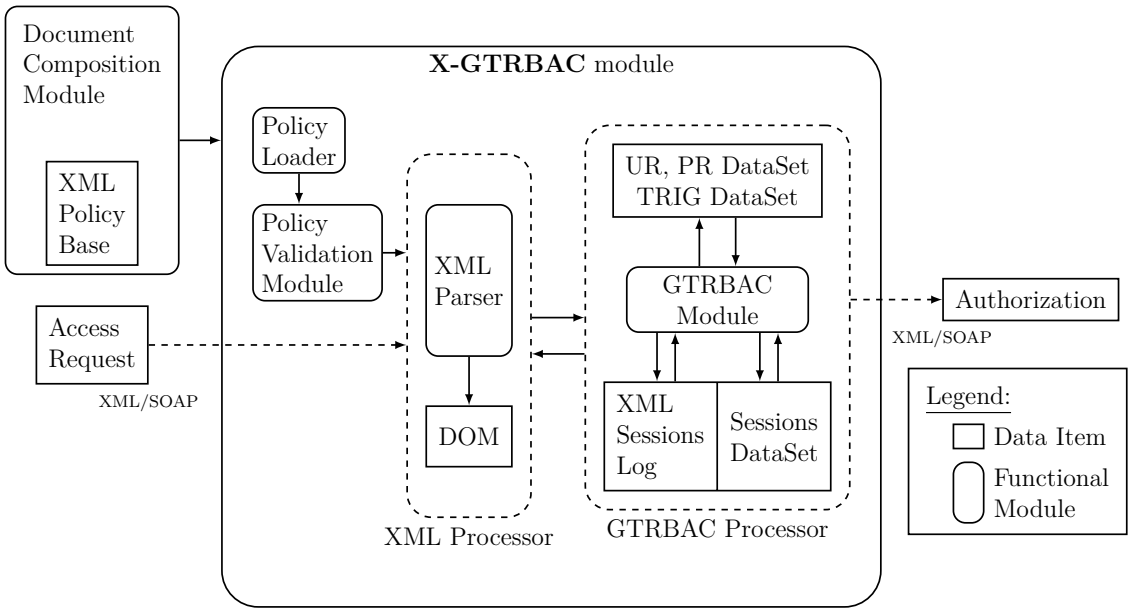


Figure 3.3 – Architecture de composants pour une décision X-GTRBAC [6]

L'arbre construit par le processeur XML est transmis au processeur GTRBAC. Le module GTRBAC enregistre l'arbre DOM dans sa propre base de données et procède à des vérifications de cohérence supplémentaires. L'information contenue dans cette base de données est ensuite utilisée par le module GTRBAC pour gérer les sessions des utilisateurs lorsqu'ils se connectent et pour calculer les décisions d'autorisation en fonction des requêtes d'accès aux ressources du système. Le module X-GTRBAC, illustré à la figure 3.3, est implémenté comme une application Java avec une interface graphique nécessaire pour afficher les données de la politique de contrôle d'accès telles que les utilisateurs ou les rôles.

3.2.2 Cadre d'applications W-RBAC pour les workflows

L'expression « Workflow Role-Based Access Control » (W-RBAC) [50] désigne un cadre d'applications pour le développement d'applications de workflows sécurisés. Un *workflow* est un ensemble partiellement ordonné de tâches dans lequel chaque tâche est une unité atomique de travail à réaliser par un intervenant humain ou par un ou plusieurs systèmes automatiques. W-RBAC est constitué de deux modèles. Le premier

modèle, W0-RBAC, combine un modèle de politiques de contrôle d'accès basé sur RBAC à un système d'exécution de workflow. Le second modèle, W1-RBAC, définit un mécanisme de gestion des exceptions qui permet d'outrepasser les contraintes spécifier par le modèle de base pour le fonctionnement normal du système.

Un système de gestion de workflows (Workflow Management System (WfMS) en anglais) utilise les spécifications des workflows pour instancier des cas particuliers. Cette instanciation est semblable à la création de n-uplets dans une base de données relationnelles suivant son schéma. L'admission à une université est un exemple de workflow et les demandes d'admission des étudiants prospecteurs sont des cas de ce workflow. Les tâches à exécuter immédiatement par des intervenants humains sont attribuées aux intervenants qui disposent de la compétence nécessaire. Par exemple, des tâches comptables seront affectées aux employés du service de comptabilité présents. Lorsqu'une tâche est complétée, d'autres tâches deviennent exécutables suivant la relation d'ordre partielle qui définit le workflow. W-RBAC prend en compte la nature spécifique de l'exécution des workflows.

Le modèle W0-RBAC est basée sur la norme RBAC. Une politique W0-RBAC comporte des utilisateurs, des rôles et des permissions, ainsi que les deux relations usuelles entre ces relations : l'assignation des rôles à des utilisateurs et l'assignation des permissions à des rôles. Les permissions dans le contexte des workflows sont les privilèges d'exécuter des tâches. En plus des entités du modèle RBAC de base, W0-RBAC ajoute les entités *cas* et *unité organisationnelle*. Un *cas* représente une instance particulière d'un workflow en cours d'exécution. Une *unité organisationnelle* regroupe des utilisateurs du système, de façon à mieux correspondre à la division du travail au sein d'une organisation. Cette entité et les relations qui sont définies à partir de celle-ci facilitent l'attribution des tâches aux utilisateurs en fonction de leur implication dans les unités de l'organisation. Par exemple, dans une entreprise un employé fait partie du département assurance qualité des produits matériels et il est impliqué pour quelques mois dans un projet de développement logiciel. Dans cet exemple, l'employé fait partie de deux unités organisationnelles : celle du département de la qualité et celle du projet de développement en cours. W0-RBAC définit les

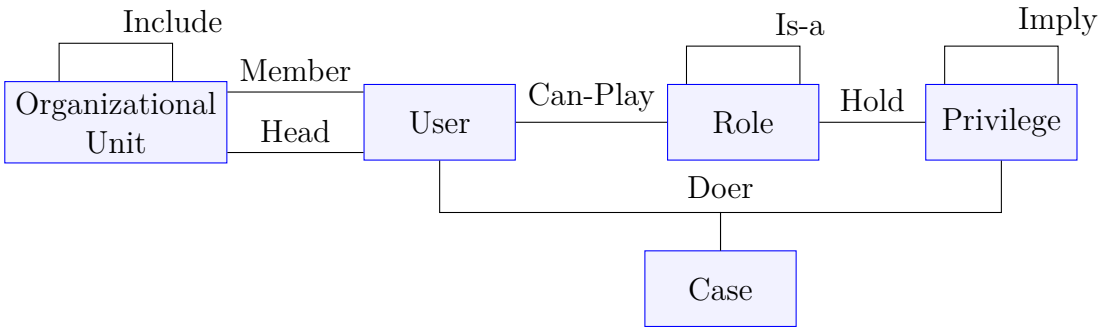


Figure 3.4 – Métamodèle W0-RBAC [50]

relations pour spécifier l'appartenance d'un utilisateur à une unité organisationnelle, et également pour définir le responsable de celle-ci. Il est aussi possible, à la manière de la hiérarchie sur les rôles, d'avoir une hiérarchie sur les unités organisationnelles par une relation réflexive sur les unités. Enfin, W0-RBAC définit une relation ternaire entre les utilisateurs, les permissions et les cas. Cette relation est utilisée pour savoir quel utilisateur a exécuté une certaine tâche pour un cas particulier. Le métamodèle qui comprend toutes ces relations est représenté à la figure 3.4.

À partir du métamodèle ainsi défini, W0-RBAC permet de raffiner le contrôle d'accès par des contraintes. Celles-ci sont écrites comme des programmes logiques par lesquels les états indésirables du système sont spécifiés. W0-RBAC distingue les *contraintes statiques* et les *contraintes dynamiques*. Les contraintes statiques définissent les modifications qui peuvent être faites aux données d'une politique. Ces données comprennent, en autres, les assignations des rôles aux utilisateurs tout comme les assignations des permissions aux rôles. Les contraintes dynamiques font intervenir les cas ou les tâches dans l'expression des programmes logiques. Elles sont plus larges que les contraintes statiques et sont utilisées pour préciser les conditions sous lesquelles des utilisateurs peuvent exécuter des tâches dans des cas de workflows. Les contraintes dynamiques peuvent exprimer par exemple des contraintes de type SoD.

W0-RBAC définit aussi un service de permission qui interagit avec le système de gestion de workflows. Le service de permission a un accès immédiat à toute l'informa-

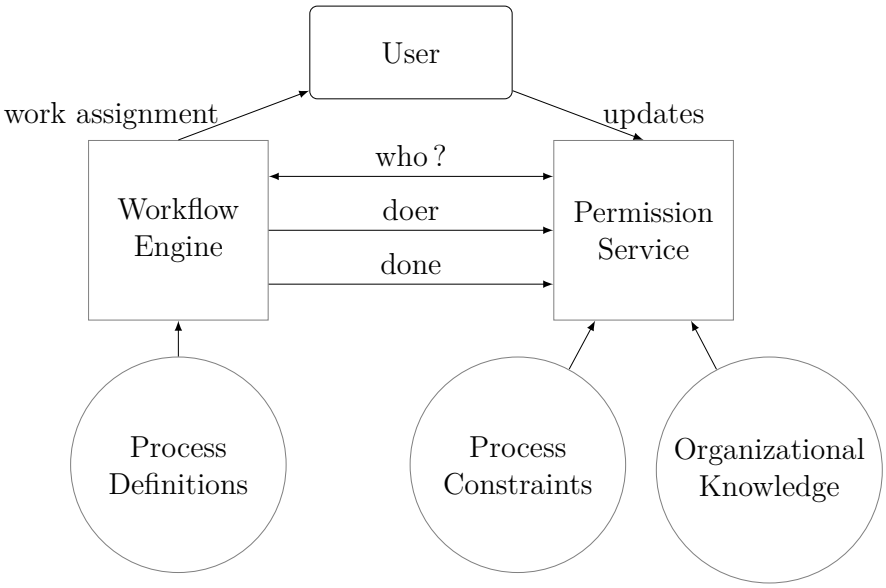


Figure 3.5 – Interactions entre les composants et les utilisateurs [50]

tion organisationnelle (par exemple, utilisateurs, rôles, unités organisationnelles) et aux contraintes sur les workflows, tandis que le service de gestion de workflows gère l'information sur les workflows et les cas en cours d'exécution. Cette architecture est illustrée à la figure 3.5. Le service de permission et le système de gestion des workflows interagissent par trois canaux. Le premier canal, nommé *who ?*, permet au composant de gestion des processeurs d'interroger le service de permission pour déterminer les utilisateurs qui peuvent exécuter une tâche donnée. Ce même canal est utilisé par le service de permission pour répondre à cette requête. La réponse est une liste ordonnée de groupes d'utilisateurs. Ces utilisateurs peuvent tous exécuter la tâche dans le cas passé en paramètre car ils ne violent aucune contrainte, en particulier celles inhérentes aux assignations des rôles aux utilisateurs et des permissions aux rôles. Ils sont groupés et listés par ordre de préférence décroissante. Le second canal, nommé *doer*, informe le service de permission sur l'identité de l'utilisateur qui a exécuté une tâche dans un cas d'un workflow. Cette information est utilisée pour mettre à jour la relation ternaire du modèle correspondant. Le dernier canal, nommé *done*, indique au service de permission que l'exécution d'un cas d'un workflow est complétée.

W1-RBAC vient compléter W0-RBAC pour la gestion des situations exceptionnelles. Il peut arriver souvent avec les workflows qu'un cas *bloque* parce que les contraintes exprimées sont restrictives à un point que la requête *who ?* retourne une liste vide. Dans ces situations, un utilisateur approprié du système peut outrepasser la politique établie pour débloquer le cas, par exemple en affectant un utilisateur à la tâche bloquée du système. W1-RBAC permet de représenter cette fonctionnalité avec deux éléments de spécification. Le premier élément est l'attribution d'un *niveau de priorité* aux contraintes de la politique de contrôle d'accès. Un niveau de priorité est tout simplement un entier positif non nul attaché à une contrainte. Ainsi les contraintes de même niveau de priorité peuvent être regroupées ensemble. Le second élément de spécification est la permission spéciale *override(n)* où n est un entier positif. Cette permission signifie qu'un utilisateur qui la possède peut outrepasser toutes les contraintes ayant un niveau de priorité inférieur ou égal à n.

W1-RBAC est intégré à l'architecture précédente illustrée à la figure 3.5 par l'ajout d'un canal spécial $assign?(u_1, u_2, t, c)$. Ce canal est utilisé par le système de gestion de workflows pour communiquer au service de permission que l'utilisateur u_1 veut forcer la réalisation de la tâche t dans le cas c d'un workflow par l'utilisateur u_2. Dans ce cas, l'utilisateur u_2 doit posséder le droit RBAC de réaliser t. De plus le métamodèle de la figure 3.4 doit être complété avec des éléments pour représenter la délégation ou le transfert de droits qui se produit entre les utilisateurs u_1 et u_2. Forcer la réalisation d'une tâche peut violer des contraintes exprimées au niveau W0-RBAC et l'utilisateur u_1 doit posséder par l'un de ses rôles actifs une permission *override(n)* telle que l'entier n soit plus grand que toutes les contraintes violées. Dans le cas où l'utilisateur u_1 ne possède pas une telle permission alors le canal spécial $assign?(u_1, u_2, t, c)$ retourne la valeur *false* au système de gestion de workflows pour indiquer que l'affectation de la tâche ne peut être exécutée. Dans le cas contraire, le service de permission met à jour la base de données avec la nouvelle affection et retourne le niveau de priorité de l'utilisateur u_1 qui lui permet d'outrepasser la politique de contrôle d'accès normale du système.

3.2.3 Discussion

Le cadre d'applications X-GTRBAC est propre à une extension de RBAC qui ajoute des contraintes temporelles. La mise en œuvre de politiques de contrôle d'accès X-GTRBAC n'utilise pas les composants du PEM de la norme XACML bien que Batthi et al. aient proposé avec la notation un moteur de calcul de décision. Leur prototype d'implémentation est réalisé en Java mais n'est pas accompagné d'une étude de performance. Les implémentations réalisées dans nos travaux sont différentes car elles ont une architecture SOA et sont accompagnées d'une étude de performance. De plus, les mécanismes de contrôle d'accès sont basés sur les composants introduits par la norme XACML.

Wainer et al. ont proposé W-RBAC qui consiste en une extension de la norme RBAC ajoutant la notion d'organisation et des programmes logiques pour exprimer des contraintes. Cette notation a été définie pour permettre le contrôle d'accès dans des WfMS. L'approche proposée par Wainer et al. diffère de nos travaux tout d'abord par la notation utilisée pour exprimer les politiques de contrôle d'accès. Le langage ASTD définit des traces acceptables pour le système tandis que W1-RBAC définit des états acceptables. D'autre part, les systèmes à sécuriser que nous considérons sont des applications SOA alors que Wainer et al. traitent des WfMS pour lesquels un prototype réalisé en Prolog a permis de valider les concepts mais pas l'efficacité en situation réelle. Dans nos travaux, des tests unitaires et de performance appuient les méthodes proposées.

Soulignons aussi les travaux de Bourdier et al. [9] qui ont développé un cadre d'applications pour des systèmes sécurisés. Leur approche est basée sur des règles formelles de réécriture. D'une part, une politique de contrôle d'accès est un système de règles réécriture de termes, et d'autre part, l'état d'une politique de contrôle d'accès est un ensemble de faits. À chaque décision d'autorisation rendue, la base de faits est mise à jour par l'application des règles de réécriture. Ce cadre d'applications permet également la validation et vérification de propriétés de sécurité. L'approche est semblable à celle proposée par Wainer et al. dans [50] qui est elle cependant spécialisée pour les WfMS.

Enfin, Mao et al. [37] ont proposé une implémentation de la norme XACML adaptée au contrôle de l'accès à des documents numériques. Leur méthode résout, sans toutefois préciser comment, les problèmes de flot d'information solutionnés par Bell-LaPadula avec les règles « no read up » et « no write down ». La solution à ces problèmes repose sur la recherche des chemins potentiels de fuite d'information et l'attribution de contraintes aux rôles XACML pour empêcher ces fuites.

3.3 Transformation de modèles

La transformation de modèles est le procédé par lequel une spécification écrite dans un langage de départ est traduite en une spécification équivalente dans un langage d'arrivée. Cette technique est à la base de méthodologies de développement logicielle telles que « Model-Driven Engineering » (MDE) [45]. Les transformations peuvent être distinguées selon plusieurs critères. Quand les langages de départ et d'arrivée sont identiques, on parle de transformation *endogène*, dans le cas contraire la transformation est dite *exogène*. La transformation est *bidirectionnelle* lorsqu'il est possible d'obtenir le modèle de départ à partir du modèle d'arrivée. Sinon elle est simplement *unidirectionnelle*.

Pour modéliser une transformation, les langages d'entrée et de sortie doivent être spécifiés d'abord. En MDE, cette spécification prend couramment la forme d'un métamodèle pour chacun des langages. Ensuite, la transformation elle-même est spécifiée comme une correspondance entre les éléments du métamodèle du langage de départ et ceux du métamodèle du langage d'arrivée. Le plus souvent, chaque élément du métamodèle du langage de départ est mis en correspondance avec un ensemble d'éléments du métamodèle du langage d'arrivée. Par exemple dans l'application de la méthode de développement MDE, un modèle UML d'entités d'un système d'information est transformé en un modèle de données relationnelles pour un SGBD moderne et une bibliothèque d'objets Java pour la couche d'accès aux données d'une application multi-tiers. Ces transformations sont illustrées à la figure 3.6.

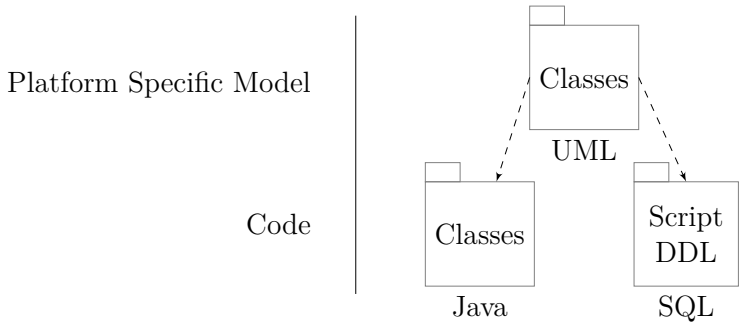


Figure 3.6 – Transformations de modèles dans MDE

Relativement à BPEL, de nombreux travaux ont été menés soit pour simplifier le développement de processus BPEL [54, 10], soit pour préciser leur sémantique [49] avec des réseaux de Petri [43], avec des réseaux de Petri colorés [53], avec Event-B [3], avec des automates [25, 52], avec des machines à états abstraites (Abstract-State Machines (ASM) en anglais) [20] ou encore des algèbres de processus [44]. Seuls [10, 3] sont présentés, car parmi tous ces travaux ce sont les seuls qui se rapprochent le plus des nôtres.

3.3.1 Vérification de processus BPEL

Ait-Sadoune [3] a proposé une transformation de processus BPEL en des modèles Event-B. Cette transformation a pour but la vérification et la validation des processus vis-à-vis de propriétés. Elle s'effectue en deux étapes. La première, dite statique, transforme la définition des services WSDL associés à un processus BPEL vers un **CONTEXT** d'un modèle Event-B. La seconde phase transforme le processus BPEL en un modèle Event-B. Ces deux étapes ont été implémentées dans l'outil BPEL2B.

L'étape statique traite des documents WSDL et XSD associés à des processus BPEL. Les types de données, les messages, les opérations et les types de port définis par un fichier WSDL sont traduits dans un **CONTEXT** Event-B. Par exemple, un message WSDL est transformé en un ensemble (élément **SETS** de Event-B) et chaque partie du message (élément **part** de WSDL) est transformé en une constante (élément **CONSTANT** de Event-B). Ces nouveaux éléments Event-B sont liés ensuite

par des axiomes (élément **AXIOMS** de Event-B). Un axiome spécifie que l'ensemble correspondant au message n'est pas vide et pour chaque partie du message, un axiome spécifie que la constante associée à la partie du message est une fonction. Une opération dans un document WSDL devient une constante Event-B et un axiome spécifie que cette constante est une fonction de l'ensemble encodant le type du paramètre d'entrée vers l'ensemble encodant le type paramètre de sortie. Dans l'éventualité où l'opération n'a pas de paramètre d'entrée ou de paramètre de sortie, l'ensemble correspondant est remplacé par l'ensemble spécial `Void`.

L'étape dynamique construit une machine Event-B à partir d'un processus BPEL. Cette machine utilise le contexte construit à partir du document WSDL de l'interface du processus BPEL. Ait-Sadoune distingue deux groupes d'activités BPEL : les activités simples et les activités structurées. Les activités simples comportent les activités de communication avec d'autres services Web, les activités de manipulation de données et les activités de contrôle telles que **wait**, **empty** et **exit**. Leur transformation dans la machine du processus produit un événement. Les activités structurées quant à elles sont les activités qui permettent de composer des activités de base, telles que les activités itératives, conditionnelles (**if**) ou de contrôle de flot. Leur transformation produit, dans les événements créés pour les activités englobées, des éléments pour encoder le comportement de l'activité structurée concernée. Par exemple, pour une séquence qui contient n activités simples, une variable entière est créée et sa valeur initialisée à n. Cette variable est décrémentée par l'exécution des événements produits pour chaque activité simple de la séquence et de plus, ces exécutions sont gardées par une valeur spécifique de la variable. L'événement de la $k^{ème}$ activité de la liste ordonnée des activités de la séquence BPEL ne peut s'exécuter que si la variable associée à la séquence a la valeur $n - k + 1$. Dans le cas général, cette condition d'exécution relative à la séquence est ajoutée à d'autres conditions inhérentes à la traduction de l'activité simple ou d'autres activités structurées englobantes.

Avec cette transformation, Ait-Sadoune définit une approche de conception verticale de processus BPEL. Dans celle-ci un processus BPEL est développé de façon incrémentale : une version abstraite du processus est enrichie de détails en même temps

que la machine Event-B correspondante est obtenue par raffinement de la machine de l'itération précédente. Cette approche permet l'introduction graduelle d'obligations de preuve à un niveau adéquat d'itération de la transformation, simplifiant ainsi la décharge de celles-ci.

3.3.2 Développement de processus BPEL corrects

Chirichiello et Salaün [10] ont proposé une méthode de développement de processus BPEL dans laquelle une algèbre de processus est utilisée pour spécifier le comportement attendu d'un processus. Cette approche diffère de la précédente dans laquelle le but est de valider le comportement d'un processus BPEL existant. Dans l'approche de Chirichiello et Salaün, le comportement désiré de l'interaction est spécifié et les outils disponibles pour l'algèbre de processus sont utilisés pour vérifier des propriétés désirables pour l'interaction. La transformation proposée utilise le langage LOTOS comme langage de départ et produit non seulement le processus BPEL mais aussi un document WSDL pour en décrire l'interface.

Dans [10], un guide de la traduction de LOTOS vers BPEL est décrit. Les émissions et réceptions sur des portes LOTOS sont traduites respectivement en des activités BPEL **`reply`** et **`receive`**. Lorsque l'émission et la réception sont spécifiées en séquence (opérateur « ; » de LOTOS), alors elles sont toutes deux traduites en une seule activité **`invoke`**. Ces activités de communication sont créées avec comme paramètres des opérations correspondant aux portes de l'émission ou réception correspondante. Les variables de ces activités BPEL correspondent aux variables utilisées dans LOTOS pour communiquer sur les portes. De ces opérations de communication LOTOS est aussi extraite la déclaration de l'interface du processus, notamment les messages et les opérations qui correspondent aux portes LOTOS. L'opération de séquence LOTOS est traduite en une activité BPEL **`sequence`**. Une action cachée du processus LOTOS correspond à une activité non visible de l'extérieur du processus, par exemple une activité **`assign`**. L'opérateur garde de LOTOS est traduit en l'activité **`case`** (**`if`** dans la nouvelle version de la norme BPEL). Un opérateur choix LOTOS avec des gardes pour chaque option du choix est traduit en une activité **`case`** avec différentes branches **`switch`**. Dans la nouvelle version de la norme BPEL, c'est l'activité BPEL

if qui est utilisée avec autant de branches conditionnelles **elseif** que nécessaires. Un choix LOTOS avec des réceptions sur des portes est traduit en une activité **pick** avec autant de branches **onMessage** que de réceptions sur des portes.

3.3.3 Discussion

Les travaux de Chirichiello et Salaün [10] ont pour but de construire des processus BPEL corrects vis-à-vis d'une spécification. C'est aussi l'une des démarches effectuées dans cette thèse, notamment au chapitre 5. Mais contrairement à leur approche qui constitue plutôt un guide pour la traduction de spécifications LOTOS en processus BPEL, notre approche est systématique puisqu'elle ne requiert pas d'intervention humaine. Une autre différence se manifeste dans le langage utilisé pour la spécification du comportement attendu des processus et aussi dans la finalité de ces processus BPEL. Dans nos travaux nous nous intéressons au langage ASTD, alors que dans [10] LOTOS est la notation privilégiée. De plus les processus BPEL visés dans cette thèse ont la finalité de servir à la mise en œuvre de politiques de contrôle d'accès. Nos travaux se basent aussi sur l'interprétation de spécifications formelles pour achever le même résultat, ce qui n'est pas le cas dans [10]. La pertinence des travaux de Ait-Sadoune [3] se manifeste dans la correspondance qu'il établit entre les éléments du langage BPEL et un autre langage, mais BPEL est dans ce cas le langage de départ et non le langage cible. La transformation doit cependant être guidée dans le processus de conception verticale. Au chapitre 5, les structures du langage ASTD sont mises en correspondance avec les activités du langage BPEL.

Chapitre 4

Modèle générique d'un PEM

Un modèle générique d'un gestionnaire de politiques de contrôle d'accès, appelé PEM, pour des applications de type SOA constitue un outil indispensable par rapport à l'approche méthodologique retenue dans cette thèse. Ce modèle est décrit en utilisant une technique de métamodélisation qui permet de bien mettre en évidence les principaux composants d'un PEM, mais aussi d'introduire différents types d'architecture logique. Le métamodèle est décrit à l'aide de diagrammes de séquences qui précisent les échanges de messages et de diagrammes de classes qui spécifient la structure des composants. Le travail présenté dans ce chapitre complète celui précédemment développé dans [17].

4.1 Architecture et composants d'un PEM

Les applications à sécuriser sont des applications SOA telles que présentées à la section 1.2. Celles-ci doivent être intégrées de façon transparente dans un cadre de sécurité que l'on désigne par le PEM. Différentes possibilités d'intégration existent et elles s'effectuent toutes à travers les composants actifs du PEM.

4.1.1 Composants actifs de l'architecture

L'architecture que nous proposons pour un PEM s'inspire fortement de celle décrite par la norme XACML de l'OASIS (voir la section 3.1). Cette norme présente tout d'abord un langage XML qui permet d'implémenter des politiques de contrôle d'accès à la manière de RBAC, en spécifiant des contraintes sur les *sujets*, les *ressources*, les *actions* et l'*environnement*. Mais, elle décrit aussi une possible architecture d'un cadre de mise en œuvre de politiques de contrôle d'accès. Les principaux composants de cette architecture que nous avons retenus sont le PEP, le PDP et le PIP.

Le composant PEP est le principal composant, car il met en œuvre la politique de contrôle d'accès. Cette mise en œuvre s'effectue par l'interception des requêtes aux services, la recherche d'une décision d'autorisation pour la requête courante et l'envoi de celle-ci au service ciblé si la requête est autorisée.

Le composant PDP calcule les décisions d'autorisation requises par le PEP suivant les politiques de contrôle d'accès déployées. Ce calcul se fait avec l'aide de composants PDP et de filtres subordonnés. Au niveau du PDP, ce dernier réalise une synthèse des différents résultats de décision obtenus par les composants subordonnés. L'architecture proposée supporte une hiérarchisation de composants PDP, car de nombreux champs d'applications sont soumis à des ensembles différents de contraintes d'accès. Ceux-ci peuvent être de priorités inégales et la composition des composants PDP qui calculent des décisions d'autorisation pour chacun des ensembles de contraintes doit prendre en compte ces priorités.

4.1.2 Échange informel de messages

La figure 4.1 illustre un scénario typique des échanges de messages entre un client (ou utilisateur), le PEP, le PDP et le service visé. Ce scénario suppose qu'un client, qui essaie d'utiliser ce service, a été authentifié avec succès. Le client authentifié envoie une requête à un service du système d'information (message 1 de la figure 4.1). Dans le cheminement de cette requête vers le service, les privilèges du client vis-à-vis du service demandé sont validés par le composant PEP qui a intercepté la

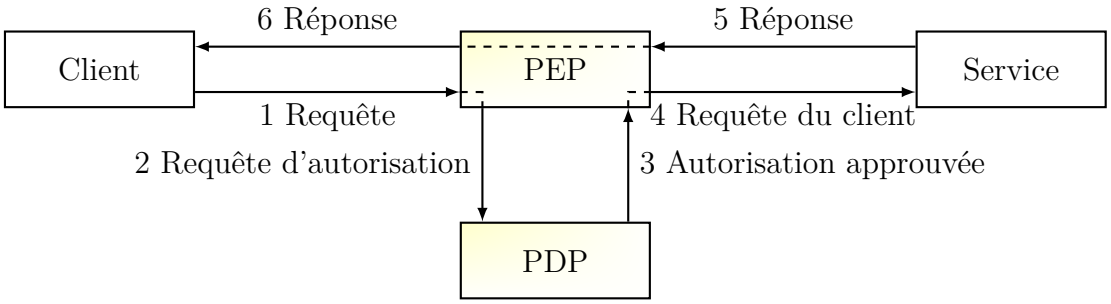


Figure 4.1 – Échange typique de messages

requête du client. La validation, initiée par la réception d'une requête d'autorisation (message 2), consiste en partie à calculer une décision d'autorisation, calcul effectué par le composant PDP suivant les politiques de contrôle d'accès déployées. Dans le scénario illustré, l'autorisation est approuvée (message 3). Le composant PEP autorise la requête initiale du client au service ciblé (message 4). Le service exécute ou pas la requête du client en procédant éventuellement à d'autres validations métiers, et la réponse est retournée au client en passant par le composant PEP (messages 5 et 6).

4.1.3 Différentes architectures possibles

La disposition logique évidente pour les composants du PEM est une répartition centralisée de ceux-ci, comme illustrée dans la figure 4.2. Dans cette vision du déploiement, le PEP est un point unique et central pour le cadre de sécurité. Pendant le fonctionnement du système, toutes les requêtes d'un client sont traitées par la même instance du PEP. Cette instance est liée à une unique instance de PDP de la plateforme. Les décisions d'accès sont donc ainsi centralisées et les requêtes et réponses sont traitées par un unique fil d'exécution. Ce mode de déploiement sied parfaitement pour des politiques de contrôle d'accès dynamique, car elles requièrent qu'un état courant de leur exécution soit maintenu. Cependant, en fonction de la charge du système et des ressources matérielles affectées à l'exécution de la plateforme de sécurité, ce mode de déploiement peut entraîner un goulot d'étranglement.

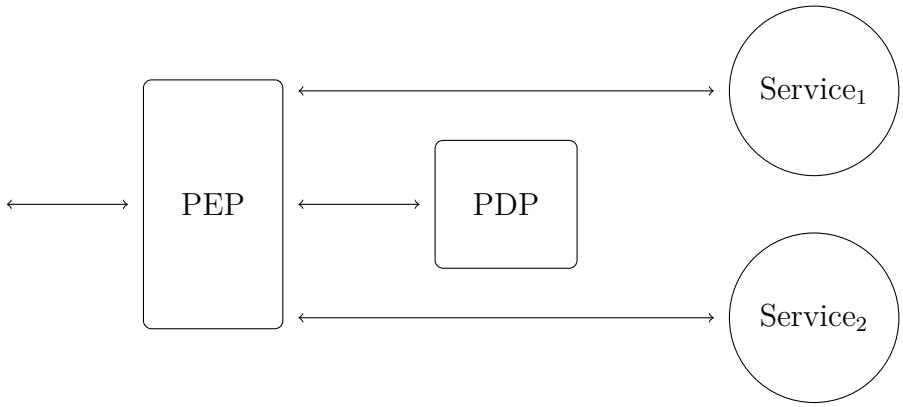


Figure 4.2 – Architecture centralisée du PEM

À l'opposé d'un déploiement centralisé, les composants du PEM peuvent être déployés de manière décentralisée. Le contrôle d'accès à chaque service du système d'information est alors assuré par une instance de chacun des composants du cadre de sécurité. Ce mode de déploiement est illustré dans la figure 4.3. Chaque requête d'un client est traitée directement pour le composant PEP qui est rattaché au service ciblé par la requête. Les différentes requêtes sont alors gérées dans différents fils d'exécution, ce qui pallie l'inconvénient des goulots d'étranglement du déploiement précédent. En revanche, le mode décentralisé convient difficilement aux politiques de contrôle d'accès dynamique, car celles-ci contraignent l'occurrence des accès aux ressources du système les uns par rapport aux autres. En particulier, les règles comme les *SoD* et les *obligations* contraignent les paramètres (métiers ou de contrôle d'accès) d'un service par rapport aux paramètres d'un autre service. Ces politiques ne peuvent donc pas être déployées individuellement car elles sont relatives à tous les services du système d'information.

À mi-chemin entre ces deux extrêmes, il est possible d'effectuer un mode de déploiement hybride (voir la figure 4.4). Dans celui-ci, une partie commune aux services de l'état de la politique de contrôle d'accès est partagé par l'ensemble de ces services. L'état du système nécessaire au calcul des décisions d'autorisation est toujours partagé pour tous les services. Une architecture hybride rend possible une meilleure répartition de la charge du cadre de sécurité. Dans la figure 4.4, les requêtes du client

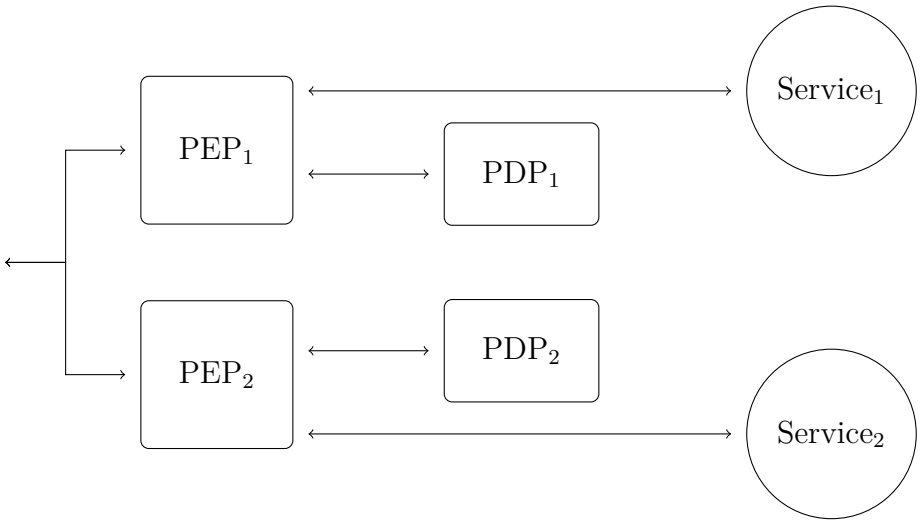


Figure 4.3 – Architecture décentralisée du PEM

adressées à un service et qui arrivent au niveau du PEP vont être autorisées ou refusées. Le calcul de la décision d'autorisation est effectué par un composant PDP rattaché au PEP et configuré avec l'ensemble des règles de contrôle d'accès qui engage le service ciblé par la requête courante. Ces règles peuvent impliquer d'autres services, comme dans le cas des *SoD* ou des *obligations*. Les décisions d'autorisation

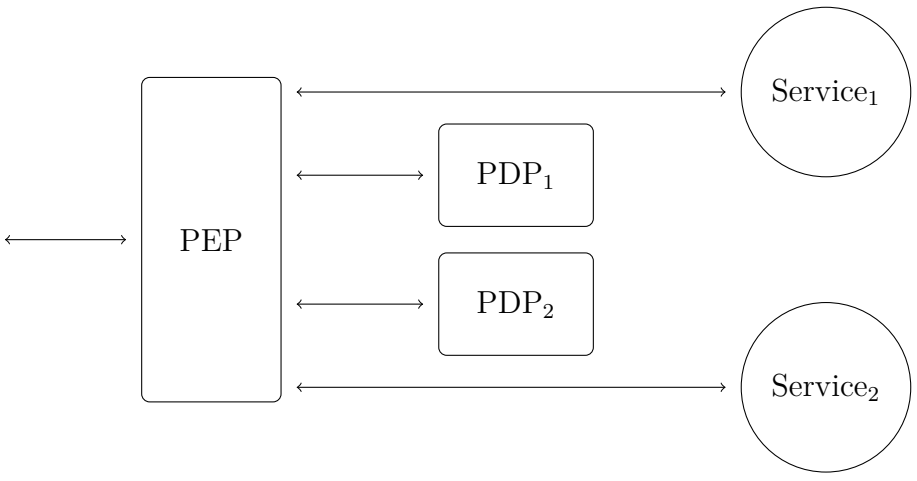


Figure 4.4 – Architecture hybride du PEM

pour les services qui apparaissent dans les mêmes règles sont calculées par la même instance du composant PDP, et cette relation s'applique de façon transitive à tous les services. Ce type d'architecture est un compromis acceptable qui offre les avantages du déploiement de politiques de contrôle d'accès dynamique et de la réduction de la charge du système lorsque cela est possible.

Dans le cadre de l'étude comparative menée pour les deux implémentations présentées dans cette thèse, l'architecture hybride est utilisée, car rappelons qu'elle sied le mieux lorsque plusieurs politiques de contrôle d'accès dynamique sont mises en œuvre avec des composants PDP distincts.

4.2 Spécification du PEM par métamodélisation

La spécification du PEM présentée dans cette section est basée sur une architecture centralisée du PEM. Cette spécification décrit de manière générique le fonctionnement d'un PEM en précisant comment les informations sont échangées entre les différents composants et en détaillant la structure de ses composants. La spécification est développée en utilisant une technique de métamodélisation. Des diagrammes de séquences décrivent le fonctionnement et des diagrammes de classes décrivent les différents objets. L'intérêt de cette approche est que ces métamodèles peuvent ensuite être instanciés afin d'obtenir différentes implémentations concrètes. Les chapitres 5 et 6 de cette thèse en décrivent deux. Dans le cadre du projet ANR SELKIS, le partenaire SWID a réalisé deux autres implémentations dérivées de ce métamodèle [11].

4.2.1 Diagrammes de séquences

Dans la description suivante des diagrammes de séquences, il est sous-entendu que les objets anonymes portent le nom de leur classe ; ils sont écrits en minuscule dans la police « sans sérif ». Les noms dans la police « `machine à écrire` » sont des noms de classe. Trois scénarios typiques sont présentés.

La figure 4.5 illustre les échanges de messages entre un utilisateur authentifié et la plateforme, quand en particulier celui-ci accède au service `E` avec les paramètres

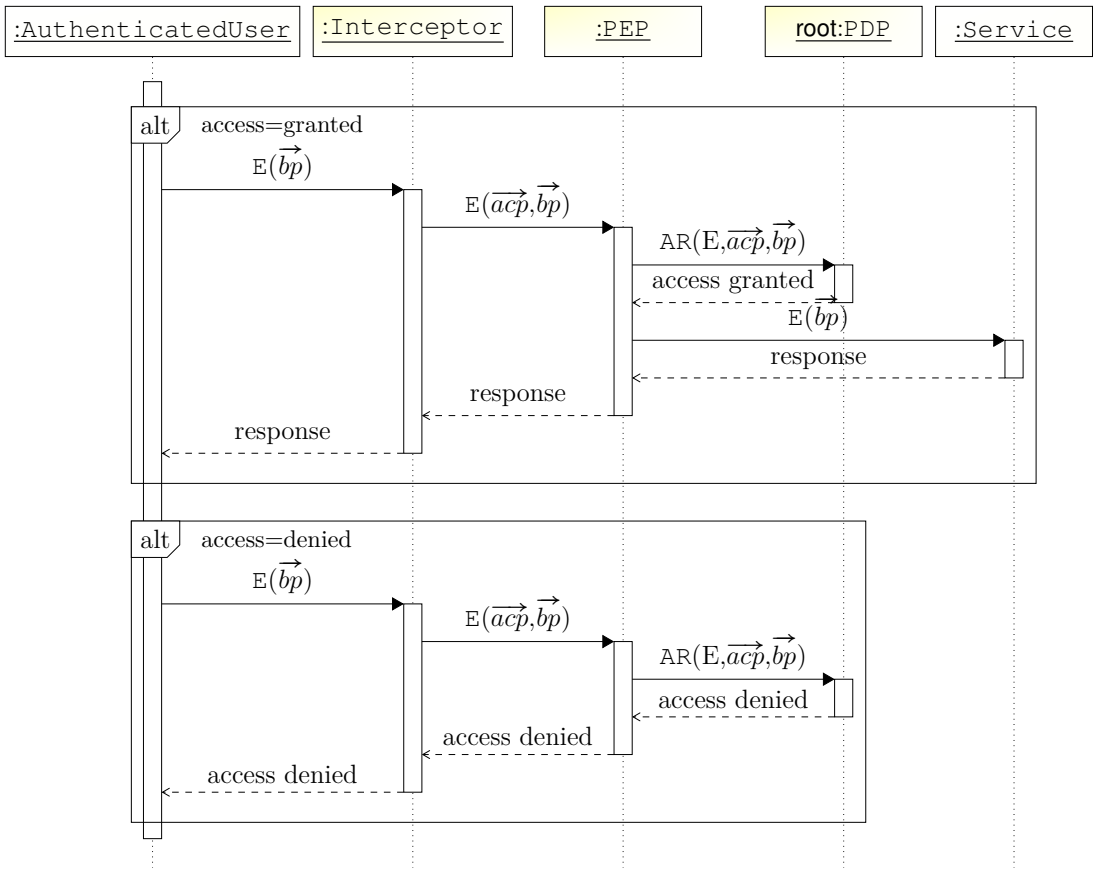


Figure 4.5 – Diagramme de séquences : PDP au plus haut niveau

métiers $\overrightarrow{bp}$. Elle montre aussi la séquence principale de messages qui passent à travers les instances interceptor, pep et root suite à la requête au service initiée par l'utilisateur. Tel qu'indiqué par son nom, une instance de la classe `Interceptor` intercepte chaque requête de l'utilisateur pour y ajouter les paramètres de contrôle d'accès $\overrightarrow{acp}$, tels que le rôle de l'utilisateur et son organisation. L'intercepteur envoie le message augmenté au PEP avec l'intention d'appliquer les politiques de contrôle d'accès déployées pour le service ciblé. Ensuite, le PEP demande au PDP de plus haut niveau dans une hiérarchie de PDP une autorisation d'exécuter le service E avec les ensembles de paramètres $\overrightarrow{acp}$ et $\overrightarrow{bp}$ (message AR(E,$\overrightarrow{acp}$,$\overrightarrow{bp}$) dans la figure 4.5, AR pour « Authori-

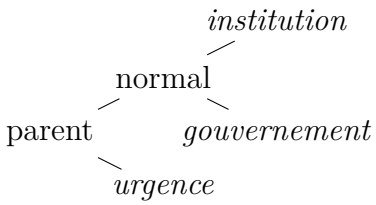


Figure 4.6 – Exemple d'une hiérarchie de PDP dans le domaine médical

zation Request » en anglais). Le PDP retourne une décision d'autorisation au PEP. Si l'accès est *autorisé*, alors la requête originale de l'utilisateur est transmise au service destination. Sinon l'utilisateur reçoit une notification qui indique que l'exécution du service demandé n'est pas autorisée. La réponse renvoyée par le service demandée, s'il s'exécute, n'est pas retournée directement à l'utilisateur. Elle passe par le PEP, qui la redirige à l'utilisateur.

Le processus de décision peut être complexe lorsque plusieurs politiques sont imposées par différents organismes de régulation internes ou externes. Dans ce cas, plusieurs instances de la classe PDP peuvent être déployées et organisées de façon hiérarchique selon le niveau de priorité de chaque politique. Pour illustration, dans le domaine médical, cinq instances du PDP (voir la figure 4.6) peuvent former une hiérarchie en trois niveaux lorsque l'on considère les politiques de contrôle d'accès suivantes :

1. une politique établie par les lois gouvernementales pour les établissements de la santé ;
2. une politique basée sur les procédures internes de l'institution en question ;
3. une politique qui est prioritaire en cas d'urgence par rapport aux deux précédentes.

À la figure 4.6, le PDP `parent` ignore les décisions rendues par le PDP `normal` lorsque le système est placé en situation d'urgence. Quand le fonctionnement système est en mode normal, la décision d'autorisation est la combinaison des décisions rendues par les composants PDP `institution` et `gouvernement`. Entre deux niveaux de la hiérarchie, il s'établit une relation parent-enfant sur laquelle le processus de décision

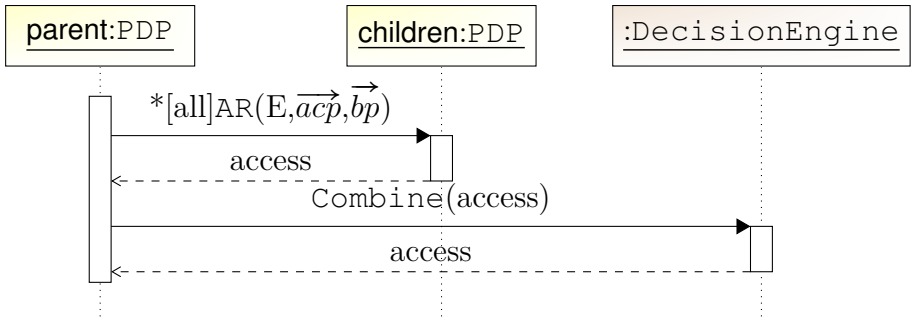


Figure 4.7 – Diagramme de séquences : PDP à un niveau intermédiaire

est basée. Comme le montre la figure 4.7, un PDP interne de la hiérarchie qui reçoit une requête d'autorisation est considéré comme un PDP parent. Ce dernier fait suivre la même requête à ses successeurs immédiats qui sont ses enfants. Les enfants deviennent parent à leur tour, s'ils ne sont pas des noeuds terminaux de la hiérarchie. Tous les enfants retournent leur décision d'accès à leur parent, qui conserve celles-ci jusqu'à ce que, soit un délai expire, soit il reçoit la dernière. Après la réception de la dernière réponse, le parent appelle une instance de la classe DecisionEngine qui combine les décisions d'accès. L'objet root est celui qui doit retourner la décision d'accès finale au PEP (voir la figure 4.5).

Comme le montre la figure 4.8, un PDP enfant à la base de la hiérarchie utilise un ensemble de filtres pour inférer une décision d'accès. Lorsqu'il reçoit un requête d'autorisation de son PDP parent, il appelle ses filtres de façon séquentielle, en leur faisant suivre le même message de requête avec cependant un paramètre additionnel (pr) résultant des filtres précédents (ce paramètre est ignoré par le premier filtre), puisqu'un filtre donné peut requérir les résultats des filtres précédents pour pouvoir calculer son propre résultat. De façon similaire à la situation décrite à la figure 4.7, les filtres retournent leurs résultats au PDP enfant, qui les conservent jusqu'à la réception du dernier message de réponse ou l'expiration d'un délai. Le PDP enfant utilise une instance de la classe DecisionEngine qui transforme les résultats des filtres en une décision d'accès. Le PDP enfant peut aussi utiliser les filtres en parallèle, et dans ce cas, le paramètre pr est tout simplement ignoré.

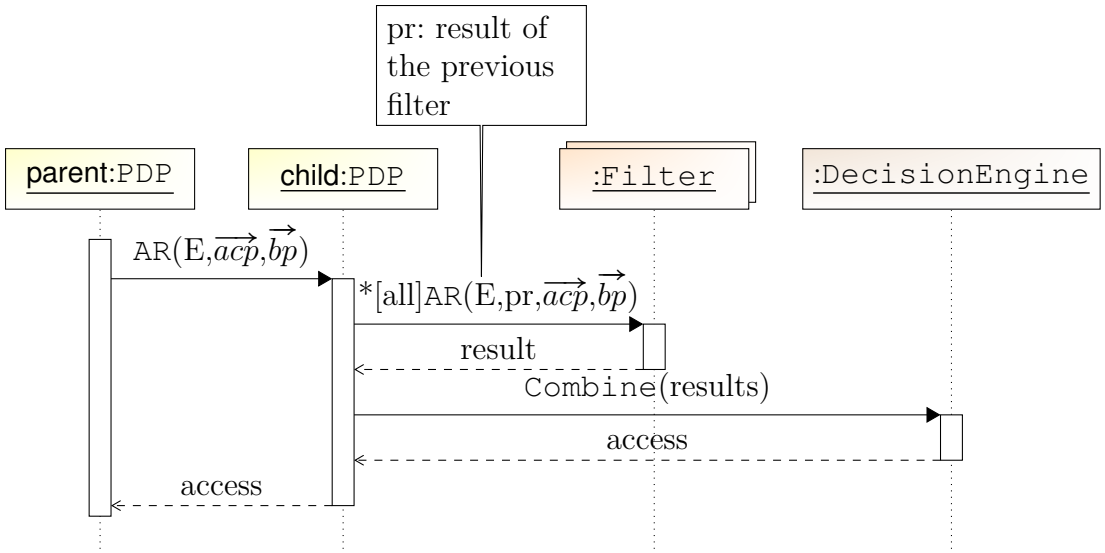


Figure 4.8 – Diagramme de séquences : PDP au plus bas niveau

4.2.2 Diagrammes de classes

Les classes d'objets sont décrites dans des diagrammes de classes pour expliciter les relations entre celles-ci. Toutefois dans un métamodèle, les classes sont en fait des métaclasses. Les noms écrits dans la police « machine à écrire » représentent les noms de classe. Le style « *italique* » est utilisé pour les noms de métaclasses abstraites et les classes abstraites.

Tous les messages sont abstraits dans une classe qui est spécialisée suivant les différents usages qui en sont faits dans le diagramme de séquences de la figure 4.5. Les métaclasses Message, AccessControlMessage et AuthorizationMessage représentent respectivement les messages E($\overrightarrow{bp}$), E($\overrightarrow{acp}$,$\overrightarrow{bp}$) et AR(E,$\overrightarrow{acp}$,$\overrightarrow{bp}$), incluant les requêtes et les réponses correspondantes, comme illustré par le diagramme de classes de la figure 4.9. Ce diagramme spécifie aussi la classe des objets qui envoient et reçoivent des messages. Les instances des métaclasses Interceptor, PEP et PDP différent suivant l'implémentation. Par exemple dans une application SOA, les instances de la métaclasse Interceptor peuvent être des classes de gestionnaires SOAP (SOAP handler classes en anglais), dont les instances s'insèrent de façon

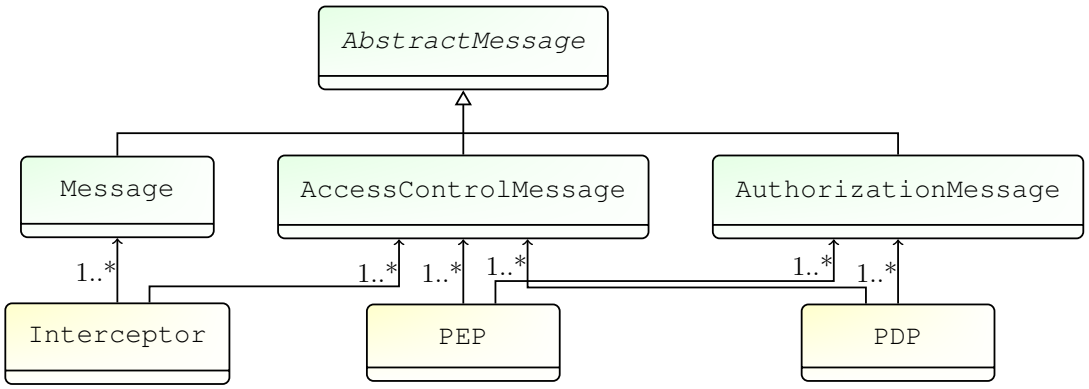


Figure 4.9 – Diagramme de classes : composants du PEM

transparente entre l'application de l'utilisateur et les services Web. Ils ont pour rôle d'injecter les paramètres de contrôle d'accès dans l'en-tête des requêtes encodées avec le protocole SOAP.

Le diagramme de classes de la figure 4.10 présente les différents types de paramètres d'un message. En général, un message contient des paramètres optionnels, qui sont catégorisés suivant le type de message auquel ils sont rattachés. Un paramètre est associé à une valeur d'entrée ou de sortie en fonction du message (requête/réponse de service ou appel/retour de méthode). Cette caractéristique est un attribut de la classe abstraite *AbstractParameter*. Suivant cet attribut, il existe des messages unidirectionnels qui comportent des paramètres d'entrée seulement et pour lesquels les émetteurs n'attendent pas de réponse du récepteur. Il existe aussi des messages bidirectionnels, avec une requête et une réponse.

La métaclasse `AccessControlParameter`, qui apparaît dans les diagrammes de classes des figures 4.10 à 4.13, est instanciée lorsqu'un modèle de contrôle d'accès est défini et les paramètres de contrôle d'accès propres à l'application considérée sont explicités (par exemple les classes `Organization`, `Role` et `User`). Tout modèle dérivé du métamodèle peut considérer d'autres paramètres de contrôle d'accès, tel que le moment de l'émission d'une requête par l'utilisateur ou l'origine géographique de celle-ci.

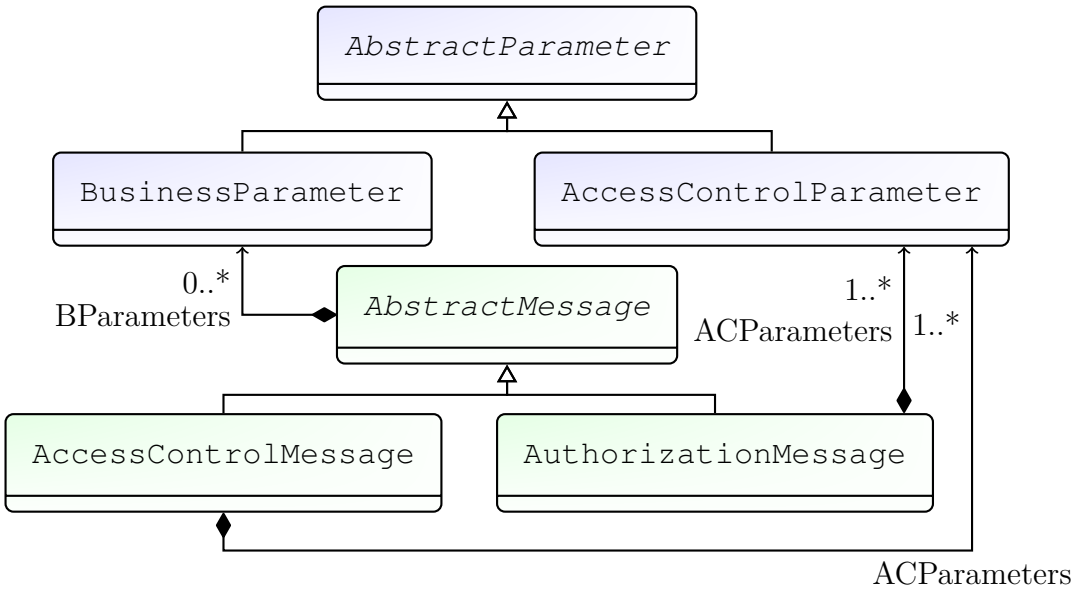


Figure 4.10 – Diagramme de classes : hiérarchie de messages

Une politique de contrôle d'accès est définie par un ensemble de règles représentées par la métaclasse abstraite `Rule` dans les figures 4.12 et 4.13. Le métamodèle fait une distinction claire entre les règles statiques et dynamiques. Les règles statiques contraignent à la manière des modèles RBAC, alors que les règles dynamiques prennent en compte l'historique des opérations exécutées avec succès par le système d'information. Les règles statiques comprennent les *permissions*, qui autorisent une opération à être exécutée, et les *interdictions*, qui empêchent l'exécution d'une opération. La *séparation des devoirs* (*SoD*), qui impose la contrainte qu'un ensemble d'opé-

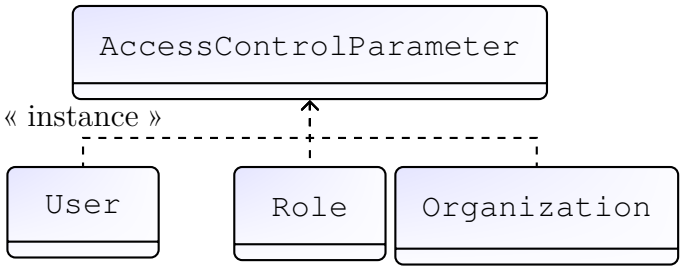


Figure 4.11 – Diagramme de classes : paramètres de contrôle d'accès

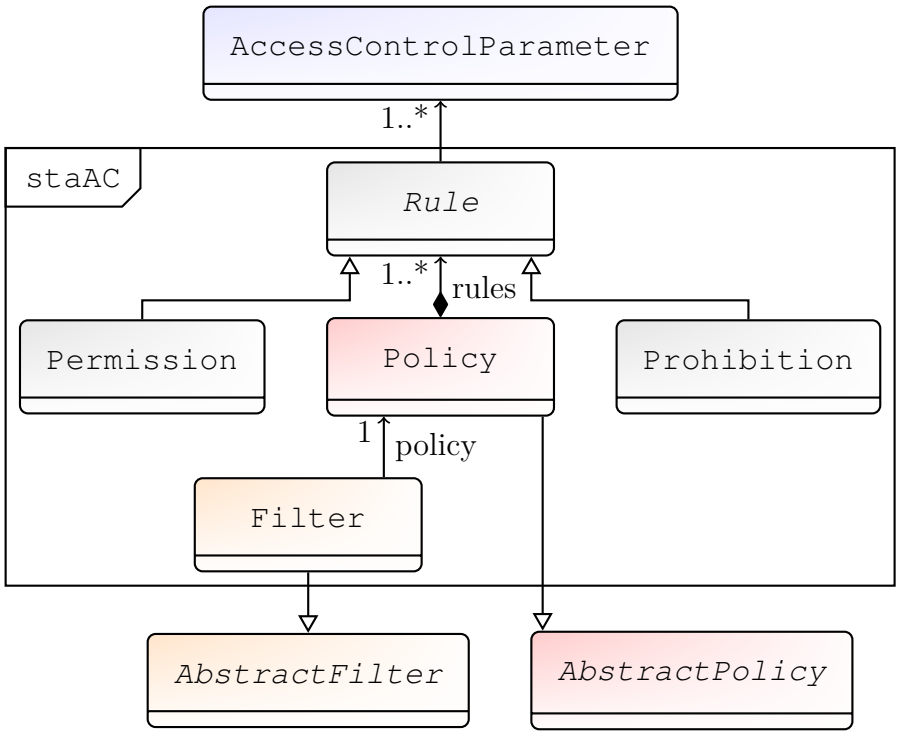


Figure 4.12 – Diagramme de classes : contrôle d'accès statique

rations doivent être exécutées par différents utilisateurs ou rôles est l'exemple le plus courant de règle dynamique. Un autre exemple de règle dynamique est l'*obligation* qui contrairement à la séparation des devoirs, oblige un utilisateur ou un rôle à exécuter un ensemble d'opérations liées. La métaclasse abstraite `staAc::`*`Rule`* (respectivement `dynAC::`*`Rule`*), qui apparaît dans le paquetage `staAC` dans la figure 4.12 (respectivement `dynAC` dans la figure 4.13), représente les règles statiques (respectivement dynamiques), qui sont utilisées pour définir les politiques statiques (respectivement dynamiques).

Les politiques de contrôle d'accès peuvent être séparées en deux, chacune étant associée à un filtre distinct. Contrairement aux classes `Organization`, `Role` et `User` du diagramme de classes de la figure 4.11, les éléments `Permission` et `Prohibition` sont véritablement des sous-métaclasses de `staAc::`*`Rule`* car le métamodèle permet l'instanciation de celles-ci pour modéliser différentes classes de règles de permission

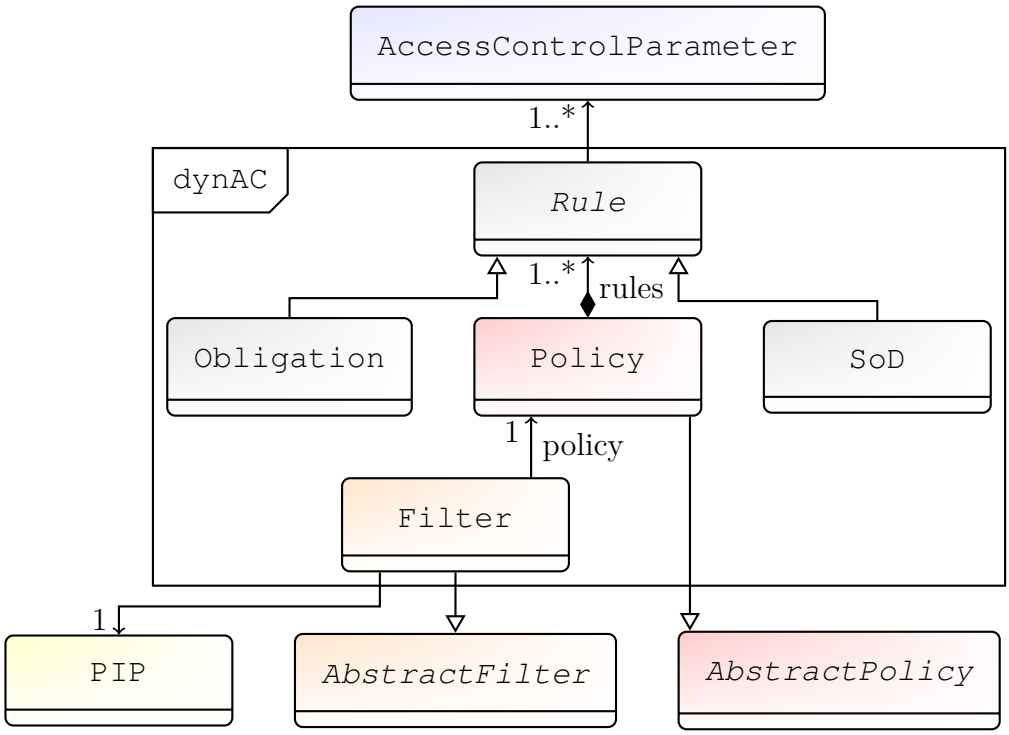


Figure 4.13 – Diagramme de classes : contrôle d'accès dynamique

ou règles d'interdiction pour des modèles de contrôle d'accès spécifiques. Par exemple, une règle de permission peut autoriser l'exécution d'opérations en fonction des rôles seulement, ou encore interdire l'exécution des opérations en fonction de l'utilisateur et de son rôle. Les instances de la métaclasse `staAC::Filter` (respectivement `dynAC::Filter`), une spécialisation de la métaclasse abstraite *`AbstractFilter`* dans le paquetage du niveau supérieur, sont des classes dont les instances utilisent une politique statique (respectivement dynamique) pour calculer une décision après évaluation de tout ou partie des règles de cette politique. En général, un filtre essaie de faire correspondre les paramètres E, $\overrightarrow{acp}$ et $\overrightarrow{bp}$ du message d'une requête d'autorisation avec les valeurs des termes correspondant de chaque règle de la politique associée au filtre. En plus de cette correspondance entre paramètres et valeurs, un filtre associé à une politique dynamique requiert de l'information sur l'historique des opérations exécutées par le système et liées aux règles à considérer. Le PIP, un composant auxiliaire du PEM, fournit cette information (voir la figure 4.13). Plusieurs

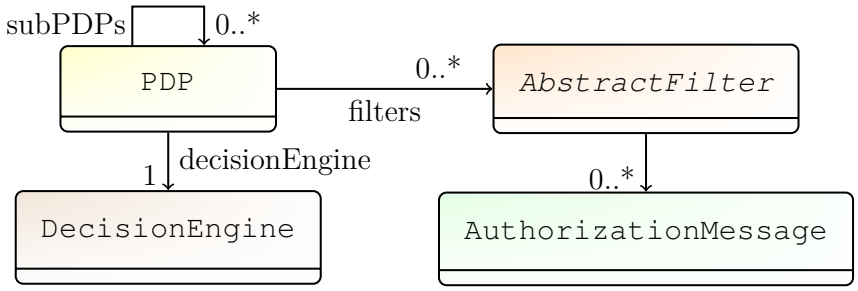


Figure 4.14 – Diagramme de classes : hiérarchie de PDP

implémentations de ce composant sont possibles. Un PIP peut mémoriser le sous-ensemble de l'état du système nécessaire pour déterminer une décision d'autorisation relativement à la définition de toutes les règles dynamiques. Une autre implémentation possible peut interroger les bases de données du système d'information ou des attributs d'environnement (par exemple la charge du système ou l'heure système) pour fournir l'information requise.

Enfin, la figure 4.14 montre les relations entre les métaclasses *AbstractFilter*, `PDP` et `DecisionEngine`. Le diagramme de classes illustre clairement que les PDP sont organisés de façon hiérarchique et qu'un seul moteur de décision est associé à chaque PDP. La synthèse d'une décision d'accès est symbolisé par la métaclasse `DecisionEngine`. Dans son implémentation élémentaire, un moteur de décision fusionne les décisions locales par une conjonction de celles-ci. Mais il est possible d'envisager des mécanismes d'inférence encore plus sophistiqués. Par exemple, un moteur de décision peut donner la priorité à une politique sur les autres dans le cas d'urgence.

4.3 Instances du métamodèle

Dans l'optique de la validation du métamodèle du PEM, celui-ci a été instancié pour des spécifications formelles de contrôle d'accès qui sont transformées ou interprétées (voir les chapitres 5 et 6). Un exemple simplifié issu du milieu bancaire permet d'illustrer une implémentation du métamodèle. Dans cette étude de cas, le

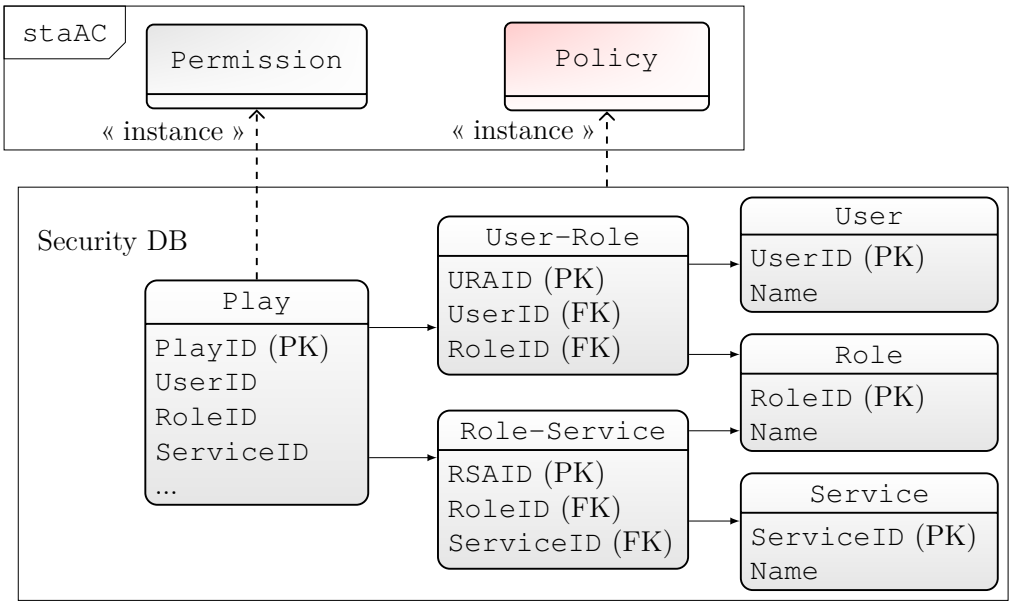


Figure 4.15 – Schéma d'une politique de contrôle d'accès statique

système d'information offre les opérations *dépôt* (`deposit`), *validation* (`validate`), *annulation* (`cancel`) aux *guichetiers* (`clerk`) et *chefs d'agence* (head office (`ho`) en anglais) d'une petite banque.

Instancier le métamodèle revient à instancier les éléments décrivant la politique de contrôle d'accès elle-même et les composants du PEM qui met en œuvre cette politique. Dans les instances du métamodèle décrites dans cette section, la politique de contrôle d'accès est implémentée par les mêmes éléments, seuls certains aspects dynamiques du PEM sont distincts suivant, entre autres, la méthode de spécification des politiques utilisée.

Une politique de contrôle d'accès est constituée d'une partie statique et d'une partie dynamique. La partie statique de la politique est implémentée par un ensemble de tables d'une base de données relationnelles. Dans cet ensemble de tables, la métaclasse `Permission` est instanciée par une vue qui répond à la question « quel *utilisateur* exerçant quel *rôle* a le droit d'effectuer quelle *action* ». Cette vue, appelée `Play` dans la figure 4.15, est une jointure des tables `User-Role` et `Role-Service`

Tableau 4.1 – Données d'une politique de contrôle d'accès statique

User		User-Role		Role		Role-Service		Service	
ID	Name	UserID	RoleID	ID	Name	RoleID	ServiceID	ID	Name
U1	Alain	*U1*	*R2*	*R1*	cashier	*R1*	*S1 .. S4*	*S1*	deposit
U2	Bob	*U1*	*R3*	*R2*	ho	*R2*	*S2 .. S3*	*S2*	validate
U3	Cedric	*U2*	*R1*	*R3*	customer			*S3*	cancel
U4	Daniel	*U3*	*R1*					*S4*	register
U5	Eric	*U5*	*R3*						

sur la colonne RoleID. Les tables User, Role et Service sont respectivement les listes des utilisateurs, des rôles et des services du système. De façon similaire, la métaclasse Prohibition pourrait être instanciée mais cela n'est pas requis pour l'étude de cas considéré. L'instanciation de la métaclasse Permission est illustrée dans la figure 4.15. Cette instanciation donne un modèle typique RBAC. Le tableau 4.1 représente un exemple de données dans cette base de données. Dans ces données, l'utilisateur Alain est à la fois client et chef d'agence de la banque, c'est-à-dire que les rôles customer et ho lui sont assignés. D'une part, le rôle ho peut exécuter les services validate et cancel, et d'autre part, le rôle cashier peut exécuter tous les services du système. La vue Play qui fait la synthèse de tous les services qu'un utilisateur peut exécuter se déduit par jointure des tables User-Role et Role-Service.

L'instanciation de la partie dynamique de la politique de contrôle d'accès se fait par l'utilisation des éléments du langage formel ASTD. L'instance de la métaclasse Policy est une spécification ASTD dont la structure principale combine des structures qui sont des instances des métaclasses SoD et Obligation. La figure 4.16 illustre l'instanciation des éléments Policy, Obligation et SoD du paquetage dynAC. La métaclasse Policy du paquetage dynAC est instanciée par la classe Specification du métamodèle du langage ASTD introduit à la section 1.1. Les types de règles SoD et Obligation sont tous les deux instanciés par la même classe ASTD.

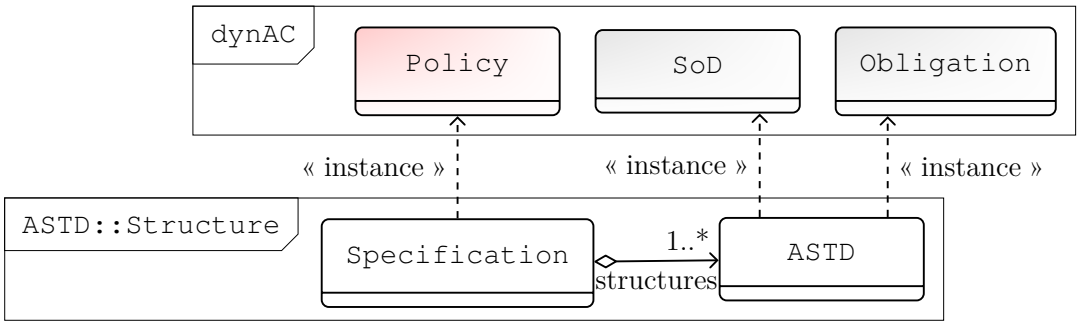


Figure 4.16 – Schéma d'une politique de contrôle d'accès dynamique

Le contrôle d'accès se fait sur les identifiants des utilisateurs ainsi que sur leurs rôles. Relativement au métamodèle présenté dans les sections précédentes, deux classes, `User` et `Role`, sont instanciées à partir de la métaclasse des paramètres de contrôle d'accès `AccessControlParameter`, comme le montre la figure 4.17.

Les instances des métaclasses `Interceptor` et `PEP` (figure 4.9) sont des « handlers » SOAP. Cette instanciation est illustrée à la figure 4.18. Elles implémentent l'interface Java `SOAPHandler<SOAPMessageContext>`. La logique d'interception est implémentée dans la méthode Java `handleMessage`. Dans celle-ci, le message correspondant à la requête SOAP est modifié et les valeurs des paramètres de contrôle d'accès sont insérées dans la section en-tête de l'enveloppe SOAP. Les « handlers » SOAP sont des objets qui interviennent entre les participants d'une conversation SOAP. La logique de la méthode `handleMessage` est invoquée pour chaque message SOAP, pendant la requête SOAP tout comme pendant la réponse correspondante.

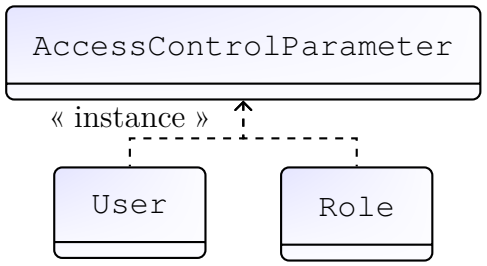


Figure 4.17 – Classes instances des paramètres de contrôle d'accès

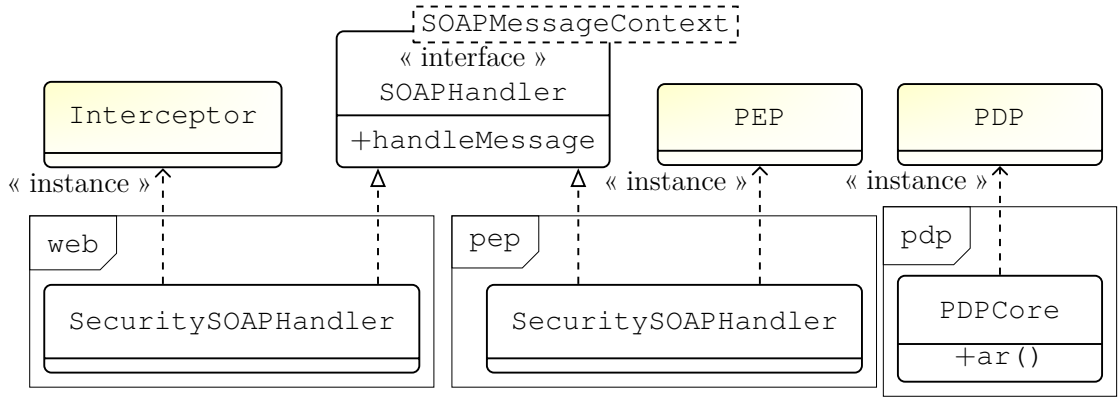
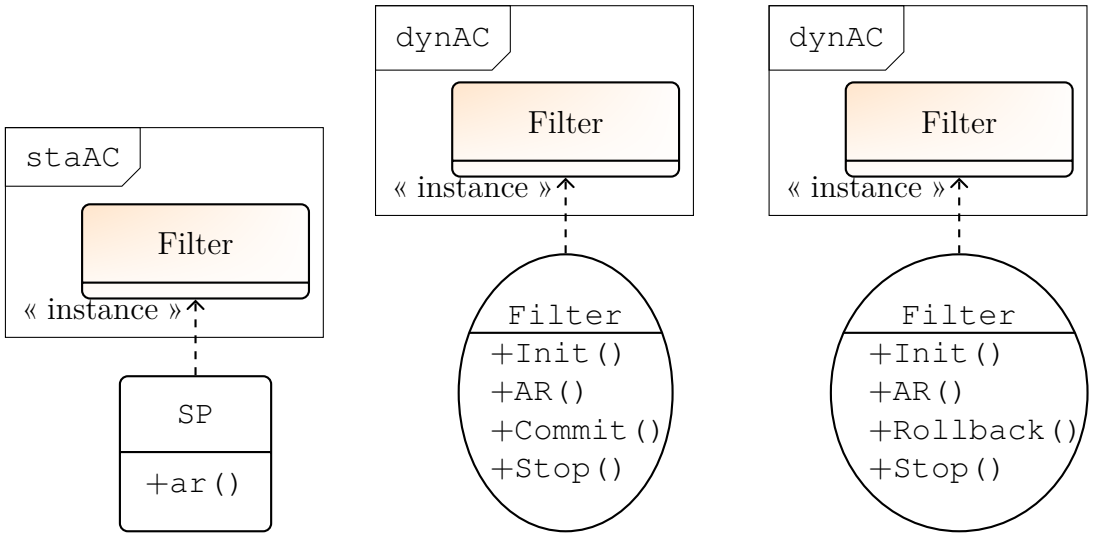


Figure 4.18 – Instanciation des composants actifs de contrôle d'accès

Dans le cas de la classe `web::SecuritySOAPHandler`, la requête SOAP est interceptée et modifiée à travers le paramètre `context` de type `SOAPMessageContext` de la méthode `handleMessage` qui contient une référence à l'enveloppe SOAP de la requête. Le but de cette modification est l'ajout, dans l'en-tête du message SOAP de la requête, des valeurs des paramètres de contrôle d'accès.

La sous-classe Java de l'interface `SOAPHandler<SOAPMessageContext>` dans le paquetage `pep`, qui implémente aussi la métaclasse `PEP`, extrait les paramètres de contrôle d'accès de la requête SOAP et applique la politique de contrôle d'accès déployée avec ces paramètres à la requête de l'utilisateur. En particulier, si la politique autorise l'exécution de la requête initiale par le service ciblé alors la méthode `handleMessage` retourne la valeur *true* qui indique à l'environnement d'exécution que la requête SOAP peut poursuivre son chemin vers le service. Dans le cas contraire, la méthode `handleMessage` retourne la valeur *false* et l'environnement d'exécution ne transmet pas la requête initiale au service ciblé.

Le composant `PDPCore`, qui implémente la métaclasse `PDP` (voir la figure 4.18), reçoit des requêtes d'autorisation (appel à la méthode `ar`) du composant PEP dans la méthode `handleMessage` de la classe Java `pep::SecuritySOAPHandler`. Le nom de la méthode `ar` est en caractères minuscules pour respecter la conven-



(a) Filtre statique (b) Filtre dynamique obtenu par transformation (c) Filtre dynamique obtenu par interprétation

Figure 4.19 – Diagrammes d'instanciation des filtres

tion de nommage usuelle de Java, mais cette méthode correspond au message `AR` de la figure 4.5. Dans l'étude de cas considérée, la politique déployée requiert un seul composant PDP. Dans cette classe, les aspects statique et dynamique de la politique de contrôle d'accès sont testés tour à tour. L'ordre séquentiel *test statique* puis *test dynamique* est important puisque la partie *décision d'accès dynamique* est plus coûteuse en termes de ressources que la partie *décision d'accès statique*. Procéder à ces requêtes d'autorisation dans cet ordre permet une optimisation élémentaire. Dans cette implémentation, la métaclasse `DecisionEngine` est *implicite* car la décision d'autorisation finale est la conjonction des décisions d'autorisation pour les parties statique et dynamique de la politique de contrôle d'accès. Le calcul par étapes de cette décision correspond bien à une implémentation de la conjonction car le calcul de la décision d'autorisation pour la politique dynamique n'est effectué que si le calcul pour la politique statique retourne le résultat *granted*.

Les instances des métaclasses `staAC::Filter` et `dynAC::Filter` sont illustrées dans la figure 4.19. Le filtre de contrôle d'accès statique est implémenté par

```
1  SELECT COUNT(*)
2   FROM Play pl
3   WHERE
4     pl.UserID = :UserName
5    AND
6     pl.RoleID = :RoleID
7    AND
8     pl.ServiceID = :ServiceID
```

Programme 4.1 – Requête SQL du prédicat statique

la classe `SP` (abréviation de static predicate en anglais). Cette classe retourne une décision d'autorisation positive (*granted*) pour la méthode `ar` si la base de données relationnelle contient un n-uplet dans la vue `Play` avec les valeurs des paramètres de contrôle d'accès introduits par le composant `Interceptor` et transférés par le composant `PEP`, c'est-à-dire l'identifiant de l'utilisateur, son rôle et le service accédé. Le programme 4.1 illustre la requête SQL utilisée pour vérifier l'existence d'une relation satisfaisant aux valeurs des paramètres de contrôle d'accès (utilisateur, rôle et service ciblé). Dans ce programme, les éléments `UserID`, `RoleID` et `ServiceID` sont respectivement les identificateurs de l'utilisateur, du rôle et du service ciblé passés au SGBD par la classe `SP`.

Dans cette thèse nos proposons deux méthodes pour l'implémentation du filtre de contrôle d'accès dynamique. Ces méthodes sont explorées dans les chapitres 5 et 6. La figure 4.19b (respectivement la figure 4.19c) est une instance de la métaclasse `Filter` obtenue par la méthode de la transformation (respectivement la méthode d'interprétation). Ce sont ces services Web qui effectuent le calcul effectif des décisions d'autorisation pour une politique de contrôle d'accès dynamique. Les opérations `Init` et `Stop` de ces services permettent respectivement de les démarrer et de les arrêter. L'opération `AR`, tout comme dans le cas du filtre de contrôle d'accès statique, calcule la décision d'autorisation. Les opérations `Commit` et `Rollback` mettent à jour l'état courant de la politique de contrôle d'accès dynamique. Ces différentes opérations sont détaillées dans les chapitres 5 et 6.

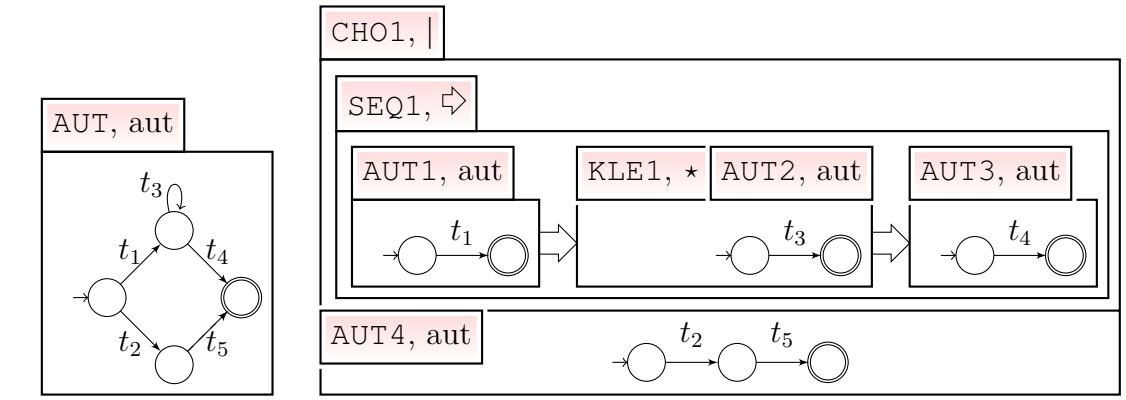
4.4 Conclusion

L'exemple d'un modèle de contrôle d'accès spécifique au domaine bancaire exposé à la section 4.3 montre comment il est aisé de définir un tel modèle lorsqu'on dispose d'un modèle générique d'un gestionnaire de contrôle d'accès exprimé sous la forme d'un métamodèle UML. Ce dernier facilite également la visualisation des interactions entre les différents composants du PEM. Dans la suite l'attention est portée principalement sur le filtre de contrôle d'accès dynamique dont l'implémentation est faite suivant deux méthodes distinctes.

Chapitre 5

Transformation d'une politique de contrôle d'accès ASTD en processus BPEL

La transformation d'une politique de contrôle d'accès (dynamique) ASTD en processus BPEL est faite en considérant des schémas génériques de code BPEL, un pour chaque structure ASTD. Chaque schéma est généralement présenté à l'aide de deux fragments de pseudocode BPEL et d'un diagramme de séquences UML pour faciliter la compréhension de l'exposé par le lecteur. Premièrement, chaque fragment de code est inséré dans l'algorithme comme résultat de la transformation de la structure ASTD correspondante. Deuxièmement, chaque diagramme de séquences remplace un fragment de pseudocode BPEL de façon à mettre clairement en relief les interactions entre l'activité BPEL d'une structure ASTD (parent) et les activités de ses sous-structures. La transformation inclut également des fonctions auxiliaires qui produisent les documents WSDL et XSD nécessaires pour l'exécution des processus dans un moteur d'exécution BPEL. L'ensemble des processus et documents constitue le filtre de contrôle d'accès dynamique (ou simplement filtre).



(a) Spécification initiale (b) Spécification prétraitée

Figure 5.1 – Prétraitement des cycles d'une spécification ASTD

5.1 Vue globale de l'algorithme de transformation

L'algorithme qui transforme une spécification ASTD d'une politique de contrôle d'accès en un ensemble de processus BPEL est récursif. L'algorithme crée les processus BPEL en même temps que les interfaces WSDL et les définitions de type XSD. Les interfaces WSDL, requises par le moteur d'exécution BPEL, décrivent les interactions avec les processus et permettent de les exposer comme des services Web. Les documents XSD décrivent les types des paramètres passés dans les messages échangés par les processus BPEL. La transformation est basée sur les événements dans le sens que les processus BPEL imitent les flots d'événements décrits par la spécification ASTD.

L'algorithme de transformation génère des processus BPEL pour des spécifications ASTD qui comportent certaines limitations. La première concerne les cycles dans les structures ASTD *automate*. En effet, la transformation d'une telle structure procède au parcours de ces transitions à l'instar du parcours d'un arbre. Dans l'éventualité où il est indispensable d'utiliser un cycle dans une structure *automate*, une étape de prétraitement de la spécification permet simplement de transformer le graphe de transition en utilisant les structures ASTD *choix*, *séquence* et *fermeture de Kleene*. L'exemple de la figure 5.1 illustre bien ceci. La figure 5.1a montre une spécification avec un cycle et la figure 5.1b un possible équivalent. L'équivalence entre les ex-

pressions régulières et les automates dans la théorie des langages assure qu'un tel prétraitement est toujours possible (voir [47]).

La seconde limitation concerne l'utilisation de macro-états dans les structures ASTD *automate*. Cette restriction empêche l'utilisation de transitions de type *from* et *to* dans les automates. Toutefois cette limitation n'est pas handicapante dans l'expression de règles de contrôle d'accès puisque les macro-états ne sont pas utilisés dans les patrons de règles.

5.2 Évolution de l'algorithme de transformation

L'algorithme décrit dans ce chapitre en est à sa troisième itération. La première, décrite dans [16], est étroitement liée aux patrons de règles de contrôle d'accès. Dans cette version, la transformation utilise un squelette d'activités BPEL spécifique pour chaque patron de règle. Ces squelettes sont ensuite rendus concrets par une exploration de la spécification ASTD vue comme un arbre. L'algorithme procède en deux phases. La première consiste en la transformation de la spécification en un arbre syntaxique abstrait annoté. Les annotations aux nœuds de l'arbre sont des résultats de calcul qui sont utilisés pour créer les activités BPEL dans la seconde phase.

L'itération suivante de l'algorithme de transformation a été publiée dans [15]. Elle permet d'éliminer l'étape du calcul de l'arbre syntaxique abstrait. De plus, elle précise mieux de façon algorithmique le passage des structures ASTD vers les activités BPEL, toujours dans le contexte des patrons des règles de contrôle d'accès. La troisième itération, présentée dans cette thèse, comporte moins de limitations que les précédentes. En effet cette transformation n'est plus restreinte aux seuls patrons de règles de contrôle d'accès, mais peut être appliquée à toute spécification ASTD pour autant que celle-ci se conforme aux restrictions exposées à la section 5.1.

5.3 Règles de transformation

Le résultat de la transformation est un ensemble de processus BPEL qui utilisent de façon intensive les liens de partenaire. Dans le cas de cette transformation, le mécanisme des liens est utilisé pour diriger les requêtes émises par un processus du filtre vers lui-même ou vers d'autres processus du filtre. Chaque structure d'une spécification ASTD est transformée en une activité **`flow`** (une activité BPEL qui exécute ses sous-activités en parallèle) avec les sous-activités suivantes.

- La première activité fonctionne comme un contrôleur pour la structure ASTD et implémente la sémantique opérationnelle décrite dans le rapport technique du langage ASTD [21]. Cette implémentation inclut, entre autres, la redirection de requêtes et le refus immédiat d'une requête si une condition n'est pas respectée.
- Les autres activités si elles existent, encodent la logique des sous-structures ASTD.

Le contrôleur dirige les requêtes vers le code BPEL correspondant pour la sous-structure ASTD à travers ses liens de partenaire. Au niveau de la structure ASTD, la transformation est effectuée par une fonction récursive `T`. Cette fonction retourne deux valeurs. La première valeur est une activité BPEL et la seconde un ensemble de sous-processus BPEL utilisés par cette activité.

5.3.1 Le contrôleur

Le *contrôleur* mentionné précédemment a la même forme pour toutes les structures ASTD. Les différences suivant la structure ASTD apparaissent dans les activités des branches de l'activité **`pick`** du contrôleur. L'algorithme qui crée ce contrôleur est listé dans le programme 5.1 et la figure 5.2 schématise son fonctionnement. Le paramètre $\mathtt{T}_{AR}$ de la routine décrite fournit la branche spécifique à la structure ASTD. Cette branche implémente des redirections des requêtes d'autorisation aux activités compagnons du contrôleur, ou même des réponses immédiates aux requêtes si cela est possible. La branche $\mathtt{T}_S$ permet de terminer l'activité liée à la structure ASTD elle-même. Cette branche est utile lorsque que l'état d'une structure ASTD ne permet plus d'accepter des actions (et donc toutes les requêtes suivantes se voient refuser

```
1  T^ctrl(astd, T_S, T_AR):
2   result ←
3   <repeatUntil>
4    <pick>
5     <@ T_S @>
6     <@ T_AR @>
7    </pick>
8    <condition>
9     RepeatVar<@Name(astd)@> = true()
10   </condition>
11  </repeatUntil>
```

Programme 5.1 – Algorithme de création du contrôleur BPEL

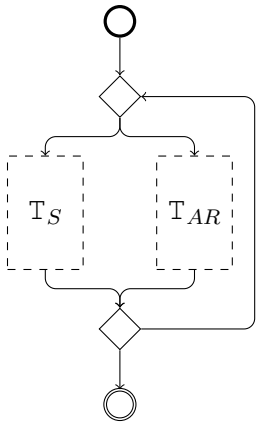


Figure 5.2 – Schéma du contrôleur

l'autorisation d'être exécutées). La variable de l'activité de répétition **repeatUntil** de la ligne 9 est positionnée à la valeur *true* par la branche T_S, pour empêcher le contrôleur d'attendre des requêtes pour des structures ASTD. Par exemple, lorsque l'état d'une structure ASTD *séquence* passe du premier composant au second composant, l'état du premier devient inutile et les ressources qui l'implémentent peuvent être alors libérées. Dans le cas de l'encodage BPEL d'une structure ASTD *séquence*, cette libération correspond à la terminaison de l'activité implémentant la première sous-structure de cette structure. Le traitement d'une requête pour l'opération `Stop` — sur l'activité BPEL encodant la première sous-structure — pour terminer l'activité de la première sous-structure, termine ses éventuelles sous-structures et arrête la boucle **repeatUntil** du contrôleur.

5.3.2 Les opérations des processus

Les processus BPEL issus de la transformation de spécifications ASTD implémentent l'opération `AR` (pour Authorization Request en anglais) décrite à la figure 4.19. Cette opération est de type requête/réponse. La requête SOAP de l'opération comporte comme paramètres l'action ciblée par la demande d'autorisation et la liste des paramètres de l'action (paramètres de contrôle d'accès et paramètres métiers). Le type de la requête est le même pour toutes les opérations correspondant à chaque structure ASTD encodée. La réponse SOAP, dont le type est lui aussi iden-

tique pour toutes les structures ASTD, contient la décision d'autorisation, c'est-à-dire *granted* ou *denied*.

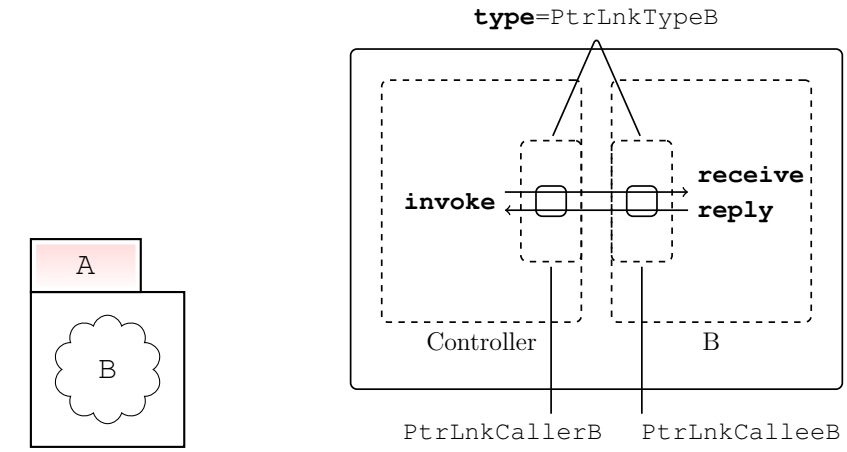
Les processus implémentent aussi l'opération `Commit`. Cette opération valide l'exécution par le système d'information de l'action autorisée par le filtre de contrôle d'accès. En utilisant cette opération, le composant PEP du PEM peut gérer les cas où une action autorisée par le PDP n'est pas exécutée par le système d'information. Lorsque cette opération est appelée avec la valeur *false*, l'instance de processus qui implémente la politique de contrôle d'accès revient dans l'état précédant la décision d'autorisation *granted* à laquelle est associée l'appel à `Commit`. Pour un appel avec la valeur *true*, l'état du processus avance pour attendre les prochaines requêtes d'autorisation.

Enfin, les processus BPEL implémentent l'opération `Stop`. Cette opération, de type requête, permet à l'activité BPEL d'une structure ASTD d'interrompre l'activité d'une sous-structure ASTD lorsque, par exemple, cette activité n'est plus nécessaire pour les décisions d'autorisation. Cette opération est offerte à la fois au composant PEP du PEM par le filtre et à toutes les activités *contrôleur* par les activités *contrôleur* imbriquées dans les processus BPEL qui encodent la politique de contrôle d'accès.

Pour permettre au composant PDP d'initialiser une instance du processus BPEL de plus haut niveau (celui du contrôleur de la structure ASTD principale), ce processus offre l'opération `Init`. L'appel à cette opération prépare une instance du processus pour qu'elle puisse recevoir les requêtes des opérations `AR`, `Commit` et `Stop`.

5.3.3 Les liens de partenaire

Les processus BPEL utilisent les liens de partenaire pour communiquer avec leurs pairs, que ceux-ci soient d'autres processus, des services Web ou les mêmes processus (dans le cas d'un appel d'opération *réflexif*). Le processus BPEL d'une structure ASTD peut recevoir des requêtes pour les opérations mentionnées dans la section précédente, à travers le canal des liens de partenaire. Lorsque le contrôleur BPEL d'une structure ASTD transmet une demande d'autorisation au contrôleur d'une sous-structure ASTD (par exemple une structure *séquence* dont l'état est dans la première



(a) Structures ASTD (b) Utilisation des liens de partenaire

Figure 5.3 – Exemple de liens de partenaire

sous-structure), cette transmission correspond à une activité **invoke**, qui utilise un lien de partenaire dans le contrôleur de la structure parent, et une paire d'activités **receive**/**reply** ou **onMessage**/**reply**, qui utilisent le même lien de partenaire dans le contrôleur de la sous-structure, ou plus simplement une activité **receive** ou **onMessage** si l'opération est à sens unique et n'attend pas de réponse. La figure 5.3a illustre les structures ASTD parent et enfant et la figure 5.3b montre l'échange par un lien de partenaire déclaré pour la sous-structure. La déclaration du type de lien de partenaire dans le fichier des interfaces WSDL est explicité dans le programme 5.2. Son utilisation dans les contrôleurs pour les structures parent (appel) et enfant (réponse) sont illustrés respectivement dans les programmes 5.3 et 5.4.

Dans l'algorithme de transformation, les liens de partenaire pour les contrôleurs *appelants* (respectivement *appelés*) sont déclarés par la fonction `PtrLinkCaller`

```
1  <partnerLinkType name="PtrLnkTypeB">
2   <role name="RoleB" portType="···"/>
3  </partnerLinkType >
```

Programme 5.2 – Déclaration d'un type de lien

```
1  ···
2  <partnerLink name="PtrLnkCallerB"
    partnerLinkType="PtrLnkTypeB"
    partnerRole="RoleB" ···/>
3  ···
4   <invoke operation="AR"
     partnerLink="PtrLnkCallerB" ···/>
5  ···
```

Programme 5.3 – Utilisation d'un lien de partenaire dans le contrôleur parent

```
1  ···
2  <partnerLink name="PtrLnkCalleeB"
    partnerLinkType="PtrLnkTypeB"
    myRole="RoleB" ···/>
3  ···
4   <receive operation="AR"
     partnerLink="PtrLnkCalleeB" ···/>
5  ···
6   <reply operation="AR"
     partnerLink="PtrLnkCalleeB" ···/>
7  ···
```

Programme 5.4 – Utilisation d'un lien de partenaire dans le contrôleur enfant

(respectivement `PtrLnkCallee`). Le programme 5.5 explicite ces fonctions, ainsi que les fonctions qui fournissent les noms de ces liens de partenaire, permettant leur utilisation dans les activités **`receive`**, **`onMessage`**, **`reply`** et **`invoke`**. La fonction `Root`, définie aux lignes 1 à 5, retourne la chaîne de caractères `'Root'` ou la chaîne vide (`''`) selon que transformation de la structure ASTD passée en paramètre donne lieu à un processus BPEL. C'est le cas lorsque la structure en question est la structure principale de la spécification ou que celle-ci a pour structure parent une structure *ap-*

```
1  Root(astd):
2   if astd has no parent or
3     has Call, Kleene or Quantified synchronization as parent
4    then result ← 'Root'
5    else result ← ''

7  PtrLinkCalleeName(astd):
8   result ← <@Name(astd)@>PtrLnkCallee

10 PtrLinkCallee(astd):
11  result ←
12   <partnerLink name="<@PtrLinkCalleeName(astd)@>"
       partnerLinkType="<@Root(astd)@>ASTDPartnerLinkType"
       myRole="<@Root(astd)@>ASTDRole" />

14 PtrLinkCallerName(astd):
15  result ← <@Name(astd)@>PtrLnkCaller

17 PtrLinkCaller(astd):
18  result ←
19   <partnerLink name="<@PtrLinkCallerName(astd)@>"
       partnerLinkType="<@Root(astd)@>ASTDPartnerLinkType"
       partnerRole="<@Root(astd)@>ASTDRole" />
```

Programme 5.5 – Fonctions de création des liens de partenaire

```
1  T_spec^BPEL(spec)
2   result ← T^proc(main)
```

Programme 5.6 – Transformation d'une spécification

pel, *fermeture de Kleene* ou *synchronisation quantifiée*. Le résultat de la fonction `Root` est utilisé comme préfixe à concaténer à la chaîne `ASTDPartnerLinkType` (respectivement la chaîne `ASTDRole`) pour former le nom du type de lien de partenaire (respectivement le nom du rôle joué par le processus qui déclare le lien) dans la fonction `PtrLnkCallee` à la ligne 12 et la fonction `PtrLnkCaller` à la ligne 19 du programme 5.5. Les types de lien de partenaire utilisés par les fonctions `PtrLnkCallee` et `PtrLnkCaller` aux lignes 12 et 19 sont définis dans le document WSDL associé aux processus BPEL (voir la section 5.6).

5.4 Transformation d'une spécification ASTD

Une spécification ASTD est une liste de structures ASTD dont l'une est marquée *main*. La fonction $\mathtt{T}_{spec}^{BPEL}$ transforme la spécification en un ensemble de processus BPEL. Elle est décrite dans le programme 5.6. La transformation est initiée depuis la structure ASTD *main* et utilise la fonction $\mathtt{T}^{proc}$ décrite dans le programme 5.7.

5.4.1 Notation

Cette section décrit le style utilisé dans la présentation des fonctions de l'algorithme de transformation. Les fonctions utilisent la variable `result` pour retourner le résultat de leurs calculs. Dans certains cas, la valeur contenue dans cette variable est le code BPEL, WSDL ou XSD à créer. Dans d'autres cas, la variable est un enregistrement à plusieurs champs qui peuvent être accédés individuellement en utilisant le point (« . ») et le nom du champ pour effectuer une indirection à la Java.

Une situation récurrente dans les fonctions décrites dans la suite est l'évaluation d'expressions pendant la création d'un programme BPEL, WSDL ou XSD. Une telle expression peut être un appel de fonction, une expression conditionnelle, une variable

```
1  T^proc(astd):
2   result_astd ← T(astd)
3   result ← {
4    <process name="<@Name(astd)@>">
5     <partnerLinks><@PtrLnkCallee(astd)@></partnerLinks>
6     <correlationSets>
7      <correlationSet name="corrSetPId" properties="PId"/>
8     </correlationSets>
9     <scope>
10     <variables>
11      <variable name="InitIn" messageType="InitRequest"/>
12     </variables>
13     <sequence>
14      <receive partnerLink="PtrLnkCalleeName(astd)"
         operation="Init" variable="InitIn"/>
15      <@ result_astd.activity @>
16      <exit/>
17     </sequence>
18    </scope>
19   </process>
20  }
21  ∪ result_astd.processes
```

Programme 5.7 – Transformation d'une structure ASTD

précédemment valuée ou encore du texte libre. Dans ces situations particulières, les expressions concernées sont marquées par les balises spéciales « <@ » et « @> ».

Les éléments BPEL **receive** et **onMessage** permettent de recevoir les requêtes d'opérations. L'élément **invoke** quant à lui permet d'envoyer la requête puis de recevoir la réponse d'une opération. Ces différents éléments sont utilisés dans la suite par les activités générées par l'algorithme de transformation. Le langage BPEL utilise le mécanisme de *corrélation* pour identifier l'instance d'un processus qui doit recevoir ou répondre à un message. Dans la suite de ce chapitre, les éléments **receive** et **onMessage** sont associés à l'ensemble de corrélation `corrSetPId` défini dans le programme 5.7 (lignes 6 à 8). Cette association n'apparaîtra pas dans les prochains programmes pour des raisons de concision.

5.4.2 Transformation de structures ASTD en processus BPEL

Le résultat de la fonction $\mathtt{T}^{proc}$ est un ensemble de processus BPEL, constitué du processus pour la structure ASTD passée en paramètre et des processus BPEL issus des structures ASTD imbriquées. Le programme 5.7 explicite cette fonction.

```
1  T(⟨| x : T, b⟩) :
2   result_b ← T(b)
3   result.activity ←
4    <scope>
5     <partnerLinks>
6      <@ PtrLnkCallee(astd) @>
7      <@ PtrLnkCaller(b) @>
8     </partnerLinks>
9     <flow>
10     <@ T^ctrl(astd, T_S^qch(astd,b),
         T_AR^qch(b)) @>
11     <@ result_b.activity @>
12    </flow>
13   </scope>
14  result.processes ← result_b.processes
```

Programme 5.8 – Transformation de la structure *choix quantifié*

Pour créer le processus BPEL, la fonction `T` est utilisée pour transformer la structure ASTD en une paire composée d'une activité BPEL et d'un ensemble de processus BPEL (ligne 2). Le premier membre de la paire, l'activité BPEL correspondant à la structure ASTD et identifié par le terme $\texttt{result}_{astd}\texttt{.activity}$, est utilisé à la ligne 15 dans le squelette de création du processus BPEL correspondant à la structure ASTD (lignes 4 à 19). Le second membre de la paire, l'ensemble de sous-processus BPEL issus de la transformation de la structure ASTD et identifié par le terme $\texttt{result}_{astd}\texttt{.processes}$, est uni à la ligne 21 au processus BPEL nouvellement créé pour former le résultat de la fonction $\texttt{T}^{proc}$.

À la ligne 5 du programme 5.7, la fonction $\texttt{T}^{proc}$ déclare le lien de partenaire par lequel les requêtes sont acceptées par le processus, ce lien de partenaire étant créé par la fonction `PtrLnkCallee`. Le même lien de partenaire, par son nom, est utilisé pour recevoir les requêtes de l'opération `Init` à la ligne 14. À la ligne 7, le processus BPEL créé définit un ensemble de corrélations utilisé pour diriger correctement les requêtes SOAP reçues de l'environnement. Cet ensemble de corrélations (élément BPEL **`correlationSet`**), appelé `corrSetPId`, utilise l'unique propriété `PId` qui identifie le processus BPEL.

5.4.3 Transformation de la structure ASTD *choix quantifié*

Le programme 5.8 décrit la transformation d'une structure ASTD *choix quantifié*. Comme expliqué à la section 1.1, le *choix quantifié* permet l'exécution de sa sous-structure, nommé b dans le programme, pour une valeur spécifique d'un ensemble de quantification. Le résultat de la transformation est une activité qui exécute en parallèle (activité BPEL **`flow`**) un contrôleur pour la structure ASTD et l'activité

```
1  T_S^qch(astd,b):
2   pl ← PtrLnkCalleeName(astd)
3   pl_b ← PtrLnkCallerName(b)
4   result ←
5    <onMessage partnerLink="pl" operation="Stop" variable="StopIn">
6     <sequence>
7      <assign><copy>
8       <from>true()</from><to>RepeatVar<@Name(astd)@></to>
9      </copy></assign>
10     <if>
11      <condition><@ compound ASTD b is not completed @></condition>
12      <invoke partnerLink="pl_b" operation="Stop" variable="StopIn"/>
13     </if>
14    </sequence>
15   </onMessage>
```

Programme 5.9 – Branche `Stop` du contrôleur du *choix quantifié*

issue de la transformation de la sous-structure du choix quantifié. Le contrôleur est implémenté avec des branches `Stop` et `AR` spécifiques aux structures ASTD *choix quantifié*.

L'algorithme de création des activités de la branche `Stop` ($\mathtt{T}_S^{qch}$) est illustré par le programme 5.9. Un élément **`onMessage`** permet au processus BPEL conteneur d'attendre l'arrivée d'une requête pour l'opération `Stop` (ligne 5). La réception d'une telle requête déclenche le positionnement à *true* de la variable de l'activité **`repeatUntil`** du contrôleur (ligne 8). Avec cette nouvelle valeur, le contrôleur s'arrête donc d'itérer après l'exécution de sa dernière activité. Cette dernière activité est l'arrêt de l'activité résultant de la transformation de la sous-structure b (ligne 12) sous la condition que celle-ci ne soit pas déjà arrêtée.

La branche `AR` utilise les règles de la sémantique opérationnelle du langage ASTD et sa logique est décrite à la figure 5.4. Le contrôleur reçoit la requête d'autorisation de la structure ASTD parent ou du composant PDP parent si la structure *choix quantifié* est la structure principale (elle n'a donc pas de structure parent) de la spécification formelle. Puisque la structure courante est *quantifiée*, la variable de quantification doit recevoir une valeur avant que le traitement de la requête `AR` se poursuive. L'activité teste si une valeur pour cette variable est déjà enregistrée dans l'état de l'instance du processus BPEL conteneur. Si aucune valeur n'est enregistrée, l'activité BPEL

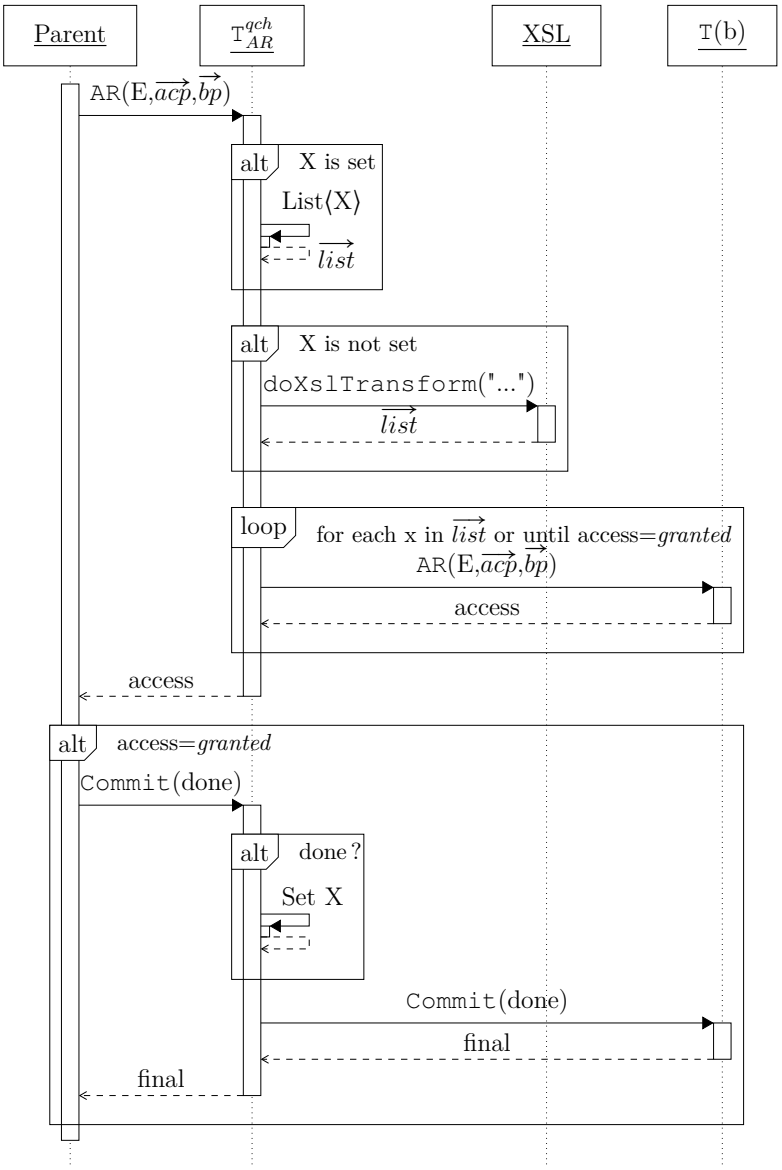


Figure 5.4 – Branche T^{qch}_{AR} de la structure *choix quantifié*

utilise l'opération `doXslTransform` pour obtenir la liste des valeurs possibles pour la variable à partir du type spécifié pour la variable de quantification. Si une valeur est déjà enregistrée, alors celle-ci est utilisée pour constituer une liste à un élément. Par la suite, toutes les valeurs de la liste constituée sont explorées les unes après les autres,

```
1  T(⟨⋆, b⟩):
2   result.activity ←
3    <scope>
4     <partnerLinks>
5      <@ PtrLnkCallee(astd) @>
6      <@ PtrLnkCaller(b) @>
7     </partnerLinks>
8     <@ T^ctrl(astd, T_S^kl(astd,b), T_AR^kl(astd)) @>
9    </scope>
10   result.processes ← T^proc(b)
```

Programme 5.10 – Transformation de la structure *fermeture de Kleene*

chaque valeur étant placée dans l'environnement et la requête `AR` est transmise à l'activité de la sous-structure du *choix quantifié*. L'exploration de la liste des valeurs s'arrête si pour l'une des valeurs la décision retournée depuis l'activité de la sous-structure est *granted*.

5.4.4 Transformation de la structure ASTD *fermeture de Kleene*

La transformation d'une structure *fermeture de Kleene* donne lieu à la création d'un processus BPEL séparé, en plus de l'activité BPEL du contrôleur. Le sous-processus ainsi créé correspond dans son essence à la transformation de la sous-structure de la *fermeture de Kleene*. Le contrôleur communique avec le processus à travers les liens de partenaire définis. Le programme 5.10 illustre la transformation d'une structure *fermeture de Kleene*. La ligne 8 crée l'activité de contrôleur avec les branches `Stop` et `AR`. La ligne 10 crée le sous-processus de la sous-structure.

Le programme 5.11 illustre la branche `Stop` de la structure ASTD *fermeture de Kleene*. Le code créé étant directement inclus dans une activité BPEL **`pick`** (voir la ligne 5 du programme 5.1). Il débute donc avec un élément **`onMessage`** qui permet de recevoir les requêtes de l'opération `Stop`. À l'intérieur de cet élément, le code créé positionne, à la valeur *true*, la variable de la condition de l'activité de répétition **`repeatUntil`** du contrôleur parent (ligne 9). Dans la même activité **`assign`**, le code créé change la valeur de l'identificateur de processus `PId` dans la variable `StopIn` (ligne 12). Cette valeur de l'identificateur de processus, combinée au lien de partenaire

```
1  T_S^kl(astd,b):
2   pl ← PtrLnkCalleeName(astd)
3   pl_b ← PtrLnkCallerName(b)
4   result ←
5    <onMessage partnerLink="pl" operation="Stop" variable="StopIn">
6     <sequence>
7      <assign>
8       <copy>
9        <from>true()</from><to>RepeatVar<@Name(astd)@></to>
10      </copy>
11      <copy>
12       <from><@Current sub-process PId@></from><to>$StopIn.PId</to>
13      </copy>
14     </assign>
15     <if>
16      <condition><@ compound ASTD b is not completed @></condition>
17      <invoke partnerLink="pl_b" operation="Stop" variable="StopIn"/>
18     </if>
19    </sequence>
20   </onMessage>
```

Programme 5.11 – Branche `Stop` du contrôleur de la *fermeture de Kleene*

$\texttt{pl}_b$, permet d'envoyer une requête pour l'opération `Stop` au sous-processus BPEL qui encode la sous-structure ASTD (ligne 17).

La branche $\texttt{T}_{AR}^{kl}$ du contrôleur d'une structure *fermeture de Kleene* est décrite par la séquence de la figure 5.5. Cette séquence suit les règles sémantiques de la structure ASTD *fermeture de Kleene*. Quand l'activité du contrôleur dans l'état courant du processus BPEL parent reçoit une requête `AR`, cette requête est transmise à l'instance courante du sous-processus BPEL. Si la réponse retournée par l'instance du sous-processus contient la décision d'autorisation *granted*, alors cette décision est retransmise à l'activité ou processus parent du contrôleur. Dans le cas contraire, la branche $\texttt{T}_{AR}^{kl}$ vérifie le statut de l'instance du sous-processus. Si cette instance du sous-processus est dans l'état *final*, la branche $\texttt{T}_{AR}^{kl}$ du contrôleur crée une autre instance du sous-processus en utilisant l'opération `Init` avec une valeur de la propriété `PId` différente de celle utilisée pour l'instance courante. Cette nouvelle instance créée encode l'*état initial* de la sous-structure ASTD correspondante. La requête `AR` est transmise à cette nouvelle instance du sous-processus, et la réponse retournée par la nouvelle instance du sous-processus est retransmise à l'activité ou processus parent du contrôleur de la structure *fermeture de Kleene*.

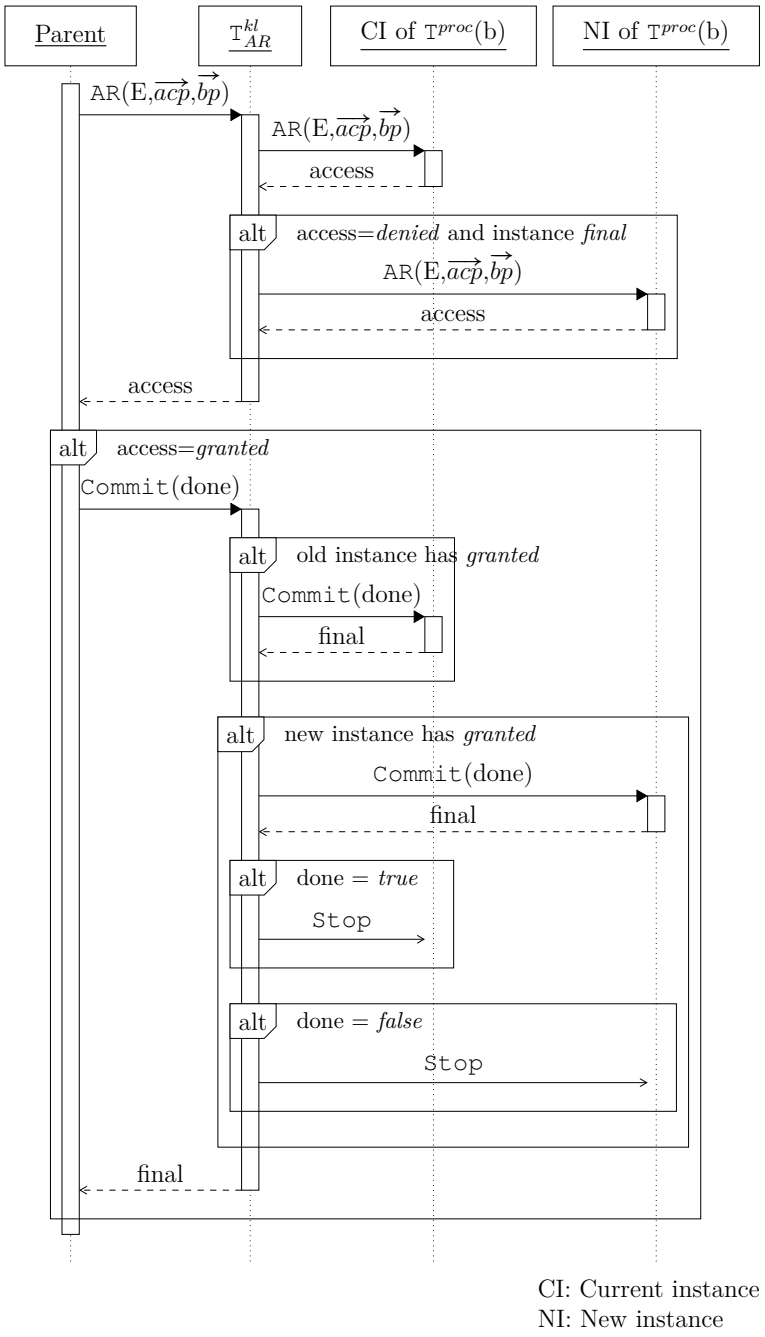


Figure 5.5 – Branche T^{kl}_{AR} de la structure *fermeture de Kleene*

```
1  T(⟨⇒, p, b⟩):
2   result_b ← T(b)
3   result.activity ←
4    <scope>
5     <partnerLinks>
6      <@ PtrLnkCallee(astd) @>
7      <@ PtrLnkCaller(b) @>
8     </partnerLinks>
9     <flow>
10     <@ T^ctrl(astd, T_S^gr(astd,b), T_AR^gr(astd)) @>
11     <@ result_b.activity @>
12    </flow>
13   </scope>
14  result.processes ← result_b.processes
```

Programme 5.12 – Transformation de la structure *garde*

Dans le cas où la réponse retournée à l'activité ou processus parent du contrôleur de la structure *fermeture de Kleene* contient la décision d'autorisation *granted*, cette activité ou processus parent envoie la requête `Commit` au contrôleur pour indiquer (suivant la valeur d'une variable booléenne contenue dans la requête) si la requête initiale de l'utilisateur a été exécutée avec succès par le système d'information. La réponse à cette requête est une variable d'état qui indique le statut *final* de l'état de la structure ASTD encodée par l'activité qui répond à l'opération `Commit`. Dans l'éventualité où la décision d'autorisation *granted* a été retournée par la nouvelle instance du sous-processus BPEL de la sous-structure ASTD et que le service ciblé a bien exécuté la requête initiale de l'utilisateur (`done` a la valeur *true* dans la requête `Commit`), l'instance courante du sous-processus est terminée avec l'opération `Stop` et la nouvelle instance est conservée pour les requêtes futures de l'opération `AR`.

5.4.5 Transformation de la structure ASTD *garde*

La transformation d'une structure ASTD *garde* résulte en une activité BPEL qui exécute en parallèle un contrôleur configuré avec les branches $\mathtt{T}_S^{gr}$ et $\mathtt{T}_{AR}^{gr}$ et la sous-activité issue de la transformation de la sous-structure de la garde. Le programme 5.12 montre la transformation de la structure ASTD *garde*. La branche $\mathtt{T}_S^{gr}$ est dans ce cas identique à celle de la structure ASTD *choix quantifié* explicitée dans le programme 5.9.

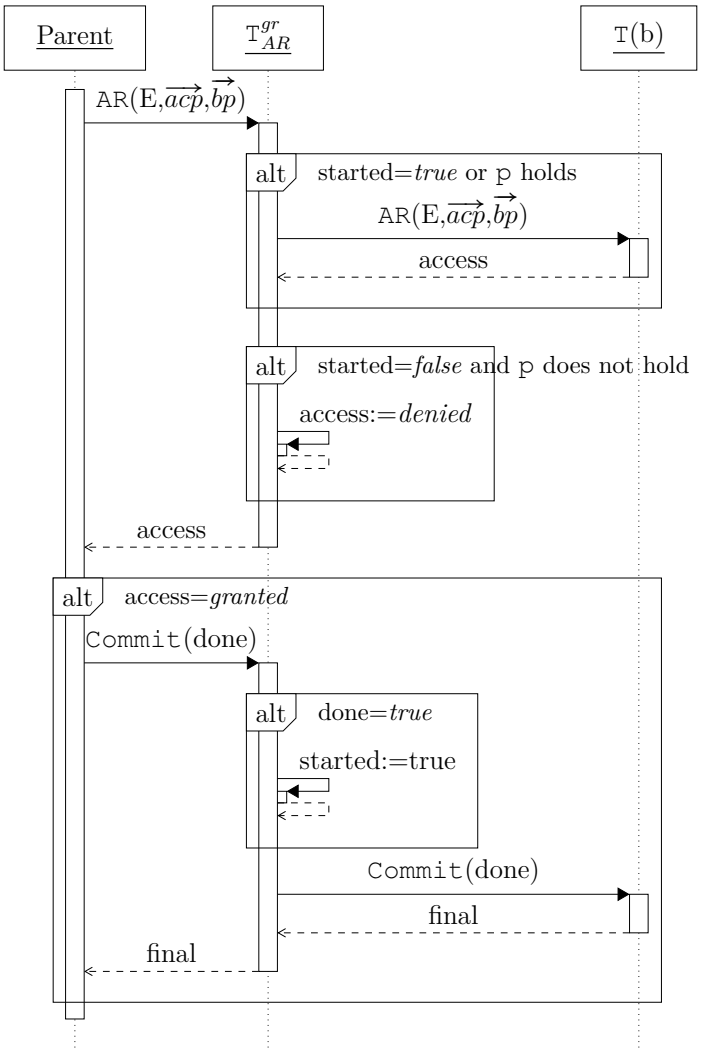


Figure 5.6 – Branche T^{gr}_{AR} de la structure *garde*

La figure 5.6 illustre le fonctionnement de l'activité T^{gr}_{AR}. La branche T^{gr}_{AR} ajoute une variable `started` à l'état de l'instance du processus BPEL conteneur. Cette variable est initialisée à la valeur *false* et indique si le prédicat de la garde a été testé. Lorsque la branche T^{gr}_{AR} reçoit la requête de l'opération AR, le prédicat de la garde est évalué dans l'environnement courant si la variable `started` a la valeur *false*. Dans le cas où l'évaluation du prédicat est positive, la requête AR est transmise à l'activité de la sous-structure de la garde. La décision retournée par l'activité de la

```
1  T(⟨⇨, f, s⟩):
2   result_f ← T(f)
3   result_s ← T(s)
4   result.activity ←
5    <scope>
6     <partnerLinks>
7      <@ PtrLnkCallee(astd) @>
8      <@ PtrLnkCaller(f) @>
9      <@ PtrLnkCaller(s) @>
10    </partnerLinks>
11    <variables><variable name="Side" type="SequenceSide"/></variables>
12    <flow>
13     <@ T^ctrl(astd, T_S^sq(astd,f,s), T_AR^sq(astd)) @>
14     <@ result_f.activity @>
15     <@ result_s.activity @>
16    </flow>
17   </scope>
18  result.processes ← result_f.processes ∪ result_s.processes
```

Programme 5.13 – Transformation de la structure *séquence*

sous-structure est retransmise à l'activité ou processus parent de l'activité contrôleur de la garde. Si cette décision est *granted*, l'activité ou processus parent envoie la requête `Commit` au contrôleur de la garde pour indiquer si la requête initiale a été exécutée par le système d'information. Dans le cas où cette requête a été en effet exécutée, la variable `started` est positionnée à la valeur *true*.

5.4.6 Transformation de la structure ASTD *séquence*

Une structure ASTD *séquence* contient deux sous-structures ASTD. La sémantique d'exécution de cette structure est d'exécuter d'abord les événements de la première sous-structure, ensuite les événements de la seconde dès que la première sous-structure est dans un état final. Le programme 5.13 illustre la transformation d'une structure ASTD *séquence*. L'activité résultant de la transformation est une activité qui exécute en parallèle un contrôleur pour la *séquence* et des activités pour chaque sous-structure de cette *séquence*. La variable `Side` à la ligne 11 indique à quelle activité les requêtes de l'opération `AR` sont transmises lorsqu'elles sont reçues par le contrôleur de la séquence. Son type `SequenceSide` est défini dans un fichier de types XSD lié au processus conteneur (voir la section 5.5).

```
1  T_S^sq(astd,f,s):
2   pl ← PtrLnkCalleeName(astd)
3   pl_f ← PtrLnkCallerName(f)
4   pl_s ← PtrLnkCallerName(s)
5   result ←
6    <onMessage partnerLink="pl" operation="Stop" variable="StopIn">
7     <sequence>
8      <assign><copy>
9       <from>true()</from><to>RepeatVar<@Name(astd)@></to>
10     </copy></assign>
11     <flow>
12      <if>
13       <condition><@ compound ASTD f is not completed @></condition>
14       <invoke partnerLink="pl_f" operation="Stop" variable="StopIn"/>
15      </if>
16      <if>
17       <condition><@ compound ASTD s is not completed @></condition>
18       <invoke partnerLink="pl_s" operation="Stop" variable="StopIn"/>
19      </if>
20     </flow>
21    </sequence>
22   </onMessage>
```

Programme 5.14 – Branche `Stop` du contrôleur de la *séquence*

La branche $\mathtt{T}_S^{sq}$ du contrôleur pour la structure ASTD *séquence* est semblable à celle de la structure ASTD *garde* ou encore celle de la structure *choix quantifié*. La différence apparente provient du fait que la structure *séquence* possède deux sous-structures, alors que les structures *garde* et *choix quantifié* n'en n'ont qu'une seule. Aux lignes 14 et 18 du programme 5.14, le processus BPEL arrête les sous-structures f et s. Ces activités **`invoke`** sont dirigées correctement par l'utilisation des liens de partenaire $\mathtt{pl}_f$ et $\mathtt{pl}_s$ créés respectivement aux lignes 3 et 4 par la fonction `PtrLnkCallerName`. L'exécution de l'envoi des requêtes de l'opération `Stop` se fait toujours sous la condition que l'exécution des activités BPEL correspondant aux sous-structures n'est pas terminée.

La figure 5.7 illustre le fonctionnement de la branche $\mathtt{T}_{AR}^{sq}$. La requête de l'opération `AR` reçue de l'activité ou processus parent est transmise à l'activité de la première sous-structure si la valeur de la variable `Side` est *first*. Si la réponse retournée par cette sous-structure est *denied* et que l'état de cette activité est *final*, la requête est retransmise à l'activité de la deuxième sous-structure. Dans le cas où la valeur de la variable `Side` est *second*, la requête `AR` reçue du parent par le contrôleur est

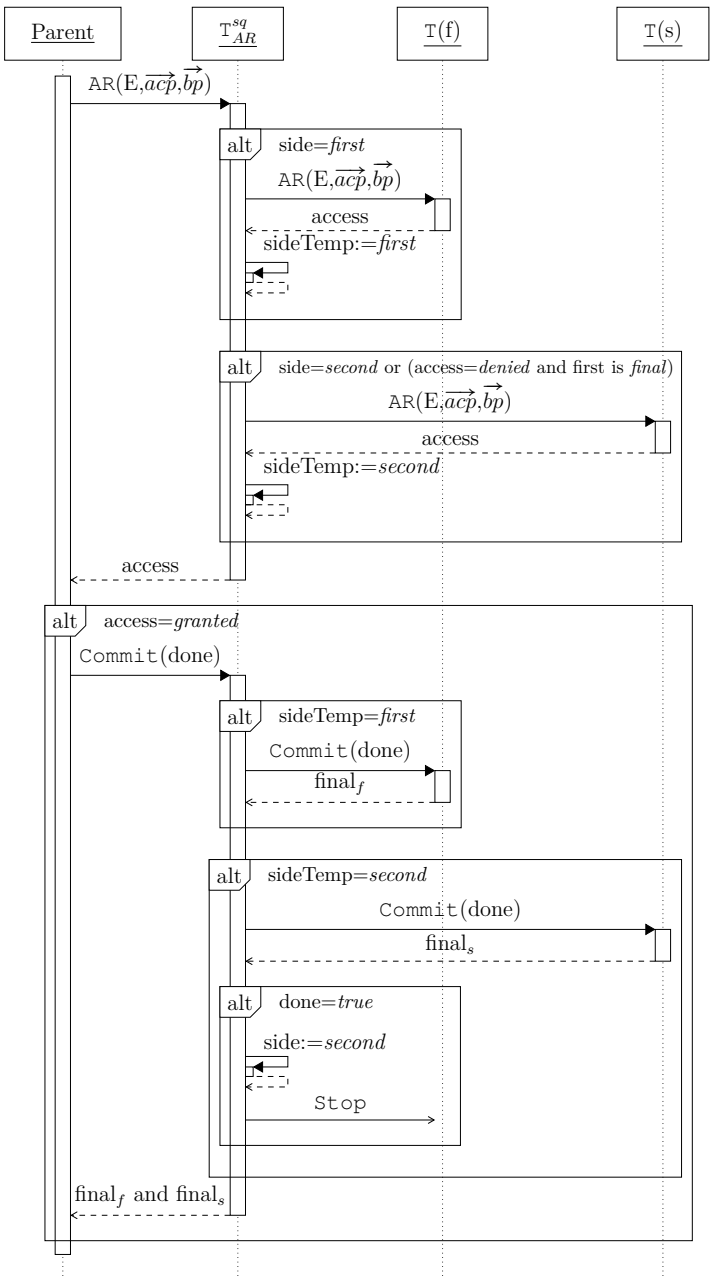


Figure 5.7 – Branche $\mathtt{T}^{sq}_{AR}$ de la structure *séquence*

transmise à l'activité de la deuxième sous-structure. Comme pour les structures ASTD précédentes, si la décision d'autorisation retournée à l'activité ou le processus parent

```
1  T(⟨| [] |, {E1, ···, En}, l, r⟩) :
2   result_l ← T(l)
3   result_r ← T(r)
4   result.activity ←
5    <scope>
6     <partnerLinks>
7      <@ PtrLnkCallee(astd) @>
8      <@ PtrLnkCaller(l) @>
9      <@ PtrLnkCaller(r) @>
10    </partnerLinks>
11    <flow>
12     <@ T^ctrl(astd, T_S^sy(astd,l,r),
        T_AR^sy(astd)) @>
13     <@ result_l.activity @>
14     <@ result_r.activity @>
15    </flow>
16   </scope>
17  result.processes ←
      result_l.processes
     ∪ result_r.processes
```

Programme 5.15 – Transformation de la structure *synchronisation paramétrée*

est *granted* alors ce parent envoie la requête `Commit` pour indiquer si la requête initiale a été exécutée par le système d'information. Pendant le traitement de cette requête, la branche $\mathtt{T}_{AR}^{sq}$ du contrôleur de la séquence met à jour la variable `Side` si la valeur courante est *first* et que la décision d'autorisation *granted* est rendue par l'activité de la première sous-structure. Le contrôleur retourne au parent le statut final de la séquence, qui est la conjonction des statuts des activités des deux sous-structures.

5.4.7 Transformation de la structure ASTD *synchronisation paramétrée*

La transformation d'une structure ASTD *synchronisation paramétrée* est semblable au squelette de l'activité BPEL créée par la transformation d'une structure ASTD *séquence*. La différence principale réside dans la branche $\mathtt{T}_{AR}^{sy}$. La branche $\mathtt{T}_{S}^{sy}$ n'est pas explicitée dans le cas de cette structure car elle est identique à la branche $\mathtt{T}_{S}^{sq}$ de la structure ASTD *séquence* décrite à la section 5.4.6. Le programme 5.15 illustre la transformation d'une structure ASTD *synchronisation paramétrée* en une activité BPEL et la figure 5.8 montre le fonctionnement de la branche $\mathtt{T}_{AR}^{sy}$ du contrôleur de la structure ASTD. L'activité résultant de la transformation exécute en parallèle le contrôleur et les activités des sous-structures ASTD gauche et droite. La branche $\mathtt{T}_{AR}^{sy}$ retransmet la requête de l'opération `AR` à ces deux activités lorsque le service ciblé par la requête initiée par l'utilisateur fait partie de l'ensemble de synchronisation de la structure ASTD *synchronisation paramétrée*.

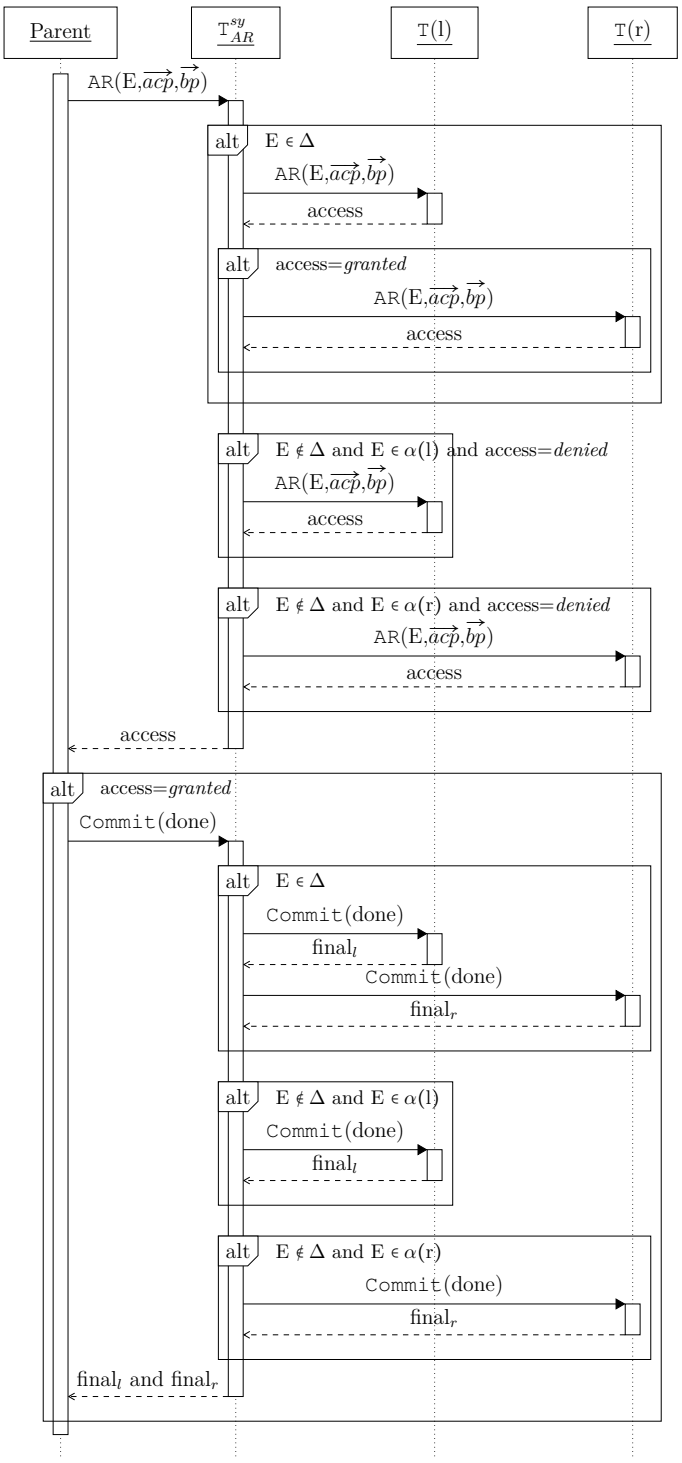


Figure 5.8 – Branche $\mathtt{T}^{sy}_{AR}$ de la structure *synchronisation paramétrée*

```
1  T(⟨aut, Σ, N, ν, δ, F, q0⟩) :
2   result.activity ←
3    <scope>
4     <partnerLinks>
5      <@ PtrLnkCallee(astd) @>
6     </partnerLinks>
7     Tau (q0, ⟨aut, Σ, N, ν, δ, F, q0⟩)
8    </scope>
9   result.processes ← {}
```

Programme 5.16 – Transformation de la structure *automate*

5.4.8 Transformation de la structure ASTD *automate*

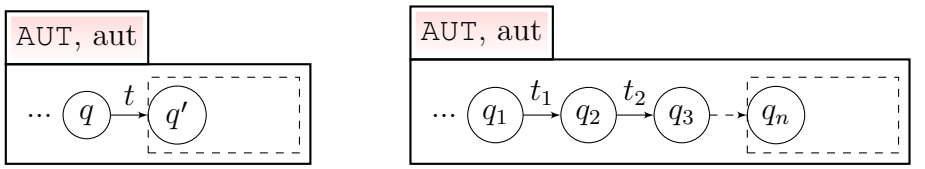
La transformation d'une structure ASTD *automate* s'effectue de façon récursive, en débutant à l'état initial de l'*automate*. L'algorithme de transformation produit du code BPEL qui pour chaque transition de l'automate compare les valeurs de la requête d'autorisation aux valeurs trouvées dans la spécification. La fonction T du programme 5.16 initie la transformation de la structure ASTD *automate* par l'état initial (ligne 7). La procédure T ne crée pas de sous-processus pour l'automate (ligne 9).

La procédure $\mathtt{T}_{au}$ utilisée à la ligne 7 du programme 5.16 est définie suivant trois cas qui peuvent survenir dans une structure *automate*.

Premier cas : l'état courant a exactement une transition sortante

Dans ce cas illustré à la figure 5.9a, l'état courant de l'*automate* traité par la procédure $\mathtt{T}_{au}$ a exactement une transition sortante. Le code BPEL créé est une activité BPEL **sequence** qui exécute en séquence, tout d'abord, l'activité correspondant au test de la transition t sortant de l'état, puis l'activité résultant de la transformation à partir de l'état destination q' de cette transition. Le programme 5.17 qui traite ce cas montre la séquence BPEL créée avec les deux activités (générées par des appels de fonction) aux lignes 4 et 5.

Ce cas peut être étendu pour inclure les chemins les plus longs d'états successifs possédant chacun une unique transition sortante, comme l'illustre la figure 5.9b. Dans cette éventualité, le programme 5.18 constitue une meilleure approche, car elle évite les activités BPEL **sequence** imbriquées multiples. À la ligne 4 du programme 5.17



(a) Cas d'un état avec une seule transition sortante

(b) Cas d'états successifs chacun avec une seule transition sortante

Figure 5.9 – Cas 1 : transition sortante unique

et aux lignes 4 à 6 du programme 5.18, la fonction $\mathtt{T}_t^1$ est utilisée. Cette fonction crée l'activité nécessaire au traitement de la réception et de la réponse à une demande d'autorisation, incluant la répétition dans le cas où la décision rendue est *denied*. Les détails de cette fonction sont explicités plus loin.

Deuxième cas : l'état courant a plus d'une transition sortante

Ce cas est illustré par la figure 5.10. Le code BPEL pourrait être une activité **`pick`** qui permet au moteur BPEL d'attendre l'arrivée des requêtes d'autorisation pour chacune des transitions partant de l'état courant q. Cependant l'opération AR est générique pour les événements dans la spécification, dans ce sens que l'événement est un paramètre de la requête, tout comme les paramètres de contrôle d'accès et les paramètres métiers sont des paramètres de l'événement. Or les branches de l'activité BPEL **`pick`** sont utilisées chacune avec une opération différente des opérations des autres branches, ce qui permet au moteur d'exécution de diriger correctement les requêtes reçues. L'approche choisie est plutôt une activité **`receive`** qui reçoit la

```
1  T_au(q,automaton):
2   result ←
3    <sequence>
4     <@ T_t^1(q,t,q',automaton) @>
5     <@ T_au(q',automaton) @>
6    </sequence>
```

Programme 5.17 – Cas d'un état avec une seule transition sortante

```
1  T_au(q1,automaton):
2   result ←
3    <sequence>
4     <@ T_t^1(q1,t1,q2,automaton) @>
5     ···
6     <@ T_t^1(qn−1,tn,qn,automaton) @>
7     <@ T_au(qn,automaton) @>
8    </sequence>
```

Programme 5.18 – Cas d'états successifs chacun avec une seule transition sortante

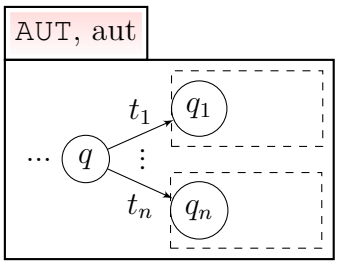


Figure 5.10 – Cas 2 : plusieurs transitions sortantes

```
1  Tau(q,automaton):
2   result ←
3    T2t(
4      q,
5      [(t1,q1); ...;(tn,qn)],
6      automaton
7    )
```

Programme 5.19 – Cas d'un état avec plusieurs transitions sortantes

requête d'autorisation. Les paramètres de la requête (service ciblé par l'utilisateur, ses paramètres de contrôle d'accès et ses paramètres métiers) sont comparés tour à tour aux différentes valeurs des transitions partant de l'état q. La génération des activités correspondant à cette approche est faite par la fonction $\mathtt{T}_t^2$ explicitée plus loin. Le programme 5.19 utilise cette fonction aux lignes 3 à 7.

Troisième cas : l'état courant n'a aucune transition sortante

L'état doit être *final* pour que la spécification soit cohérente, mais ce n'est pas requis par l'algorithme de transformation. Ce cas est illustré à la figure 5.11. Le code créé est simplement une activité **`empty`** (voir le programme 5.20).

Fonctions $\mathtt{T}_t^1$ et $\mathtt{T}_t^2$

Les fonctions $\mathtt{T}_t^1$ et $\mathtt{T}_t^2$ sont utilisées par la fonction $\mathtt{T}_{au}$ lorsque l'état courant a au moins une transition. Ces deux fonctions (voir le programme 5.21 pour la fonction $\mathtt{T}_t^1$ et le programme 5.22 pour la fonction $\mathtt{T}_t^2$) acceptent les requêtes et répondent aux opérations `AR` et `Commit`.

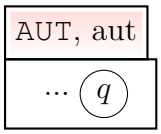


Figure 5.11 – Cas 3 : aucune transition sortante

```
1  Tau(q,automaton):
2   result ←
3    <empty/>
```

Programme 5.20 – Cas d'un état sans transition sortante

```
1  T_t^1(q,t,q',automaton):
2   pl ← PtrLnkCalleeName(automaton)
3   result ←
4    <repeatUntil>
5     <condition>$RepeatVar_t=true()</condition>
6     <sequence>
7      <receive partnerLink="pl" operation="AR" variable="ARVarIn"/>
8      <assign><copy>
9       <from><@ Access(ARVarIn,t) @></from><to>$ARVarOut.access</to>
10     </copy></assign>
11     <reply partnerLink="pl" operation="AR" variable="ARVarOut"/>
12     <if>
13      <condition>$ARVarOut.access=granted</condition>
14      <sequence>
15       <receive partnerLink="pl"
            operation="Commit" variable="CommitVarIn"/>
16       <assign><copy>
17        <from>$CommitVarIn.done</from><to>$RepeatVar_t</to>
18       </copy></assign>
19       <if>
20        <condition>$CommitVarIn.done=true()</condition>
21        <assign>
22         <copy>
23          <from><@ q' has outgoing transitions? @></from>
24          <to>$CommitVarOut.isCompleted</to>
25         </copy>
26         <copy>
27          <from><@ q' final? @></from><to>$CommitVarOut.isFinal</to>
28         </copy>
29        </assign>
30       </if>
31       <reply partnerLink="pl"
            operation="Commit" variable="CommitVarOut"/>
32      </sequence>
33     </if>
34    </sequence>
35   </repeatUntil>
```

Programme 5.21 – Fonction $\mathtt{T}_t^1$

Aux lignes 7 et 11 du programme 5.21, la fonction $\mathtt{T}_t^1$ crée des activités BPEL qui reçoivent et répondent à l'opération `AR`, à travers le lien de partenaire `pl` calculé à la ligne 2 par la fonction `PtrLnkCalleeName`. Entre la réception et la réponse à l'opération `AR`, la décision d'autorisation est calculée par la fonction `Access` à la ligne 9 en comparant, d'une part, les valeurs reçues de la requête d'autorisation et contenues dans la variable BPEL `ARVarIn` et, d'autre part, les valeurs de la spécification contenues dans la transition t de l'automate. La valeur de la décision d'autorisation est affectée à l'élément XML `access` de la variable `ARVarOut`. L'activité BPEL **`if`** des lignes 12 à 33 met le processus BPEL conteneur dans l'attente de la requête de

```
1  T_t^2(q,[(t_1,q_1); ...;(t_n,q_n)],automaton):
2   pl ← PtrLnkCalleeName(automaton)
3   result ←
4    <sequence>
5     <repeatUntil>
6      <condition>
7       $RepeatVar=true()
8      </condition>
9      <sequence>
10      <receive partnerLink="pl"
         operation="AR" variable="ARVarIn"/>
11      <while>
12       <condition>
13        $ARVarOut.access=denied and $i≤n
14       </condition>
15       <assign>
16        <copy>
17         <from>$i+1</from><to>$i</to>
18        </copy>
19        <copy>
20         <from>
            <@ Access(ARVarIn,t_i) @>
           </from>
            <to>$ARVarOut.access</to>
21         </copy>
22       </assign>
23      </while>
24      <reply partnerLink="pl"
         operation="AR"
         variable="ARVarOut"/>
25      <if>
26       <condition>
          $ARVarOut.access=granted
         </condition>
27       <sequence>
28        <receive partnerLink="pl"
            operation="Commit"
            variable="CommitVarIn"/>
29        <assign><copy>
30         <from>$CommitVarIn.done</from>
31         <to>$RepeatVar</to>
32        </copy></assign>
33        <if>
34         <condition>
            $CommitVarIn.done=true()
           </condition>
35         <assign>
36          <copy>
37           <from>
              <@
              q′ has outgoing transitions?
              @>
             </from>
             <to>
              $CommitVarOut.isCompleted
             </to>
38          </copy>
39          <copy>
40           <from><@ q′ final? @></from>
41           <to>
              $CommitVarOut.isFinal
             </to>
42          </copy>
43         </assign>
44        </if>
45        <reply variable="CommitVarOut"
         operation="Commit"
          partnerLink="pl"/>
46       </sequence>
47      </if>
48     </sequence>
49    </repeatUntil>
50    <if>
51     <condition>$i=1</condition>
52     <@ T_au(q_1,automaton) @>
53     <elseif>
54      <condition>$i=2</condition>
55      <@ T_au(q_2,automaton) @>
56     </elseif>
57     ...
58     <elseif>
59      <condition>$i=n</condition>
60      <@ T_au(q_n,automaton) @>
61     </elseif>
62    </if>
63   </sequence>
64
```

Programme 5.22 – Fonction $\mathtt{T}_t^2$

l'opération `Commit` (ligne 15), calcule les valeurs « *isCompleted* » (le chemin suivi par la transition t est complété, c'est-à-dire que son état sortant q' n'a aucune transition sortante) et « *isFinal* » (l'automate est dans un état final) si nécessaire, retourne ces valeurs dans la variable de sortie de l'opération `Commit` (ligne 31). Toute cette logique est contenue dans une activité BPEL `repeat` qui la répète tant que la décision d'autorisation n'est pas *granted* et que la requête initiale de l'utilisateur n'est pas exécutée par le système d'information.

La fonction $\mathtt{T}_t^2$ du programme 5.22 est semblable à la fonction $\mathtt{T}_t^1$. La différence notable est que la décision d'autorisation évalue tour à tour les différentes transitions possibles à partir de l'état courant q. Aux lignes 11 à 23, l'activité BPEL créée par la fonction $\mathtt{T}_t^2$ parcours la liste des transitions en calculant pour chaque transition la décision d'autorisation. La recherche s'arrête lorsque, soit la décision est *granted*, soit la liste des transitions est complètement explorée. À l'issue de l'exploration, la décision d'autorisation est retournée par l'activité BPEL **`reply`** créée à la ligne 24. Après le traitement de la requête et de la réponse à l'opération `Commit` (lignes 25 à 48), le code créé par la fonction $\mathtt{T}_t^2$ doit faire suivre le traitement à partir de l'état q_j correspondant si le service ciblé par l'utilisateur a bien été exécuté par le système d'information (la requête de l'opération `Commit` contient la valeur *true* pour l'élément `done`). Le fil d'exécution se poursuit à la branche adéquate grâce aux activités **`if`** des lignes 51 à 63.

5.5 Génération des types XSD

Le document XSD contient les définitions des types XSD et des éléments XML utilisés par les processus BPEL créés par l'algorithme $\mathtt{T}^{proc}$. Par exemple, l'élément XML `InitRequestType` (respectivement `ARRequestType`, `CommitRequestType` et `StopRequestType`) est défini dans ce document et est le type utilisé pour définir le message de la requête SOAP de l'opération `Init` (respectivement `AR`, `Commit` et `Stop`) — voir plus loin le programme 5.24 qui associe les types aux messages et les messages aux opérations. Ce document XSD est enregistré dans le fichier `Types.xsd`.

Les lignes 4 à 9 du programme 5.23 définissent le type XSD `AccessType` qui énumère les valeurs possibles d'une décision d'autorisation : *granted* (autorisé) et *denied* (refusé). Ce type est utilisé pour définir une variable dans le message de réponse à l'opération `AR`. Les lignes 10 à 15 contiennent la définition du type XSD `SequenceSide` utilisé pour des variables `Side` par le contrôleur de la structure ASTD *séquence* (voir la section 5.4.6). De même, le type XSD `ChoiceSide` est défini aux lignes 16 à 21 et est utilisé par l'activité résultant de la transformation des structures ASTD *choix*.

```
1  T_spec^XSD(spec):
2   result ←
3    <schema>
4     <simpleType name="AccessType">
5      <restriction base="string">
6       <enumeration value="granted"/>
7       <enumeration value="denied"/>
8      </restriction>
9     </simpleType>
10    <simpleType name="SequenceSide">
11     <restriction base="string">
12      <enumeration value="first"/>
13      <enumeration value="second"/>
14     </restriction>
15    </simpleType>
16    <simpleType name="ChoiceSide">
17     <restriction base="string">
18      <enumeration value="left"/>
19      <enumeration value="right"/>
20     </restriction>
21    </simpleType>
22    <complexType name="tRequestBaseType">
23     <sequence>
24      <element name="PId" type="string"/>
25     </sequence>
26    </complexType>
27    <!-- Init request / response -->
28    <element name="InitRequestType">
29     <complexType><complexContent>
30      <extension base="tRequestBaseType"/>
31     </complexContent></complexType>
32    </element>
33    <element name="InitResponseType">
34     <complexType><sequence/></complexType>
35    </element>
36    <!-- Init request / response -->
37    <!-- AR request / response -->
38    <element name="ARRequestType">
39     <complexType><complexContent>
40      <extension
         base="tRequestBaseType"><sequence>
41       <element ref="Event"/>
42       <element name="Environment"
         type="tEnvironment"/>
43      </sequence></extension>
44     </complexContent></complexType>
45    </element>
46    <element name="ARResponseType">
47     <complexType><sequence>
48      <element name="Access"
         type="AccessType"/>
49     </sequence></complexType>
50    </element>
51    <!-- AR request / response -->
52    <!-- Commit request / response -->
53    <element name="CommitRequestType">
54     <complexType><complexContent>
55      <extension
         base="tRequestBaseType">
         <sequence>
56       <element name="Done"
         type="boolean"/>
57      </sequence></extension>
58     </complexContent></complexType>
59    </element>
60    <element name="CommitResponseType">
61     <complexType><sequence>
62      <element name="IsFinal"
         type="boolean"/>
63      <element name="IsCompleted"
         type="boolean"/>
64     </sequence></complexType>
65    </element>
66    <!-- Commit request / response -->
67    <!-- Stop request / response -->
68    <element name="StopRequestType">
69     <complexType><complexContent>
70      <extension
         base="tRequestBaseType"/>
71     </complexContent></complexType>
72    </element>
73    <!-- Stop request / response -->
74    <@ T^XSD(main, "") @>
75   </schema>
```

Programme 5.23 – Document XSD du filtre de contrôle d'accès dynamique

Les lignes 28 à 32 décrivent l'élément XML `InitRequestType` qui est le type des messages des requêtes de l'opération `Init`. La donnée `PId`, qui est portée par ces messages et qui identifie l'instance de processus BPEL ciblée par la requête, est définie dans le message de base `tRequestBaseType` à la ligne 24. Il en est de même pour l'élément XML `StopRequestType` défini aux lignes 68 à 72. L'élément XML `InitResponseType` (lignes 33 à 35) est le type des messages des réponses à l'opération `Init`.

Les requêtes et les réponses de l'opération AR sont définies à partir des éléments XML ARRequestType et ARResponseType respectivement déclarés aux lignes 38 à 45 et aux lignes 46 à 50. L'élément XML ARResponseType porte la décision d'autorisation dans l'élément Access dont le type XSD AccessType est défini aux lignes 4 à 9. L'élément XML ARRequestType contient les éléments XML suivants :

- PId, c'est-à-dire l'identifiant du processus qui reçoit la requête ;
- Event, c'est-à-dire la requête initiale de l'utilisateur authentifié (incluant le nom de service, les paramètres de contrôle d'accès et les paramètres métiers) ;
- Environment, c'est-à-dire une liste d'éléments XML <PV X=""V=""/> qui associent les variables à leurs valeurs (pour des raisons de concision, leur définition est omise du programme 5.23).

Les éléments XML CommitRequestType et CommitResponseType sont respectivement les types des messages des requêtes et des réponses qui définissent l'opération Commit. Une instance de l'élément XML CommitRequestType, défini aux lignes 53 à 59, contient un élément XML Done dont la valeur booléenne indique que le service ciblé a été exécuté par le système d'information après qu'il ait été autorisé. L'élément XML CommitResponseType, défini aux lignes 60 à 65, porte deux variable booléennes, IsFinal qui indique le statut *final* de l'état ASTD correspondant à l'activité qui retourne la réponse à l'opération Commit et IsCompleted qui indique que l'activité qui retourne la réponse ne pourra plus traiter de requêtes AR. La ligne 74 initie l'appel de la procédure $\mathtt{T}^{XSD}$ avec la structure principale de la spécification pour créer les types XSD des variables de quantification pour les structures ASTD quantifiées (structures ASTD *choix quantifié* et *synchronisation quantifiée*).

5.6 Génération du document d'interfaces WSDL

Tout processus BPEL utilise un document WSDL pour exposer les opérations qu'il offre en tant que services Web. Tous les processus BPEL créés par la procédure $\mathtt{T}^{BPEL}_{spec}$ utilise le document WSDL du fichier FilterInterface.wsdl. Ce document contient, par exemple, les déclarations des messages et des opérations. Il est créé par la procédure $\mathtt{T}^{WSDL}_{spec}$ du programme 5.24.

```
1  T_spec^WSDL(spec):
2   result ←
3   <definitions name="FilterInterface">
4    //messages
5    <message name="InitRequest">
6     <part name="payload"
       type="InitRequestType"/>
7    </message>
8    <message name="ARRequest">
9     <part name="payload"
       type="ARRequestType"/>
10   </message>
11   <message name="ARResponse">
12    <part name="result"
       type="ARResponseType"/>
13   </message>
14   <message name="CommitRequest">
15    <part name="payload"
       type="CommitRequestType"/>
16   </message>
17   <message name="CommitResponse">
18    <part name="result"
       type="CommitResponseType"/>
19   </message>
20   <message name="StopRequest">
21    <part name="payload"
       type="StopRequestType"/>
22   </message>
23   //port types
24   <portType name="RootASTDPortType">
25    <operation name="Init">
26     <input name="inputInit"
        message="InitRequest"/>
27     <output name="outputInit"
        message="InitResponse"/>
28    </operation>
29    <operation name="AR">
30     <input name="inputAR"
        message="ARRequest"/>
31     <output name="outputAR"
        message="ARResponse"/>
32    </operation>
33    <operation name="Commit">
34     <input name="inputCommit"
        message="CommitRequest"/>
35     <output name="outputCommit"
        message="CommitResponse"/>
36    </operation>
37    <operation name="Stop">
38     <input name="inputStop"
        message="StopRequest"/>
39    </operation>
40   </portType>
41   <portType name="ASTDPortType">
42    <operation name="AR">
43     <input name="inputAR"
        message="ARRequest"/>
44     <output name="outputAR"
        message="ARResponse"/>
45    </operation>
46    <operation name="Commit">
47     <input name="inputCommit"
        message="CommitRequest"/>
48     <output name="outputCommit"
        message="CommitResponse"/>
49    </operation>
50    <operation name="Stop">
51     <input name="inputStop"
        message="StopRequest"/>
52    </operation>
53   </portType>
54   //bindings
55   <binding name="RootASTDBinding"
      type="RootASTDPortType">
56    ...
57   </binding>
58   <binding name="ASTDBinding"
      type="ASTDPortType">
59    ...
60   </binding>
61   //services
62   <@ T^WSDL(main) @>
63   //partner link types
64   <partnerLinkType
      name="RootASTDPartnerLinkType">
65    <role name="RootASTDRole"
       portType="RootASTDPortType"/>
66   </partnerLinkType>
67   <partnerLinkType
      name="ASTDPartnerLinkType">
68    <role name="ASTDRole"
       portType="ASTDPortType"/>
69   </partnerLinkType>
70   //property, property aliases
71   <property name="PId" type="string"/>
72   <propertyAlias propertyName="PId"
       part="payload"
       messageType="InitRequest">
      <query>PId</query>
73   </propertyAlias>
74   <propertyAlias propertyName="PId"
       part="payload"
       messageType="ARRequest">
75    <query>PId</query>
76   </propertyAlias>
77   <propertyAlias propertyName="PId"
       part="payload"
       messageType="CommitRequest">
78    <query>PId</query>
79   </propertyAlias>
80   <propertyAlias propertyName="PId"
       part="payload"
       messageType="StopRequest">
81   <query>PId</query>
82   </propertyAlias>
83  </definitions>
84
```

Programme 5.24 – Document WSDL du filtre de contrôle d'accès dynamique

Les lignes 5 à 7 déclarent le type de message `InitRequest` utilisé comme requête pour l'opération `Init`. Ce type de message porte un seul élément, `payload`, dont le type `InitRequestType` est déjà défini dans le document XSD lié. Il est associé à l'opération `Init`, créée dans le type de port `RootASTDPortType` aux lignes 25 à 28. Les types de message `ARRequest` et `ARResponse` définis aux lignes 8 à 13, `CommitRequest` et `CommitResponse` définis aux lignes 14 à 19 et `StopRequest` défini aux lignes 20 à 22, sont respectivement les messages des requêtes et réponses des opérations `AR`, `Commit` et `Stop`. Le lien entre tous ces messages et les opérations correspondantes se fait dans la déclaration des types de port `RootASTDPortType` aux lignes 24 à 40, et `ASTDPortType` aux lignes 41 à 53.

L'algorithme de transformation génère un processus BPEL pour la structure ASTD principale et un processus BPEL pour chaque structure ASTD *appel*, *fermeture de Kleene* ou *synchronisation quantifiée*. La création d'un processus pour une sous-structure est illustrée à la section 5.4.4 dans le cas de la transformation d'une structure ASTD *fermeture de Kleene*. Le premier type de port créé dans le programme 5.24, nommé `RootASTDPortType` (ligne 24), est utilisé par les activités résultant des structures ASTD qui sont soit la structure principale de la spécification, soit une sous-structure d'une structure *appel*, *fermeture de Kleene* ou *synchronisation quantifiée*. Ces sous-processus requièrent l'opération `Init` pour pouvoir effectuer leur instanciation, d'où la nécessité du type de port `RootASTDPortType` qui comporte en plus la déclaration de l'opération `Init`. Le second type de port, `ASTDPortType` (ligne 41), diffère du premier par l'absence de l'opération `Init`. Les lignes 29 à 32 (respectivement lignes 42 à 45) déclarent l'opération `AR` pour le type de port `RootASTDPortType` (respectivement `ASTDPortType`) et lui associent comme message d'entrée (élément XML **`input`**) le message `ARRequest` déclaré aux lignes 8 à 10 et comme message de sortie (élément XML **`output`**) le message `ARResponse` déclaré aux lignes 11 à 13. Les opérations `Commit` (lignes 33 à 36 dans le type de port `RootASTDPortType` et lignes 46 à 49 dans le type de port `ASTDPortType`), `Stop` (lignes 37 à 39 dans le type de port `RootASTDPortType` et lignes 50 à 52 dans le type de port `ASTDPortType`) sont associées aux messages correspondants de façon similaire. Les lignes 55 à 60 déclarent les éléments **`binding`** qui précisent

```
1  T^WSDL(astd):
2   result ←
3    <service name="<@Name(astd)@>Service">
4     <port name="<@Name(astd)@>Port" binding="<@Root(astd)@>ASTDBinding">
5      <soap:address location="http://.../<@Name(astd)@>Service"/>
6     </port>
7    </service>
8
9    <@ foreach compound ASTD b in astd @> T^WSDL(b) <@ endfor @>
```

Programme 5.25 – Procédure de création des services Web

le protocole et l'encodage utilisés dans la communication avec le filtre de contrôle d'accès. L'implémentation de l'algorithme utilise la combinaison protocole SOAP et encodage *document*. Cependant, ces détails sont omis du programme 5.24, mais ils sont présentés dans le programme B.1 en annexe B.

La propriété `PId` est définie à la ligne 71, avec le type XSD de base `string`. Elle est utilisée pour définir les corrélations qui servent à identifier les instances de processus. Les lignes 72 à 83 définissent des *alias de propriétés* qui sont un moyen de « calculer » la valeur de la propriété concernée (`PId`) pour chaque type de message. Par exemple, pour une instance du message `ARRequest`, les lignes 75 à 77 indiquent que le chemin XPath `./payload/PId` retourne la valeur de la propriété `PId`.

La procédure $\mathtt{T}^{WSDL}$, décrite dans le programme 5.25, crée les services Web qui sont des points d'accès aux différentes opérations (attentes de requêtes par les activités **`receive`** et **`pick`**/**`onMessage`**) implémentées dans les activités résultant de la transformation de structures ASTD. Le service, son port ainsi que l'adresse URL à partir de laquelle il est accessible sont différenciés par l'utilisation du nom de la structure ASTD comme préfixe.

5.7 Conclusion

La solution au problème de la mise en œuvre automatique d'un filtre de contrôle d'accès à partir d'une spécification ASTD présentée dans ce chapitre repose sur une méthode de transformation complexe. En effet, elle doit tenir compte de plusieurs

aspects et limitations du langage BPEL et de nombreux détails nécessaires pour le déploiement des processus BPEL comme services Web. Bien qu'il ait fallu imposer des restrictions aux spécifications ASTD, ces dernières ne sont pas contraignantes vis-à-vis les patrons de contrôle d'accès introduits au chapitre 2. Globalement, le filtre de contrôle d'accès est obtenu par l'appel des fonctions $\mathtt{T}_{spec}^{BPEL}$, $\mathtt{T}_{spec}^{WSDL}$ et $\mathtt{T}_{spec}^{XSD}$.

Chapitre 6

Interprétation d'une politique de contrôle d'accès ASTD par des processus BPEL

L'*interprétation* permet l'exécution d'un programme ou d'une spécification instruction par instruction, sans toutefois *compiler* celui-ci dans un langage de bas niveau. Un des traitements centraux dans l'utilisation de cette méthode, pour implémenter un filtre de contrôle d'accès dynamique (ou simplement filtre), consiste à faire évoluer l'état courant de la politique de contrôle d'accès pour chaque action du système qui exige une autorisation. Lorsque cette évolution est possible, l'action est autorisée. Dans le cas contraire, son exécution par le système d'information est interdite. La présentation de l'implémentation du filtre suivant cette méthode est faite en trois points. Tout d'abord, un portrait de l'architecture interne du filtre permet de distinguer le rôle des principaux processus BPEL et leur organisation. Ensuite, une présentation des structures de données (et leur utilisation) et des formats d'encodage (des politiques, des états et des environnements) permet de mieux cerner comment les politiques de contrôle d'accès sont interprétées. Enfin, une explication des interactions entre les processus BPEL de même qu'une description de l'implémentation des opérations `Execute`, `IsFinal` et `IS` exposées par ces processus permettent de comprendre comment ils fonctionnent comme services Web. Cette implémentation du filtre est une solution complète des idées présentées dans [18].

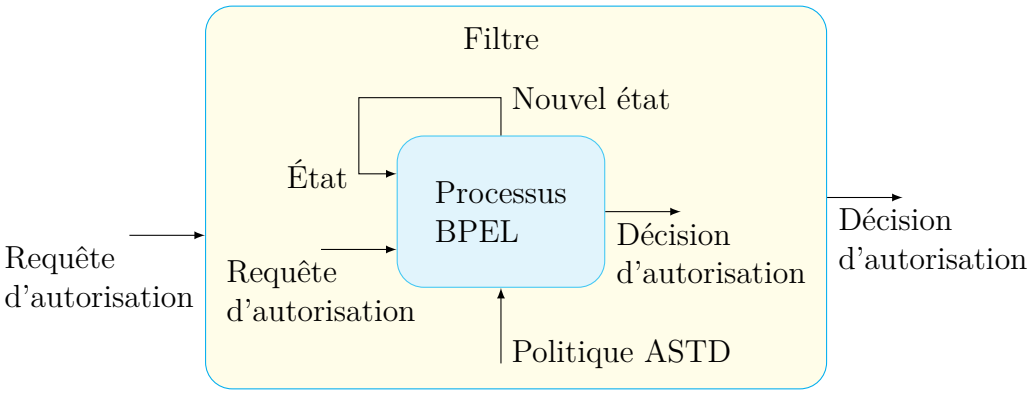


Figure 6.1 – Vue d'ensemble du filtre de contrôle d'accès dynamique

6.1 Architecture du filtre de contrôle d'accès dynamique

Le filtre interprète une spécification ASTD suite aux requêtes d'autorisation reçues du composant PDP correspondant. La figure 6.1 illustre une vue d'ensemble du filtre. La requête d'autorisation est l'événement pour lequel le filtre interprète la spécification. Les processus BPEL du filtre calculent le prochain état à partir de l'état courant de la politique ASTD afin de retourner une décision d'autorisation. Le filtre est constitué par un ensemble de onze processus BPEL, un processus par structure ASTD (*automate*, *séquence*, *choix*, *fermeture de Kleene*, *synchronisation*, *garde*, *appel*, *choix quantifié* et *synchronisation quantifiée*) ainsi qu'un processus principal, nommé `Filter`, et un processus aiguilleur, nommé `ASTD`. La figure 6.2 montre l'architecture du filtre.

6.1.1 Processus BPEL `Filter`

Le filtre est exposé comme service Web à travers le processus `Filter` qui constitue le point d'entrée avec l'extérieur, nommément le composant PDP. La figure 6.3 montre l'interface `FilterPT` (pour Filter Port Type en anglais) exposée par ce processus et déclarée dans celui-ci par le lien de partenaire `FilterPtrLnk`. Elle offre quatre opérations : `Init`, `AR`, `Rollback` et `Stop`. Les opérations `Init` et `Stop` permettent

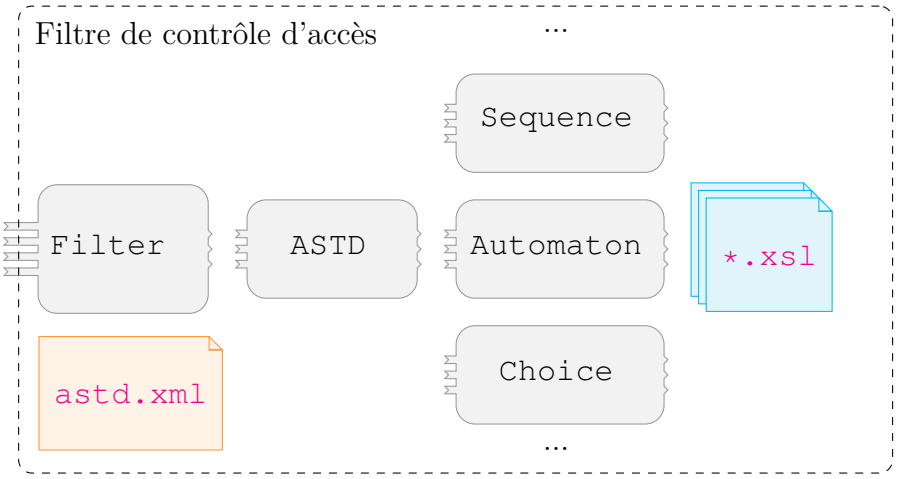


Figure 6.2 – Architecture du filtre de contrôle d'accès dynamique

respectivement de démarrer et d'arrêter le filtre. L'opération `Init` calcule en plus l'état initial de la spécification ASTD et le conserve comme état courant. L'opération `Stop` permet de quitter la boucle d'attente des opérations `AR` et `Rollback` et conduit le flot d'exécution de l'instance courante du processus vers une activité **`exit`**.

L'opération `AR` (pour Authorization Request en anglais) permet d'obtenir une décision d'autorisation. Elle prend en paramètres dans la requête SOAP, entre autres, le nom de l'action concernée avec ses paramètres de contrôle d'accès et ses paramètres métiers. À la réception de la requête, le filtre *calcule* le prochain état de la politique de contrôle d'accès ASTD à partir de son état courant. Si le calcul échoue, c'est-à-dire qu'un nouvel état n'a pas pu être calculé, la réponse retournée par le filtre contient la valeur `denied` pour la décision d'autorisation. Dans le cas contraire, le nouvel état devient l'état courant et la réponse retournée au composant PDP qui a initié la requête contient la décision `granted`.

Dans cette implémentation, le filtre a une vue optimiste de l'exécution des actions du système d'information. Cette vue implique que les actions qui sont autorisées par le filtre seront *probablement* exécutées par le système d'information. Si tel n'est pas le cas, l'opération `Rollback` permet de revenir à l'état précédent, et ainsi annuler dans le filtre la validation de l'exécution de la dernière action pour laquelle une décision

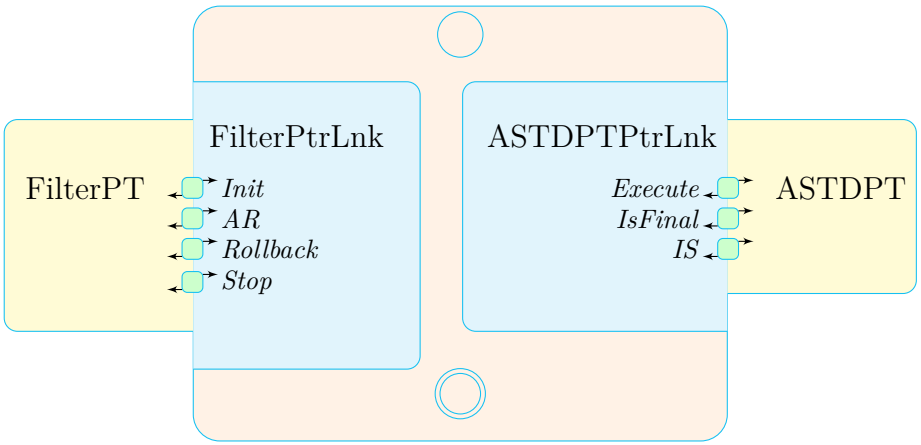


Figure 6.3 – Processus `Filter`

d'autorisation a été positivement accordée. Elle n'est utilisée par le composant PDP que lorsque le système d'information n'a pas exécuté l'action, par exemple à cause de contraintes d'intégrité ou de règles d'affaires violées. À titre d'illustration, la décision d'autorisation peut être accordée pour l'exécution du service de retrait d'un montant dans un compte bancaire, mais le retrait ne s'effectue pas pour insuffisance de fonds. Dans cette éventualité l'appel de l'opération `Rollback` rétablit le filtre dans l'état précédant la décision d'autorisation positive. La requête SOAP pour cette opération n'a pas de paramètre, de même que la réponse SOAP.

La figure 6.3 montre également le lien de partenaire déclaré `ASTDPtrLnk` du type de lien de partenaire `ASTD` associé au type de port `ASTDPT`. Il permet aux instances du processus `Filter` d'utiliser les opérations `Execute`, `IsFinal` et `IS` associées au processus aiguilleur `ASTD`.

6.1.2 Processus BPEL `ASTD`

Le processus `ASTD` agit comme aiguilleur vers les neuf autres processus associés aux structures ASTD. Son interface, montrée à la figure 6.4, comporte les opérations `Execute`, `IsFinal` et `IS` dont les requêtes et réponses sont aiguillées vers les neufs autres processus ASTD qui ont la même interface `ASTDPT` (pour ASTD Port Type en

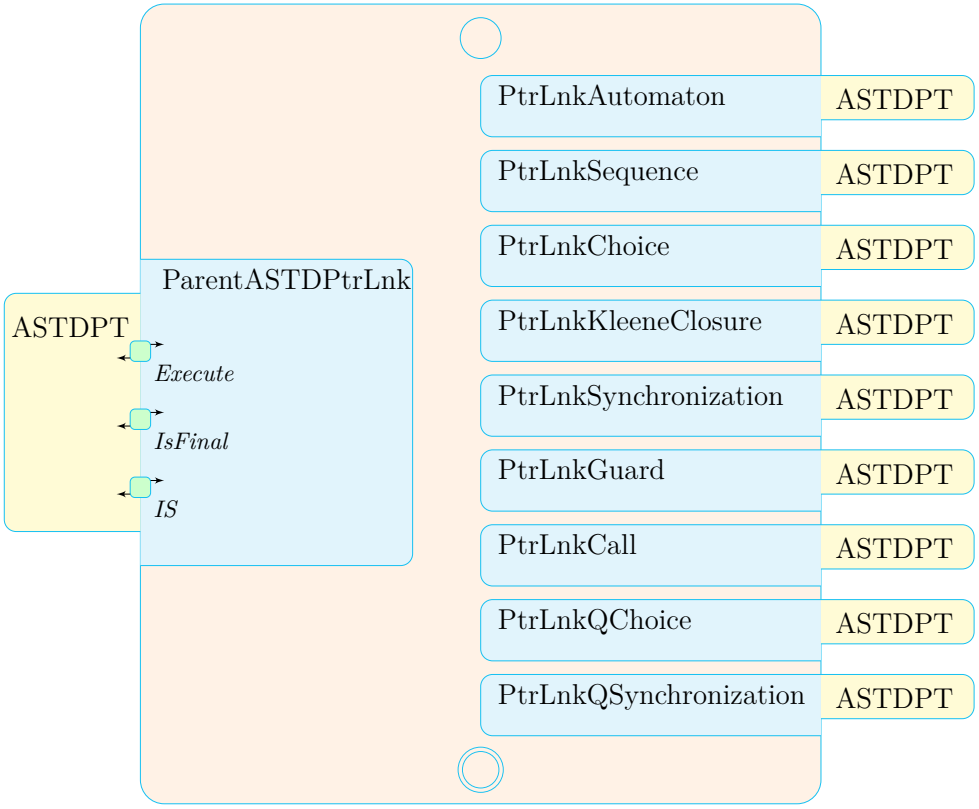


Figure 6.4 – Processus `ASTD`

anglais). L'opération `Execute` prend en paramètres dans sa requête SOAP la spécification d'une structure ASTD, l'état courant de cette spécification, l'événement avec ses paramètres pour lequel un nouvel état doit être calculé, et enfin l'environnement courant d'exécution. La réponse SOAP de cette opération est le nouvel état calculé. L'opération `IsFinal` détermine si un état courant de la spécification est *final*. L'opération `IS` (pour Initial State en anglais) calcule l'état initial d'une structure ASTD. Sa requête SOAP accepte en paramètres la spécification ASTD pour lequel le nouvel état doit être calculé et l'environnement courant. La réponse SOAP correspondante contient le nouvel état calculé.

Dans la figure 6.4, les interfaces WSDL exposées et requises sont schématisées. Le processus `ASTD` déclare plusieurs liens de partenaire, en particulier le lien de

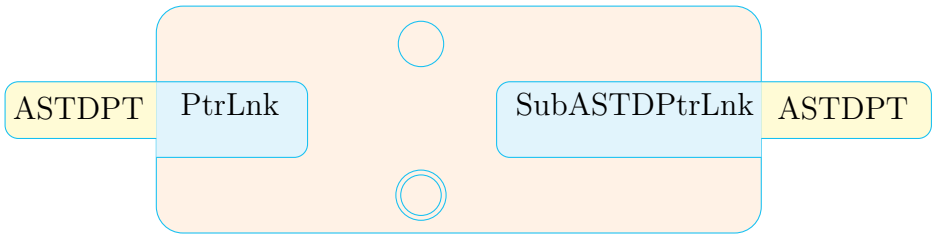


Figure 6.5 – Processus `QChoice`

partenaire `ParentASTDPtrLnk` de type `ASTDPT` est déclaré et exposé aux processus BPEL du filtre. Le type de port `ASTDPT` est inclus dans le document d'interfaces WSDL du fichier `ASTD.wsdl`. Le processus `ASTD` déclare aussi neuf autres liens de partenaire (`PtrLnkAutomaton` à `PtrLnkQSynchronization`) par lesquels il aiguille les requêtes et les réponses SOAP vers le processus BPEL qui implémente les règles sémantiques de la structure ASTD correspondante. Ces liens de partenaire sont aussi de type `ASTDPT`, c'est-à-dire que le processus `ASTD` est client de neuf services qui fournissent également les opérations `Execute`, `IsFinal` et `IS`.

6.1.3 Processus BPEL des structures ASTD

Le langage formel ASTD comporte neuf structures ASTD. L'interprétation d'une spécification ASTD est possible grâce à neuf processus BPEL correspondant chacun une structure ASTD et implémentant la sémantique opérationnelle du langage. Le processus `QChoice` correspond à la structure ASTD *choix quantifié* et ses interfaces (exposée et requise) sont illustrées à la figure 6.5. Tous les autres processus BPEL du filtre exposent et requièrent les mêmes interfaces à travers des liens de partenaire identiques.

6.2 La structure et le format de données

Les processus BPEL correspondant à chacune des structures ASTD communiquent par des requêtes et des réponses SOAP qui utilisent des formats d'encodage de spécifications ASTD et leur état courant. En plus de contenir ces informations, les messages

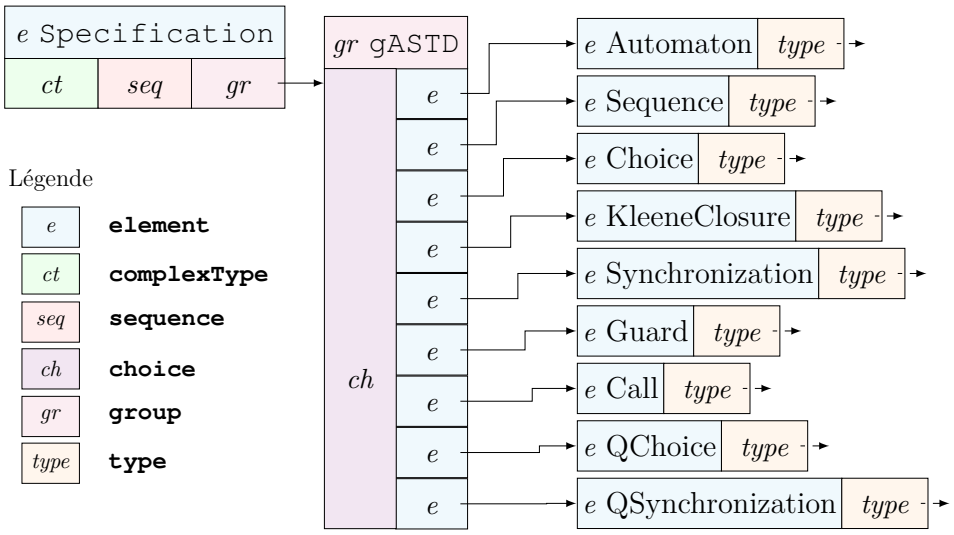


Figure 6.6 – Définition de l'élément `Specification`

SOAP contiennent suivant les besoins :

- l'*environnement* courant au contexte de la requête d'autorisation ;
- la *requête d'autorisation* elle-même.

6.2.1 Document XML d'une spécification ASTD

La politique contrôle d'accès dynamique modélisée avec le langage ASTD est encodée dans un document XML pour pouvoir être utilisée par le filtre pour les décisions d'autorisation. Le métamodèle du langage ASTD sur lequel cet encodage repose est brièvement présenté à la section 1.1. Ce modèle est défini par le schéma XSD du fichier `ASTD.xsd` dont le contenu est listé dans le programme A.1 à l'annexe A.

L'élément XML `Specification` est l'élément racine du document XML. Il est un type complexe anonyme qui contient une suite non vide de structures ASTD, comme le montre la figure 6.6. La suite non vide est introduite dans le type complexe de la spécification par une séquence de références à un groupe XML qui offre le choix entre les neuf éléments XML des structures ASTD. L'initiale *e* signifie que dans le document XSD, `Specification` est un élément XML. Cet élément contient un

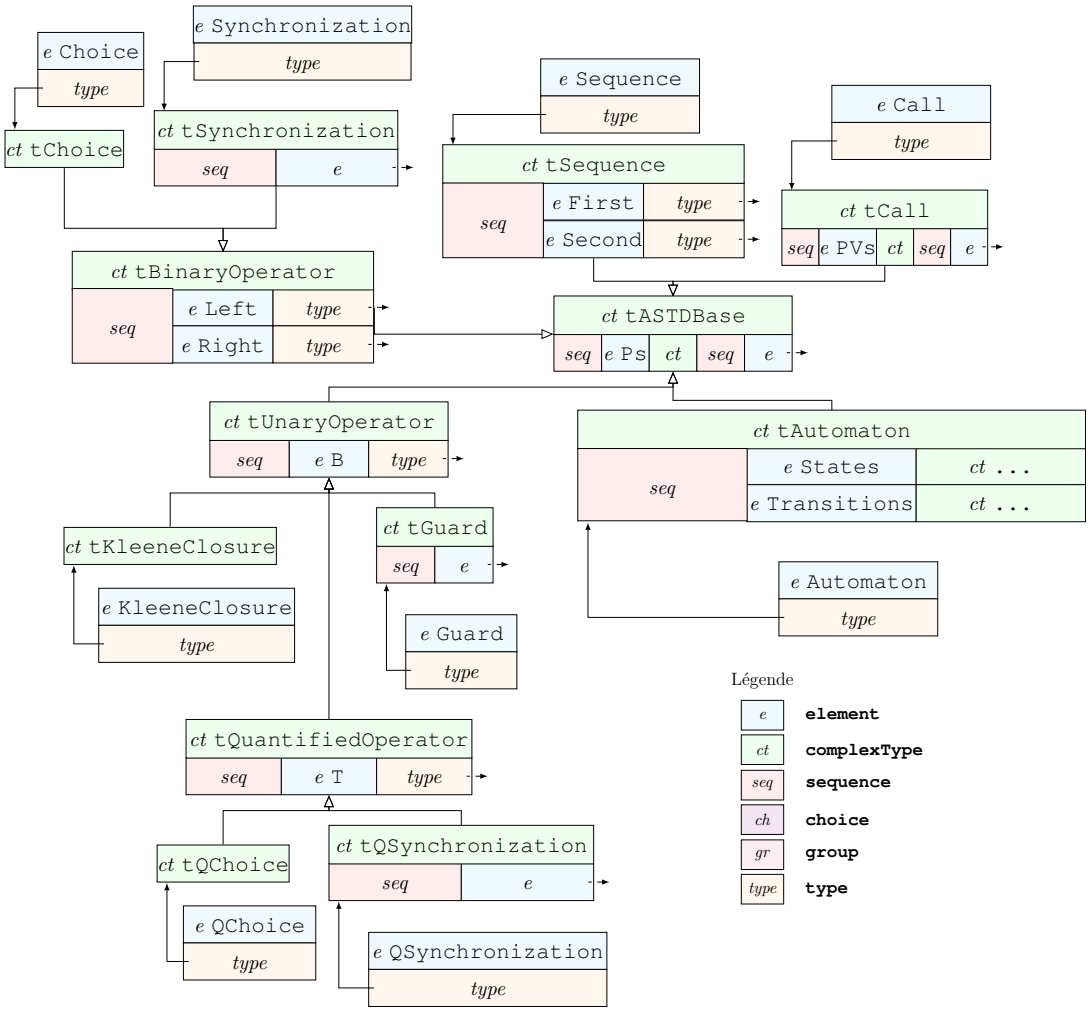


Figure 6.7 – Schéma XSD des structures ASTD

type complexe (initiales *ct*) qui contient une séquence (initiales *seq*). Cette dernière contient à son tour un groupe XSD (initiales *gr*), tout cela sous la forme de balises XSD imbriquées.

Les principaux éléments XML pour les structures ASTD sont montrés dans la figure 6.7. Le schéma XSD définit le type complexe (élément **`complexType`**) abstrait `tASTDBase` qui sert de base à la définition des types complexes des différentes structures ASTD. Ce type comprend les éléments de base de toute structure ASTD : le nom de la structure ASTD et les paramètres de celle-ci. Le type `tASTDBase` spécifie

que chaque paramètre d'une structure ASTD est un élément XML contenant un sous-élément pour le nom du paramètre et également un sous-élément pour sa valeur. Les types complexes abstraits `tUnaryOperator` et `tBinaryOperator` sont définis à partir du type complexe `tASTDBase`. Ils représentent la généralisation des structures ASTD qui ont respectivement une seule sous-structure et deux sous-structures ASTD.

Le type complexe abstrait `tUnaryOperator` hérite du type `tASTDBase`. Il est utilisé comme type de base pour les types complexes des éléments XML des structures ASTD qui possèdent une sous-structure telles les structures ASTD *fermeture de Kleene* et *garde*. La définition du type `tUnaryOperator` ajoute, à son type de base, un élément XML `B` qui contient un élément encodant la sous-structure. Les types complexes concrets `tKleeneClosure`, `tGuard` et le type complexe abstrait `tQuantifiedOperator` sont définis à partir du type `tUnaryOperator` et donc du type `tASTDBase`. Le type `tKleeneClosure` (respectivement le type `tGuard`) permet de définir l'élément XML `KleeneClosure` (respectivement l'élément XML `Guard`) qui est l'élément de la structure ASTD *fermeture de Kleene* (respectivement la structure ASTD *garde*). Les structures ASTD qui utilisent une quantification, c'est-à-dire une variable dont les valeurs sont tirées d'un ensemble dit ensemble de quantification, sont encodées par des éléments XML définis à partir du type `tQuantifiedOperator`. C'est le type de base du type complexe concret `tQChoice` (respectivement `tQSynchronization`), qui est le type de l'élément XML `QChoice` (respectivement `QSynchronization`). Il est l'élément XML qui encode la structure ASTD *choix quantifié* (respectivement la structure ASTD *synchronisation quantifiée*).

Le type complexe abstrait `tBinaryOperator` hérite du type `tASTDBase`, tout comme le type `tUnaryOperator`. Il étend le type de base en ajoutant deux éléments enfants `Left` et `Right`, qui chacun contiennent un élément encodant une structure ASTD. Les types complexes concrets `tChoice` et `tSynchronization` héritent de ce type et sont les types des éléments XML `Choice` et `Synchronization` qui encodent les structures ASTD *choix* et *synchronisation*. Le type complexe concret `tAutomaton` (respectivement `tSequence` et `tCall`) permettant de définir l'élé-

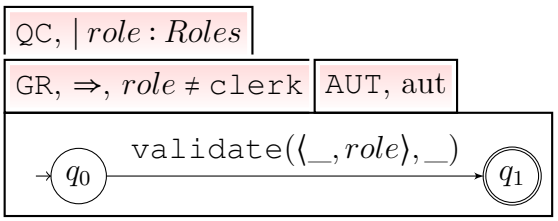


Figure 6.8 – Exemple d'une structure ASTD *choix quantifié*

ment XML `Automaton` (respectivement `Sequence` et `Call`) hérite directement du type abstrait `tASTDBase`. Le type `tSequence` en particulier n'hérite pas de `tBinaryOperator` puisque le langage ASTD désigne par *first* et *second* les sous-structures de la séquence, au lieu de *left* et *right*. L'élément XML `Automaton` (respectivement `Sequence` et `Call`) encode la structure ASTD *automate* (respectivement *séquence* et *appel*).

La figure 6.8 est un exemple d'une structure ASTD *choix quantifié*. Il contient une sous-structure ASTD *garde*, qui elle comporte une sous-structure *automate*. Le programme 6.1 illustre l'encodage XML de la spécification de la figure 6.8. Les lignes 1 et 54 de ce programme correspondent respectivement à l'ouverture et à la fermeture des balises de l'élément XML `QChoice` qui encode la structure ASTD *choix quantifié*. En particulier, la ligne 1 contient les attributs `a:Name` et `a:X` qui encodent respectivement le nom de la structure et le nom de la variable quantifiée. Entre ces lignes d'ouverture et fermeture de l'élément `QChoice`, les lignes 5 et 12, qui correspondent à l'ouverture et à la fermeture des balises de l'élément XML `a:T`, encodent le type des valeurs de la variable de quantification de la structure ASTD. Ce type est représenté comme une énumération XSD des valeurs `clerk` et `ho` des rôles possibles (dans la spécification de la figure 6.8 *Roles* = {`clerk`, `ho`}). Le sous-élément XML `a:B` (lignes 13 à 53) contiennent la sous-structure ASTD du choix quantifié qui est une *garde*. Le procédé d'encodage continue ainsi de façon récursive aux lignes 14 à 52.

```
<a:QChoice a:Name="QC" a:X="role"
 xmlns:xsd="http://.../XMLSchema"
 xmlns:a="http://.../schema/ASTD"
 xmlns:p="http://.../schema/Predicate">
 <a:T>
  <xsd:simpleType xsd:name="Roles">
   <xsd:restriction xsd:base="string">
    <xsd:enumeration xsd:value="clerk"/>
    <xsd:enumeration xsd:value="ho"/>
   </xsd:restriction>
  </xsd:simpleType>
 </a:T>
 <a:B>
  <a:Guard a:Name="GR">
   <p:Predicate>
    <p:NotEqual>
     <p:Left>
      <p:Variable>role</p:Variable>
     </p:Left>
     <p:Right>
      <p:String>clerk</p:String>
     </p:Right>
    </p:NotEqual>
   </p:Predicate>
   <a:B>
    <a:Automaton a:Name="AUT" a:N0="q0">
     <a:States>
      <a:State a:Name="q0">
       <a:Elementary/>
      </a:State>
      <a:State a:Name="q1">
       <a:Elementary a:Final="true"/>
      </a:State>
     </a:States>
     <a:Transitions>
      <a:Transition a:Final="false">
       <a:Phi>
        <p:Predicate>
         <p:Boolean>true</p:Boolean>
        </p:Predicate>
       </a:Phi>
       <a:LocalArrow a:N1="q0"
        a:N2="q1"/>
       <a:Event a:Name="validate">
        <a:PV a:X="user" a:V="_"/>
        <a:PV a:X="role" a:V="$role"/>
        <a:PV a:X="chId" a:V="_"/>
       </a:Event>
      </a:Transition>
     </a:Transitions>
    </a:Automaton>
   </a:B>
  </a:Guard>
 </a:B>
</a:QChoice>
```

Programme 6.1 – Encodage XML d'une structure ASTD

6.2.2 Document XML de l'état d'une spécification ASTD

Pendant l'interprétation d'une spécification ASTD, les opérations `Execute` des processus BPEL correspondant aux différentes structures ASTD, calculent de nouveaux états. Ces états sont encodés dans des documents XML qui contiennent un type d'élément XML pour chaque structure ASTD. Le programme 6.2 correspond à

```
<a:QChoiceState
 xmlns:a="http://gril.udes.ca/astd/schema/ASTD">
 <a:S>
  <a:GuardState>
   <a:S>
    <a:AutomatonState a:N="q0">
     <a:H/>
    </a:AutomatonState>
   </a:S>
  </a:GuardState>
 </a:S>
</a:QChoice>
```

Programme 6.2 – Encodage XML de l'état d'une structure ASTD

l'encodage XML d'un état possible de la structure ASTD de la figure 6.8, celui obtenu après l'exécution de l'opération `IS` avec cette spécification en paramètre.

La définition des éléments XML de l'état d'une spécification ASTD suit une approche similaire à celle des structures ASTD décrite à la section précédente. Le type `tUnaryState` est le type complexe abstrait de base de tous les états ASTD qui comportent un seul sous-état. Les types concrets `tSequenceState`, `tCallState`, `tChoiceState`, `tGuardState`, `tQChoiceState`, `tKleeneClosureState` et `tQSynchronizationState` héritent du type abstrait `tUnaryState` et ajoutent des attributs relatifs aux états de la structure ASTD qu'ils encodent. La figure 6.9 illustre le modèle utilisé pour l'encodage XML des états d'une spécification ASTD. Les états des structures ASTD *automate*, *synchronisation* et *synchronisation quantifiée* sont des cas particuliers. La définition du type complexe qui encode l'état d'une structure *automate* permet d'encoder aussi l'historique des états des sous-structures de l'*automate*. Pour une structure ASTD *synchronisation*, le type complexe concret `tSynchronizationState` encode les états et définit les éléments enfants XML `LeftState` et `RightState` qui encodent les états des sous-structures membres de la structure ASTD *synchronisation*.

Dans le cas de la version quantifiée de la *synchronisation*, les états sont maintenus par une fonction qui associe à une valeur de la variable de quantification, l'état courant de la sous-structure membre de la *quantification synchronisée*. Aussi, bien qu'héritant du type `tUnaryState`, le type complexe `tQSynchronizationState` est complété par l'encodage de cette fonction. La fonction est encodée par un élément XML `F` et chaque relation valeur/état par un élément XML `FV` qui possède un attribut `V` pour la valeur et un élément enfant pour l'état. L'élément `F` est défini dans le type complexe abstrait `tQSynchronizationState`. Cet héritage permet une optimisation pour l'exécution de l'opération `Execute` avec une structure ASTD *synchronisation quantifiée*. En effet, l'état initial de la sous-structure ASTD d'une synchronisation quantifiée est calculé une seule fois à l'issue de l'opération `IS` et cet état initial est conservé dans l'élément `S` défini par le type complexe `tUnaryState` dont hérite le type complexe `tQSynchronizationState` (voir la figure 6.10). La définition des

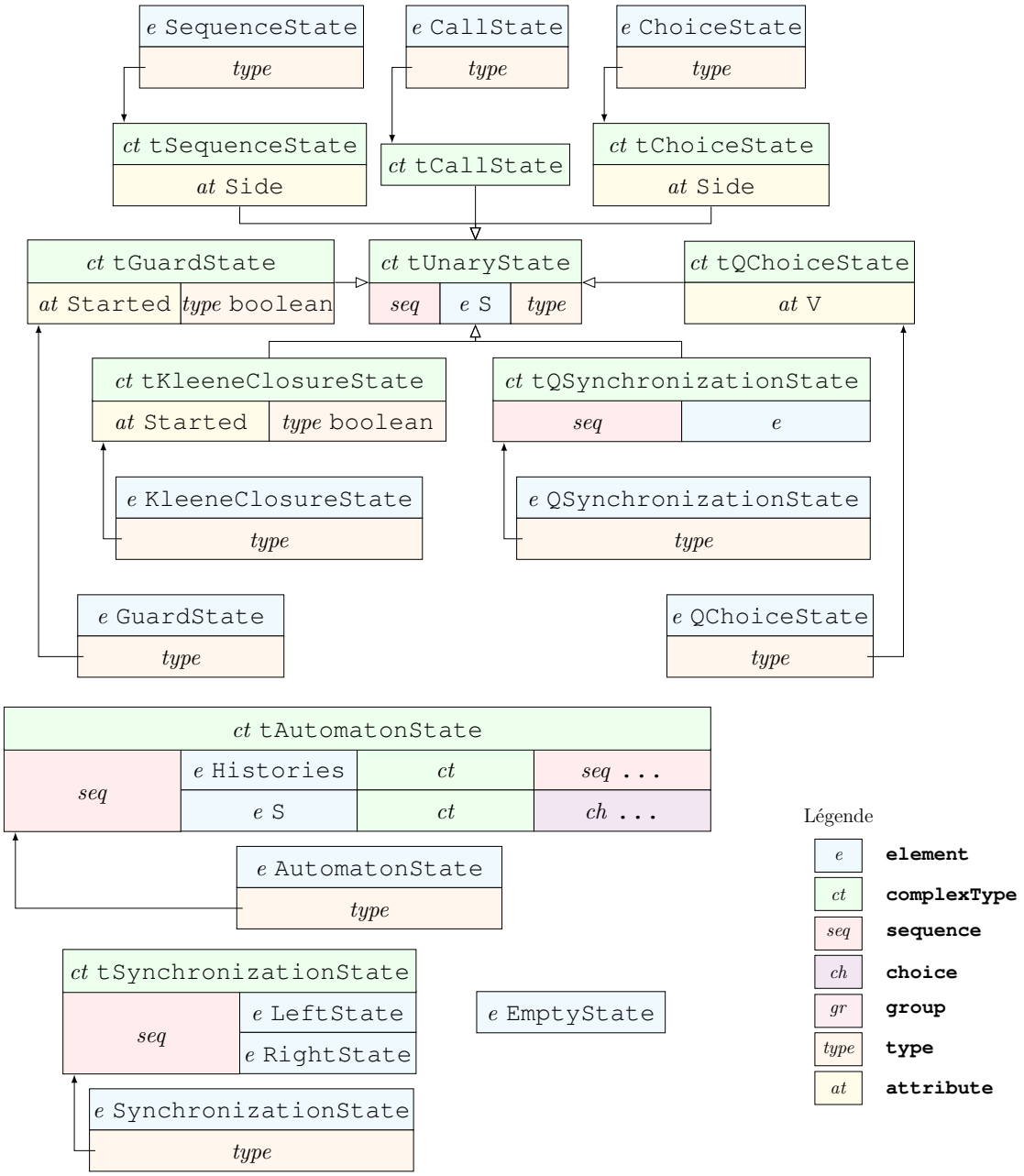


Figure 6.9 – Schéma XSD des états d'une structure ASTD

états comporte aussi l'élément spécial `EmptyState` qui symbolise l'échec du calcul d'un nouvel état.

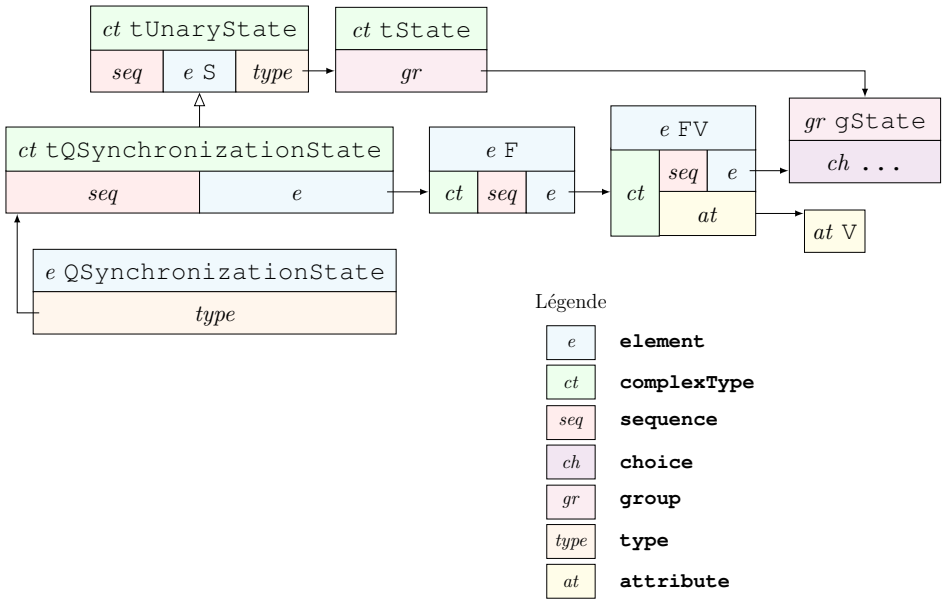


Figure 6.10 – Éléments XML de l'état d'une synchronisation quantifiée

6.2.3 Environnement

Les structures quantifiées *choix* et *synchronisation* introduisent des variables dans les spécifications. Ces variables permettent l'exécution de la spécification pour des valeurs qui ne sont pas figées à l'avance lors de l'écriture de la spécification. Pendant l'exécution d'une spécification formelle, un environnement qui contient les valeurs des variables est utilisé et permet de définir le contexte dans lequel les événements sont acceptés. Les variables peuvent aussi être introduites dans l'environnement à la suite de l'utilisation d'une structure ASTD *appel*. Dans le filtre qui interprète la spécification, l'environnement est un élément XML qui liste les variables dans l'environnement ainsi que les valeurs de chacune de celles-ci. Le programme 6.3 montre un environnement avec trois variables (`user`, `role` et `clId`) qui peuvent être, par exemple, les variables de trois structures ASTD quantifiées imbriquées l'une dans l'autre. Les valeurs `Bob`, `clerk` et `3` sont affectées respectivement à ces variables dans l'environnement considéré.

```
1  <t:Environment
2    xmlns:t="http://gril.udes.ca/astd/schema/Types"
3    xmlns:a="http://gril.udes.ca/astd/schema/ASTD">
4   <a:PV a:X="user" a:V="Bob"/>
5   <a:PV a:X="role" a:V="clerk"/>
6   <a:PV a:X="clId" a:V="3"/>
7  </t:Environment>
```

Programme 6.3 – Un environnement XML avec trois variables

Au cours de l'interprétation d'une spécification ASTD, l'environnement ne peut être modifié que par la sémantique opérationnelle des structures ASTD *appel*, *choix quantifié* et *synchronisation quantifiée*. Pour une structure ASTD *appel*, les valeurs spécifiées pour les variables de la structure appelée sont placées dans l'environnement avant l'exécution d'une action. De même, pour les structures ASTD qui possèdent une variable (structures *choix quantifié* et *synchronisation quantifiée*), une valeur doit être affectée à cette variable dans l'environnement pour pouvoir exécuter une action. La mise à jour de l'environnement lors de l'exécution des opérations `Execute` et `IsFinal` de ces trois types de structure ASTD se fait par l'utilisation d'une transformation XSLT, puisque le langage BPEL ne permet pas de façon native d'ajouter un nouvel élément XML à un autre élément XML et d'en faire son élément enfant.

Dans le cas des structures ASTD ayant une variable quantifiée (*choix quantifié* et *synchronisation quantifiée*), le résultat de la mise à jour de l'environnement est un nouvel environnement comportant la nouvelle variable (si elle n'existait pas déjà dans l'environnement) avec une valeur vide. Le programme 6.4 correspond à la mise à jour de l'environnement du programme 6.3 pour une structure ASTD *choix quantifié* ou *synchronisation quantifiée* dont la variable de quantification est `chId`. La

```
1  <t:Environment
2    xmlns:t="http://gril.udes.ca/astd/schema/Types"
3    xmlns:a="http://gril.udes.ca/astd/schema/ASTD">
4   <a:PV a:X="user" a:V="Bob"/>
5   <a:PV a:X="role" a:V="clerk"/>
6   <a:PV a:X="clId" a:V="3"/>
7   <a:PV a:X="chId" a:V=""/>
8  </t:Environment>
```

Programme 6.4 – Environnement mis à jour pour une structure quantifiée

```
1  <t:EnvironmentXslInput
2    xmlns:t="http://gril.udes.ca/astd/schema/Types"
3    xmlns:a="http://gril.udes.ca/astd/schema/ASTD">
4   <t:Environment>
5    <a:PV a:X="user" a:V="Bob"/>
6    <a:PV a:X="role" a:V="clerk"/>
7    <a:PV a:X="clId" a:V="3"/>
8   </t:Environment>
9   <t:NewEnvironment>
10   <a:PV a:X="clId" a:V="$user"/>
11   <a:PV a:X="chId" a:V="101"/>
12  </t:NewEnvironment>
13 </t:EnvironmentXslInput>
```

Programme 6.5 – Document XML pour la mise à jour d'un environnement

nouvelle variable insérée dans le programme 6.4 se trouve à la ligne 7. La mise à jour de l'environnement se fait avec la méthode BPEL `doXslTransform` qui prend en paramètres la feuille de transformation XSLT `GetNewEnvironmentForX.xsl`, l'environnement à mettre à jour et le nom de la nouvelle variable à ajouter à l'environnement.

Pour les opérations `Execute` et `IsFinal` du processus BPEL `Call` (celui qui implémente la sémantique opérationnelle de la structure ASTD *appel*), il peut y avoir plus d'une variable à insérer. La méthode décrite dans le paragraphe précédent n'est pas adéquate même si elle est utilisée de façon répétitive pour chaque variable à rajouter dans l'environnement. Dans ce cas, une feuille de style XSLT particulière est utilisée. Celle-ci prend en paramètre un document XML contenant l'environnement à mettre à jour ainsi que les nouvelles paires variables/valeurs à ajouter à l'environnement. Le programme 6.5 correspond à un exemple de document passé en paramètre à la feuille de style `GetNewEnvironment.xsl` par la méthode `doXslTransform`. Le nouvel environnement retourné par la fonction `doXslTransform` est explicité au programme 6.6.

6.2.4 Ensemble de valeurs

Les structures ASTD quantifiées (*choix* et *synchronisation*) utilisent une variable quantifiée dont le domaine est fini. Cet ensemble est décrit dans l'encodage XML de la spécification formelle par un type simple XSD. Il est possible de procéder à l'énu-

```
1  <t:Environment
2    xmlns:t="http://gril.udes.ca/astd/schema/Types"
3    xmlns:a="http://gril.udes.ca/astd/schema/ASTD">
4   <a:PV a:X="user" a:V="Bob"/>
5   <a:PV a:X="role" a:V="clerk"/>
6   <a:PV a:X="clId" a:V="Bob"/>
7   <a:PV a:X="chId" a:V="101"/>
8  </t:Environment>
```

Programme 6.6 – Nouvel environnement

mération des valeurs de l'ensemble ou encore d'expliciter une plage de valeurs pour des entiers. Avec ces deux types d'ensemble de base, des ensembles plus complexes peuvent être construits par union. Le programme 6.7 montre l'exemple de la définition du domaine pour la variable `role` (lignes 2 à 10) de la structure ASTD de la figure 6.8.

Les processus BPEL correspondant aux structures ASTD quantifiées requièrent, pour le fonctionnement des opérations `Execute`, `IsFinal` et `IS`, de pouvoir parcourir la liste explicite des valeurs du type de la variable de quantification. Cette liste est obtenue en utilisant la feuille de style `GetTypeValues.xsl`, qui peut prendre en paramètre le document XML correspondant à l'élément XML `a:T`. La liste retournée par la fonction `doXslTransform` pour le type `Roles` de l'exemple du programme 6.7 est donnée dans le programme 6.8.

```
1  <a:QChoice a:Name="QC" a:X="role">
2   <a:T>
3    <xsd:simpleType xsd:name="Roles"
4      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
5     <xsd:restriction xsd:base="string">
6      <xsd:enumeration xsd:value="clerk"/>
7      <xsd:enumeration xsd:value="ho"/>
8     </xsd:restriction>
9    </xsd:simpleType>
10  </a:T>
11  <a:B>
12   ...
```

Programme 6.7 – Définition de l'ensemble de quantification d'une variable

```
1  <t:TypeValuesXslOutput
2    xmlns:t="http://gril.udes.ca/astd/schema/Types"
3    xmlns:a="http://gril.udes.ca/astd/schema/ASTD">
4   <t:V a:V="clerk"/>
5   <t:V a:V="ho"/>
6  </t:TypeValuesXslOutput>
```

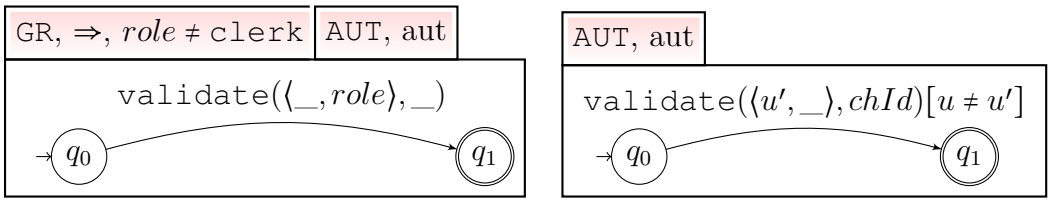
Programme 6.8 – Liste de valeurs pour le type `Roles`

6.2.5 Garde et prédicat

La structure ASTD *garde* utilise un prédicat pour procéder à l'interprétation de la sous-structure ASTD. De même, les transitions dans les structures ASTD *automate* possèdent un prédicat optionnel qui fait partie de la condition à évaluer — dans l'environnement — pour passer de l'état courant de l'automate à un autre état. Les deux cas sont illustrés dans la figure 6.11.

Ces prédicats sont aussi encodés dans des éléments XML avec les structures ASTD et les transitions auxquelles ils sont associés. Pour y parvenir, chaque prédicat, vu sous la forme d'un arbre syntaxique abstrait, est encodé en utilisant un modèle XSD propre à la syntaxe des prédicats. Pour le prédicat $3 \geq x * 2 + 1$, l'arbre syntaxique abstrait est donné dans la figure 6.12.

Le programme 6.9 est l'encodage XML de l'arbre syntaxique abstrait de la figure 6.12. Les différents opérateurs supportés sont les opérateurs de base de l'arithmétique entière et de l'arithmétique en virgule flottante (addition, soustraction, multiplication et division), les opérateurs booléens de base (la négation, la conjonction et



(a) Prédicat associé à une garde (b) Prédicat associé à une transition

Figure 6.11 – Exemples de prédicats

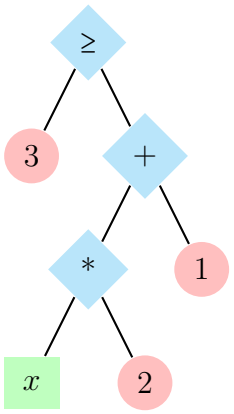


Figure 6.12 – Arbre syntaxique abstrait du prédicat $3 \geq x * 2 + 1$

```
1  <p:Predicate
2    xmlns:p="http://.../schema/Predicate">
3   <p:Geq>
4    <p:Left>
5     <p:Number>3</p:Number>
6    </p:Left>
7    <p:Right>
8     <p:Addition>
9      <p:Left>
10      <p:Multiplication>
11       <p:Left>
12        <p:Variable>x</p:Variable>
13       </p:Left>
14       <p:Right>
15        <p:Number>2</p:Number>
16       </p:Right>
17      </p:Multiplication>
18     </p:Left>
19     <p:Right>
20      <p:Number>1</p:Number>
21     </p:Right>
22    </p:Addition>
23   </p:Right>
24  </p:Geq>
25 </p:Predicate>
```

Programme 6.9 – Encodage XML du prédicat $3 \geq x * 2 + 1$

la disjonction) ainsi que quelques opérateurs relationnels (égal, différent, supérieur, supérieur ou égal, inférieur et inférieur ou égal). Le modèle sur lequel repose cet encodage permet une évaluation directe du prédicat en utilisant une feuille de style XSLT.

Le langage XSLT permet l'application récursive de règles à un document XML. Cette caractéristique du langage est utilisée pour l'évaluation des prédicats. Par exemple, pour une addition rencontrée lors du parcours de l'élément XML d'un prédicat, la feuille de style `Predicat.xsl` évalue les opérandes de l'opérateur sous les éléments XML `p:Left` et `p:Right` en utilisant la récursivité du langage XSLT. Les résultats de ces évaluations sont temporairement stockées dans deux variables dont les valeurs sont simplement additionnées pour compléter l'évaluation de l'addition. Les autres opérateurs sont évalués de façon similaire. L'élément XML `p:Variable` requiert une attention particulière. Les valeurs associées aux variables sont stockées dans l'environnement qui définit le contexte dans lequel la spécification formelle est

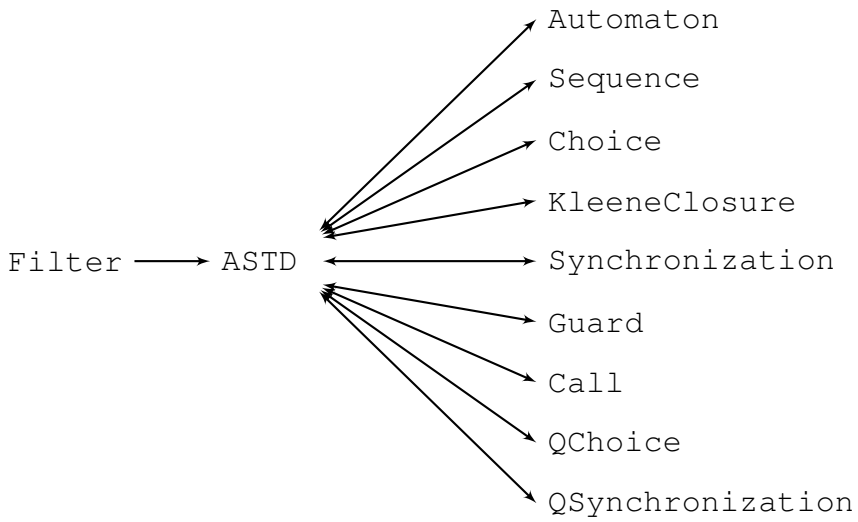


Figure 6.13 – Graphe d'interactions entre les processus du filtre de contrôle d'accès

interprétée. L'environnement est aussi passé en paramètre à la feuille de style par la fonction `doXslTransform`. L'évaluation de l'élément XML `p:Variable` consiste donc à retourner la valeur associée à la variable dans l'environnement.

6.3 Interactions entre les processus BPEL

Pour que le filtre puisse retourner des décisions d'autorisation, les processus BPEL doivent coopérer entre eux à travers les liens de partenaire qu'ils déclarent. La figure 6.13 illustre le graphe d'interactions entre les processus du filtre. Dans ce graphe, les arcs représentent des appels d'opérations (requêtes/réponses SOAP) émis depuis la source de l'arc vers sa cible.

Le processus `Filter` interagit avec le processus `ASTD` afin de retourner des décisions d'autorisation, comme illustré à la figure 6.14. Celle-ci montre deux instances de ces processus. Les instances du processus `Filter` utilisent les opérations `Execute` et `IS` des instances du processus `ASTD`, mais pas l'opération `IsFinal`. L'opération `IS` calcule l'état initial de la spécification lors de l'initialisation du filtre. L'opération `Execute` est appelée depuis le processus `Filter` lorsque celui-ci doit calculer une

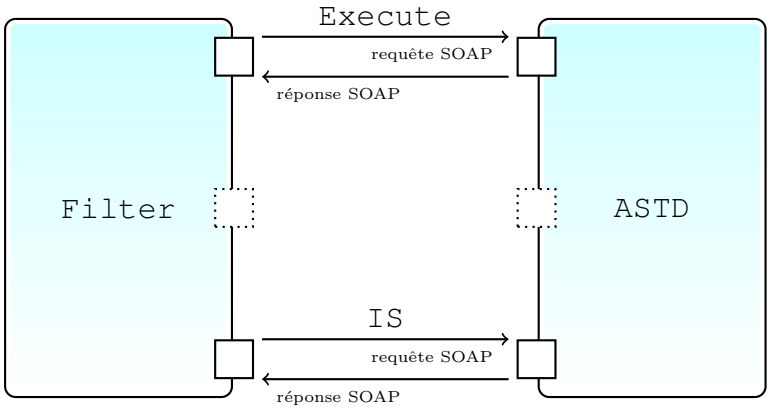


Figure 6.14 – Interactions entre les processus `Filter` et `ASTD`

décision d'autorisation. Le processus `Filter` passe en paramètres à cette opération la structure ASTD *principale* de la spécification, l'état courant de la spécification, un environnement vide (sans paramètre, ni valeur) et l'événement contenu dans la requête d'autorisation `AR` reçu par le processus `Filter`.

Le processus `ASTD` aiguille les requêtes (`Execute`, `IsFinal` ou `IS`) vers les processus BPEL qui implémentent la sémantique opérationnelle des structures ASTD correspondantes. Il détermine le type de la structure ASTD contenu dans la requête SOAP (toutes les opérations du type de port `ASTDPT` portent dans leurs requêtes SOAP une structure ASTD) et suivant ce type, fait suivre la requête au processus BPEL correspondant en utilisant les liens de partenaire déclarés (voir la partie droite de la figure 6.4). Dans le cas de la spécification de la figure 6.8, les instances du processus `ASTD` interagissent avec les instances du processus `QChoice`, comme illustré dans la figure 6.15.

La structure ASTD *choix quantifié* de la figure 6.8 contient une structure ASTD *garde*. Pour que les opérations appelées sur l'instance du processus `QChoice` puissent s'exécuter, l'instance doit aussi pouvoir interagir avec les instances du processus `Guard`. Cette interaction n'est pas directe, mais se fait à travers une *autre* instance du processus `ASTD`, car la logique d'aiguillage est mise en commun dans le processus `ASTD` et non disséminée dans les neuf processus BPEL correspondant à chaque type

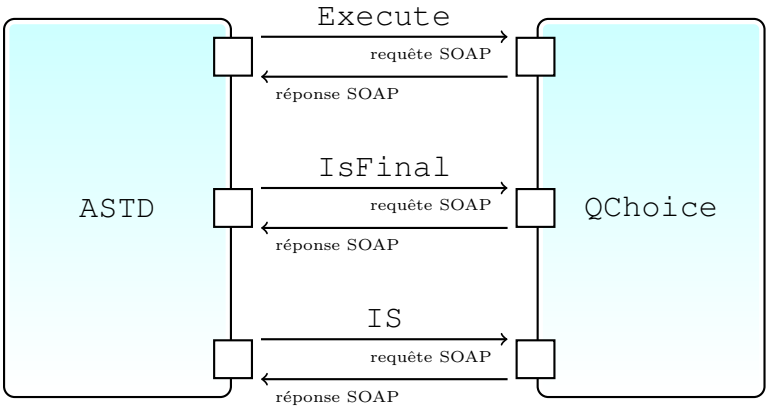


Figure 6.15 – Interactions entre les processus `ASTD` et `QChoice`

de structure ASTD. La figure 6.16 montre une instance de processus `QChoice` qui interagit avec une instance du processus `Guard` à travers une instance du processus `ASTD`. La création de nouvelles instances repose sur le mécanisme des *corrélations* utilisé par le langage BPEL.

Tous les processus du filtre construisent un identifiant unique pour la corrélation du processus avec lequel ils doivent interagir, à partir de leur propre identifiant. Ce procédé assure un réservoir d'identifiants uniques. La figure 6.17 illustre l'évolution de l'identifiant des processus pour une séquence d'opérations donnée. Dans cet exemple, le processus `Filter` est instantié avec l'identifiant `T101`. Lorsque cette instance uti-

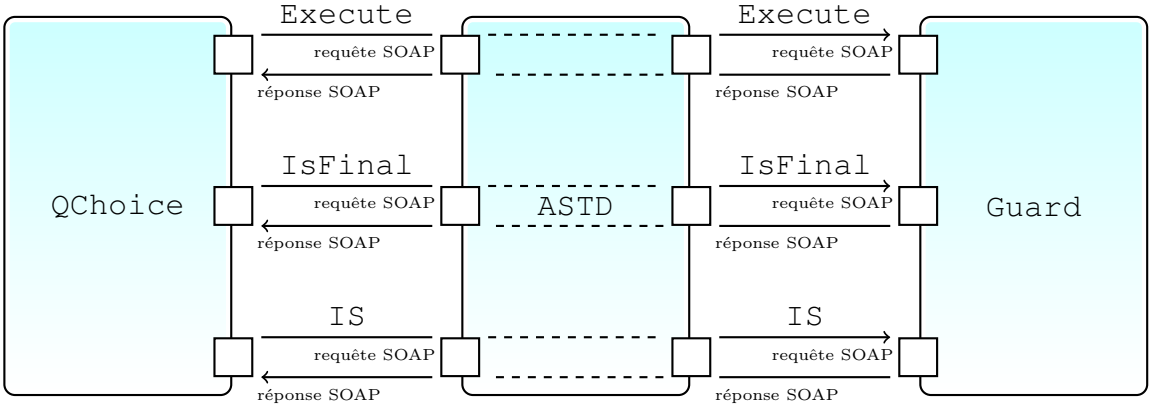


Figure 6.16 – Interactions entre `QChoice` et `Guard` à travers `ASTD`

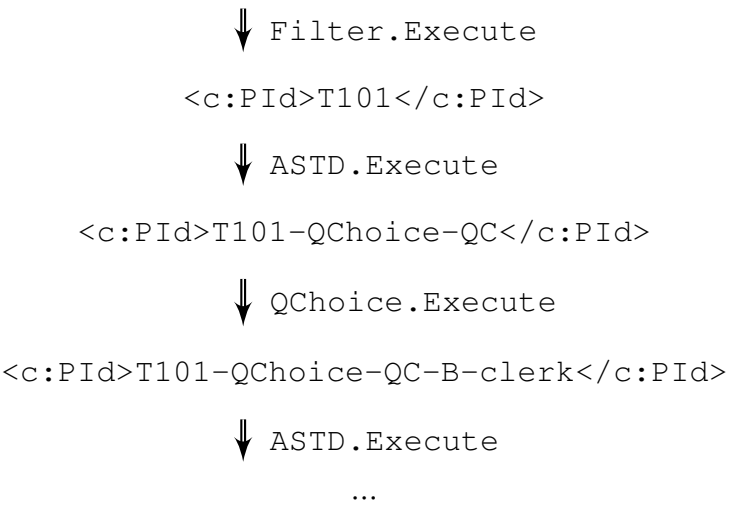


Figure 6.17 – Évolution des identifiants des processus

lise l'opération `Execute` du processus `ASTD`, elle utilise le même identifiant. L'instance du processus `ASTD` est alors identifiée par l'identifiant `T101`. Cette instance crée ensuite l'identifiant `T101-QChoice-QC` qu'elle utilise pour appeler l'opération `Execute` du processus `QChoice`.

6.4 Interprétation de spécifications ASTD

Comme il a été mentionné précédemment, le filtre comporte dix processus BPEL qui permettent d'interpréter des spécifications ASTD. Les processus BPEL `Automaton`, `Sequence`, `Choice`, `KleeneClosure`, `Synchronization`, `Guard`, `QChoice`, `QSynchronization` et également `Call` implémentent les règles sémantiques formelles du langage ASTD, tandis que le processus `ASTD` permet d'aiguiller les appels d'opérations entre ceux-ci. Dans cette section, l'implémentation faite de l'interprétation des structures ASTD est décrite à travers l'exemple de la figure 6.8.

Lorsque le processus `QChoice` reçoit la requête SOAP de l'opération `Execute` avec un nouvel identifiant de processus unique, une nouvelle instance de processus est créée puisque la requête est récupérée par une branche **`onMessage`**. La branche fait partie d'une activité **`pick`** marquée avec l'attribut **`createInstance`** et la valeur

```
1  <soap:Envelope
2    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
3    xmlns:t="http://gril.udes.ca/astd/schema/Types"
4    xmlns:a="http://gril.udes.ca/astd/schema/ASTD"
5    xmlns:c="http://gril.udes.ca/astd/schema/Common">
6   <soap:Header/>
7   <soap:Body>
8    <t:ExecuteRequestType>
9     <c:PId>T101-QChoice-QC</c:PId>
10    <t:ASTD>
11     <a:QChoice a:Name="QC">...</a:QChoice>
12    </t:ASTD>
13    <t:State>
14     <a:QChoiceState a:V="">...</a:QChoiceState>
15    </t:State>
16    <t:Event c:Name="deposit">...</t:Event>
17    <t:Environment/>
18   </t:ExecuteRequestType>
19  </soap:Body>
20 </soap:Envelope>
```

Programme 6.10 – Requête SOAP de l'opération `Execute`

`yes` pour cet attribut. Comme le montre le programme 6.10, la requête contient l'encodage XML de la spécification (lignes 10 à 12) et son état courant (lignes 13 à 15), l'événement (ligne 16) pour lequel un nouvel état est calculé et un environnement vide (ligne 17).

Le traitement de la requête SOAP pour l'opération `Execute` dans le processus `QChoice` est illustré à la figure 6.18. À la réception de la requête, l'environnement initialement vide est mis à jour (comme décrit à la section 6.2.3) pour insérer la variable de quantification de la structure ASTD. Deux situations peuvent se présenter ensuite : une valeur a été attribuée à la variable de quantification ou pas (attribut `a:V` de la ligne 14 du programme 6.10) dans l'état courant de la spécification. Dans les deux cas, une liste de valeurs est constituée, avec soit la valeur attribuée à la variable de quantification, soit la liste des valeurs du type de la variable de quantification obtenue en utilisant la feuille de style `GetTypeValues.xsl` (voir la section 6.2.4). Dans une activité **`while`**, chacune des valeurs de cette liste est insérée dans l'environnement pour la variable de quantification, un nouvel identifiant de processus est créé et une nouvelle requête SOAP est préparée.

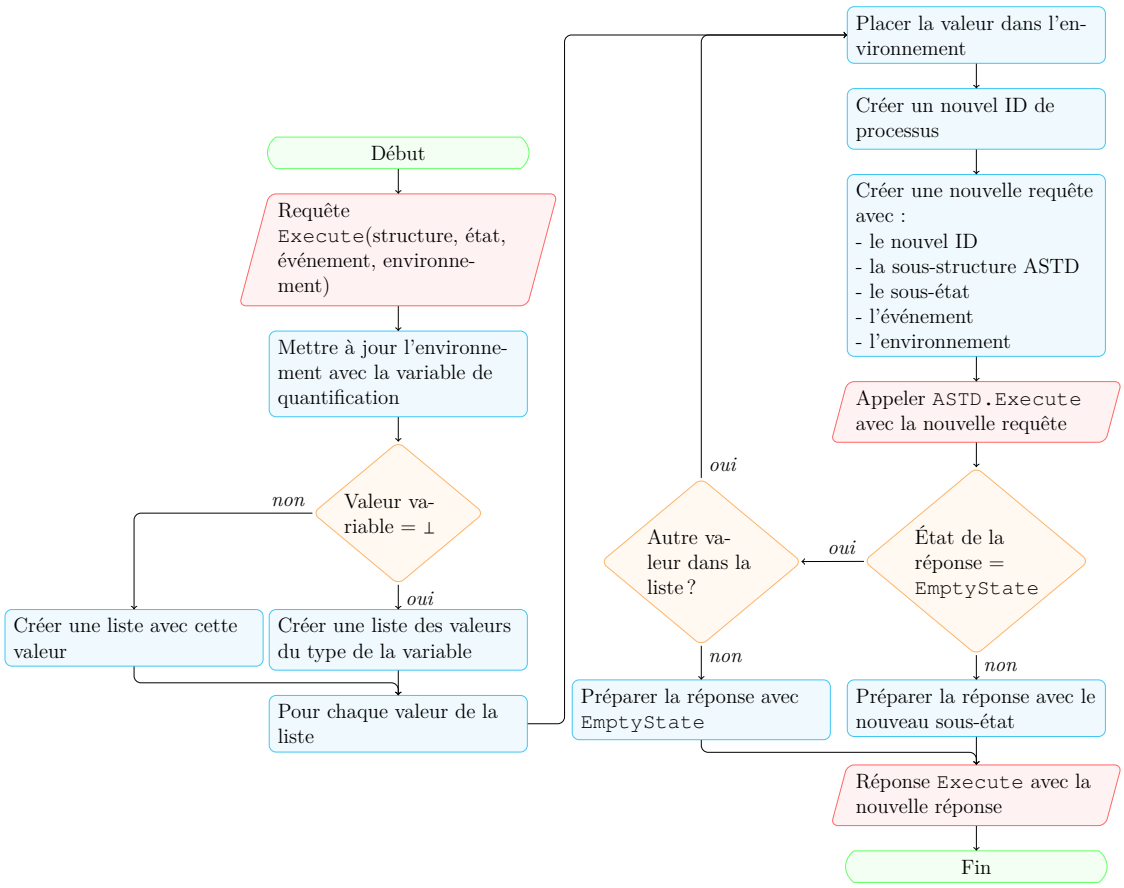


Figure 6.18 – Organigramme de l'opération `Execute` du processus `QChoice`

La nouvelle requête SOAP contient le nouvel identifiant, la sous-structure ASTD *garde*, le sous-état inclus dans l'état du *choix quantifié* (par exemple, les lignes 4 à 10 dans le programme 6.2), l'événement reçu dans la requête qui a permis de créer l'instance courante du processus et enfin l'environnement avec la valeur courante de la liste. L'instance de processus émet ensuite la nouvelle requête SOAP au processus `ASTD` par le lien de partenaire `SubASTDPtrLnk` et reçoit la réponse SOAP contenant l'état de la structure ASTD *garde* pour l'événement considéré. Si ce nouvel état de la garde est différent de `EmptyState`, la recherche s'arrête et une réponse SOAP, est préparée contenant l'état de la garde, puis retournée. La réponse préparée contient un état de type `QChoiceState` dont le sous-élément XML `a:S` est le sous-état retourné par l'appel précédent à `Execute`. Dans le cas contraire les valeurs suivantes de la

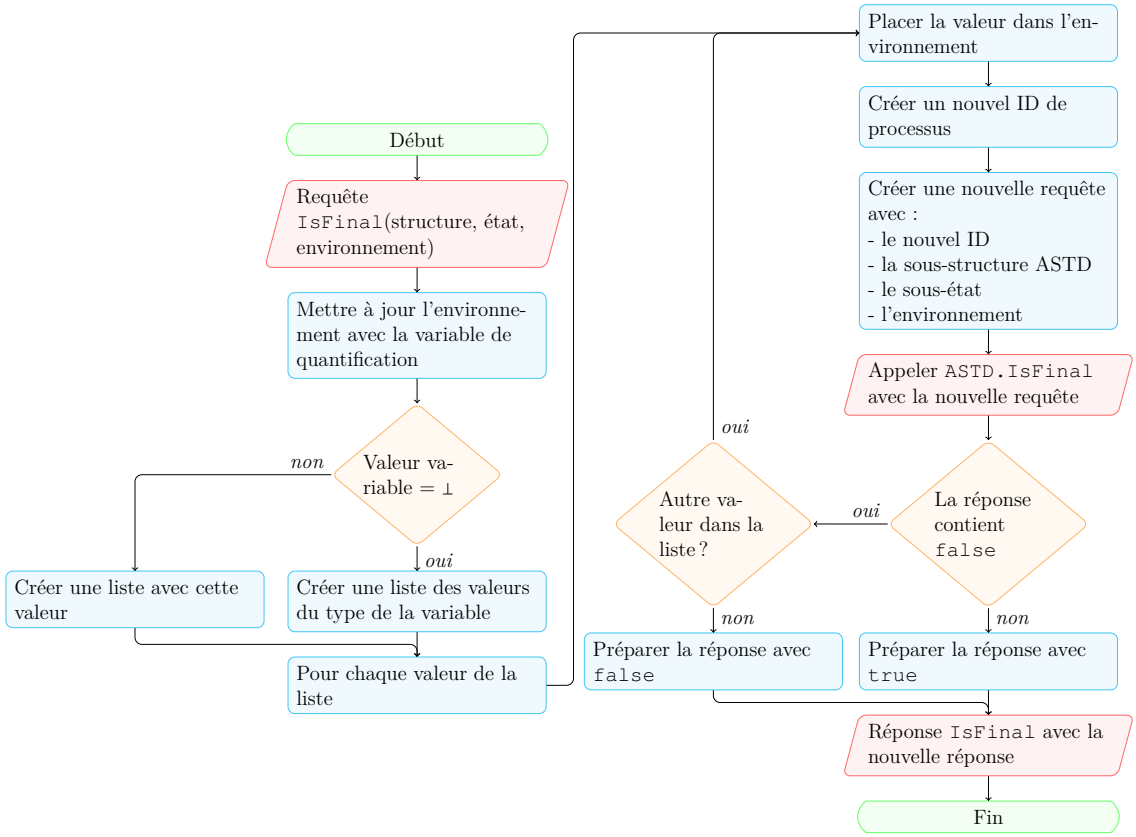


Figure 6.19 – Organigramme de l'opération `IsFinal` du processus `QChoice`

liste sont explorées de façon similaire. À l'issue de l'exploration de toutes les valeurs de la liste, une réponse SOAP contenant l'état spécial `EmptyState` est préparée, puis retournée à l'instance de processus `ASTD` appelant. Cette réponse indique qu'un nouvel état de la spécification n'a pas pu être atteint pour l'événement considéré.

Les opérations `IsFinal` et `IS` (voir respectivement les figures 6.19 et 6.20) sont implémentées de façon similaire, toujours en exploitant la sémantique formelle du langage ASTD.

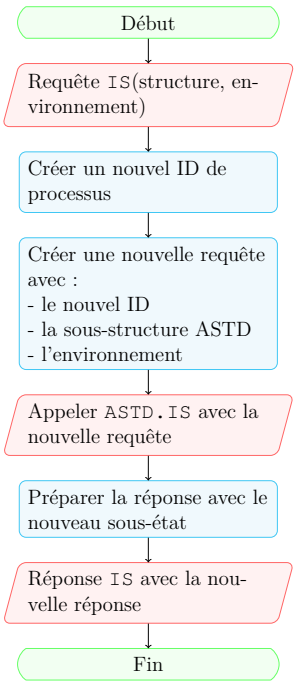


Figure 6.20 – Organigramme de l'opération `IS` du processus `QChoice`

6.5 Conclusion

La solution présentée dans ce chapitre repose sur une méthode d'interprétation de spécifications formelles écrites en ASTD. Tout comme la solution du chapitre précédent, elle constitue une approche de calcul des décisions d'autorisation pour des politiques de contrôle d'accès dynamique. Cette solution semble plus élégante que celle du chapitre précédent car elle est calquée sur la sémantique opérationnelle du langage ASTD, aussi bien pour les traitements que pour les structures des données. Sa mise en œuvre demeure néanmoins complexe considérant les formats d'encodage des spécifications ASTD et des messages émis ainsi que des détails de programmation des processus BPEL déployés dans un environnement SOA.

Chapitre 7

Expérimentation

Une expérience réalisée à partir d'un banc d'essai a permis de valider la faisabilité et d'évaluer la performance des deux méthodes d'implémentation du filtre de contrôle d'accès dynamique (simplement nommé filtre dans la suite). La validation est faite sous la forme de tests unitaires qui sont aussi utilisés pour une partie de l'évaluation de la performance. L'expérience inclut également l'intégration du filtre dans un PEM qui interagit avec un système d'information simplifié d'une banque, tous deux déployés dans un environnement SOA. Cette partie du banc d'essai permet d'évaluer la performance globale d'un système sécurisé.

7.1 Description des jeux d'essai

Les jeux d'essai visent à évaluer les deux implémentations du filtre. Les données d'essai comportent des spécifications ASTD et pour chacune d'elles des séquences d'actions à tester ainsi que les résultats prédéterminés afin de les comparer avec les résultats obtenus lors de l'expérience. Ces données d'essai sont rassemblées dans un fichier qui respecte le format XML décrit dans le programme 7.1. Les lignes 3 à 5 contiennent une spécification ASTD et les lignes 6 à 16 incluent ses séquences d'actions (`TestCase`) imbriquées dans la balise `TestSuite`. Chaque séquence commence avec l'action `Init` (opération d'initialisation du filtre) et termine avec l'action `Stop`

```
1  <ax:Tests xmlns:ax="http://gril.udes.ca/astd/schema/ASTD">
2   <ax:Test ax:Name="T001">
3    <ax:Specification>
4     ...
5    </ax:Specification>
6    <ax:TestSuite>
7     <ax:TestCase ax:Functionality="true">
8      <ax:Init/>
9      <ax:Event ax:Name="T1" ax:ResultAccess="granted"/>
10     <ax:Commit ax:ResultIsFinal="true" ax:ResultIsCompleted="true"/>
11     ...
12     <ax:Stop/>
13    </ax:TestCase>
14    <ax:TestCase ax:Functionality="true">
15    ...
16   </ax:TestSuite>
17  </ax:Test>
18 ...
19 </ax:Tests>
```

Programme 7.1 – Fichier des données d'essai

(opération d'arrêt du filtre). Les autres actions possibles sont `Event` (opération de demande d'autorisation pour un événement) et les actions `Commit`/`Rollback`. Dans le cas de l'action `Event` (ligne 9), l'élément XML comporte le nom de l'événement et ses paramètres (le cas échéant) ainsi que la valeur de retour attendue pour la demande d'autorisation (`granted` ou `denied`). À la ligne 10, l'action `Commit` est utilisée avec les valeurs booléennes attendues des variables de débogage `IsFinal` et `IsCompleted`. L'action `Rollback`, qui n'est pas illustrée dans cet exemple, utilise aussi les même variables de débogage. À partir du fichier de données d'essai, trois groupes de fichiers sont créés : le groupe de fichiers relatif à la méthode de transformation, le groupe de fichiers relatif à la méthode d'interprétation et le groupe de fichiers projet de l'outil d'exécution des tests et des fichiers de scripts pour piloter l'exécution des jeux d'essai et générer les statistiques.

L'algorithme de transformation détaillé au chapitre 5 est implémenté dans un programme écrit en OCaml [1] et dans un ensemble de feuilles de style XSL. Puisque les résultats attendus de la transformation sont i) des processus BPEL, ii) le document d'interfaces WSDL et iii) les fichiers de déploiement pour le moteur d'exécution ODE, l'utilisation d'un processeur XSLT et de feuilles de style XSL semble appropriée car

1. Site Web du langage OCaml : caml.inria.fr/ocaml

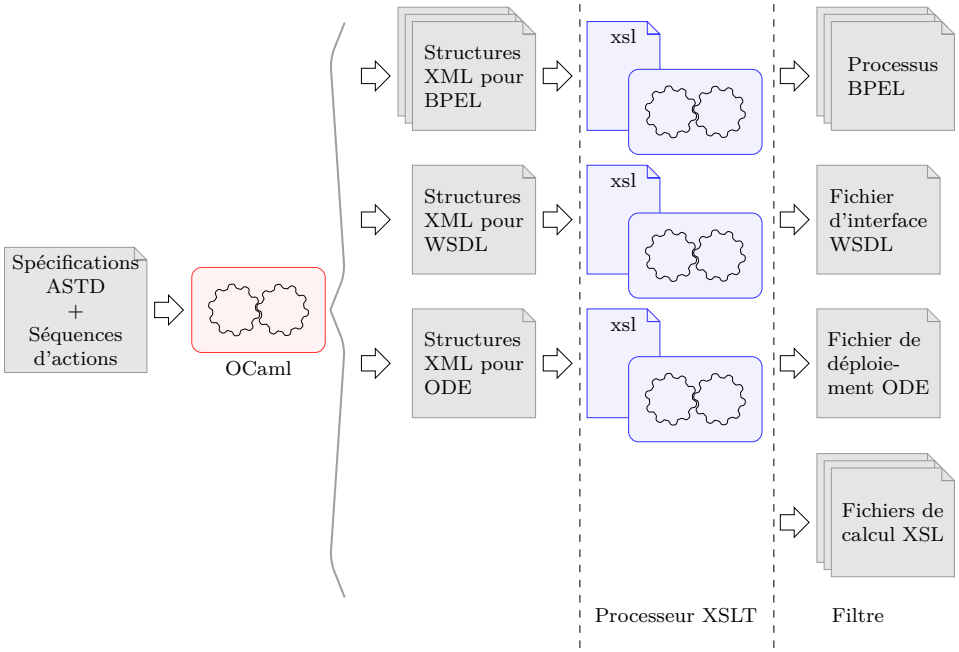


Figure 7.1 – Création du filtre de contrôle d'accès

tous ces fichiers sont dans un format XML. Le rôle du programme OCaml est de diriger la transformation en extrayant les spécifications ASTD du fichier de données d'essai. Une fois une spécification extraite, le programme la réorganise en fonction de la sortie attendue (respectivement les processus BPEL, le document d'interfaces WSDL et le fichier de déploiement) et traite ces nouveaux documents XML à l'aide du processeur XSLT et de la feuille de style correspondante (respectivement `Process.xsl`, `FilterProcessInterface.xsl` et `Deploy.xsl`). Ce traitement est illustré à la figure 7.1 et il est répété pour chaque spécification ASTD du fichier de données d'essai. Le nombre de documents XML (désigné par d) générés par le programme OCaml est fonction de la spécification ASTD :

$d = 1 + $*nombre de structures appel, fermeture de Kleene et synchronisation quantifiée.*

Ce nombre est également le nombre de processus BPEL créés. Chaque groupe de fichiers créés pour le filtre est enregistré dans un dossier correspondant à la spécification ASTD transformée.

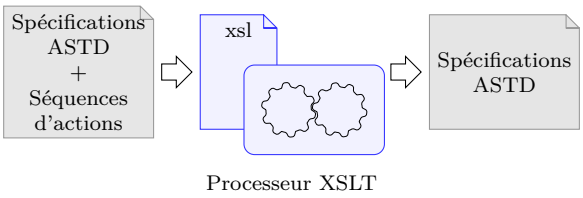


Figure 7.2 – Extraction des spécifications du filtre de contrôle d'accès

Dans le cas de la méthode d'interprétation décrite au chapitre 6, les fichiers de processus ne sont pas à créer parce qu'ils constituent l'interpréteur. Cependant, les spécifications ASTD sont extraites du fichier de données d'essai et rassemblées dans un fichier XML pour être interprétées par le filtre. L'action `Init` d'une séquence d'actions indique, par la valeur de son paramètre `PId`, laquelle des spécifications sera testée. L'extraction des spécifications se fait par un processeur XSLT et une feuille XSL appropriée, comme illustrée à la figure 7.2.

Les séquences d'actions exécutées sont concrètement des appels de services Web. Ces appels sont des messages SOAP envoyés par un outil approprié de test des services Web. L'outil choisi pour notre expérience est *soapUI*[2]. En effet, il offre une interface graphique ainsi qu'un programme utilisable en ligne de commande. Les requêtes SOAP sont créées à partir du fichier de données d'essai et sont rassemblées dans deux fichiers projet (un pour la transformation et un pour l'interprétation) propres à l'outil *soapUI*. À chaque requête SOAP sont attachées des assertions qui valident que la réponse SOAP reçue à l'issue de l'envoi du message est conforme aux valeurs attendues. À titre d'illustration pour la demande d'autorisation de la ligne 9 du programme 7.1, la requête SOAP correspondante comporte trois assertions dans les deux fichiers projet. La première assertion (*SOAP Response* dans l'outil *soapUI*) valide que le format XML de la réponse reçue est conforme à celui des réponses SOAP. La seconde assertion (*Not SOAP Fault* dans l'outil *soapUI*) valide que la réponse reçue n'est pas une erreur SOAP. La troisième assertion (*Access granted* ou *Access denied* dans l'outil *soapUI*), plus intéressante pour notre étude, valide que la réponse obtenue est conforme à la décision d'autorisation attendue, c'est-à-dire *granted* dans le cas de

2. Site Web de *soapUI* : soapui.org

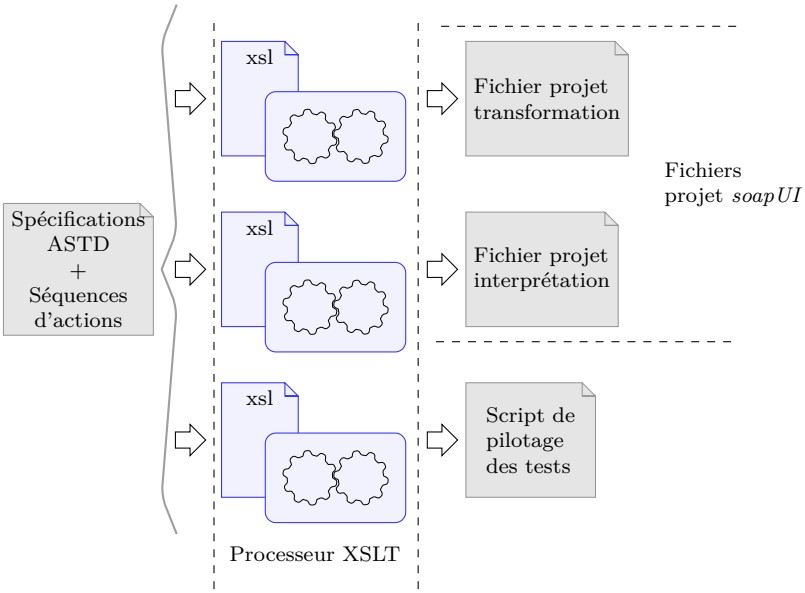


Figure 7.3 – Création des fichiers projet *soapUI*

la ligne 9 du programme 7.1. Dans le cas des actions `Commit/Rollback`, des assertions spécifiques aux valeurs des variables de débogage `IsFinal` et `IsCompleted` sont ajoutées aux messages SOAP correspondants. Les deux fichiers projet sont créés à partir de feuilles de style XSL passées en paramètres (en plus du fichier de données d'essai) à un processeur XSLT (voir la figure 7.3).

À partir du fichier de données d'essai, un script de pilotage des tests unitaires est également créé. À son exécution, ce script déploie les processus BPEL propres à chacune des méthodes et exécute les mêmes séquences d'actions sur les processus déployés. L'exécution de ces séquences d'actions est en fait l'envoi des requêtes SOAP correspondantes dans les fichiers projet *soapUI* et cet envoi se fait par l'utilisation du programme *soapUI* en ligne de commande. L'exécution inclut également la validation des assertions attachées aux messages SOAP ainsi que la prise de mesures de performance (par exemple temps minimum, maximum et moyen). Enfin, le script rassemble dans un même fichier XML les résultats obtenus pour toutes les séquences d'actions de chaque spécification ASTD. La figure 7.3 illustre la création de ce script à partir du fichier de données d'essai.

Tableau 7.1 – Caractéristiques des machines de l'expérience

	Serveur	Client
Processeur	Intel(R) Xeon(R) CPU X5550 @ 2.67GHz	Intel(R) Core(TM) 2 CPU 6400 @ 2.13GHz
Mémoire vive	4 Go	4 Go
Système d'exploitation	Linux Ubuntu 10.04.4 LTS Noyau 2.6.32-41-generic	Linux Ubuntu 10.04.4 LTS Noyau 2.6.32-42-generic
Logiciel	Apache Tomcat 7.0.28 ODE 1.3.5 MySQL 5.1	*soapUI* 4.5.1

7.2 Environnements de test

Les tests sont réalisés sur le réseau local de la Faculté des sciences de l'Université de Sherbrooke. Ils sont exécutés sur deux machines dans une configuration client-serveur. La machine serveur est celle sur laquelle s'exécute le moteur d'exécution ODE[3]. Il s'agit d'une machine virtuelle Linux dont les caractéristiques matérielles et logicielles sont indiquées dans le tableau 7.1. La machine client exécute le programme *soapUI*. Elle est l'hôte des différents fils d'exécution qui agissent comme des clients virtuels pour les processus déployés sur le serveur. Les caractéristiques de cette machine sont aussi incluses dans le tableau 7.1.

7.3 Tests unitaires

Le filtre est déployé et exécuté avec une politique de contrôle d'accès. Cette dernière est exprimée en utilisant les différentes structures du langage ASTD. Les tests réalisés visent à vérifier que la sémantique de chacune des structures ASTD est respectée par chaque implémentation du filtre. Avec l'outil utilisé, il s'agit d'un test de performance qui utilise la stratégie qualifiée de *simple* selon les paramètres du tableau 7.2. Ce qualificatif signifie que, pour chaque séquence d'actions, cinq clients virtuels exécutent de façon concurrente la séquence d'actions pendant 60 secondes avec une pause de 0,5s

3. Site Web du moteur ODE : ode.apache.org

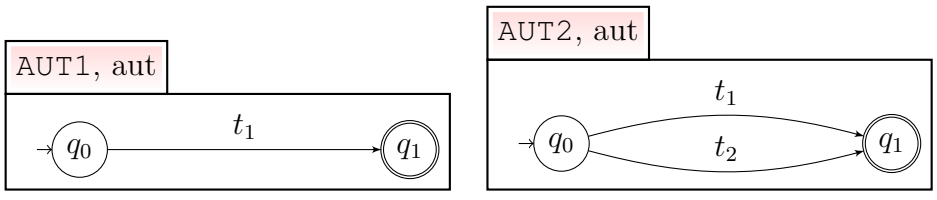
Tableau 7.2 – Paramètres des tests unitaires

N^{bre} de clients	Durée (s)	Délai (s)	Aléatoire
5	60	1	0,5

(*Aléatoire* × *Délai*) à 1s entre chaque exécution de la séquence. De plus, quelque soit la méthode d'implémentation, les processus BPEL du filtre déployé résident toujours en mémoire, car avec le moteur d'exécution ODE il est possible d'avoir une exécution avec ou sans persistance des processus.

7.3.1 La structure ASTD *automate*

Les spécifications des figures 7.4a et 7.4b sont construites uniquement avec une structure ASTD *automate*. Une fois les spécifications déployées dans le filtre, l'outil *soapUI* vérifie que, dans l'état initial q_0, la décision d'autorisation rendue est toujours *granted* à l'occurrence de l'événement t_1 pour les deux spécifications et à l'occurrence de l'événement t_2 pour la deuxième. Il vérifie également qu'il est possible, après avoir reçu la réponse *granted*, de faire un retour en arrière avec l'opération *Commit* (et la valeur *false* de son paramètre) dans le cas de la transformation et *Rollback* dans le cas de l'interprétation. Une fois ce pas en arrière effectué, dans le cas de la méthode par transformation, l'outil s'assure que la réponse obtenue à la suite d'une demande d'autorisation reçue entre une réponse d'autorisation *granted* et l'opération *Commit* suivante est toujours *denied*.



(a) Première spécification (b) Deuxième spécification

Figure 7.4 – Spécifications d'une structure ASTD *automate*

Tableau 7.3 – Exécution sans persistance pour la spécification `AUT1`

		Transformation (ms)	Interprétation (ms)	Résultat
Séquence 1				
1	Init	15.70	67.11	
2	Stop	8.70	12.34	
Séquence 2				
1	Init	14.41	38.01	
2	t_1	15.14	34.14	Granted
3	Commit	11.06	—	
4	Stop	7.88	11.61	
Séquence 3				
1	Init	15.86	36.83	
2	t_1	19.19	28.03	Granted
3	Rollback	10.11	6.16	
4	t_1	7.89	19.55	Granted
5	Rollback	9.32	5.97	
6	t_1	8.81	19.90	Granted
7	t_1	7.22	18.22	Denied
8	Commit	9.34	—	
9	t_1	6.98	21.32	Denied
10	t_2	6.89	16.42	Denied
11	t_3	6.80	19.57	Denied
12	Stop	6.43	6.09	

Le tableau 7.3 contient les temps moyens d'exécution de séquences d'actions pour la spécification de la figure 7.4a. Ces résultats sont présentés graphiquement dans la figure 7.5. Dans la suite, les temps moyens des tests unitaires sont présentés par des histogrammes. L'histogramme de la figure 7.5 montre un temps important au démarrage du filtre de la méthode d'interprétation. Ce temps est attribuable à l'initialisation du moteur d'exécution BPEL. On remarque aussi dans cet histogramme que les temps moyens de calcul des décisions d'autorisation dans le cas de la méthode de transformation sont moins élevés que ceux de la méthode d'interprétation. Cette tendance est observable aussi dans l'histogramme des temps moyens de la figure 7.6 qui correspond aux séquences d'actions pour la spécification de la figure 7.4b.

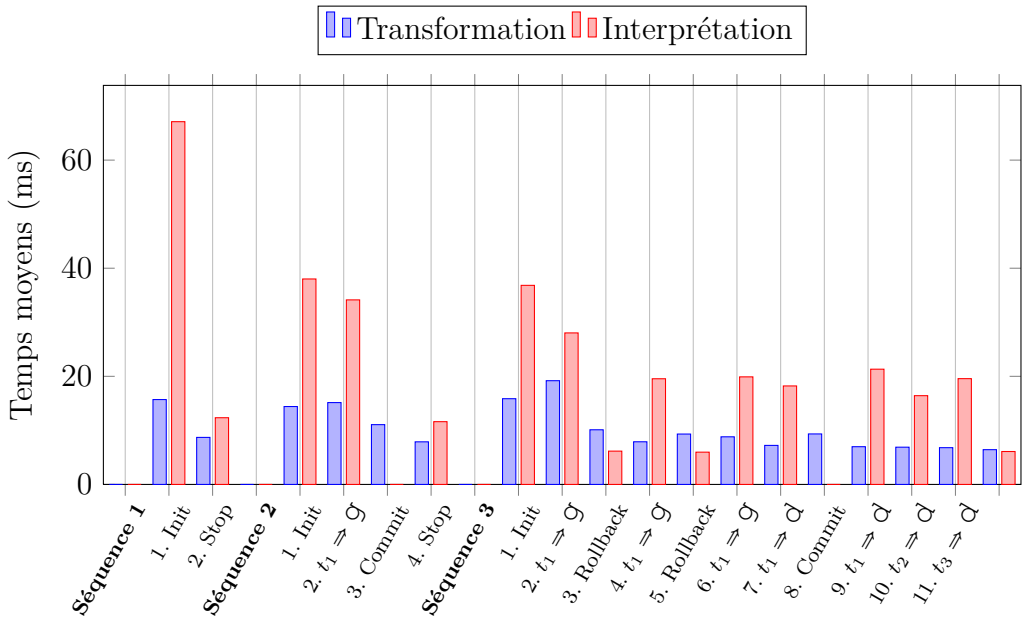


Figure 7.5 – Histogramme des temps moyens de l'*automate* `AUT1`

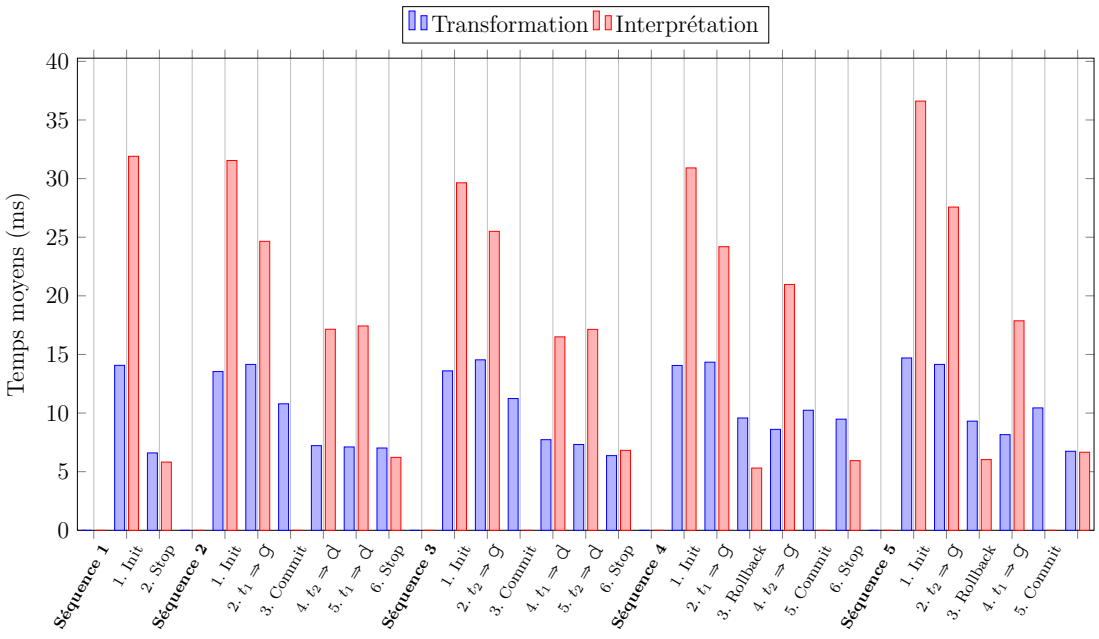


Figure 7.6 – Histogramme des temps moyens de l'*automate* `AUT2`

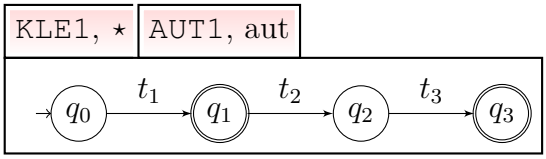


Figure 7.7 – Spécification d'une structure ASTD *fermeture de Kleene*

7.3.2 La structure ASTD *fermeture de Kleene*

La structure ASTD *fermeture de Kleene* permet la mise en œuvre répétée de la même règle de contrôle d'accès spécifiée par la sous-structure membre. La figure 7.7 illustre la structure ASTD *fermeture de Kleene* utilisée pour le test unitaire. La figure 7.8 montre l'histogramme des temps moyens du calcul de décisions d'autorisation pour quelques séquences d'actions. La première séquence est l'initialisation puis l'arrêt du filtre. La seconde séquence est essentiellement la répétition (trois fois) de la suite $t_1 \rightarrow t_2 \rightarrow t_3$. La troisième séquence montre que la fermeture de Kleene peut recommencer à son état initial dès que la sous-structure est dans l'état final q_1 par la répétition de l'action t_1. Dans cet histogramme, le lecteur peut observer qu'encore une fois les temps moyens pour la méthode d'interprétation sont plus élevés que ceux de la méthode de transformation. Cet histogramme montre également un temps moyen de calcul important lorsque la sous-structure est dans son état final q_3 et que le filtre re-

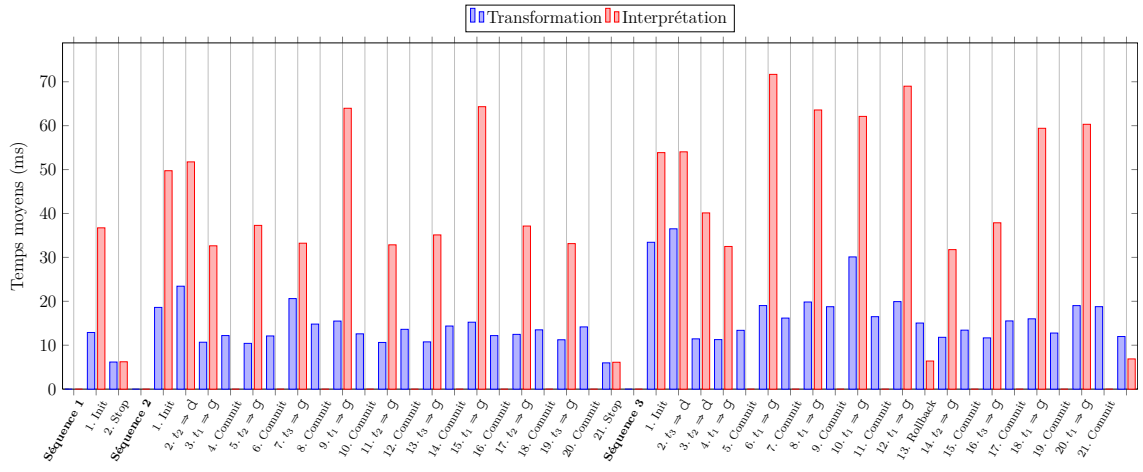


Figure 7.8 – Histogramme des temps moyens de la structure `KLE1`

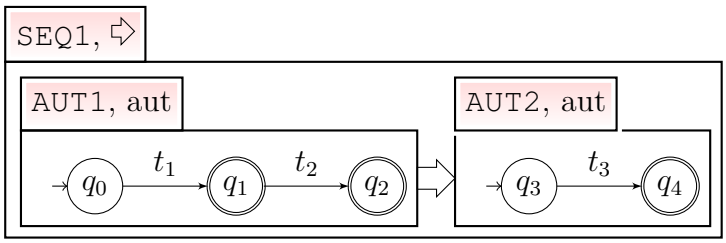


Figure 7.9 – Spécification d'une structure ASTD *séquence*

çoit la requête t_1 qui place la sous-structure dans l'état initial q_0 et ensuite dans l'état q_1 après la réponse à la requête d'autorisation (voir la séquence 2 de l'histogramme).

7.3.3 La structure ASTD *séquence*

La structure ASTD *séquence* permet l'exécution à la suite l'une de l'autre de deux sous-structures ASTD. Dans le cas de politiques de contrôle d'accès, elle permet d'imposer la mise en œuvre d'une règle spécifiée par le premier membre de la structure *séquence* avant d'imposer la mise en œuvre de la règle spécifiée par le second membre. La structure *séquence* de la figure 7.9 est utilisée dans le test unitaire. La séquence 2 de l'histogramme de la figure 7.10 montre que la structure ASTD *séquence* telle que spécifiée accepte les occurrences des événements t_1, t_2 et t_3 dans cet ordre, c'est-à-dire que les décisions d'autorisation rendues sont toutes `granted` (abrégé `g` dans

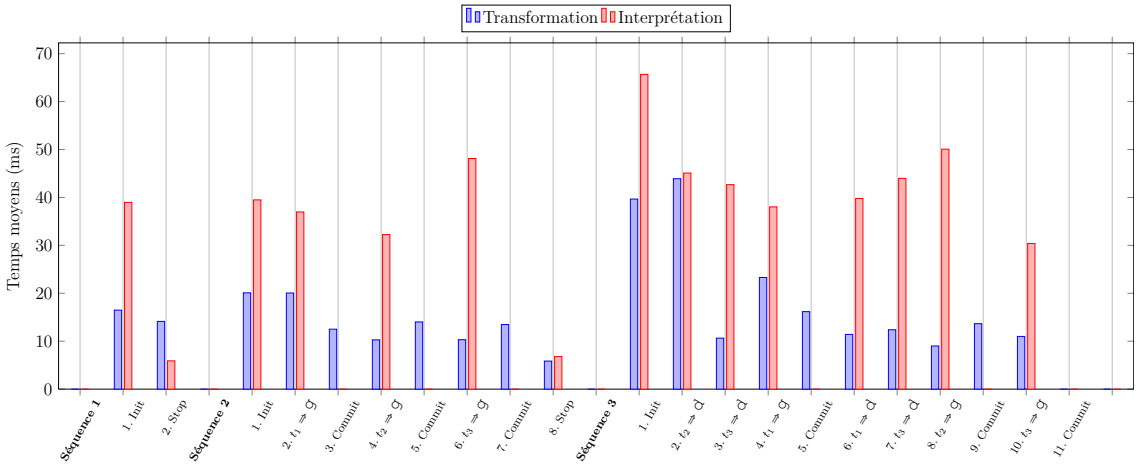


Figure 7.10 – Histogramme des temps moyens de la structure `SEQ1`

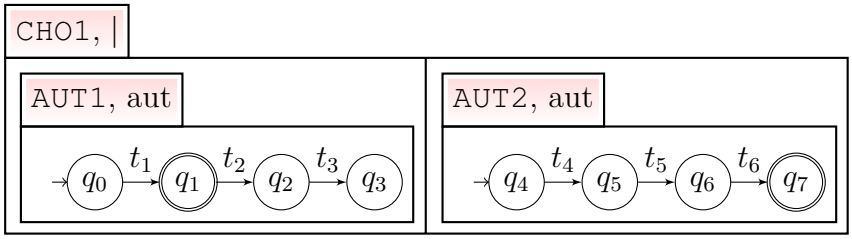


Figure 7.11 – Spécification d'une structure ASTD *choix*

l'histogramme). La séquence 3 montre que l'événement t_2 n'est pas autorisé avant l'événement t_1 et que l'événement t_3 n'est pas autorisé avant les événements t_1 et t_2.

7.3.4 La structure ASTD *choix*

La structure ASTD *choix* permet de sélectionner l'exécution d'une sous-structure membre suivant l'occurrence d'un événement acceptable. La figure 7.11 est la spécification de la structure ASTD *choix* utilisée pour le test. Dans l'histogramme de la figure 7.12 obtenu avec les paramètres de tests du tableau 7.2, la séquence d'actions 2 (respectivement 3) montre que la structure `CHO1` accepte l'occurrence des événements t_1, t_2 et t_3 (respectivement t_4, t_5 et t_6) dans cet ordre. La séquence d'actions 4 (respectivement 5) montre qu'une fois que l'occurrence de l'événement t_1 (respectivement t_4) est acceptée, c'est-à-dire que la décision *granted* est rendue et que l'action est

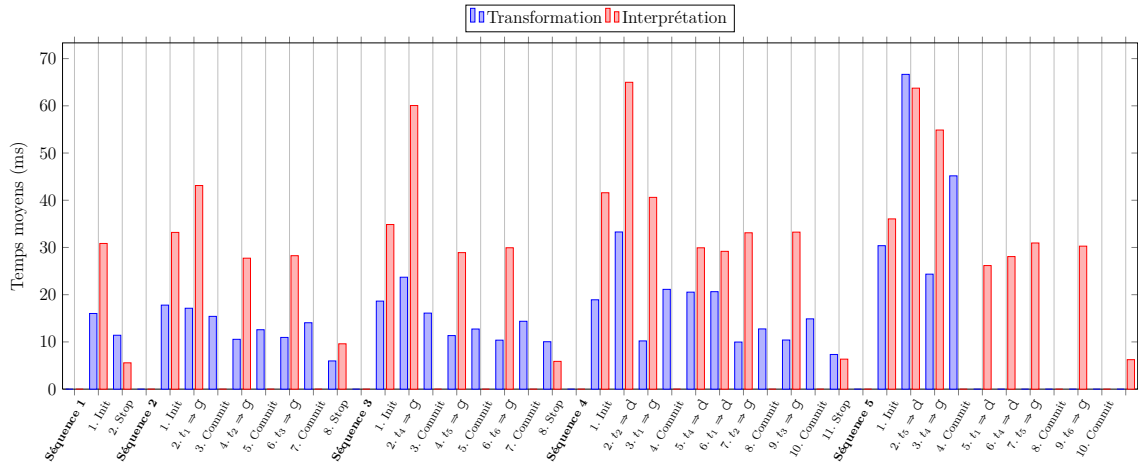


Figure 7.12 – Histogramme des temps moyens de la structure `CHO1`

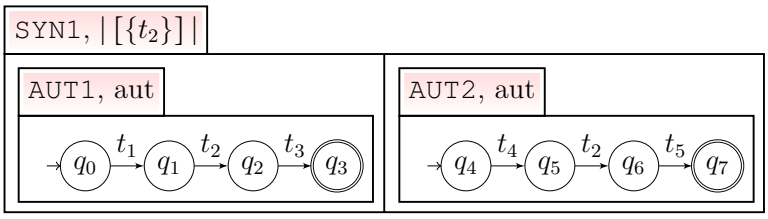


Figure 7.13 – Spécification d'une structure ASTD *synchronisation paramétrée*

validée par l'opération `Commit`, l'événement t_4 (respectivement t_1) obtient la décision d'autorisation *denied*. Cette situation illustre bien le fait qu'à l'occurrence de l'événement t_1 (respectivement t_4) le choix de l'exécution de la sous-structure `AUT1` (respectivement `AUT2`) est correctement fait. L'histogramme dévoile deux anomalies dans la séquence 5 pour la méthode de transformation. Premièrement, l'action t_5 qui est refusée (réponse *denied* abrégée `d` dans l'histogramme) consomme un temps démesuré par rapport aux autres actions. Cette anomalie est difficilement explicable. Deuxièmement, les temps moyens pour les dernières actions de la séquence 5 sont absents. Cette anomalie peut s'expliquer par une surcharge du serveur qui a abandonné l'exécution des processus à partir de la cinquièmement action.

7.3.5 La structure ASTD *synchronisation paramétrée*

Une structure ASTD *synchronisation paramétrée* permet l'exécution en parallèle de deux sous-structures tout en forçant leur synchronisation sur un ensemble d'événements spécifié, c'est-à-dire que les occurrences des événements de l'ensemble de synchronisation doivent être acceptées simultanément par les deux sous-structures. La structure ASTD *synchronisation paramétrée* utilisée dans le test unitaire est illustrée dans la figure 7.13. Dans celle-ci, les deux sous-structures `AUT1` et `AUT2` doivent se synchroniser sur l'événement commun t_2. L'histogramme obtenu pour les temps moyens de l'exécution de quelques séquences d'actions est montré à la figure 7.14. Dans celui-ci, la séquence d'actions 2 montre que l'événement t_2 est accepté après les événements t_1 et t_4 et avant les événements t_3 et t_5. La séquence d'actions 3 montre que l'événement t_2 ne peut pas être accepté avant que les sous-structures `AUT1` et `AUT2` ne soient toutes deux prêtes à l'accepter, c'est-à-dire que les deux événements t_1 et t_4 aient été acceptés.

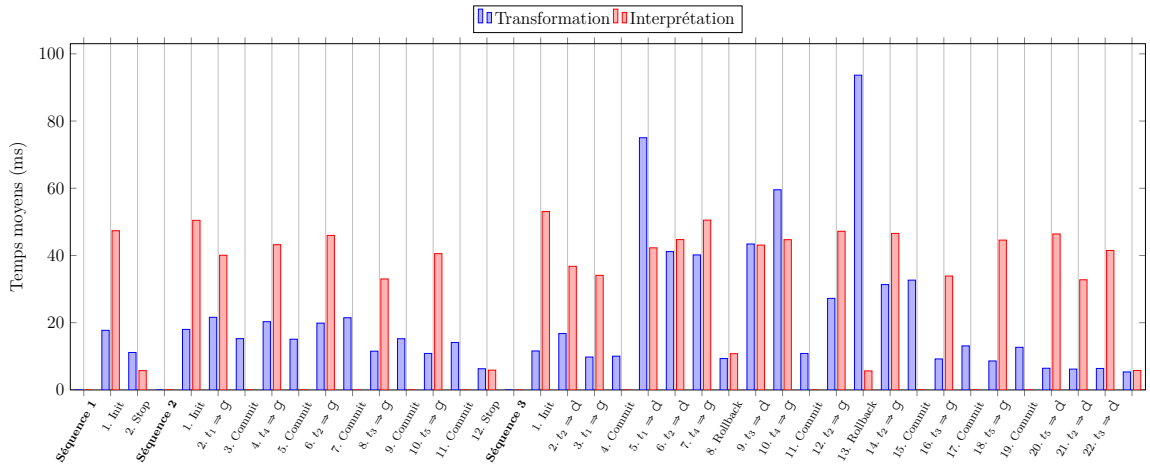


Figure 7.14 – Histogramme des temps moyens de la structure `SYN1`

7.3.6 La structure ASTD *appel*

La structure ASTD *appel* permet de faire référence à la spécification d'une autre structure ASTD en lui passant des valeurs pour ses paramètres. Cette référence s'effectue à la manière d'un appel de fonction. Dans la spécification de la figure 7.15, la structure automate `AUT1` est appelée avec la valeur 3 pour son paramètre x. L'histogramme obtenu pour les temps moyens d'exécution de quelques séquences d'actions est donné à la figure 7.16. Dans la séquence d'actions 2, les événements t_1 à t_3 sont acceptés avec la valeur 3 du paramètre x tel que spécifié par la structure *appel*. Cette séquence d'actions montre aussi une valeur importante du temps moyen d'exécution pour l'occurrence de l'événement t_1. Dans le cas de la méthode de transformation, elle est attribuable à la mise à jour de l'environnement à l'aide d'une feuille de style XSL et à la création d'une instance du processus résultant de la transformation de la structure `AUT1` par l'appel de l'opération `Init` de ce processus BPEL. Les occur-

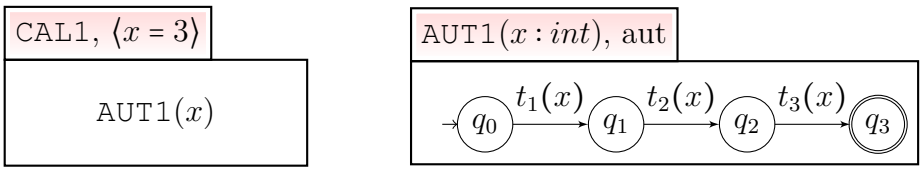


Figure 7.15 – Spécification d'une structure ASTD *appel*

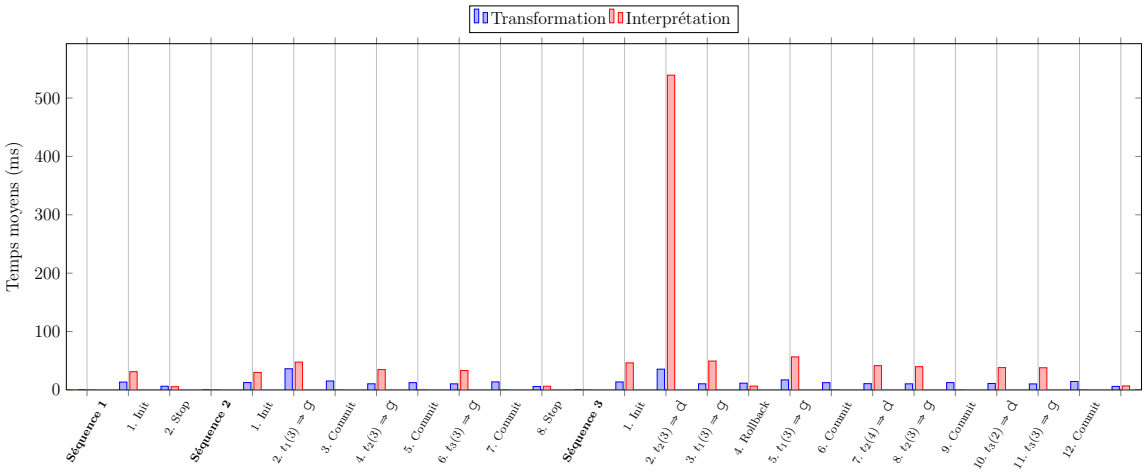


Figure 7.16 – Histogramme des temps moyens de la structure `CAL1`

rences des événements suivants t_2 et t_3 ont des temps moyens moins importants car le processus BPEL pour la sous-structure est déjà initialisé et l'environnement pour la structure `CAL1`, contenant la variable x et sa valeur 3, est déjà calculé. La séquence d'actions 3 montre que l'événement t_1 obtient la décision *`denied`* pour une autre valeur que celle spécifiée par la structure `CAL1`.

7.3.7 La structure ASTD *garde*

La structure ASTD *garde* permet l'exécution de sa sous-structure lorsque le prédicat de la garde s'évalue à la valeur *vrai*. Après l'occurrence d'un événement accepté, le prédicat n'est plus évalué. La structure `GR1` spécifiée dans la figure 7.17 accepte des occurrences d'événements seulement pour des valeurs de la variable quantifiée x comprise entre −1 et 1. Les prédicats spécifiés pour les structures ASTD *garde* portent le plus souvent sur des variables et, dans le cas de la structure `GR1`, la variable est introduite par la structure parente `QCH1` qui est un choix quantifié. La figure 7.18 contient l'histogramme des temps moyens de l'exécution de quelques séquences d'actions. Il montre, dans le cas de la transformation et de l'interprétation, des temps moyens importants pour les actions tant qu'un premier événement n'est pas accepté. Ce temps moyen correspond à l'évaluation de la garde à l'aide d'une expression XPath dans une activité BPEL **`assign`** dans le cas de la méthode de transformation et à

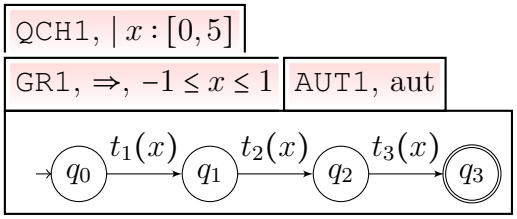


Figure 7.17 – Spécification d'une structure ASTD *garde*

l'aide d'une feuille de style XSL dans le cas de la méthode d'interprétation, ainsi qu'à l'utilisation de la structure ASTD *choix quantifié* `QCH1` qui induit une recherche d'une valeur acceptable de la variable x. La séquence d'actions 2 de l'histogramme montre que les événements t_1 et t_2 sont acceptés avec la valeur 0 de la variable x. Il faut remarquer que bien que le prédicat de la garde autorise les valeurs entières −1, 0 et 1 pour cette variable, la structure parente `QCH1` ne permet que les valeurs 0 à 5. Cette situation est d'ailleurs illustrée par les séquences d'actions 3 et 4 qui montrent que l'événement t_1 reçoit la décision d'autorisation *denied* pour les valeurs −1 et 2.

7.3.8 La structure ASTD *choix quantifié*

La structure ASTD *choix quantifié* spécifie l'exécution d'une sous-structure pour une valeur de sa variable de quantification prise dans un ensemble de quantification.

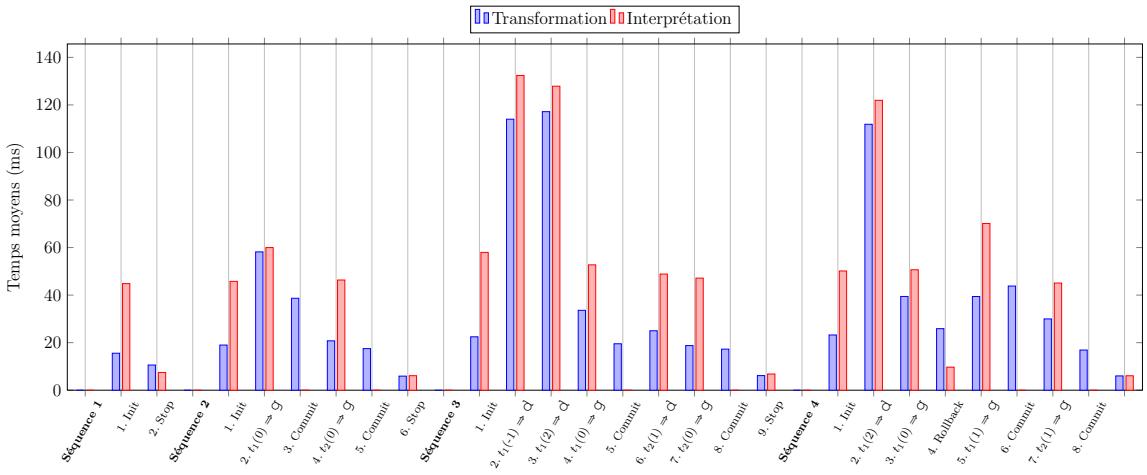


Figure 7.18 – Histogramme des temps moyens de la structure `GR1`

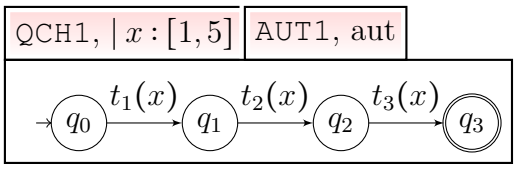


Figure 7.19 – Spécification d'une structure ASTD *choix quantifié*

Le choix de la valeur se fait à la première acceptation de l'occurrence d'un événement. Dans l'exemple de la figure 7.19, la variable de quantification est x et l'ensemble de quantification est l'intervalle $[1, 5]$. L'histogramme obtenu pour les temps moyens de l'exécution de quelques séquences d'actions pour cette structure est montré à la figure 7.20. La séquence d'actions 2 montre que les événements t_1 à t_3 sont acceptés pour la valeur 3 de la variable x. La séquence d'actions 3 montre que l'événement t_1 obtient la décision d'autorisation `granted` pour la valeur 3 de la variable x et l'action `Commit` (avec le valeur *false*) (respectivement `Rollback`) annule dans le filtre implémenté par la méthode de transformation (respectivement la méthode d'interprétation) la décision précédemment obtenue. Cette séquence montre aussi que l'événement suivant t_1 avec la valeur 2 du paramètre x est acceptée. Dans le cas des deux séquences, les temps moyens des premières actions (celle refusée et celle permise et validée) sont plus importants car une valeur acceptable pour la variable de quantification doit être recherchée dans l'ensemble de quantification. Pour une valeur

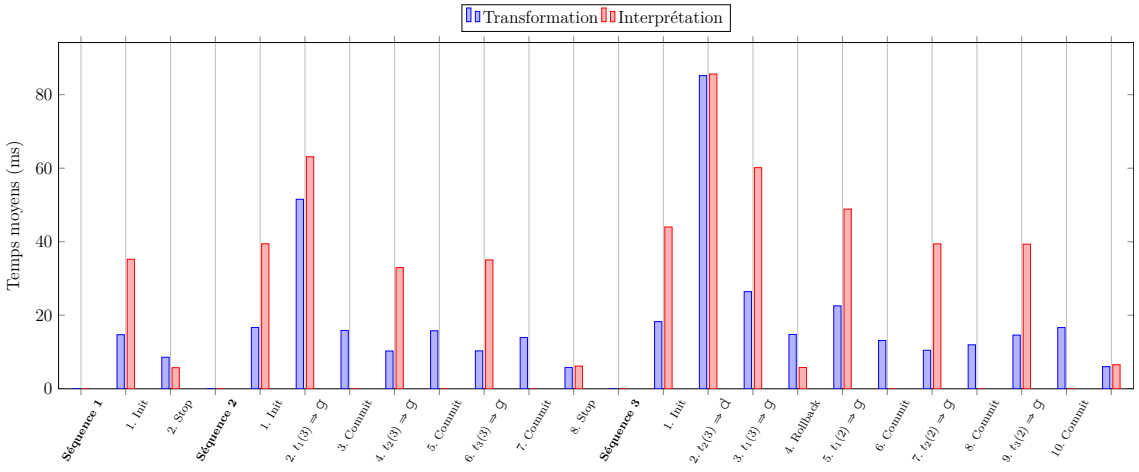


Figure 7.20 – Histogramme des temps moyens de la structure `QCH1`

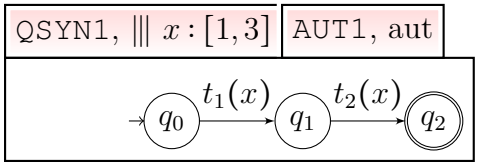


Figure 7.21 – Spécification d'une structure ASTD *synchronisation quantifiée*

trouvée et un événement accepté, c'est-à-dire permis et validé, les occurrences des événements subséquents n'entraînent plus de recherche car la valeur trouvée précédente est réutilisée dans les deux solutions d'implémentation.

7.3.9 La structure ASTD *synchronisation quantifiée*

La structure ASTD *synchronisation quantifiée* permet l'exécution en parallèle de sa sous-structure pour différentes valeurs de la variable de quantification. Les différentes exécutions de cette structure ASTD sont synchronisées sur les événements d'un ensemble de synchronisation, à la manière d'une structure ASTD *synchronisation paramétrée*. La structure ASTD *synchronisation quantifiée* de la figure 7.21 utilisée dans le test unitaire a un ensemble de synchronisation vide, c'est donc un *entrelacement quantifié*. Les temps moyens obtenus pour quelques séquences d'actions sont représentés sous la forme d'un histogramme dans la figure 7.22. Cet histogramme montre que les temps moyens pour la méthode de transformation sont moins élevés que les temps moyens pour la méthode d'interprétation. Cette observation est valide quelque soit la structure ASTD. La séquence d'actions 4 montre que la sous-structure `AUT1` s'exécute pour les valeurs 2, 3 et 1.

7.3.10 Conclusion sur les tests unitaires

À la vue des mesures prises pendant l'exécution des tests unitaires, il s'avère que les temps d'exécution des filtres obtenus par la méthode d'interprétation sont plus élevés que ceux obtenus par la méthode de transformation. Ces résultats concordent avec nos attentes. La méthode par interprétation présente cependant des avantages par rapport à celle par transformation. Premièrement, la charge en mémoire du filtre est limitée. Cette charge n'augmente que pendant l'exécution des actions `Init` et

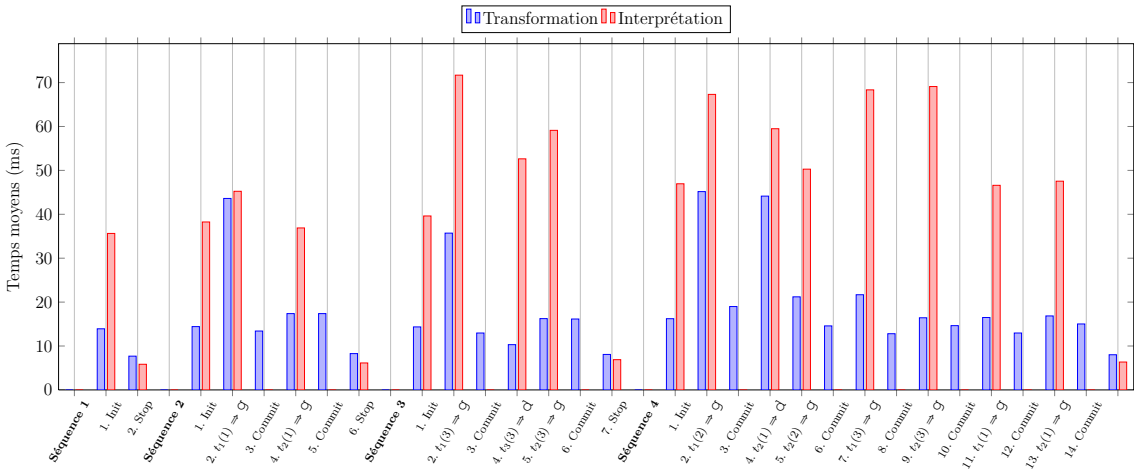


Figure 7.22 – Histogramme des temps moyens de la structure `QSYN1`

`Event` et retourne à un niveau stable à la fin de l'exécution. Alors que dans la méthode par transformation, les processus BPEL résident en mémoire et n'y sont retirés que lorsqu'ils ont fini d'être exécutés suivant la spécification ASTD à laquelle ils correspondent. Deuxièmement, la méthode par interprétation simplifie la reprise en cas de panne car l'état de la spécification ASTD interprétée est entièrement contenu dans un document XML. Cet état peut être aisément enregistré, par exemple dans une base de données, après chaque modification de façon à permettre la reprise du traitement après une panne ou même de permettre plus qu'un retour en arrière (multiples `Rollback`). Troisièmement, la méthode d'interprétation est la plus simple à implémenter et à maintenir car les processus BPEL et leurs interactions sont moins complexes. Simplement, il s'agit d'écrire un programme alors que dans le cas de la méthode par transformation il s'agit d'écrire un programme qui génère un programme.

La méthode par transformation présente deux principaux avantages par rapport à la méthode par interprétation. D'une part, elle présente de meilleurs temps d'exécution comme le montre les mesures prises dans cette section. D'autre part, elle offre l'opportunité d'implémenter des optimisations spécifiques à la spécification ASTD à transformer. Par exemple, deux structures ASTD *fermeture de Kleene* immédiatement imbriquées équivalent à une seule structure et l'implémentation de l'algorithme pourrait détecter cette situation et générer du code BPEL pour une seule structure ASTD

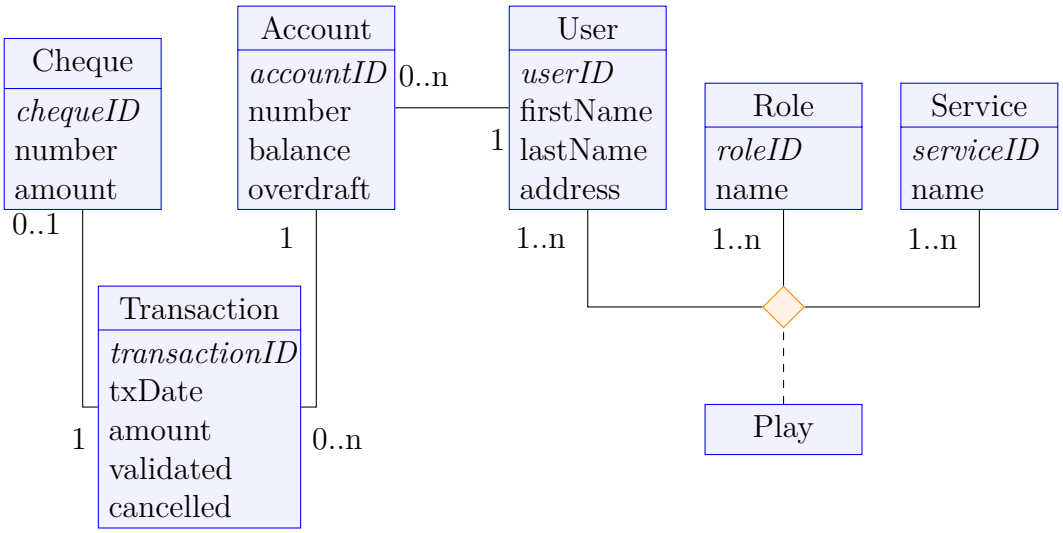


Figure 7.23 – Diagramme de classes du système d'information de la banque

fermeture de Kleene. La méthode par interprétation pourrait aussi implémenter une telle optimisation, mais au détriment des performances à l'exécution du filtre.

7.4 Tests de performance

Les tests de performance réalisés sont basés sur l'étude de cas d'un système d'information simplifié d'une banque. Ce système d'information est implémenté par des services Web. L'étude de cas comporte les entités décrites dans la figure 7.23.

7.4.1 Description de l'étude de cas

La banque emploie trois catégories de personnes : des caissiers, des conseillers et des chefs d'agence. Elle gère des comptes de banque pour des clients dont certains peuvent être aussi des employés. Cependant aucun employé ne peut faire partie de deux catégories distinctes de personnes. Par exemple, un caissier ne peut être un conseiller ou un chef d'agence. Les mouvements sur un compte donnent lieu à une transaction liée au compte. Le système d'information de la banque offre plusieurs opérations. Celles visées par la politique de contrôle d'accès testée dans cette section sont :

- `balance` qui retourne le solde d'un compte, elle prend en paramètre le numéro de compte et retourne le solde du compte ;
- `deposit` qui permet d'effectuer le dépôt d'un chèque ou d'argent liquide sur un compte, elle prend en paramètres le numéro de compte sur lequel doit être déposé le chèque ou l'argent liquide, le numéro du chèque à déposer (0 s'il s'agit d'argent liquide) et le montant à déposer sur le compte et retourne la valeur *vrai* si le dépôt s'est effectué correctement ;
- `withdraw` qui permet de faire un retrait d'argent depuis un compte, elle prend en paramètres le numéro de compte et le montant du retrait et retourne le montant retiré ;
- `validate` (respectivement `cancel`) qui permet de valider (respectivement d'annuler) un dépôt de chèque, elle prend en paramètre le numéro de chèque à valider (respectivement à annuler) et retourne la valeur *vrai* lorsque l'opération est réalisée ;
- `register` qui permet d'enregistrer un client auprès d'un caissier particulier, elle prend en paramètres l'identifiant du client à enregistrer et l'identifiant du caissier en charge du client et retourne la valeur *vrai* lorsque l'enregistrement est effectué.

Ces opérations sont fournies par trois services du système d'information et leur utilisation doit se conformer aux règles suivantes qui constituent la politique de contrôle d'accès :

1. le client peut consulter le solde de ses comptes et seulement de ceux-ci ;
2. les caissiers ont accès au solde de tous les comptes des clients, mais pas à celui de leurs propres comptes s'ils sont aussi clients de la banque (pendant qu'ils travaillent comme caissiers de la banque) ;
3. les conseillers et les chefs d'agence n'ont pas accès au solde des comptes des clients ;
4. seuls un caissier, un conseiller et un chef d'agence peuvent faire un dépôt de chèque dans un compte d'un client, aucun de ces employés ne peut faire de dépôt (chèque ou d'argent liquide) dans son propre compte s'il est aussi client

de la banque, les dépôts d'argent liquide ne peuvent être effectués que par un cassier ;

5. pour pouvoir effectuer un dépôt de chèque dans un compte de client, un caissier doit être enregistré comme caissier particulier de ce client, cette règle ne s'applique pas aux autres employés de la banque ;
6. un chef d'agence peut enregistrer un client auprès d'un caissier, cependant en tant que chef d'agence il ne peut s'enregistrer lui-même s'il est client auprès d'un caissier ;
7. un client peut s'enregistrer auprès d'un caissier, un caissier qui est aussi client ne peut pas s'enregistrer auprès de lui-même ;
8. la validation et l'annulation d'un dépôt de chèque sont toujours effectuées par un conseiller ou un chef d'agence ;
9. lorsqu'un dépôt de chèque est effectué par un caissier, la validation ou l'annulation du dépôt doit être faite par un conseiller ou un chef d'agence ;
10. lorsqu'un dépôt de chèque est effectué par un conseiller, la validation ou l'annulation du dépôt doit être faite par un chef d'agence ;
11. lorsqu'un dépôt de chèque est effectué par un chef d'agence, la validation ou l'annulation du dépôt doit être faite par un autre chef d'agence ;
12. le retrait d'argent depuis un compte est autorisé à tous les employés de la banque, cependant un employé ne peut pas effectuer lui-même de retrait depuis son propre compte.

Les rôles `Customer`, `Cashier`, `Advisor` et `Head Office` sont dégagés de cette description. Ils sont insérés dans une table `Role` de la base de données du système d'information et reliés suivant les règles prescrites aux opérations pour constituer une partie de la politique de contrôle d'accès statique du système. Les opérations sont insérés dans une table `Service` de la base de données. Par exemple, la règle 12 permet d'attribuer aux rôles `Cashier`, `Advisor` et `Head Office` le droit d'exécuter l'opération `withdraw`. Le tableau 7.4 détaille la liste complète des opérations associées aux différents rôles. Les valeurs des noms y sont montrées mais pas les identifiants pour des raisons de clarté. Les règles qui présentent un aspect dynamique

Tableau 7.4 – Listes des opérations permises pour chaque rôle

Rôle	Opération
`Customer`	`balance`, `register`
`Cashier`	`balance`, `deposit`, `withdraw`
`Advisor`	`deposit`, `withdraw`, `validate`, `cancel`
`Head Office`	`deposit`, `withdraw`, `validate`, `cancel`, `register`

sont spécifiées dans la notation ASTD. Les paramètres de contrôle d'accès choisis sont l'identifiant de l'utilisateur qui initie l'action, le rôle de cet utilisateur et l'identifiant du client qui est concerné par l'opération. La figure 7.24 (respectivement le programme 7.2) est la représentation graphique (respectivement la représentation au format XML) de la spécification de la règle 2 qui concerne les cassiers et le solde des comptes des clients.

Les opérations sont séparées en trois groupes suivant les règles et l'accès aux opérations d'un groupe est contrôlé par une politique de contrôle d'accès dynamique. Le premier groupe contient uniquement l'opération `balance` et l'accès à cette opération est contrôlé par la politique identifiée par `BankACPolicyBalance`. La politique `BankACPolicySoDObl` filtre les accès au second groupe constitué par les opérations `deposit`, `validate`, `cancel` et `register`. Le dernier groupe est constitué par l'opération `withdraw` dont l'accès est contrôlé par la politique nommée

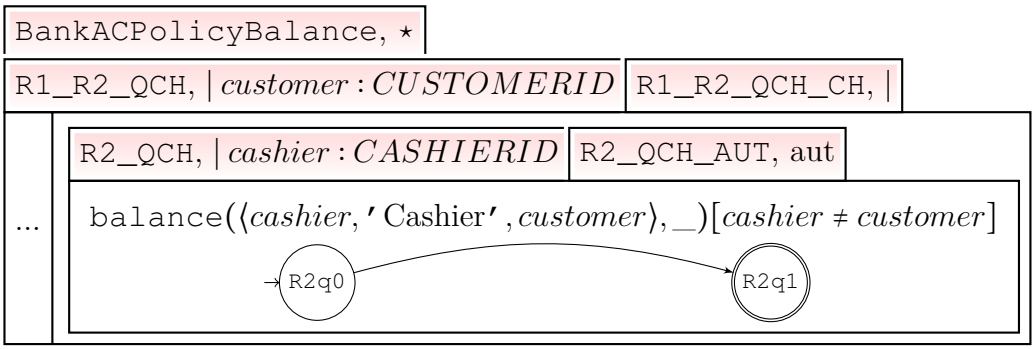


Figure 7.24 – Spécification ASTD de la règle 2

```
<!-- Rules 1, 2 -->
<ax:KleeneClosure ax:Name="BankACPolicyBalance">
 <ax:B>
  <ax:QChoice ax:Name="R1_R2_QCH" ax:X="customer" ax:T="CUSTOMERID">
   <ax:B>
    <ax:Choice ax:Name="R1_R2_QCH_CH">
     <!-- Rule 1 -->
     <ax:Left>...</ax:Left>
     <ax:Right>
      <!-- Rule 2 : A cashier can check the balance on
      all customer accounts except on his own accounts -->
      <ax:QChoice ax:Name="R2_QCH" ax:X="cashier" ax:T="CASHIERID">
       <ax:B>
        <ax:Automaton ax:Name="R2_QCH_AUT" ax:N0="R2q0">
         <ax:States>
          <ax:State ax:Name="R2q0"><ax:Elementary ax:Final="false"/></ax:State>
          <ax:State ax:Name="R2q1"><ax:Elementary ax:Final="true"/></ax:State>
         </ax:States>
         <ax:Transitions>
          <ax:Transition ax:Final="false">
           <ax:Phi><px:Predicate>
              <px:NotEqual>
               <px:Left><px:Variable>cashier</px:Variable></px:Left>
               <px:Right><px:Variable>customer</px:Variable></px:Right>
              </px:NotEqual>
           </px:Predicate></ax:Phi>
           <ax:LocalArrow ax:N1="R2q0" ax:N2="R2q1"/>
           <ax:Event ax:Name="balance">
            <ax:PV ax:X="userId" ax:V="$cashier"/>
            <ax:PV ax:X="roleId" ax:V="Cashier"/>
            <ax:PV ax:X="customerId" ax:V="$customer"/>
            <ax:PV ax:X="accountId" ax:V="_"/>
           </ax:Event>
          </ax:Transition>
         </ax:Transitions>
        </ax:Automaton>
       </ax:B>
      </ax:QChoice><!-- Rule 2 -->
     </ax:Right>
    </ax:Choice>
   </ax:B>
  </ax:QChoice>
 </ax:B>
</ax:KleeneClosure><!-- Rules 1, 2 -->
```

Programme 7.2 – Spécification ASTD de la règle 2

`BankACPolicyWithdraw`. Séparer les opérations en groupes permet d'implémenter le contrôle d'accès dans une architecture hybride présentée à la section 4.1.3. Cette séparation est orientée par les règles qui contraignent les occurrences d'opérations les unes par rapport aux autres. Les spécifications ASTD complètes de ces politiques de contrôle d'accès sont données dans les programmes C.1, C.2 et C.3 de l'annexe C.

Tableau 7.5 – Statistiques sur la base de données

Entité	Nombre	
Utilisateurs	1000	
Clients	800	
Caissiers	75	(31 clients)
Conseillers	30	(6 clients)
Chefs d'agence	10	(2 clients)
Comptes	1579	(0 à 4 par clients)
Rôles	4	
Services	6	

7.4.2 Données d'essai

Les données d'essai sont constituées essentiellement de données dans la base de données et d'un fichier projet *soapUI*. Le tableau 7.5 résume quelques informations sur les données insérées dans la base de données. Celles-ci ont été obtenues, d'une part, avec l'outil en ligne www.generatedata.com pour obtenir la liste des utilisateurs du système et, d'autre part, avec une feuille de style XSL pour attribuer des droits à ces utilisateurs. Un programme OCaml a été utilisé pour créer de façon aléatoire des comptes pour les utilisateurs du système. Ces données sont rassemblées dans un fichier de commandes SQL `INSERT` qui crée la base de données du système d'information et insère les données dans un SGBD MySQL. Le système d'information lui-même est déployé dans un serveur Apache Tomcat, avec le cadre d'applications de mise en œuvre des politiques de contrôle d'accès spécifiées avec la notation ASTD. Le déploiement du système est illustré à la figure 7.25. Trois services constituent le système d'information et les opérations sont groupées dans le service suivant les entités du système manipulées. Les classes `Interceptor` et `PEP` sont des sous-classes de l'interface `SOAPHandler` qui est un gestionnaire de messages SOAP. Les accès à la base données se font à travers une couche d'accès au données (Data Access Layer en anglais) implémentée avec le cadre d'applications Hibernate [4]. Le système comprend, pendant son exécution, trois instances distinctes du composant `PDPCore` ainsi que trois instances

4. Site Web du cadre d'applications Hibernate : www.hibernate.org

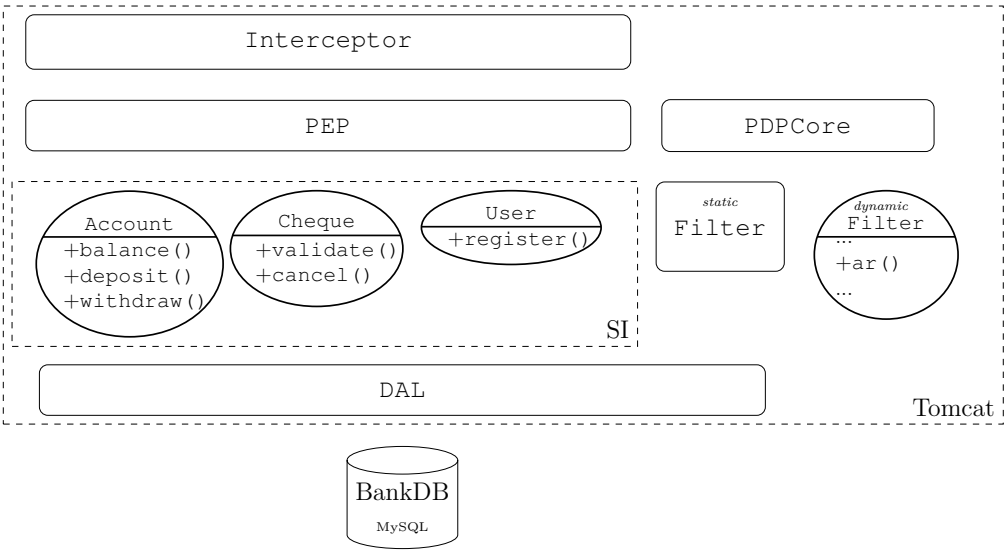


Figure 7.25 – Système sécurisé de la banque déployé

distinctes du composant `DynamicFilter`. Ces instances correspondent respectivement aux trois politiques `BankACPolicyBalance`, `BankACPolicySoDObl` et `BankACPolicyWithdraw`. Le composant `DynamicFilter` est en fait une classe proxy d'un filtre dynamique déployé dans le moteur d'exécution BPEL et permet à l'instance du composant `PDPCore` auquel il est associé de communiquer avec le filtre.

Le fichier projet *soapUI* comporte une séquence spéciale d'actions qui teste chaque opération du système d'information. Des fragments de script permettent, de façon aléatoire, de choisir à chaque pas du test réalisé la prochaine opération à exécuter. Les données de la séquence en cours d'exécution sont également déterminées de façon aléatoire. Par exemple, l'utilisateur et le rôle sont déterminés au début de l'exécution. Ensuite, suivant le rôle choisi, une opération à exécuter est choisie. Ce choix, bien qu'aléatoire, est paramétré par des pondérations attribuées aux opérations en fonction du rôle courant. Ainsi, le rôle `customer` a plus de *chance* que le rôle `advisor` d'exécuter l'action `balance`. En fonction de l'opération choisie, un autre fragment de script détermine (de façon aléatoire) des valeurs pour les paramètres de l'opération. L'opération elle-même est ensuite appelée, c'est-à-dire que la requête SOAP correspondante est envoyée au service système d'information et la réponse reçue est validée

Tableau 7.6 – Paramètres des tests de charge

Stratégie	N^{bre} de clients	Durée (s)	Délai (s)	Aléatoire	Période (s)	Variance
Simple	25	600	1	0,5	—	—
Variance	25	600	—	—	120	0,3

par des assertions. Ce mécanisme reprend à partir du choix de l'opération et se répète un nombre de fois prédéterminé à l'avance de façon aléatoire et pondérée suivant le rôle courant (par exemple, le rôle `cashier` exécutera toujours généralement plus d'opérations que le rôle `head office`).

Les tests exécutés avec ces données utilisent deux stratégies de test de charge. La première, nommée simple, détermine le comportement de base du système sécurisé avec des paramètres de nombre de clients virtuels fixés à l'avance. C'est celle qui a déjà été utilisée à la section 7.3 pour obtenir des histogrammes de temps moyens. La seconde, dite de variance, maintient un nombre de clients virtuels variable en fonction du temps. La fonction du nombre de clients est une fonction périodique en *dents de scie*. Les paramètres pour ces deux stratégies sont dans le tableau 7.6. Suivant ces paramètres, le nombre de clients virtuels pour la stratégie de variance augmente puis diminue de façon périodique dans l'intervalle $[25*(1-0,3); 25*(1+0,3)]$. La figure 7.26 montre l'évolution du nombre de clients virtuels actifs en fonction du temps pour la

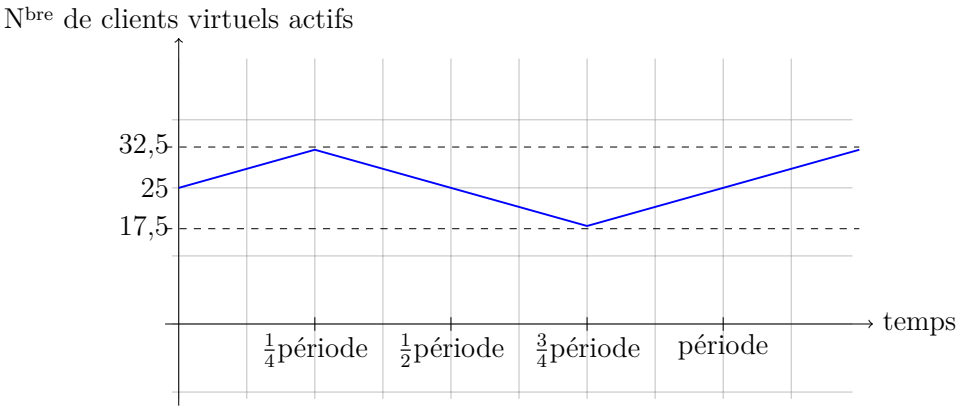


Figure 7.26 – Nombre de clients virtuels actifs en fonction du temps

stratégie de variance suivant les paramètres du tableau 7.6. La charge du système d'information est ainsi variable au cours du temps de test.

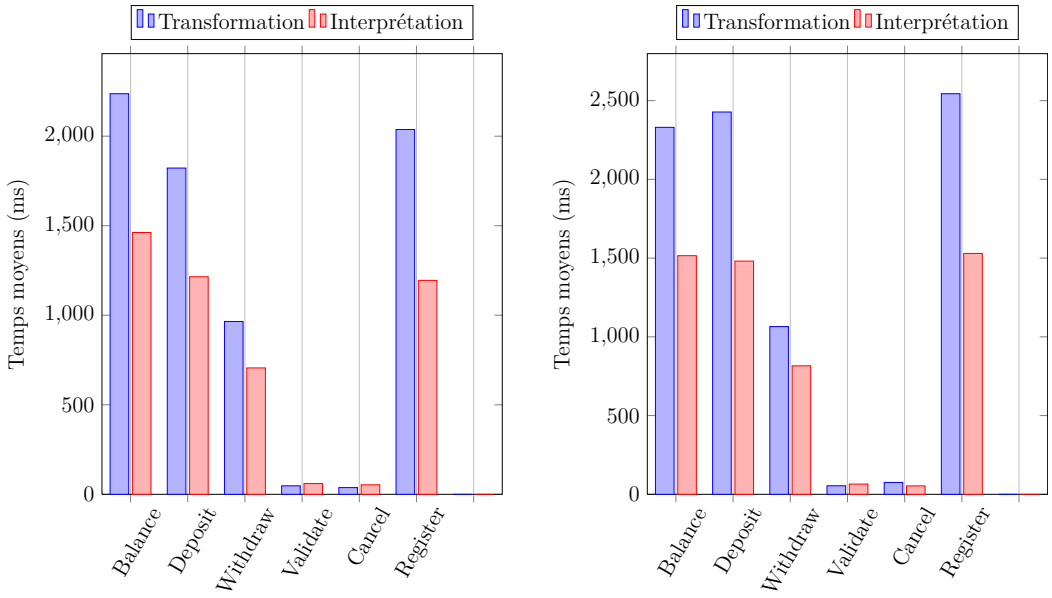
7.4.3 Résultats d'expériences

Les spécifications de politiques de contrôle d'accès ont subi chacune des tests unitaires (sans stratégie de charge particulière) pour vérifier qu'elles implémentent bien les règles énoncées à la section 7.4.1. Ces tests unitaires sont constitués, pour chaque politique de contrôle d'accès, d'une séquence d'actions qui ciblent spécifiquement les contraintes des règles. Chaque action est une requête SOAP qui teste directement le filtre dynamique correspondant, sans l'intervention du système d'information. Ainsi, pour la méthode de transformation et celle d'interprétation, les trois politiques de contrôle d'accès `BankACPolicyBalance`, `BankACPolicySoDObl` et `BankACPolicyWithdraw` ont passé les tests unitaires.

Les tests de charge ont été réalisés après que les tests unitaires des filtres ont été satisfaisants. Les résultats des expériences de charge ont été relevés pour chaque méthode de contrôle d'accès et pour les deux stratégies sélectionnées. Les temps moyens d'exécution sont présentés dans la figure 7.27. Dans les deux stratégies, les temps moyens d'exécution des opérations, sous la forme de services, par le système d'information sécurisé sont plus élevés lorsque la méthode de transformation est utilisée, sauf dans le cas des opérations `Validate` et `Cancel`.

7.4.4 Conclusion sur les tests de performance

Les résultats des tests montrent que les temps d'exécution sont cohérents pour les deux stratégies de test évaluées. L'expérience montre aussi que la méthode d'interprétation donne de meilleurs temps d'exécution que celle de transformation. Cette situation est attribuable à la quantité importante de ressources utilisées par le moteur d'exécution pour la méthode de transformation. En effet, dans le cas de la méthode d'interprétation, le nombre d'instances de processus BPEL maintenues en mémoire est limité par la profondeur de la spécification ASTD déployée dans le filtre, c'est-à-dire le nombre de structures ASTD traversées depuis la structure principale jusqu'à



(a) Stratégie simple

(b) Stratégie de variance

Figure 7.27 – Résultats des tests de charge

la structure ASTD *automate* la plus imbriquée. Ces instances sont retirées de la mémoire, à l'exception des instances du processus `Filter` de la figure 6.2 (une pour chacune des trois politiques `BankACPolicyBalance`, `BankACPolicySoDObl` et `BankACPolicyWithdraw` déployées), après chaque requête d'autorisation. Dans le cas de la méthode de transformation, les instances de processus BPEL résident en mémoire une fois qu'elles sont créées et ce pour toute la durée d'exécution du filtre. Les exceptions avec cette méthode sont les instances de processus qui sont arrêtées car l'automate qu'elles implémentent est dans un état qui ne possède pas de transition sortante. Plus encore, dans le cas de la structure ASTD *synchronisation quantifiée*, le filtre utilise une instance d'un processus BPEL pour chaque valeur de l'ensemble de quantification. Pour la politique de contrôle d'accès `BankACPolicySoDObl`, le filtre crée au moins 800 (clients) + 2000 (chèques) instances de processus BPEL.

7.5 Conclusion générale

Les tests effectués ont montré que les deux méthodes d'implémentation du filtre de contrôle d'accès dynamique sont viables dans un système d'information prototype. Ils ont aussi exhibés les différences de performance entre la méthode de transformation et la méthode d'interprétation. Bien que cette étude soit limitée aux temps d'exécution, elle pourrait être étendue à la taille des messages échangés si celle-ci est critique pour l'application cible.

Conclusion

Le sujet traité dans cette thèse se situe en aval de deux vastes projets de recherche (EB^{3}SEC et SELKIS) qui s'articulent autour du contrôle d'accès dans les systèmes d'information. Dans ces deux projets, l'approche retenue pour la construction de systèmes d'information, accessibles depuis le Web, est en accord avec celle du MDA (pour Model Driven Architecture en anglais) qui est une norme de l'OMG. Cette norme recommande d'élaborer plusieurs modèles pour un même système, du plus abstrait au plus concret. Ceci s'effectue en trois étapes successives qui représentent chacune un niveau de modélisation. Dans la première étape, un modèle du système d'information, nommé CIM (pour Computation Independant Model en anglais), décrit les différents aspects du système, en particulier les politiques de contrôle d'accès, dans un langage naturel. Dans la deuxième étape, un modèle, nommé PIM (pour Platform Independant Model en anglais), définit le système d'une manière plus précise à l'aide de langages formels ou semi-formels dans un niveau d'abstraction élevé, c'est-à-dire en omettant les détails d'implémentation. Enfin dans la troisième étape, un modèle, nommé PSM (pour Platform Specific Model en anglais), est un raffinement du modèle PIM qui inclut les détails d'implémentation dans un environnement SOA. Ainsi, en considérant uniquement le contrôle d'accès aux ressources d'un système, le niveau PSM contient tous les éléments nécessaires d'un gestionnaire d'exécution de politiques de contrôle d'accès définies dans une notation formelle à partir de règles informelles imposées par une ou plusieurs organisations. C'est donc par rapport à ce dernier niveau que s'inscrivent les travaux présentés dans cette thèse, plus particulièrement la transformation et l'interprétation de politiques de contrôle d'accès modélisées dans le niveau PIM et exprimées dans la notation ASTD afin qu'elles puissent être mise en œuvre de manière automatique dans une architecture orientée services.

Rappel des principales contributions

Dans cette thèse, nous avons notamment proposé deux implémentations d'un filtre de contrôle d'accès dynamique basées sur deux méthodes fréquemment rencontrées dans la théorie des langages de programmation : la compilation et l'interprétation. D'une part, nous avons élaboré un algorithme de transformation de spécifications ASTD en processus BPEL prêts à être exécutés comme services sur le Web. D'autre part, nous avons conçu un interpréteur en BPEL qui permet de faire évoluer les états des structures ASTD parallèlement à l'exécution d'un système d'information. Dans les deux cas, les spécifications ASTD encodent des politiques de contrôle d'accès aux ressources d'un système d'information. En amont, dans le contexte du traitement de programmes (spécifications formelles de haut niveau dans le cadre de cette thèse), la méthode associée à chacune de ces deux implémentations est complètement transparente à l'utilisateur. En aval, chaque implémentation offre un environnement d'exécution propre à la méthode qui lui est associée. Sur le plan méthodologique, notre solution repose sur l'utilisation de méthodes formelles avec comme conséquence le traitement automatique des politiques de contrôle d'accès, c'est-à-dire sans intervention humaine pour leur implémentation une fois exprimée dans la notation ASTD. Notre solution s'appuie également sur un modèle générique d'un gestionnaire d'exécution de politiques de contrôle d'accès. Enfin, nous avons validé nos deux implémentations à l'aide d'un banc d'essai tant pour nous assurer de leur bon fonctionnement que pour déterminer leur performance individuelle et celle de l'une par rapport à l'autre.

Les travaux de recherche présentés dans cette thèse se démarquent par rapport aux points suivants. Premièrement, à notre connaissance, c'est la première fois que des politiques de contrôles d'accès écrites dans un langage formel basé sur une algèbre de processus sont traduites en des processus BPEL ou interprétées par des processus BPEL. Deuxièmement, les deux implémentations d'un filtre de contrôle d'accès démontrent que l'approche conceptuelle proposée dans les projets EB^{3}SEC et SELKIS pour le contrôle d'accès aux ressources d'un système d'information est réalisable dans une architecture SOA avec des temps de traitement acceptables. Enfin, dans une perspective plus générale, nos travaux posent des jalons pour une nouvelle approche de transformation et d'interprétation de spécifications formelles lorsque ces

deux méthodes pour un langage donné comme BPEL sont offertes comme des services Web.

Évaluation de la solution proposée et travaux futurs

Bien que la solution présentée dans cette thèse soit complète par rapport aux objectifs poursuivis dans les projets EB³SEC et SELKIS, elle soulève de nombreuses interrogations, chacune pouvant faire l'objet soit d'une amélioration ou d'une extension, soit d'une étude sur un sujet connexe.

Parmi les améliorations, mentionnons d'abord le point relatif à la justesse de l'algorithme de transformation. Sa preuve d'exactitude est omise dans cette thèse, pour les deux raisons suivantes. Premièrement, une telle preuve d'exactitude ne permet pas d'assurer un fonctionnement correct du filtre de contrôle d'accès dynamique considérant tous les détails d'implémentation des processus BPEL, en particulier ceux concernant l'échange des messages entre les processus BPEL. Deuxièmement, en privilégiant une approche plus traditionnelle pour valider l'algorithme de transformation, c'est-à-dire une approche qui englobe des tests unitaires, des tests d'intégration et des tests systèmes, il nous a été possible de produire un ensemble de statistiques afin d'évaluer la performance de cette solution dans un environnement orienté services. Bien qu'omise, cette preuve consisterait en une preuve par récurrence sur la hiérarchie des structures ASTD imbriquées. La preuve d'exactitude de l'algorithme d'interprétation de spécifications ASTD a été également omise, bien qu'elle serait plus aisée à effectuer que celle de l'algorithme de transformation car, comme il a été mentionné à la fin du chapitre 6, ce dernier algorithme est calqué sur la sémantique opérationnelle du langage ASTD, aussi bien pour les traitements que pour les structures des données. Néanmoins, la réalisation de ces preuves constituerait un exercice intéressant qui permettrait de renforcer sur le plan scientifique notre solution.

Une autre amélioration consiste à lever les restrictions imposées aux spécifications ASTD qui sont traduites en processus BPEL. Parmi ces restrictions, il y a celles qui sont inhérentes aux limites du langage BPEL et pour lesquelles des solutions potentielles sont plus difficiles à trouver à moins d'implémenter le filtre de contrôle

d'accès dynamique dans un langage plus puissant que BPEL. Bien que les restrictions imposées aux spécifications ASTD n'empêchent pas de spécifier des politiques de contrôle d'accès les plus courantes (celles représentées par les quatre patrons introduits à la section 2.8), il y en a une qui ne permet pas l'utilisation d'appel de fonctions dans les gardes. Ceci rend l'écriture des gardes moins expressive. Réduire le plus possible ces restrictions constituerait un progrès intéressant, bien que ceci ne remettrait pas en cause notre approche globale de transformation de spécifications ASTD afin de les rendre opérationnelles comme services Web. Nonobstant l'utilisation de BPEL comme langage d'implémentation, l'approche par transformation défendue dans cette thèse reste pertinente bien que l'expérience montre qu'elle est moins performante que celle par interprétation car un meilleur dimensionnement du conteneur d'applications pourrait aboutir à de meilleurs résultats.

Suite aux travaux réalisés dans cette thèse, nous avons identifié au moins trois sujets qui pourraient faire l'objet d'études connexes. D'un point de vue technique, il serait pertinent de reproduire l'évaluation de la performance des implémentations du filtre de contrôle d'accès pour différents environnements aussi bien matériels que logiciels. En effet, l'évaluation a été faite pour un moteur d'exécution BPEL particulier sur un ordinateur bas de gamme par rapport à des serveurs beaucoup plus puissants utilisés dans les entreprises bancaires. De plus, nos expériences ont été réalisées sur un réseau local. Explorer d'autres technologies permettrait de déterminer dans quelles mesures notre approche est viable pour des systèmes de grande envergure. D'un point de vue théorique, l'ajout de fonctions administratives au PEM, comme la mise à jour de politiques de contrôle d'accès dynamique pendant l'exécution du système d'information et du PEM, pose de sérieux problèmes, en particulier celui de maintenir le système dans un état cohérent pendant et après la mise à jour. Enfin, les implémentations que nous avons développées l'ont été pour une architecture hybride, sous l'hypothèse qu'une telle architecture est la plus appropriée pour mettre en œuvre plusieurs politiques de contrôle d'accès dynamique indépendantes. Envisager une autre architecture parmi celles présentées à la section 4.1.3 constitue aussi une autre piste à explorer.

Annexe A

Métamodèle XSD de la notation ASTD

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3   targetNamespace="http://gril.udes.ca/astd/schema/ASTD"
4   xmlns="http://gril.udes.ca/astd/schema/ASTD"
5   elementFormDefault="qualified"
6   attributeFormDefault="qualified"
7   xmlns:px="http://gril.udes.ca/astd/schema/Predicate"
8   >
9
10   <xsd:import schemaLocation="Predicate.xsd"
      namespace="http://gril.udes.ca/astd/schema/Predicate"/>
11
12
13  <xsd:simpleType name="tName">
14   <xsd:restriction base="xsd:string"/>
15  </xsd:simpleType>
16
17  <xsd:attribute name="Name" type="tName"/>
18
19  <!-- Possible values for a parameter's type -->
20  <xsd:simpleType name="tParameterType">
21   <xsd:restriction base="xsd:string">
22    <xsd:enumeration value="string" />
23    <xsd:enumeration value="integer" />
24    <xsd:enumeration value="float" />
25   </xsd:restriction>
26  </xsd:simpleType>
27
```

```
28  <xsd:simpleType name="tLabels">
29   <xsd:list itemType="tName"/>
30  </xsd:simpleType>
31
32
33  <xsd:element name="Delta" type="tLabels"/>
34
35
36  <xsd:simpleType name="tPredicate">
37   <xsd:restriction base="xsd:string" />
38  </xsd:simpleType>
39
40
41  <xsd:simpleType name="tV">
42   <xsd:union memberTypes="xsd:string xsd:integer xsd:float" />
43  </xsd:simpleType>
44
45
46
47  <!-- Parameter's name -->
48  <xsd:attribute name="X" type="tName"/>
49
50  <!-- Parameter's value / a value -->
51  <xsd:attribute name="V" type="tV"/>
52
53
54  <!-- Parameter's value in ASTD Call and incomming event -->
55  <xsd:element name="PV" type="tPV"/>
56
57  <xsd:complexType name="tPV">
58   <xsd:attribute ref="X" use="required"/>
59   <xsd:attribute ref="V" use="required"/>
60  </xsd:complexType>
61
62
63  <!-- Incomming event definition -->
64  <xsd:element name="IncommingEvent" type="tIncommingEvent"/>
65
66  <xsd:complexType name="tIncommingEvent">
67   <xsd:sequence>
68    <xsd:element ref="PV" minOccurs="0" maxOccurs="unbounded"/>
69   </xsd:sequence>
70   <xsd:attribute ref="Name" use="required"/>
71  </xsd:complexType>
72  <!-- Incomming event definition - end -->
73
74
75  <!-- Type of parameters in an event's signature and ASTD parameters -->
76  <xsd:element name="P" type="tP"/>
```

```
77
78  <xsd:complexType name="tP">
79   <xsd:attribute ref="X" use="required"/>
80   <xsd:attribute name="T" type="tParameterType" use="required"/>
81  </xsd:complexType>
82
83
84  <!-- Signature definition -->
85  <xsd:element name="Signature" type="tSignature"/>
86
87  <xsd:complexType name="tSignature">
88   <xsd:sequence>
89    <xsd:element name="Event" minOccurs="0" maxOccurs="unbounded">
90     <xsd:complexType>
91      <xsd:sequence>
92       <xsd:element ref="P" minOccurs="0" maxOccurs="unbounded"/>
93      </xsd:sequence>
94      <xsd:attribute ref="Name" use="required"/>
95     </xsd:complexType>
96    </xsd:element>
97   </xsd:sequence>
98  </xsd:complexType>
99  <!-- Signature definition - end -->
100
101
102 <!-- An ASTD specification -->
103 <xsd:element name="Specification">
104  <xsd:complexType>
105   <xsd:sequence>
106    <xsd:group ref="gASTD" maxOccurs="unbounded"/>
107   </xsd:sequence>
108  </xsd:complexType>
109 </xsd:element>
110
111 <xsd:complexType name="tASTD">
112  <xsd:group ref="gASTD"/>
113 </xsd:complexType>
114
115 <xsd:group name="gASTD">
116  <xsd:choice>
117   <xsd:element ref="Automaton"/>
118   <xsd:element ref="Sequence"/>
119   <xsd:element ref="Choice"/>
120   <xsd:element ref="KleeneClosure"/>
121   <xsd:element ref="Synchronization"/>
122   <xsd:element ref="Guard"/>
123   <xsd:element ref="Call"/>
124   <xsd:element ref="QChoice"/>
125   <xsd:element ref="QSynchronization"/>
```

```
126   </xsd:choice>
127  </xsd:group>
128
129
130  <!-- An ASTD State -->
131  <xsd:element name="State" type="tState"/>
132
133  <xsd:complexType name="tState">
134   <xsd:group ref="gState"/>
135  </xsd:complexType>
136
137  <xsd:group name="gState">
138   <xsd:choice>
139    <xsd:element ref="EmptyState"/>
140    <xsd:element ref="AutomatonState"/>
141    <xsd:element ref="SequenceState"/>
142    <xsd:element ref="ChoiceState"/>
143    <xsd:element ref="KleeneClosureState"/>
144    <xsd:element ref="SynchronizationState"/>
145    <xsd:element ref="GuardState"/>
146    <xsd:element ref="CallState"/>
147    <xsd:element ref="QChoiceState"/>
148    <xsd:element ref="QSynchronizationState"/>
149   </xsd:choice>
150  </xsd:group>
151
152
153  <!-- Base type for all ASTD structures-->
154  <xsd:complexType name="tASTDBase" abstract="true">
155   <xsd:sequence>
156    <xsd:element name="Ps" minOccurs="0"><!-- Parameters of the ASTD -->
157     <xsd:complexType>
158      <xsd:sequence>
159       <xsd:element ref="P" minOccurs="0" maxOccurs="unbounded"/>
160      </xsd:sequence>
161     </xsd:complexType>
162    </xsd:element>
163   </xsd:sequence>
164   <xsd:attribute ref="Name" use="required"/>
165  </xsd:complexType>
166
167  <!--
168   Base type for all ASTD structures that have a unique ASTD
169   substructure
170  -->
171  <xsd:complexType name="tUnaryOperator" abstract="true">
172   <xsd:complexContent>
173    <xsd:extension base="tASTDBase">
174      <xsd:sequence>
```

```
175         <xsd:element name="B" type="tASTD"/>
176       </xsd:sequence>
177     </xsd:extension>
178   </xsd:complexContent>
179 </xsd:complexType>
180
181
182 <!--
183  Base type for all ASTD structures that have a pair ASTD
184  substructures
185 -->
186 <xsd:complexType name="tBinaryOperator" abstract="true">
187   <xsd:complexContent>
188     <xsd:extension base="tASTDBase">
189       <xsd:sequence>
190         <xsd:element name="Left" type="tASTD"/>
191         <xsd:element name="Right" type="tASTD"/>
192       </xsd:sequence>
193     </xsd:extension>
194   </xsd:complexContent>
195 </xsd:complexType>
196
197
198 <!--
199  Base type for all ASTD states that have a unique ASTD
200  substate
201 -->
202 <xsd:complexType name="tUnaryState" abstract="true">
203   <xsd:sequence>
204     <xsd:element name="S" type="tState"/>
205   </xsd:sequence>
206 </xsd:complexType>
207
208
209 <!-- Event definition in ASTD specification -->
210 <xsd:element name="Event" type="tEvent"/>
211
212 <xsd:complexType name="tEvent">
213   <xsd:sequence>
214     <xsd:element ref="PV" minOccurs="0" maxOccurs="unbounded"/>
215   </xsd:sequence>
216   <xsd:attribute ref="Name" use="required"/>
217 </xsd:complexType>
218 <!-- Event definition in ASTD specification - end -->
219
220
221 <!-- Transition definition in Automaton -->
222   <xsd:element name="Transition">
223     <xsd:complexType>
```

```
224         <xsd:sequence>
225           <xsd:group ref="gArrow" />
226           <xsd:element ref="Event"/>
227         </xsd:sequence>
228         <xsd:attribute name="Final" type="xsd:boolean" default="false"/>
229         <xsd:attribute name="Phi" type="tPredicate" default=""/>
230       </xsd:complexType>
231     </xsd:element>
232   <!-- Transition definition in Automaton - end -->
233
234    <xsd:element name="Automaton" type="tAutomaton"/>
235
236    <xsd:complexType name="tAutomaton">
237     <xsd:complexContent>
238      <xsd:extension base="tASTDBase">
239       <xsd:sequence>
240
241        <xsd:element name="States">
242         <xsd:complexType>
243          <xsd:sequence>
244           <xsd:element name="State" minOccurs="0" maxOccurs="unbounded">
245            <xsd:complexType>
246             <xsd:choice>
247              <xsd:element name="Elementary">
248               <xsd:complexType>
249                <xsd:attribute name="Final" type="xsd:boolean" default="false"/>
250               </xsd:complexType>
251              </xsd:element>
252              <xsd:group ref="gASTD"/>
253             </xsd:choice>
254             <xsd:attribute ref="Name" use="required"/>
255            </xsd:complexType>
256           </xsd:element>
257          </xsd:sequence>
258         </xsd:complexType>
259        </xsd:element>
260
261        <xsd:element name="Transitions">
262         <xsd:complexType>
263          <xsd:sequence>
264           <xsd:element ref="Transition" maxOccurs="unbounded"/>
265          </xsd:sequence>
266         </xsd:complexType>
267        </xsd:element>
268
269       </xsd:sequence>
270       <xsd:attribute name="N0" use="required"/>
271      </xsd:extension>
272     </xsd:complexContent>
```

```
</xsd:complexType>

<!-- Base type of arrows in Automaton transitions -->
<!-- See 2.5.3 Empty Content of http://www.w3.org/TR/xmlschema-0/ -->
<xsd:complexType name="tArrowBase" abstract="true">
 <xsd:attribute name="N1" type="tName" use="required"/>
 <xsd:attribute name="N2" type="tName" use="required"/>
</xsd:complexType>

<xsd:group name="gArrow">
 <xsd:choice>
  <xsd:element name="LocalArrow">
   <xsd:complexType>
    <xsd:complexContent>
     <xsd:extension base="tArrowBase"/>
    </xsd:complexContent>
   </xsd:complexType>
  </xsd:element>

  <xsd:element name="ToSubArrow">
   <xsd:complexType>
    <xsd:complexContent>
     <xsd:extension base="tArrowBase">
      <xsd:attribute name="SubN2" type="tName" use="required"/>
     </xsd:extension>
    </xsd:complexContent>
   </xsd:complexType>
  </xsd:element>

  <xsd:element name="FromSubArrow">
   <xsd:complexType>
    <xsd:complexContent>
     <xsd:extension base="tArrowBase">
      <xsd:attribute name="SubN1" type="tName" use="required"/>
     </xsd:extension>
    </xsd:complexContent>
   </xsd:complexType>
  </xsd:element>
 </xsd:choice>
</xsd:group>

<xsd:element name="AutomatonState" type="tAutomatonState"/>

<xsd:complexType name="tAutomatonState">
 <xsd:sequence>

  <xsd:element name="Histories">
   <xsd:complexType>
    <xsd:sequence>
```

```
322       <xsd:element name="History" minOccurs="0" maxOccurs="unbounded" >
323        <xsd:complexType>
324         <xsd:complexContent>
325          <xsd:extension base="tState">
326           <xsd:attribute ref="Name" use="required"/>
327          </xsd:extension>
328         </xsd:complexContent>
329        </xsd:complexType>
330       </xsd:element>
331      </xsd:sequence>
332     </xsd:complexType>
333    </xsd:element>
334
335    <xsd:element name="S">
336     <xsd:complexType>
337      <xsd:choice>
338       <xsd:element name="ElementaryState" />
339       <xsd:group ref="gState"/>
340      </xsd:choice>
341     </xsd:complexType>
342    </xsd:element>
343
344   </xsd:sequence>
345   <xsd:attribute name="N" type="tName" use="required" />
346  </xsd:complexType>
347
348
349  <xsd:simpleType name="tSequenceSide">
350   <xsd:restriction base="xsd:string">
351    <xsd:enumeration value="first" />
352    <xsd:enumeration value="second" />
353   </xsd:restriction>
354  </xsd:simpleType>
355
356  <xsd:element name="Sequence" type="tSequence"/>
357
358  <xsd:complexType name="tSequence">
359   <xsd:complexContent>
360    <xsd:extension base="tASTDBase">
361     <xsd:sequence>
362      <xsd:element name="First" type="tASTD"/>
363      <xsd:element name="Second" type="tASTD"/>
364     </xsd:sequence>
365    </xsd:extension>
366   </xsd:complexContent>
367  </xsd:complexType>
368
369  <xsd:element name="SequenceState" type="tSequenceState"/>
370
```

```
<xsd:complexType name="tSequenceState">
 <xsd:complexContent>
  <xsd:extension base="tUnaryState">
   <xsd:attribute name="Side" type="tSequenceSide" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>


<xsd:simpleType name="tChoiceSide">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="left" />
  <xsd:enumeration value="right" />
  <xsd:enumeration value="bottom" />
 </xsd:restriction>
</xsd:simpleType>

<xsd:element name="Choice" type="tChoice"/>

<xsd:complexType name="tChoice">
 <xsd:complexContent>
  <xsd:extension base="tBinaryOperator"/>
 </xsd:complexContent>
</xsd:complexType>

<xsd:element name="ChoiceState" type="tChoiceState"/>

<xsd:complexType name="tChoiceState">
 <xsd:complexContent>
  <xsd:extension base="tUnaryState">
   <xsd:attribute name="Side" type="tChoiceSide" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>


<xsd:element name="KleeneClosure" type="tKleeneClosure"/>

<xsd:complexType name="tKleeneClosure">
 <xsd:complexContent>
  <xsd:extension base="tUnaryOperator"/>
 </xsd:complexContent>
</xsd:complexType>

<xsd:element name="KleeneClosureState" type="tKleeneClosureState"/>

<xsd:complexType name="tKleeneClosureState">
 <xsd:complexContent>
  <xsd:extension base="tUnaryState">
```

```
420      <xsd:attribute name="Started" type="xsd:boolean" use="required"/>
421     </xsd:extension>
422    </xsd:complexContent>
423   </xsd:complexType>
424
425
426   <xsd:element name="Synchronization" type="tSynchronization"/>
427
428   <xsd:complexType name="tSynchronization">
429    <xsd:complexContent>
430     <xsd:extension base="tBinaryOperator">
431      <xsd:sequence>
432       <xsd:element ref="Delta" />
433      </xsd:sequence>
434     </xsd:extension>
435    </xsd:complexContent>
436   </xsd:complexType>
437
438   <xsd:element name="SynchronizationState" type="tSynchronizationState"/>
439
440   <xsd:complexType name="tSynchronizationState">
441    <xsd:sequence>
442     <xsd:element name="LeftState" type="tState"/>
443     <xsd:element name="RightState" type="tState"/>
444    </xsd:sequence>
445   </xsd:complexType>
446
447
448   <xsd:complexType name="tT">
449    <xsd:sequence>
450     <xsd:element name="V" type="tV" maxOccurs="unbounded"/>
451    </xsd:sequence>
452    <xsd:attribute name="BaseType" type="tName" use="required"/>
453   </xsd:complexType>
454
455
456   <!-- Base type of quantified unary operator -->
457   <xsd:complexType name="tQuantifiedOperator" abstract="true">
458    <xsd:complexContent>
459     <xsd:extension base="tUnaryOperator">
460      <xsd:sequence>
461       <xsd:element name="T" type="tT" />
462      </xsd:sequence>
463      <xsd:attribute ref="X" use="required"/>
         <!-- Name of the quantification variable -->
464     </xsd:extension>
465    </xsd:complexContent>
466   </xsd:complexType>
467
```

```
468
469  <xsd:element name="QChoice" type="tQChoice"/>
470
471  <xsd:complexType name="tQChoice">
472   <xsd:complexContent>
473    <xsd:extension base="tQuantifiedOperator"/>
474   </xsd:complexContent>
475  </xsd:complexType>
476
477  <xsd:element name="QChoiceState" type="tQChoiceState"/>
478
479  <xsd:complexType name="tQChoiceState">
480   <xsd:complexContent>
481    <xsd:extension base="tUnaryState">
482     <xsd:attribute ref="V" use="required"/>
       <!-- Value of the quantification variable -->
483    </xsd:extension>
484   </xsd:complexContent>
485  </xsd:complexType>
486
487
488  <xsd:element name="QSynchronization" type="tQSynchronization"/>
489
490  <xsd:complexType name="tQSynchronization">
491   <xsd:complexContent>
492    <xsd:extension base="tQuantifiedOperator">
493     <xsd:sequence>
494      <xsd:element ref="Delta" />
495     </xsd:sequence>
496    </xsd:extension>
497   </xsd:complexContent>
498  </xsd:complexType>
499
500  <xsd:element name="QSynchronizationState" type="tQSynchronizationState"/>
501
502  <xsd:element name="FV">
503   <xsd:complexType>
504    <xsd:sequence>
505     <xsd:group ref="gState"/>
506    </xsd:sequence>
507    <xsd:attribute ref="V" use="required"/>
508   </xsd:complexType>
509  </xsd:element>
510
511  <xsd:element name="F">
512   <xsd:complexType>
513    <xsd:sequence>
514     <xsd:element ref="FV" minOccurs="0" maxOccurs="unbounded"/>
515    </xsd:sequence>
```

```
516    </xsd:complexType>
517   </xsd:element>
518
519   <xsd:complexType name="tQSynchronizationState">
520    <xsd:complexContent>
521     <xsd:extension base="tUnaryState">
        <!-- For optimization purpose, so that we can have S -->
522      <xsd:sequence>
523       <xsd:element ref="F"/>
524      </xsd:sequence>
525     </xsd:extension>
526    </xsd:complexContent>
527   </xsd:complexType>
528
529
530   <xsd:element name="Guard" type="tGuard"/>
531
532   <xsd:complexType name="tGuard">
533    <xsd:complexContent>
534     <xsd:extension base="tUnaryOperator">
535      <xsd:sequence>
536        <xsd:element ref="px:Predicate"/>
537      </xsd:sequence>
538      <xsd:attribute name="G" type="tPredicate" use="required"/>
539     </xsd:extension>
540    </xsd:complexContent>
541   </xsd:complexType>
542
543   <xsd:element name="GuardState" type="tGuardState"/>
544
545   <xsd:complexType name="tGuardState">
546    <xsd:complexContent>
547     <xsd:extension base="tUnaryState">
548      <xsd:attribute name="Started" type="xsd:boolean" use="required"/>
549     </xsd:extension>
550    </xsd:complexContent>
551   </xsd:complexType>
552
553
554   <xsd:element name="Call" type="tCall"/>
555
556   <xsd:complexType name="tCall">
557    <xsd:complexContent>
558     <xsd:extension base="tASTDBase">
559      <xsd:sequence>
560       <xsd:element name="PVs">
561        <xsd:complexType>
562         <xsd:sequence>
563          <xsd:element ref="PV" minOccurs="0" maxOccurs="unbounded"/>
```

```
564          </xsd:sequence>
565         </xsd:complexType>
566        </xsd:element>
567       </xsd:sequence>
568       <xsd:attribute name="B" type="tName" use="required"/>
569     </xsd:extension>
570    </xsd:complexContent>
571   </xsd:complexType>
572
573   <xsd:element name="CallState" type="tCallState"/>
574
575   <xsd:complexType name="tCallState">
576    <xsd:complexContent>
577     <xsd:extension base="tUnaryState"/>
578    </xsd:complexContent>
579   </xsd:complexType>
580
581    <xsd:element name="EmptyState"/>
582
583  </xsd:schema>
```

Programme A.1 – Document XSD du métamodèle de la notation ASTD

Annexe B

Extrait du fichier WSDL

```
1  <binding name="RootASTDBinding" type="RootASTDPortType">
2   <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
3   <operation name="Init">
4    <soap:operation
       soapAction="http://gril.udes.ca/astd/wsdl/RootASTDInit"
       style="document"/>
5    <input name="inputInit">
6     <soap:body use="literal"
        namespace="http://gril.udes.ca/astd/wsdl"/>
7    </input>
8    <output name="outputInit">
9     <soap:body use="literal"
        namespace="http://gril.udes.ca/astd/wsdl"/>
10   </output>
11  </operation>
12  <operation name="AR">
13   <soap:operation
       soapAction="http://gril.udes.ca/astd/wsdl/RootASTDAR"
       style="document"/>
14   <input name="inputAR">
15    <soap:body use="literal"
        namespace="http://gril.udes.ca/astd/wsdl"/>
16   </input>
17   <output name="outputAR">
18    <soap:body use="literal"
        namespace="http://gril.udes.ca/astd/wsdl"/>
19   </output>
20  </operation>
21  <operation name="Commit">
22   <soap:operation
       soapAction="http://gril.udes.ca/astd/wsdl/RootASTDCommit"
```

```
        style="document"/>
23      <input name="inputCommit">
24       <soap:body use="literal"
          namespace="http://gril.udes.ca/astd/wsdl"/>
25      </input>
26      <output name="outputCommit">
27       <soap:body use="literal"
          namespace="http://gril.udes.ca/astd/wsdl"/>
28      </output>
29     </operation>
30     <operation name="Stop">
31      <soap:operation
         soapAction="http://gril.udes.ca/astd/wsdl/RootASTDStop"
         style="document"/>
32      <input name="inputStop">
33       <soap:body use="literal"
          namespace="http://gril.udes.ca/astd/wsdl"/>
34      </input>
35     </operation>
36    </binding>
37
38
39    <binding name="ASTDBinding" type="ASTDPortType">
40     <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
41     <operation name="AR">
42      <soap:operation
         soapAction="http://gril.udes.ca/astd/wsdl/ASTDAR"
         style="document"/>
43      <input name="inputAR">
44       <soap:body use="literal"
          namespace="http://gril.udes.ca/astd/wsdl"/>
45      </input>
46      <output name="outputAR">
47       <soap:body use="literal"
          namespace="http://gril.udes.ca/astd/wsdl"/>
48      </output>
49     </operation>
50     <operation name="Commit">
51      <soap:operation
         soapAction="http://gril.udes.ca/astd/wsdl/ASTDCommit"
         style="document"/>
52      <input name="inputCommit">
53       <soap:body use="literal"
          namespace="http://gril.udes.ca/astd/wsdl"/>
54      </input>
55      <output name="outputCommit">
56       <soap:body use="literal"
          namespace="http://gril.udes.ca/astd/wsdl"/>
```

```
57      </output>
58     </operation>
59     <operation name="Stop">
60      <soap:operation
          soapAction="http://gril.udes.ca/astd/wsdl/ASTDStop"
          style="document"/>
61      <input name="inputStop">
62       <soap:body use="literal"
           namespace="http://gril.udes.ca/astd/wsdl"/>
63      </input>
64     </operation>
65    </binding>
```

Programme B.1 – Éléments **binding** du fichier WSDL

Annexe C

Spécifications ASTD des politiques de la banque

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- Balance AC Policy -->
3  <ax:Specification
4   xmlns:ax="http://gril.udes.ca/astd/schema/ASTD"
5   xmlns:px="http://gril.udes.ca/astd/schema/Predicate"
6   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8   xsi:schemaLocation="http://gril.udes.ca/astd/schema/ASTD ASTD.xsd">
9   <!-- Rules 1, 2 -->
10  <ax:KleeneClosure ax:Name="BankACPolicyBalance">
11   <ax:B>
12    <ax:QChoice ax:Name="R1_R2_QCH" ax:X="customer" ax:T="CUSTOMERID">
13     <ax:B>
14      <ax:Choice ax:Name="R1_R2_QCH_CH">
15       <ax:Left>
16        <!-- Rule 1 : A customer can check the balance
17        on his accounts only -->
18        <ax:Automaton ax:Name="R1_AUT" ax:N0="R1q0">
19         <ax:States>
20          <ax:State ax:Name="R1q0"><ax:Elementary ax:Final="false"/></ax:State>
21          <ax:State ax:Name="R1q1"><ax:Elementary ax:Final="true"/></ax:State>
22         </ax:States>
23         <ax:Transitions>
24          <ax:Transition ax:Final="false">
25           <ax:Phi><px:Predicate>
26            <px:Boolean>true</px:Boolean>
27           </px:Predicate></ax:Phi>
28           <ax:LocalArrow ax:N1="R1q0" ax:N2="R1q1"/>
```

```
29            <ax:Event ax:Name="balance">
30             <ax:PV ax:X="userId" ax:V="$customer"/>
31             <ax:PV ax:X="roleId" ax:V="Customer"/>
32             <ax:PV ax:X="customerId" ax:V="$customer"/>
33             <ax:PV ax:X="accountId" ax:V="_"/>
34            </ax:Event>
35           </ax:Transition>
36          </ax:Transitions>
37         </ax:Automaton><!-- Rule 1 -->
38        </ax:Left>
39        <ax:Right>
40         <!-- Rule 2 : A cashier can check the balance on
41         all customer accounts except on his own accounts -->
42         <ax:QChoice ax:Name="R2_QCH" ax:X="cashier" ax:T="CASHIERID">
43          <ax:B>
44           <ax:Automaton ax:Name="R2_QCH_AUT" ax:N0="R2q0">
45            <ax:States>
46             <ax:State ax:Name="R2q0"><ax:Elementary ax:Final="false"/></ax:State>
47             <ax:State ax:Name="R2q1"><ax:Elementary ax:Final="true"/></ax:State>
48            </ax:States>
49            <ax:Transitions>
50             <ax:Transition ax:Final="false">
51              <ax:Phi><px:Predicate>
52                 <px:NotEqual>
53                  <px:Left><px:Variable>cashier</px:Variable></px:Left>
54                  <px:Right><px:Variable>customer</px:Variable></px:Right>
55                 </px:NotEqual>
56              </px:Predicate></ax:Phi>
57              <ax:LocalArrow ax:N1="R2q0" ax:N2="R2q1"/>
58              <ax:Event ax:Name="balance">
59               <ax:PV ax:X="userId" ax:V="$cashier"/>
60               <ax:PV ax:X="roleId" ax:V="Cashier"/>
61               <ax:PV ax:X="customerId" ax:V="$customer"/>
62               <ax:PV ax:X="accountId" ax:V="_"/>
63              </ax:Event>
64             </ax:Transition>
65            </ax:Transitions>
66           </ax:Automaton>
67          </ax:B>
68         </ax:QChoice><!-- Rule 2 -->
69        </ax:Right>
70       </ax:Choice>
71      </ax:B>
72     </ax:QChoice>
73    </ax:B>
74   </ax:KleeneClosure><!-- Rules 1, 2 -->
75   <ax:Types>
76    <xsd:simpleType xsd:name="CUSTOMERID">
77     <xsd:union>
```

```
78      <!-- Cashiers that are also customers -->
79      <xsd:simpleType>
80       <xsd:restriction xsd:base="integer">
81        <xsd:minInclusive xsd:value="45"/>
82        <xsd:maxInclusive xsd:value="75"/>
83       </xsd:restriction>
84      </xsd:simpleType>
85      <!-- Advisors that are also customers -->
86      <xsd:simpleType>
87       <xsd:restriction xsd:base="integer">
88        <xsd:minInclusive xsd:value="100"/>
89        <xsd:maxInclusive xsd:value="105"/>
90       </xsd:restriction>
91      </xsd:simpleType>
92      <!-- Head offices that are also customers -->
93      <xsd:simpleType>
94       <xsd:restriction xsd:base="integer">
95        <xsd:minInclusive xsd:value="114"/>
96        <xsd:maxInclusive xsd:value="115"/>
97       </xsd:restriction>
98      </xsd:simpleType>
99      <!-- Pure customers -->
100     <xsd:simpleType>
101      <xsd:restriction xsd:base="integer">
102       <xsd:minInclusive xsd:value="116"/>
103       <xsd:maxInclusive xsd:value="876"/>
104      </xsd:restriction>
105     </xsd:simpleType>
106    </xsd:union>
107   </xsd:simpleType>
108   <xsd:simpleType xsd:name="CASHIERID">
109    <xsd:restriction xsd:base="integer">
110     <xsd:minInclusive xsd:value="1"/>
111     <xsd:maxInclusive xsd:value="75"/>
112    </xsd:restriction>
113   </xsd:simpleType>
114  </ax:Types>
115 <ax:Specification>
```

Programme C.1 – Politique de contrôle d'accès `BankACPolicyBalance`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!-- SoD, Obligations AC Policy -->
3  <ax:Specification
4   xmlns:ax="http://gril.udes.ca/astd/schema/ASTD"
5   xmlns:px="http://gril.udes.ca/astd/schema/Predicate"
6   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8   xsi:schemaLocation="http://gril.udes.ca/astd/schema/ASTD ASTD.xsd">
9   <ax:Synchronization ax:Name="BankACPolicySoDObl">
10   <ax:Delta>deposit</ax:Delta>
11   <ax:Left>
12    <!-- Rules pertaining to a customer (CuR=Customer Rules) -->
13    <ax:QSynchronization ax:Name="CuR" ax:X="customer" ax:T="CUSTOMERID">
14     <ax:Delta></ax:Delta>
15     <ax:B>
16      <!-- CsG = Customer Glue -->
17      <ax:Synchronization ax:Name="CsG1">
18       <ax:Delta></ax:Delta>
19       <ax:Left>
20        <!-- Rules 4.1, 4.3 -->
21        <!-- Rule 4.1 : A cheque deposit can be made by an advisor
22        or a head office, he cannot make it on his own accounts -->
23        <!-- Rule 4.3 : A cash deposit can only be made by
24        a cashier, he cannot make it on his own accounts -->
25        <ax:KleeneClosure ax:Name="R4_1_R4_3">
26         <ax:B>
27          <ax:QChoice ax:Name="R4_1_R4_3_QCH" ax:X="user" ax:T="EMPLOYEEID">
28           <ax:B>
29            <ax:QChoice ax:Name="R4_1_R4_3_QCH_QCH" ax:X="role" ax:T="ROLE">
30             <ax:B>
31              <ax:Choice ax:Name="R4_1_R4_3_QCH_QCH_CH">
32               <ax:Left>
33                <ax:Automaton ax:Name="R4_1_AUT" ax:N0="R4_1_AUTq0">
34                 <ax:States>
35                  <ax:State ax:Name="R4_1_AUTq0">
                     <ax:Elementary ax:Final="false"/>
                    </ax:State>
36                  <ax:State ax:Name="R4_1_AUTq1">
                     <ax:Elementary ax:Final="true"/>
                    </ax:State>
37                 </ax:States>
38                 <ax:Transitions>
39                  <ax:Transition ax:Final="false">
40                   <ax:Phi><px:Predicate>
41                       <px:And>
42                        <!-- user != customer -->
43                        <px:Left><px:NotEqual>
44                         <px:Left><px:Variable>user</px:Variable></px:Left>
45                         <px:Right><px:Variable>customer</px:Variable></px:Right>
```

```
                    </px:NotEqual></px:Left>
                    <px:Right><px:Or>
                     <px:Left><px:Equal>
                       <px:Left><px:Variable>role</px:Variable></px:Left>
                       <px:Right><px:String>Advisor</px:String></px:Right>
                     </px:Equal></px:Left>
                     <px:Right><px:Equal>
                      <px:Left><px:Variable>role</px:Variable></px:Left>
                      <px:Right><px:String>Head Office</px:String></px:Right>
                     </px:Equal></px:Right>
                    </px:Or></px:Right>
                   </px:And>
                 </px:Predicate></ax:Phi>
                 <ax:LocalArrow ax:N1="R4_1_AUTq0" ax:N2="R4_1_AUTq1"/>
                 <ax:Event ax:Name="deposit">
                  <ax:PV ax:X="userId" ax:V="$user"/>
                  <ax:PV ax:X="roleId" ax:V="$role"/>
                  <ax:PV ax:X="customerId" ax:V="$customer"/>
                  <ax:PV ax:X="accountId" ax:V="_"/>
                  <ax:PV ax:X="chequeId" ax:V="_"/>
                  <ax:PV ax:X="chequeNumber" ax:V="_"/>
                  <ax:PV ax:X="amount" ax:V="_"/>
                 </ax:Event>
                </ax:Transition>
               </ax:Transitions>
              </ax:Automaton>
             </ax:Left>
             <ax:Right>
              <ax:Automaton ax:Name="R4_3_AUT" ax:N0="R4_3_AUTq0">
               <ax:States>
                <ax:State ax:Name="R4_3_AUTq0">
                 <ax:Elementary ax:Final="false"/>
                </ax:State>
                <ax:State ax:Name="R4_3_AUTq1">
                 <ax:Elementary ax:Final="true"/>
                </ax:State>
               </ax:States>
               <ax:Transitions>
                <ax:Transition ax:Final="false">
                 <ax:Phi><px:Predicate>
                     <!-- user != customer -->
                     <px:NotEqual>
                      <px:Left><px:Variable>user</px:Variable></px:Left>
                      <px:Right><px:Variable>customer</px:Variable></px:Right>
                     </px:NotEqual>
                 </px:Predicate></ax:Phi>
                 <ax:LocalArrow ax:N1="R4_3_AUTq0" ax:N2="R4_3_AUTq1"/>
                 <ax:Event ax:Name="deposit">
                  <ax:PV ax:X="userId" ax:V="$user"/>
```

```
91                  <ax:PV ax:X="roleId" ax:V="Cashier"/>
92                  <ax:PV ax:X="customerId" ax:V="$customer"/>
93                  <ax:PV ax:X="accountId" ax:V="_"/>
94                  <ax:PV ax:X="chequeId" ax:V="0"/>
95                  <ax:PV ax:X="chequeNumber" ax:V="_"/>
96                  <ax:PV ax:X="amount" ax:V="_"/>
97                 </ax:Event>
98                </ax:Transition>
99               </ax:Transitions>
100             </ax:Automaton>
101            </ax:Right>
102           </ax:Choice>
103          </ax:B>
104         </ax:QChoice>
105        </ax:B>
106       </ax:QChoice>
107      </ax:B>
108     </ax:KleeneClosure><!-- Rules 4.1, 4.3 -->
109    </ax:Left>
110
111    <ax:Right>
112     <!-- Rule 5 : A cashier must be registered with a
113     customer BEFORE doing deposits for the customer's
114     accounts, a cashier can't be registered with himself
115     as a customer -->
116     <ax:KleeneClosure ax:Name="R5">
117      <ax:B>
118       <ax:QChoice ax:Name="R5_QCH" ax:X="cashier" ax:T="CASHIERID">
119        <ax:B>
120         <ax:Sequence ax:Name="R5_QCH_SEQ">
121          <ax:First>
122           <!-- Rules 6, 7 -->
123           <!-- Rule 6 A head office can register a customer
124           to a cashier -->
125           <!-- Rule 7 -->
126           <ax:QChoice ax:Name="R6_R7" ax:X="user2"
               ax:T="CUSTOMER_AND_HEADOFFICE_ID">
127            <ax:B>
128             <ax:QChoice ax:Name="R6_R7_QCH" ax:X="role" ax:T="ROLE">
129              <ax:B>
130               <ax:Automaton ax:Name="R6_R7_AUT" ax:N0="R6_R7_AUTq0">
131                <ax:States>
132                 <ax:State ax:Name="R6_R7_AUTq0">
                     <ax:Elementary ax:Final="false"/>
                    </ax:State>
133                 <ax:State ax:Name="R6_R7_AUTq1">
                     <ax:Elementary ax:Final="true"/>
                    </ax:State>
134                </ax:States>
```

```
<ax:Transitions>
 <ax:Transition ax:Final="false">
  <ax:Phi><px:Predicate>
     <px:And>
      <!-- Rule 7.2, 6.3 -->
      <!-- cashier != customer -->
      <px:Left><px:NotEqual>
       <px:Left><px:Variable>cashier</px:Variable></px:Left>
       <px:Right><px:Variable>customer</px:Variable></px:Right>
      </px:NotEqual></px:Left>
      <px:Right><px:And>
       <!-- 7.1 -->
       <!-- user2 = customer => r = 'Customer' -->
       <px:Left>
        <px:Or>
         <px:Left><px:NotEqual>
          <px:Left><px:Variable>user2</px:Variable></px:Left>
          <px:Right><px:Variable>customer</px:Variable></px:Right>
         </px:NotEqual></px:Left>
         <px:Right><px:Equal>
          <px:Left><px:Variable>role</px:Variable></px:Left>
          <px:Right><px:String>Customer</px:String></px:Right>
         </px:Equal></px:Right>
        </px:Or>
       </px:Left>
       <!-- Rule 6.2 -->
       <!-- user2 != customer => r = 'Head Office' -->
       <px:Right>
        <px:Or>
         <px:Left><px:Equal>
          <px:Left><px:Variable>user2</px:Variable></px:Left>
          <px:Right><px:Variable>customer</px:Variable></px:Right>
         </px:Equal></px:Left>
         <px:Right><px:Equal>
          <px:Left><px:Variable>role</px:Variable></px:Left>
          <px:Right><px:String>Head Office</px:String></px:Right>
         </px:Equal></px:Right>
        </px:Or>
       </px:Right>
      </px:And></px:Right>
     </px:And>
  </px:Predicate></ax:Phi>
  <ax:LocalArrow ax:N1="R6_R7_AUTq0" ax:N2="R6_R7_AUTq1"/>
  <ax:Event ax:Name="register">
   <ax:PV ax:X="userId" ax:V="$user2"/>
   <ax:PV ax:X="roleId" ax:V="$role"/>
   <ax:PV ax:X="customerId" ax:V="$customer"/>
   <ax:PV ax:X="customerIdToRegister" ax:V="_"/>
   <ax:PV ax:X="cashierIdToRegister" ax:V="$cashier"/>
```

```
                    </ax:Event>
                   </ax:Transition>
                  </ax:Transitions>
                 </ax:Automaton>
                </ax:B>
               </ax:QChoice>
              </ax:B>
             </ax:QChoice><!-- Rules 6, 7 -->
            </ax:First>
            <ax:Second>
             <!-- Rule 5.2 : Check deposits by a cashier
             AFTER he is registered with the customer -->
             <ax:KleeneClosure ax:Name="R5_2">
              <ax:B>
               <ax:QChoice ax:Name="R5_2_QCH" ax:X="cheque" ax:T="CHEQUEID2">
                <ax:B>
                 <ax:Automaton ax:Name="R5_2_QCH_AUT" ax:N0="R5_2_QCH_AUTq0">
                  <ax:States>
                   <ax:State ax:Name="R5_2_QCH_AUTq0">
                   <ax:Elementary ax:Final="false"/>
                    </ax:State>
                   <ax:State ax:Name="R5_2_QCH_AUTq1">
                   <ax:Elementary ax:Final="true"/>
                    </ax:State>
                  </ax:States>
                  <ax:Transitions>
                   <ax:Transition ax:Final="false">
                    <ax:Phi><px:Predicate>
                       <px:NotEqual>
                        <px:Left><px:Variable>cheque</px:Variable></px:Left>
                        <px:Right><px:Number>0</px:Number></px:Right>
                       </px:NotEqual>
                    </px:Predicate></ax:Phi>
                    <ax:LocalArrow ax:N1="R5_2_QCH_AUTq0" ax:N2="R5_2_QCH_AUTq1"/>
                    <ax:Event ax:Name="deposit">
                     <ax:PV ax:X="userId" ax:V="$cashier"/>
                     <ax:PV ax:X="roleId" ax:V="Cashier"/>
                     <ax:PV ax:X="customerId" ax:V="$customer"/>
                     <ax:PV ax:X="accountId" ax:V="_"/>
                     <ax:PV ax:X="chequeId" ax:V="$cheque"/>
                     <ax:PV ax:X="chequeNumber" ax:V="_"/>
                     <ax:PV ax:X="amount" ax:V="_"/>
                    </ax:Event>
                   </ax:Transition>
                  </ax:Transitions>
                 </ax:Automaton>
                </ax:B>
               </ax:QChoice>
              </ax:B>
```

```
229              </ax:KleeneClosure><!-- Rule 5.2 -->
230             </ax:Second>
231            </ax:Sequence>
232           </ax:B>
233          </ax:QChoice>
234         </ax:B>
235        </ax:KleeneClosure><!-- Rule 5 -->
236       </ax:Right>
237      </ax:Synchronization>
238     </ax:B>
239    </ax:QSynchronization><!-- Rules pertaining to a customer -->
240   </ax:Left>
241   <ax:Right>
242    <!-- Rules pertaining to a cheque (ChR = Cheque Rules) -->
243    <ax:QSynchronization ax:Name="ChR" ax:X="cheque" ax:T="CHEQUEID">
244     <ax:Delta></ax:Delta>
245     <ax:B>
246      <!-- ChG=Cheque Glue -->
247      <ax:Choice ax:Name="ChG">
248       <ax:Left>
249        <!-- Rule 4.3.1 : A cash deposit (cheque=0) can only be made
250        by a cashier -->
251        <ax:KleeneClosure ax:Name="R4_3_1">
252         <ax:B>
253          <ax:Automaton ax:Name="R4_3_1_AUT" ax:N0="R4_3_1_AUTq0">
254           <ax:States>
255            <ax:State ax:Name="R4_3_1_AUTq0">
                <ax:Elementary ax:Final="false"/>
               </ax:State>
256            <ax:State ax:Name="R4_3_1_AUTq1">
                <ax:Elementary ax:Final="true"/>
               </ax:State>
257           </ax:States>
258           <ax:Transitions>
259            <ax:Transition ax:Final="false">
260             <ax:Phi><px:Predicate>
261               <px:Equal>
262                <px:Left><px:Variable>cheque</px:Variable></px:Left>
263                <px:Right><px:Number>0</px:Number></px:Right>
264               </px:Equal>
265             </px:Predicate></ax:Phi>
266             <ax:LocalArrow ax:N1="R4_3_1_AUTq0" ax:N2="R4_3_1_AUTq1"/>
267             <ax:Event ax:Name="deposit">
268              <ax:PV ax:X="userId" ax:V="_"/>
269              <ax:PV ax:X="roleId" ax:V="Cashier"/>
270              <ax:PV ax:X="customerId" ax:V="_"/>
271              <ax:PV ax:X="accountId" ax:V="_"/>
272              <ax:PV ax:X="chequeId" ax:V="$cheque"/>
273              <ax:PV ax:X="chequeNumber" ax:V="_"/>
```

```
274              <ax:PV ax:X="amount" ax:V="_"/>
275             </ax:Event>
276            </ax:Transition>
277           </ax:Transitions>
278          </ax:Automaton>
279         </ax:B>
280        </ax:KleeneClosure><!-- Rule 4.3.1 -->
281       </ax:Left>
282       <ax:Right>
283        <!-- Rules 9, 10, 11: SoD -->
284        <!-- Rule 9 : A cheque deposit made by a cashier can be
285        validated/cancelled by an advisor or a head office -->
286        <!-- Rule 10 : A cheque deposit made by an advisor must
287        be validated/cancelled only by a head office -->
288        <!-- Rule 11 : A cheque deposit made by a head office must
289        be validated/cancelled only by another head office -->
290        <ax:QChoice ax:Name="R9_10_11" ax:X="user1" ax:T="EMPLOYEEID">
291         <ax:B>
292          <ax:QChoice ax:Name="R9_10_11_QCH" ax:X="role1" ax:T="ROLE">
293           <ax:B>
294            <ax:Sequence ax:Name="R9_10_11_QCH_SEQ">
295             <ax:First>
296              <ax:Automaton ax:Name="R9_10_11_QCH_SEQ_F_AUT"
                  ax:N0="R9_10_11_QCH_SEQ_F_AUTq0">
297               <ax:States>
298                <ax:State ax:Name="R9_10_11_QCH_SEQ_F_AUTq0">
                    <ax:Elementary ax:Final="false"/>
                   </ax:State>
299                <ax:State ax:Name="R9_10_11_QCH_SEQ_F_AUTq1">
                    <ax:Elementary ax:Final="true"/>
                   </ax:State>
300               </ax:States>
301               <ax:Transitions>
302                <ax:Transition ax:Final="false">
303                 <ax:Phi><px:Predicate>
304                    <px:NotEqual>
305                     <px:Left><px:Variable>cheque</px:Variable></px:Left>
306                     <px:Right><px:Number>0</px:Number></px:Right>
307                    </px:NotEqual>
308                 </px:Predicate></ax:Phi>
309                 <ax:LocalArrow ax:N1="R9_10_11_QCH_SEQ_F_AUTq0"
                     ax:N2="R9_10_11_QCH_SEQ_F_AUTq1"/>
310                 <ax:Event ax:Name="deposit">
311                  <ax:PV ax:X="userId" ax:V="$user1"/>
312                  <ax:PV ax:X="roleId" ax:V="$role1"/>
313                  <ax:PV ax:X="customerId" ax:V="_"/>
314                  <ax:PV ax:X="accountId" ax:V="_"/>
315                  <ax:PV ax:X="chequeId" ax:V="$cheque"/>
316                  <ax:PV ax:X="chequeNumber" ax:V="_"/>
```

```
317                  <ax:PV ax:X="amount" ax:V="_"/>
318                 </ax:Event>
319                </ax:Transition>
320               </ax:Transitions>
321              </ax:Automaton>
322             </ax:First>
323             <ax:Second>
324              <ax:QChoice ax:Name="R9_10_11_QCH_SEQ_S_QCH" ax:X="user2"
                  ax:T="EMPLOYEEID">
325               <ax:B>
326                <ax:QChoice ax:Name="R9_10_11_QCH_SEQ_S_QCH_QCH" ax:X="role2"
                    ax:T="ROLE">
327                <ax:B>
328                 <ax:Guard ax:Name="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD">
329                  <px:Predicate>
330                   <!--
331                    (Rule 9) r1 = 'Cashier'
                          => (r2 = 'Advisor' or r2 = 'Head Office')
332                    (Rule 10) and r1 = 'Advisor' => r2 = 'Head Office'
333                    (Rule 11) and r1 = 'Head Office'
                          => (r2 = 'Head Office' and u1 != u2)
334                   -->
335                   <px:And>
336                    <px:Left><px:And>
337                     <!-- r1 = 'Cashier'
                          => (r2 = 'Advisor' or r2 = 'Head Office') -->
338                     <px:Left>
339                      <px:Or>
340                       <px:Left><px:NotEqual>
341                        <px:Left><px:Variable>role1</px:Variable></px:Left>
342                        <px:Right><px:String>Cashier</px:String></px:Right>
343                       </px:NotEqual></px:Left>
344                       <px:Right><px:Or>
345                         <px:Left><px:Equal>
346                          <px:Left><px:Variable>role2</px:Variable></px:Left>
347                          <px:Right><px:String>Advisor</px:String></px:Right>
348                         </px:Equal></px:Left>
349                         <px:Right><px:Equal>
350                          <px:Left><px:Variable>role2</px:Variable></px:Left>
351                          <px:Right><px:String>Head Office</px:String></px:Right>
352                         </px:Equal></px:Right>
353                       </px:Or></px:Right>
354                      </px:Or>
355                     </px:Left>
356                     <!-- r1 = 'Advisor' => r2 = 'Head Office' -->
357                     <px:Right>
358                      <px:Or>
359                       <px:Left><px:NotEqual>
360                        <px:Left><px:Variable>role1</px:Variable></px:Left>
```

```
361                      <px:Right><px:String>Advisor</px:String></px:Right>
362                     </px:NotEqual></px:Left>
363                     <px:Right><px:Equal>
364                      <px:Left><px:Variable>role2</px:Variable></px:Left>
365                      <px:Right><px:String>Head Office</px:String></px:Right>
366                     </px:Equal></px:Right>
367                    </px:Or>
368                   </px:Right>
369                  </px:And></px:Left>
370                  <!-- r1 = 'Head Office' => (r2 = 'Head Office' and u1 != u2) -->
371                  <px:Right>
372                    <px:Or>
373                     <px:Left><px:NotEqual>
374                      <px:Left><px:Variable>role1</px:Variable></px:Left>
375                      <px:Right><px:String>Head Office</px:String></px:Right>
376                     </px:NotEqual></px:Left>
377                     <px:Right><px:And>
378                       <px:Left><px:Equal>
379                        <px:Left><px:Variable>role2</px:Variable></px:Left>
380                        <px:Right><px:String>Head Office</px:String></px:Right>
381                       </px:Equal></px:Left>
382                       <px:Right><px:NotEqual>
383                        <px:Left><px:Variable>user1</px:Variable></px:Left>
384                        <px:Right><px:Variable>user2</px:Variable></px:Right>
385                       </px:NotEqual></px:Right>
386                     </px:And></px:Right>
387                    </px:Or>
388                  </px:Right>
389                 </px:And>
390                </px:Predicate>
391                <ax:B>
392                 <ax:Automaton ax:Name="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD_AUT"
                   ax:N0="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD_AUTq0">
393                  <ax:States>
394                   <ax:State ax:Name="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD_AUTq0">
                     <ax:Elementary ax:Final="false"/>
                    </ax:State>
395                   <ax:State ax:Name="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD_AUTq1">
                     <ax:Elementary ax:Final="true"/>
                    </ax:State>
396                  </ax:States>
397                  <ax:Transitions>
398                   <ax:Transition ax:Final="false">
399                    <ax:Phi><px:Predicate>
400                     <px:Boolean>true</px:Boolean>
401                    </px:Predicate></ax:Phi>
402                    <ax:LocalArrow ax:N1="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD_AUTq0"
                     ax:N2="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD_AUTq1"/>
403                    <ax:Event ax:Name="validate">
```

```
                        <ax:PV ax:X="userId" ax:V="$user2"/>
                        <ax:PV ax:X="roleId" ax:V="$role2"/>
                        <ax:PV ax:X="customerId" ax:V="_"/>
                        <ax:PV ax:X="chequeId" ax:V="$cheque"/>
                       </ax:Event>
                      </ax:Transition>
                      <ax:Transition ax:Final="false">
                       <ax:Phi><px:Predicate>
                        <px:Boolean>true</px:Boolean>
                       </px:Predicate></ax:Phi>
                       <ax:LocalArrow ax:N1="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD_AUTq0"
                        ax:N2="R9_10_11_QCH_SEQ_S_QCH_QCH_GRD_AUTq1"/>
                       <ax:Event ax:Name="cancel">
                        <ax:PV ax:X="userId" ax:V="$user2"/>
                        <ax:PV ax:X="roleId" ax:V="$role2"/>
                        <ax:PV ax:X="customerId" ax:V="_"/>
                        <ax:PV ax:X="chequeId" ax:V="$cheque"/>
                       </ax:Event>
                      </ax:Transition>
                     </ax:Transitions>
                    </ax:Automaton>
                   </ax:B>
                  </ax:Guard>
                 </ax:B>
                </ax:QChoice>
               </ax:B>
              </ax:QChoice>
             </ax:Second>
            </ax:Sequence>
           </ax:B>
          </ax:QChoice>
         </ax:B>
        </ax:QChoice><!-- Rules 9, 10, 11 -->
       </ax:Right>
      </ax:Choice><!-- Cheque glue -->
     </ax:B>
    </ax:QSynchronization><!-- Rules pertaining to a cheque -->
   </ax:Right>
  </ax:Synchronization>
  <ax:Types>
   <xsd:simpleType xsd:name="CUSTOMERID">
    <xsd:union>
     <!-- Cashiers that are also customers -->
     <xsd:simpleType>
      <xsd:restriction xsd:base="integer">
       <xsd:minInclusive xsd:value="45"/>
       <xsd:maxInclusive xsd:value="75"/>
      </xsd:restriction>
     </xsd:simpleType>
```

```
452      <!-- Advisors that are also customers -->
453      <xsd:simpleType>
454       <xsd:restriction xsd:base="integer">
455        <xsd:minInclusive xsd:value="100"/>
456        <xsd:maxInclusive xsd:value="105"/>
457       </xsd:restriction>
458      </xsd:simpleType>
459      <!-- Head offices that are also customers -->
460      <xsd:simpleType>
461       <xsd:restriction xsd:base="integer">
462        <xsd:minInclusive xsd:value="114"/>
463        <xsd:maxInclusive xsd:value="115"/>
464       </xsd:restriction>
465      </xsd:simpleType>
466      <!-- Pure customers -->
467      <xsd:simpleType>
468       <xsd:restriction xsd:base="integer">
469        <xsd:minInclusive xsd:value="116"/>
470        <xsd:maxInclusive xsd:value="876"/>
471       </xsd:restriction>
472      </xsd:simpleType>
473     </xsd:union>
474    </xsd:simpleType>
475    <xsd:simpleType xsd:name="CASHIERID">
476     <xsd:restriction xsd:base="integer">
477      <xsd:minInclusive xsd:value="1"/>
478      <xsd:maxInclusive xsd:value="75"/>
479     </xsd:restriction>
480    </xsd:simpleType>
481    <xsd:simpleType xsd:name="ADVISORID">
482     <xsd:restriction xsd:base="integer">
483      <xsd:minInclusive xsd:value="76"/>
484      <xsd:maxInclusive xsd:value="105"/>
485     </xsd:restriction>
486    </xsd:simpleType>
487    <xsd:simpleType xsd:name="HEADOFFICEID">
488     <xsd:restriction xsd:base="integer">
489      <xsd:minInclusive xsd:value="106"/>
490      <xsd:maxInclusive xsd:value="115"/>
491     </xsd:restriction>
492    </xsd:simpleType>
493    <xsd:simpleType xsd:name="EMPLOYEEID">
494     <xsd:union xsd:memberTypes="CASHIERID ADVISORID HEADOFFICEID"/>
495    </xsd:simpleType>
496    <xsd:simpleType xsd:name="CUSTOMER_AND_HEADOFFICE_ID">
497     <xsd:union xsd:memberTypes="CUSTOMERID HEADOFFICEID"/>
498    </xsd:simpleType>
499    <xsd:simpleType xsd:name="ROLE">
500     <xsd:restriction xsd:base="string">
```

```
501       <xsd:enumeration xsd:value="Customer"/>
502       <xsd:enumeration xsd:value="Cashier"/>
503       <xsd:enumeration xsd:value="Advisor"/>
504       <xsd:enumeration xsd:value="Head Office"/>
505      </xsd:restriction>
506     </xsd:simpleType>
507     <xsd:simpleType xsd:name="CHEQUEID">
508      <xsd:restriction xsd:base="integer">
509       <xsd:minInclusive xsd:value="0"/>
510       <xsd:maxInclusive xsd:value="2000"/>
511      </xsd:restriction>
512     </xsd:simpleType>
513     <xsd:simpleType xsd:name="CHEQUEID2">
514      <xsd:restriction xsd:base="integer">
515       <xsd:minInclusive xsd:value="1"/>
516       <xsd:maxInclusive xsd:value="2000"/>
517      </xsd:restriction>
518     </xsd:simpleType>
519    </ax:Types>
520   </ax:Specification>
```

Programme C.2 – Politique de contrôle d'accès `BankACPolicySoDObl`

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Withdraw AC Policy -->
<ax:Specification
 xmlns:ax="http://gril.udes.ca/astd/schema/ASTD"
 xmlns:px="http://gril.udes.ca/astd/schema/Predicate"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://gril.udes.ca/astd/schema/ASTD ASTD.xsd">
 <!-- Rule 12 : A withdrawal cannot be performed by a customer
 and an employee cannot perform a withdraw from his own accounts -->
 <ax:KleeneClosure ax:Name="BankACPolicyWithdraw">
  <ax:B>
   <ax:QChoice ax:Name="R12_QCH" ax:X="customer" ax:T="CUSTOMERID">
    <ax:B>
     <ax:QChoice ax:Name="R12_QCH_QCH" ax:X="employee" ax:T="EMPLOYEEID">
      <ax:B>
       <ax:QChoice ax:Name="R12_QCH_QCH_QCH" ax:X="role" ax:T="ROLE">
        <ax:B>
         <ax:Automaton ax:Name="R12_QCH_QCH_QCH_AUT" ax:N0="R12q0">
          <ax:States>
           <ax:State ax:Name="R12q0"><ax:Elementary ax:Final="false"/></ax:State>
           <ax:State ax:Name="R12q1"><ax:Elementary ax:Final="true"/></ax:State>
          </ax:States>
          <ax:Transitions>
           <ax:Transition ax:Final="false">
            <!-- employee != customer and role != 'Customer' -->
            <ax:Phi><px:Predicate>
               <px:And>
                <px:Left><px:NotEqual>
                 <px:Left><px:Variable>employee</px:Variable></px:Left>
                 <px:Right><px:Variable>customer</px:Variable></px:Right>
                </px:NotEqual></px:Left>
                <px:Right><px:NotEqual>
                 <px:Left><px:Variable>role</px:Variable></px:Left>
                 <px:Right><px:String>Customer</px:String></px:Right>
                </px:NotEqual></px:Right>
               </px:And>
            </px:Predicate></ax:Phi>
            <ax:LocalArrow ax:N1="R12q0" ax:N2="R12q1"/>
            <ax:Event ax:Name="withdraw">
             <ax:PV ax:X="userId" ax:V="$employee"/>
             <ax:PV ax:X="roleId" ax:V="$role"/>
             <ax:PV ax:X="customerId" ax:V="$customer"/>
             <ax:PV ax:X="accountId" ax:V="_"/>
             <ax:PV ax:X="amount" ax:V="_"/>
            </ax:Event>
           </ax:Transition>
          </ax:Transitions>
         </ax:Automaton>
```

```
          </ax:B>
         </ax:QChoice>
        </ax:B>
       </ax:QChoice>
      </ax:B>
     </ax:QChoice>
    </ax:B>
  </ax:KleeneClosure><!-- Rule 12 -->
  <ax:Types>
   <xsd:simpleType xsd:name="CUSTOMERID">
    <xsd:union>
     <!-- Cashiers that are also customers -->
     <xsd:simpleType>
      <xsd:restriction xsd:base="integer">
       <xsd:minInclusive xsd:value="45"/>
       <xsd:maxInclusive xsd:value="75"/>
      </xsd:restriction>
     </xsd:simpleType>
     <!-- Advisors that are also customers -->
     <xsd:simpleType>
      <xsd:restriction xsd:base="integer">
       <xsd:minInclusive xsd:value="100"/>
       <xsd:maxInclusive xsd:value="105"/>
      </xsd:restriction>
     </xsd:simpleType>
     <!-- Head offices that are also customers -->
     <xsd:simpleType>
      <xsd:restriction xsd:base="integer">
       <xsd:minInclusive xsd:value="114"/>
       <xsd:maxInclusive xsd:value="115"/>
      </xsd:restriction>
     </xsd:simpleType>
     <!-- Pure customers -->
     <xsd:simpleType>
      <xsd:restriction xsd:base="integer">
       <xsd:minInclusive xsd:value="116"/>
       <xsd:maxInclusive xsd:value="876"/>
      </xsd:restriction>
     </xsd:simpleType>
    </xsd:union>
   </xsd:simpleType>
   <xsd:simpleType xsd:name="CASHIERID">
    <xsd:restriction xsd:base="integer">
     <xsd:minInclusive xsd:value="1"/>
     <xsd:maxInclusive xsd:value="75"/>
    </xsd:restriction>
   </xsd:simpleType>
   <xsd:simpleType xsd:name="ADVISORID">
    <xsd:restriction xsd:base="integer">
```

```
99      <xsd:minInclusive xsd:value="76"/>
100     <xsd:maxInclusive xsd:value="105"/>
101    </xsd:restriction>
102   </xsd:simpleType>
103   <xsd:simpleType xsd:name="HEADOFFICEID">
104    <xsd:restriction xsd:base="integer">
105     <xsd:minInclusive xsd:value="106"/>
106     <xsd:maxInclusive xsd:value="115"/>
107    </xsd:restriction>
108   </xsd:simpleType>
109   <xsd:simpleType xsd:name="EMPLOYEEID">
110    <xsd:union xsd:memberTypes="CASHIERID ADVISORID HEADOFFICEID"/>
111   </xsd:simpleType>
112   <xsd:simpleType xsd:name="ROLE">
113    <xsd:restriction xsd:base="string">
114     <xsd:enumeration xsd:value="Customer"/>
115     <xsd:enumeration xsd:value="Cashier"/>
116     <xsd:enumeration xsd:value="Advisor"/>
117     <xsd:enumeration xsd:value="Head Office"/>
118    </xsd:restriction>
119   </xsd:simpleType>
120  </ax:Types>
121 <ax:Specification>
```

Programme C.3 – Politique de contrôle d'accès `BankACPolicyWithdraw`

Bibliographie

[1] J.-R. Abrial.
Modeling in Event-B : System and Software Engineering.
Cambridge University Press, 2010.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta et L. Voisin.
« Rodin : An open toolset for modelling and reasoning in Event-B ».
International Journal on Software Tools for Technology Transfer, 12(6):447–466, 2010.

[3] I. Ait-Sadoune.
« *Modélisation et vérification formelles de composition de services. Une approche fondée sur le raffinement et la preuve* ».
Thèse de doctorat, École Doctorale Sciences et Ingénierie pour l'Information, Université de Poitiers, Limoges, France, décembre 2010.

[4] D. E. Bell et L. J. LaPadula.
« Secure Computer Systems: Mathematical Foundations ».
Rapport Technique 2547, MITRE, mars 1973.

[5] J. Bendisposto, M. Leuschel, O. Ligot et M. Samia.
« La validation de modèles Event-B avec le plug-in ProB pour Rodin ».
Technique et Science Informatiques, 27(8):1065–1084, 2008.

[6] R. Bhatti, A. Ghafoor, E. Bertino et J. B. D. Joshi.
« X-GTRBAC : An XML-based policy specification framework and architecture for enterprise-wide access control ».
ACM Transactions on Information and System Security, 8(2):187–227, 2005.

[7] A. Bianchi, D. Caivano, F. Lanubile et G. Visaggio.
« Evaluating software degradation through entropy ».

Dans *Proceedings of the 7th IEEE International Software Metrics Symposium (METRICS'01)*, pages 210–219. IEEE Computer Society, 2001.

[8] T. Bolognesi et E. Brinksma.
« Introduction to the ISO specification language LOTOS ».
Dans P. H. J. van Eijk, C. A. Vissers et M. Diaz, éditeurs, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers, 1989.

[9] T. Bourdier, H. Cirstea, M. Jaume et H. Kirchner.
« Formal specification and validation of security policies ».
Dans J. Garcia-Alfaro et P. Lafourcade, éditeurs, *Foundations and Practice of Security (FPS'11)*, volume 6888 de *Lecture Notes in Computer Science*, pages 148–163. Springer Berlin / Heidelberg, 2012.

[10] A. Chirichiello et G. Salaün.
« Encoding process algebraic descriptions of Web services into BPEL ».
Web Intelligence and Agent Systems, 5(4):419–434, 2007.

[11] G. Coatrieux, M. Embe Jiague et S. Morucci.
« *Implementation of Web Services for Security Enforcement, Livrable numéro 4.2* ».
Projet SELKIS ANR-08-SEGI-018, février 2011.

[12] Department of Defense.
« *Trusted Computer System Evaluation Criteria* ».
Department of Defense, USA, décembre 1985.

[13] M. Embe Jiague, M. Frappier, F. Gervais, P. Konopacki, R. Laleau, J. Milhau et R. St-Denis.
« Model-driven engineering of functional security policies ».
Dans J. Filipe et J. Cordeiro, éditeurs, *Proceedings of the 12th International Conference on Enterprise Information Systems*, pages 374–379. SciTePress, 2010.

[14] M. Embe Jiague, M. Frappier, F. Gervais, P. Konopacki, R. Laleau, J. Milhau et R. St-Denis.
« A four-concern-oriented secure IS development approach ».
Dans J. Lopez et P. Samarati, éditeurs, *Proceedings of the 8th International*

Conference on Security and Cryptography (SECRYPT'11), pages 464–471. SciTePress, 2011.

[15] M. Embe Jiague, M. Frappier, F. Gervais, R. Laleau et R. St-Denis.
« Enforcing ASTD access-control policies with WS-BPEL processes in SOA environments ».
International Journal of Systems and Service-Oriented Engineering, 2(2):37–59, 2011.

[16] M. Embe Jiague, M. Frappier, F. Gervais, R. Laleau et R. St-Denis.
« From ASTD access control policies to WS-BPEL processes deployed in a SOA environment ».
Dans D. Chiu, L. Bellatreche, H. Sasaki, H. Leung, S.-C. Cheung, H. Hu et J. Shao, éditeurs, *Web Information Systems Engineering – WISE 2010 Workshops*, volume 6724 de *Lecture Notes in Computer Science*, pages 126–141. Springer Berlin / Heidelberg, 2011.

[17] M. Embe Jiague, M. Frappier, F. Gervais, R. Laleau et R. St-Denis.
« A metamodel for the design of access-control policy enforcement managers : Work in progress ».
Dans J. Garcia-Alfaro et P. Lafourcade, éditeurs, *Foundations and Practice of Security (FPS'11)*, volume 6888 de *Lecture Notes in Computer Science*, pages 218–226. Springer Berlin / Heidelberg, 2012.

[18] M. Embe Jiague, R. St-Denis, R. Laleau et F. Gervais.
« A BPEL Implementation of a Security Filter ».
Rapport Technique 2010/04, Institute of Computer Science, University Halle-Wittenberg, novembre 2010.

[19] T. Erl.
SOA Principles of Service Design.
Service-Oriented Computing Series. Prentice Hall, 2007.

[20] D. Fahland.
« Complete Abstract Operational Semantics for the Web Service Business Process Execution Language ».

Rapport Technique 190, Institut für Informatik, Humboldt-Universität zu Berlin, septembre 2005.

[21] B. Fraikin, M. Frappier, F. Gervais et R. Laleau.
« Algebraic State Transition Diagrams ».
Rapport Technique 24, Département d'informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada, octobre 2010.

[22] M. Frappier, B. Fraikin, F. Gervais, R. Laleau et M. Richard.
« Synthesizing information systems : The APIS project ».
Dans C. Rolland, O. Pastor et J.-L. Cavarero, éditeurs, *Proceedings of the 1st International Conference on Research Challenges in Information Science (RCIS'07)*, pages 73–84, 2007.

[23] M. Frappier, F. Gervais, R. Laleau, B. Fraikin et R. St-Denis.
« Extending statecharts with process algebra operators ».
Innovations in Systems and Software Engineering, 4(3):285–292, 2008.

[24] M. Frappier et R. St-Denis.
« EB^3 : An entity-based black-box specification method for information systems ».
Software and System Modeling, 2(2):134–149, 2003.

[25] X. Fu.
« *Formal Specification and Verification of Asynchronously Communicating Web Services* ».
Thèse de doctorat, University of California, Santa Barbarba, California, USA, juin 2004.

[26] M. T. Goodrich et R. Tamassia.
Introduction to Computer Security.
Addison-Wesley, 2010.

[27] G. S. Graham et P. J. Denning.
« Protection : Principles and Practice ».
Dans *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference (AFIPS'72, Spring)*, pages 417–429. ACM, 1972.

[28] T. Hallwyl, F. Henglein et T. Hildebrandt.
« Analysis of the WS-BPEL 2.0 standard using standard-driven implementation ».
Rapport Technique 607, Department of Computer Science, University of Copenhagen, Sierre, Switzerland, juin 2009.

[29] T. Hallwyl, F. Henglein et T. Hildebrandt.
« A standard-driven implementaion of WS-BPEL 2.0 ».
Dans *Proceedings of the 25th ACM Symposium on Applied Computing (SAC'10)*, pages 2472–2476. ACM, 2010.

[30] D. Harel.
« Statecharts : A visual formalism for complex systems ».
Science of Computer Programming, 8(3):231–274, 1987.

[31] C. A. R. Hoare.
Communicating Sequential Processes.
Prentice-Hall, 1985.

[32] International Committee for Information Technology Standards.
« *Information Technology — Role Based Access Control* ».
ANSI/INCITS, 2012.

[33] J. Joshi, E. Bertino, U. Latif et A. Ghafoor.
« A generalized temporal role-based access control model ».
IEEE Transactions on Knowledge and Data Engineering, 17(1):4–23, 2005.

[34] P. Konopacki.
« *Une approche événementielle pour la description de politiques de contrôle d'accès* ».
Thèse de doctorat, École Doctorale Mathématiques et Sciences et Technologies de l'Information et de la Communication, Université Paris-Est Créteil Val-de-Marne, Créteil, France, mai 2012.

[35] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells et B. Zhao.
« OceanStore : An architecture for global-scale persistent storage ».
Special Interest Group on Programming Languages Notices, 35(11):190–201,

2000.

[36] B. W. Lampson.
« Protection ».
Dans *5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443. Department of Electrical Engineering, Princeton University, Princeton New Jersey, 1971.

[37] L. Mao, S. Yao, K. Zhang et K. Sakurai.
« Design and implementation of document access control model based on role and security policy ».
Dans L. Chen et M. Yung, éditeurs, *Proceedings of the 2nd International Conference on Trusted Systems (INTRUST'10)*, volume 6802 de *Lecture Notes in Computer Science*, pages 26–36. Springer-Verlag, 2011.

[38] J. Milhau, M. Frappier, F. Gervais et R. Laleau.
« Systematic Translation Rules from ASTD to Event-B ».
Dans D. Méry et S. Merz, éditeurs, *Integrated Formal Methods*, volume 6396 de *Lecture Notes in Computer Science*, pages 245–259. Springer Berlin / Heidelberg, 2010.

[39] OASIS Access Control Technical Commitee.
« *eXtensible Access Control Markup Language (XACML) Version 2.0* ».
OASIS, février 2005.

[40] OASIS Web Services Business Process Execution Language (WSBPEL) Technical Commitee.
« *Web Services Business Process Execution Language Version 2.0* ».
OASIS, avril 2007.

[41] Object Management Group.
« *OMG Unified Modeling Language™ (OMG UML), Infrastructure* ».
Object Management Group, août 2011.

[42] Object Management Group.
« *OMG Unified Modeling Language™ (OMG UML), Superstructure* ».
Object Management Group, août 2011.

[43] C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas et A. H. M. ter Hofstede.
« Formal semantics and analysis of control flow in WS-BPEL ».
Science of Computer Programming, 67(2–3):161–198, 2007.

[44] G. Salaün, L. Bordeaux et M. Schaerf.
« Describing and reasoning on Web services using process algebra ».
Dans *Proceedings of the 11th IEEE International Conference on Web Services (ICWS'04)*, pages 43–50. IEEE Computer Society, 2004.

[45] D. C. Schmidt.
« Guest editor's introduction : Model-driven engineering ».
IEEE Computer, 39(2):25–31, 2006.

[46] B. Schneier.
Applied Cryptography, 2nd Edition.
John Wiley & Sons, 1996.

[47] T. A. Sudkamp.
Languages and Machines : An Introduction to the Theory of Computer Science, 3rd Edition.
Addison-Wesley, 2005.

[48] E. Tryggeseth.
« *Support for Understanding in Software Maintenance* ».
Thèse de doctorat, Department of Computer and Information Science, Faculty of Physics, Informatics and Mathematics, Norwegian University of Science and Technology, Trondheim, Norway, février 1997.

[49] F. van Breugel et M. Koshkina.
« Models and Verification of BPEL ».
Rapport Technique, Department of Computer Science and Engineering, York University, Toronto, Ontario, Canada, septembre 2006.

[50] J. Wainer, P. Barthelmess et A. Kumar.
« W-RBAC — A workflow security model incorporating controlled overriding of constraints ».
International Journal of Cooperative Information Systems, 12(4):455–485, 2003.

[51] L. Willcocks.
« Evaluating information technology investments : research findings and reappraisal ».
Journal of Information Systems, 2(4):243–268, 1992.

[52] A. Wombacher, P. Fankhauser et E. Neuhold.
« Transforming BPEL into annotated deterministic finite state automata for service discovery ».
Dans *Proceedings of the 11th IEEE International Conference on Web Services (ICWS'04)*, pages 316–323. IEEE Computer Society, 2004.

[53] Y. Yang, Q. Tan, Y. Xiao, J. Yu et F. Liu.
« Verifying Web services composition : A transformation-based approach ».
Dans *Proceedings of the 6th International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'05)*, pages 546–548. IEEE Computer Society, 2005.

[54] X. Yi et K. J. Kochut.
« A CP-nets-based design and verification framework for Web services composition ».
Dans *Proceedings of the 11th IEEE International Conference on Web Services (ICWS'04)*, pages 756–760. IEEE Computer Society, 2004.