Politique de contrôle de flux d'information définie par les utilisateurs pour les orchestrations de services. Mise en oeuvre dans un orchestrateur BPEL

Thomas Demongeot

► To cite this version:

Thomas Demongeot. Politique de contrôle de flux d'information définie par les utilisateurs pour les orchestrations de services. Mise en oeuvre dans un orchestrateur BPEL. Web. Télécom Bretagne, Université de Rennes 1, 2013. Français. NNT : . tel-00959447

HAL Id: tel-00959447
https://theses.hal.science/tel-00959447

Submitted on 14 Mar 2014

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

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

N° d'ordre : 2013telb0290

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En habilitation conjointe avec l'Université de Rennes 1

Ecole Doctorale – MATISSE

Politique de contrôle de flux d'information définie par les utilisateurs pour les orchestrations de services. Mise en œuvre dans un orchestrateur BPEL.

Thèse de Doctorat

Mention : Informatique

Présentée par **Thomas Demongeot**

Département : Réseau Sécurité Multimédia

Laboratoire : SERVAL

Directeur de thèse : Yves Le Traon

Soutenue le 19 décembre 2013

Jury :
Mme Maryline Laurent, Professeur, Telecom Sud Paris (Rapporteur)
M. Mathieu Jaume, HDR, LIP6 (Rapporteur)
M. Jean-Marie Bonnin, Professeur, Telecom Bretagne (Examinateur)
M. Olivier Heen, Technicolor (Examinateur)
M. Benoit Martin, DGA Maitrise de l'Information (Examinateur)
M. Yves Le Traon, Professeur, Université du Luxembourg (Directeur de thèse)
M. Eric Totel, Professeur, Supelec (Co-directeur de thèse)
Mme Valérie Viet Triem Tong, Professeur Assistant, Supelec (Co-directeur de thèse)

Résumé

Les *Web Services* sont aujourd'hui à la base de nombreuses applications de commerce électronique. Aujourd'hui les solutions de sécurité qui sont proposées pour les *Web-Services* permettent de protéger en confidentialité et en intégrité les communications entre les *Web-Services* et de contrôler l'accès aux *Web-Services*. Cependant, le plus souvent, les clients utilisent ces applications sans en connaître le fonctionnement interne. De ce fait, ils n'ont aucune connaissance sur l'usage de leurs données qui est effectué à l'intérieur d'une application.

BPEL (*Business Process Execution Language*) est un langage de programmation permettant d'orchestrer des *Web-Services* au sein d'une architecture orientée services. Une des fonctionnalités des architectures orientées services est la découverte dynamique des services effectivement utilisés au cours de l'exécution d'une orchestration. De ce fait, l'utilisateur d'un programme BPEL ne peut connaitre avant l'exécution comment et par qui les données qu'il va fournir seront utilisées.

Dans le cadre de cette thèse, nous proposons un modèle de politique de contrôle de flux d'information adapté aux architectures orientées services et en particulier aux orchestrations de services. Ce modèle offre aux utilisateurs la possibilité de spécifier des contraintes sur l'usage qui est fait de leurs données personnelles. Les politiques de contrôle de flux d'information définies suivant ce modèle sont définies au cours de l'exécution par l'utilisateur d'une orchestration de services. Les aspects dynamiques des *Web-Services* conduisent, en effet, à des situations où la politique peut empêcher l'exécution nominale de l'orchestration (par exemple lorsqu'il est fait usage d'un service que l'utilisateur ne connaissait pas avant l'exécution). A cet effet, nous proposons à l'utilisateur d'autoriser dynamiquement des modifications de la politique de contrôle de flux d'information. Afin de lui permettre d'effectuer cette modification, nous proposons dans le cadre de cette thèse un mécanisme permettant à l'utilisateur de disposer d'un maximum d'informations sur l'usage qui est fait de ses données et qui conduit à une violation de la politique de contrôle de flux d'information.

Nous avons ensuite appliqué cette approche au langage d'orchestration BPEL. Nous avons en particulier présenté les principaux flux d'information qui interviennent dans ce langage. Nous avons enfin proposé une implémentation de ces mécanismes de contrôle de flux d'information en modifiant un orchestrateur BPEL existant.

Abstract

Web Services are currently the base of a lot a e-commerce applications. Today's security solutions proposed for Web services help to protect communications confidentiality and integrity between Web services and to control access to Web services. Nevertheless, clients often use these services without knowing anything about their internals. Moreover, they have no clue about the use of their personal data inside the global application.

BPEL (Business Process Execution Language) is a programming language orchestrating Web Services within Service-Oriented Architecture (SOA). As one feature of SOAs is the dynamic discovery of services actually used during execution, a BPEL user does not know prior to the execution how, and by whom, the data he provides will be used.

In this thesis we propose a model of political control information flow suitable for service-oriented architectures and in particular orchestrations of services. This model offers the opportunity to the user to specify constraints on the use of its personal data. This policy is configured at runtime by the user of the BPEL program. However, the dynamic aspects of the web services lead to situations in which the policy prohibits the nominal operation of the orchestration (e.g., when using a service that is unknown by the user). To solve this problem, we suggest the user to dynamically allow exceptional unauthorized flows. In order to make hi decision, the user is provided with all information necessary for decision-making.

We then applied this approach to BPEL orchestration language. In particular, we presented the main information flow involved in this language. Finally, we proposed an implementation of these information flow control mechanisms by modifying an existing BPEL orchestrator.

Remerciements

En arrivant à la fin d'une thèse, on se rend compte qu'il y a de très nombreuses personnes qui nous ont accompagnées au cours de cette démarche et sans qui ce manuscrit ne serait arrivé à son terme. Je souhaiterai pouvoir toutes les remercier, malheureusement je suis sûr d'en oublier...

Tout d'abord un grand merci (immense devrais-je dire) à Eric et Valérie. Ils se sont démenés, n'ont pas ménagé leur temps pour m'accompagner dans cet apprentissage de la recherche. Même quand ils n'y croyaient plus, ils n'ont cessés de m'accompagner, de relire, de corriger, de proposer,... Bref un grand merci à vous.

Je tiens également à remercier Yves qui a accepté de diriger cette thèse. Il m'a accompagné lors de la première année. Malgré son départ au Luxembourg, il a quand même continué de suivre mes travaux. Merci.

Un grand merci à l'ensemble du jury qui a accepté d'évaluer cette thèse. En particulier je voudrais citer les rapporteurs Mathieu et Maryline qui ont patiemment relu l'ensemble du manuscrit. Un grand merci pour vos remarques qui ont permis d'améliorer ce manuscrit. Et un grand merci pour vos encouragements.

Se lancer dans une thèse, c'est également grâce au soutien et à l'accompagnement de ceux qui nous ont introduit dans le monde de la recherche. Merci Julien de m'avoir introduit au monde des politiques de contrôle de flux d'information et à celui de la recherche. Ingénieur militaire, je n'étais pas destiné à préparer une thèse. Néanmoins en sortie d'école d'ingénieur, la DGA m'a proposé de prolonger mon cursus. Merci à Daniel Menez de m'avoir proposé ce projet et d'avoir soutenu ma candidature. Merci à André Parriel d'avoir soutenu ce projet. Merci à Yves Correc de m'avoir trouvé une équipe d'encadrement.

Merci à Télécom Bretagne et à Supélec de m'avoir accueilli dans leurs locaux. Merci à Tej et au doctorants (docteurs) pour cette première année de thèse à Télécom Bretagne. Merci à Sylvain d'avoir continué à faire le lien avec Télécom Bretagne. Merci à toute l'équipe Cidre pour m'avoir accueilli les années suivantes.

Merci aux laboratoires TD, ALID, EA, de DGA-Maitrise de l'Information de m'avoir permis de terminer cette thèse. Merci à Laurent, Fred, Nathalie, Constant, Guillaume, Marie-Thérèse, Jamila, Gilles, Thierry, Erwan pour ces deux dernières années.

Enfin merci à Anne-Charlotte et Sixtine de m'avoir rejoint sur cette route du thésard. J'espère que la route du docteur vous sera un peu plus agréable.

Bref merci à tous, en particulier ceux que j'ai oublié de citer.

Table des matières

Introduction **1**

1 Sécurité des architectures orientées services **5**

1.1 Architecture orientée service 5

1.1.1 Généralités 5

1.1.2 Une implémentation des SOA : les *Web Services* 7

1.1.3 BPEL : l'orchestration des *Web Services* 10

1.2 Sécurité des Web-Services 12

1.2.1 Briques standards pour la sécurité des Web-Services . . . 12

1.2.2 Protection des *Web Services* et du contenu des messages . 14

1.3 Bilan 15

2 Politique de sécurité pour les architectures orientées services **17**

2.1 Modèles d'accès discrétionnaire et obligatoire 17

2.1.1 Le modèle d'accès discrétionnaire 17

2.1.2 Le modèle d'accès obligatoire 18

2.1.3 Utilisation de ces modèles dans les architectures orientées services 18

2.2 Modèle RBAC 19

2.3 Modèles de sécurité pour les architectures orientées services . . . 21

2.4 Bilan 22

3 Contrôle de flux d'information **23**

3.1 Suivi des flux d'information 23

3.1.1 Flux d'information 23

3.1.2 Flux d'information au sein des programmes 23

3.1.3 Flux d'information dans un système d'exploitation 24

3.1.4 Flux d'information dans les orchestrations de service . . . 25

3.2 Politiques de flux d'information 26

3.2.1 Politiques en treillis 26

3.2.2 Le modèle décentralisé 31

3.2.3 Le modèle du système Blare 34

3.2.4 Le modèle de la non-interférence 36

3.3 Vérification statique des politiques de flux 37
3.3.1 Principe du contrôle statique de flux d'information 38
3.3.2 Contrôle de flux d'information par typage 38
3.3.3 Application du contrôle de flux par typage 40
3.3.4 Au niveau des orchestrations de service 43
3.4 Suivi dynamique des flux d'information 44
3.4.1 Au niveau des programmes 44
3.4.2 Au niveau des systèmes d'exploitation 49
3.5 Modification de la politique de sécurité 50
3.5.1 Comment sont effectuées les déclassifications 51
3.5.2 Qui peut déclassifier ? 51
3.5.3 Quelles informations peut-on déclassifier ? 52
3.5.4 Où peut-on déclassifier ? 53
3.5.5 Quand peut-on déclassifier ? 54
3.6 Bilan 55

4 Politique de flux d'information pour les orchestrations de service 59
4.1 Modèle du système 59
4.2 Politique de flux d'informations 62
4.3 Premier modèle de suivi de la politique de flux 64
4.3.1 Principe général 64
4.3.2 Labels de sécurité 65
4.3.3 Propagation des labels 66
4.3.4 Respect de la politique de flux par les labels 69
4.3.5 Propagation de la politique de flux entre les services . . . 71
4.3.6 Les limites du modèle 72
4.4 Modification dynamique de la politique de flux d'information . . . 73
4.4.1 Illustration des problèmes 73
4.4.2 Proposition d'une solution 76
4.5 Flux d'informations et modification des labels 82
4.5.1 Définition des labels de suivi des flux d'informations . . . 82
4.5.2 Initialisation et modification des labels de suivi des flux d'informations 85
4.5.3 Lien avec le modèle de labels proposé par Myers 89
4.5.4 Modification de la politique à l'aide d'un service externe . 95
4.6 Bilan 98

5 Suivi des flux d'information dans le langage BPEL 99
5.1 Introduction 99
5.1.1 Flux d'information dans le langage BPEL 99
5.1.2 Initialisation de la politique de flux d'informations 100
5.1.3 Labels des variables de BPEL 102
5.2 Communication avec les services 103
5.2.1 Reception des messages 105

5.2.2 Envoi des messages 105
5.2.3 Appel de services 107
5.3 Flux d'informations explicites 109
5.3.1 Les affectations 109
5.3.2 Les requêtes et expressions XPath 112
5.4 Flux d'information implicites 113
5.4.1 Les conditions 113
5.4.2 Les boucles 115
5.5 Bilan 116

6 OrchestraFlow, un moniteur permettant le suivi des flux d'information dans un programme BPEL 117
6.1 Introduction 117
6.2 Orchestra, un interpréteur BPEL 118
6.2.1 Choix d'Orchestra 118
6.2.2 Exécution des programmes BPEL dans Orchestra 119
6.3 Implémentation d'OrchestraFlow 122
6.3.1 De Orchestra à OrchestraFlow 122
6.3.2 Association de la politique de flux aux conteneurs 124
6.3.3 Propagation de la politique de flux d'information 126
6.3.4 Communication avec les services 129
6.3.5 Performances d'OrchestraFlow 133
6.4 Bilan 133

Conclusion 135
Bilan 135
Perspectives 136

Table des figures

1.1 Composants d'une architecture orientée services 6
1.2 Chorégraphie de services 6
1.3 Orchestration de services 6
1.4 Composition dynamique de services 7
1.5 Structure d'un message SOAP [CDK$^+$02] 8
1.6 Les principaux éléments d'un fichier WSDL 9
1.7 Structure d'un processus BPEL 10
1.8 Les briques de sécurité standards des *Web Services* (d'après [SWS07]) 12

3.1 Exemples de politiques de flux d'information 28
3.2 Grammaire du langage « While » 39
3.3 Système de type sécurisé 39
3.4 Exemple d'utilisation de label dynamique dans JIF 41
3.5 Classes de solutions pour le suivi dynamique des flux d'informations 47

4.1 Composants d'une architecture orientée services 60
4.2 Service de vente en ligne 61
4.3 Une politique de flux d'informations pour l'exemple 1 64
4.4 Grammaire des labels 66
4.5 Propagation d'une politique de flux entre des services 72
4.6 Extrait d'une orchestration non conforme à la politique de flux . . 74
4.7 Extrait d'une orchestration appelant dynamiquement un service de paiement 75
4.8 Extrait d'une orchestration présentant un flux implicite d'informations 81
4.9 Grammaire des labels 83
4.10 Modification de la politique à l'aide d'un service externe 96

5.1 Extrait de programme BPEL additionnant les variables x et y dans z 100
5.2 Exemple d'un type entier XML labélisé 101
5.3 Exemple d'un message XML labélisé 102
5.4 Exemple d'un schema XML de message 102
5.5 Structure d'une déclaration de variables 103
5.6 Exemple d'une variable BPEL et de l'arbre des labels associés . . 104

5.7 Structure d'un élément wsa:EndpointReference 104
5.8 Structure d'une activité <receive> 105
5.9 Variable créée à partir du message de la figure 5.3 et son arbre des labels associé 106
5.10 Structure d'une activité <reply> 106
5.11 Structure d'une activité <invoke> 107
5.12 Communication avec les services 109
5.13 Syntaxe des affectations 109
5.14 Formes possibles d'une source <from> 111
5.15 Formes possibles d'une destination <to> 111
5.16 Syntaxe de la fonction getVariableData 112
5.17 Syntaxe de l'activité <if> 113
5.18 Exemple d'une activité <if> produisant un flux implicite d'information 114
5.19 Syntaxe de l'activité <while> 115
5.20 Extrait d'une boucle <while> contenant un flux implicite d'information 115

6.1 Extrait de la définition de la classe BpelProcess 119
6.2 Extrait de la définition de la classe Scope 119
6.3 Syntaxe de l'activité <if> et diagramme de classe correspondant 120
6.4 Extrait de la définition de la classe Variable 121
6.5 Principales classes du package org.ow2.orchestra.runtime 121
6.6 Diagramme simplifié de la classe Label et des classes associées . 124
6.7 Association d'un arbre des labels à une variable instanciée : diagramme de classe 125
6.8 Méthode execute de la classe If modifiée afin de prendre en compte les flux implicites d'informations 127
6.9 Méthode execute de la classe While modifiée afin de prendre en compte les flux implicites d'informations 128
6.10 Méthode execute de la classe Copy modifiée afin de propager la politique de flux d'information 129
6.11 Interface graphique permettant la modification de la politique de flux 131
6.12 Extrait du fichier WSDL du service de modification de la politique de flux 132

Liste des tableaux

1.1 Activités basiques de BPEL (d'après [RVC07]) 11
1.2 Activités structurées de BPEL (d'après [RVC07]) 11

4.1 Opérations sur les labels 84

5.1 Les commandes principales du langage BPEL 100

6.1 Principales modifications apportées à Orchestra par OrchestraFlow 122
6.2 Surcoût d'exécution 133

Introduction

L'architecture orientée service [LZV08, PH07, RVC07] propose un modèle d'architecture distribuée basé sur la notion de service. Un service est un ensemble de fonctions qui reçoivent des messages et qui restituent le résultat de leur traitement. L'architecture orientée service est bâtie autours de trois composants : un annuaire des services qui liste l'ensemble des services disponibles, les fournisseurs de services, et les consommateurs de services. Une implémentation de ce concept se retrouve dans les *Web Services* qui sont articulés autour du langage pivot XML.

Les *Web Services* [Cer02] ont été au départ conçus, selon les principes du réseau Internet, comme un ensemble de services réutilisables, librement mis à la disposition de chacun. Le développement des *Web-Services* a permis la création de nouveaux services utilisant des services différents pour fournir un service complet. Nous pouvons prendre comme exemple un service complet de vente de livres en ligne qui utilisera plusieurs services différents, comme celui d'une librairie, d'une banque fournissant un service de paiement et d'un livreur permettant de délivrer le produit au client. Les accès à ces multiples services sont orchestrés dans un programme qui peut en particulier être écrit dans un langage dédié nommé BPEL (*Business Process Execution Language*).

Le langage BPEL [BPE07] est un langage relativement simple qui permet de décrire dans quel ordre les appels aux différents services nécessaires au bon fonctionnement du programme sont réalisés. Le programme BPEL peut recevoir des informations des utilisateurs, et utiliser ces données pour fournir des informations aux services qu'il invoque. Par conséquent, le programme BPEL réalise des flux d'information depuis les données de l'utilisateur vers les services qu'il utilise. Le problème est de savoir si ces flux d'information sont légaux du point de vue de l'utilisateur. Au moins deux problèmes peuvent être distingués :

– L'utilisateur n'a pas forcément confiance dans le code BPEL qu'il invoque. Il y a deux raisons. Tout d'abord, le programme peut très bien envoyer des données vers des services en lesquels l'utilisateur n'a pas confiance. Ensuite, il peut envoyer des données prévues pour un type de service vers un autre type de service, mais qui ne devrait pas recevoir ces informations. Ainsi, pour reprendre l'exemple du service d'achat en ligne d'un livre, l'utilisateur pourrait vouloir s'assurer que ses coordonnées bancaires ne sont fournies qu'à un service bancaire, et pas à la librairie en ligne ou au service de livraison. Dans la pratique, rien ne certifie qu'un *Web Service* écrit en BPEL

répondra aux besoins de l'utilisateur en terme de confidentialité, les données que celui-ci a fourni pouvant être utilisées à son insu, sans qu'il en soit informé.
- Les fonctions invoquées par le programme BPEL peuvent être découvertes dynamiquement à l'exécution : par exemple, le service de paiement par carte bancaire peut être celui de n'importe quelle banque en ligne. Cela signifie qu'il est impossible, lors de l'écriture du programme BPEL, de savoir vers quels services certains flux d'information seront réalisés. Or, l'utilisateur n'a peut-être pas confiance dans tous les services bancaires disponibles. Il peut donc vouloir restreindre la diffusion de ses coordonnées bancaires à seulement quelques services de paiement qu'il connaît et en lesquels il place sa confiance.

Les solutions actuelles en terme de sécurité dans les *Web Services* consistent à mettre en place des mécanismes de contrôle d'accès qui vérifient si un service ou un utilisateur a le droit d'invoquer un *Web Service*. Ces solutions sont insuffisantes en ce qui concerne la confidentialité des données utilisateurs, comme nous venons de le décrire dans l'exemple précédent (les services ont bien le droit d'être invoqués, mais ne doivent pas utiliser certaines données fournies par l'utilisateur). Afin de vérifier dynamiquement l'accès aux données de l'utilisateur par les *Web Services* invoqués, nous proposons dans le cadre de cette thèse de mettre en place des mécanismes de contrôle de flux d'information au niveau de l'interpréteur BPEL.

Afin d'assurer la confidentialité des données envoyées par un utilisateur vers une orchestration de services nous défendons dans cette thèse les positions suivantes :
- la politique de contrôle de flux d'information liée aux données qu'il transmet à un *Web-Service* doit être définie par l'utilisateur. En effet seul l'utilisateur d'un service est légitime pour le choix d'une politique de confidentialité des données qu'il transmet à une orchestration de services ;
- la politique de contrôle de flux d'information doit pouvoir être suivie entre les différents services. Ainsi une donnée envoyée par un service A vers un service B ne doit pouvoir être envoyée par le service B à un service C si le service C n'est pas autorisé par la politique de confidentialité ;
- la politique de contrôle de flux d'information doit pouvoir être modifiée au cours de l'exécution d'une orchestration de services. En effet l'ensemble des services utilisés n'est pas forcément connu avant l'exécution d'une orchestration de services. De plus la définition d'une politique de contrôle de flux d'information peut-être une démarche compliquée pour l'utilisateur, il est donc plus facile pour lui de la modifier au fur et à mesure de l'exécution de l'orchestration de services ;
- les modifications dynamiques de la politique de contrôle de flux d'information ne doivent pas produire de fuites d'information non portées à la connaissance de l'utilisateur. Il est donc nécessaire de fournir à l'utilisateur les informations suffisantes lui permettant d'autoriser ou interdire une modification de la politique de contrôle de flux d'information ;

– l'application de la politique de contrôle de flux d'information doit-être indépendante des services et en particulier des orchestrations de services. Elle doit en effet pouvoir être garantie indépendamment de l'orchestration de services initiale et ne doit pas nécessiter la réécriture de l'orchestration de services. C'est pourquoi nous proposons de localiser les mécanismes de contrôle de flux d'information au niveau des serveurs de services et plus particulièrement des orchestrateurs de services.

Pour atteindre ces objectifs, nous proposons la démarche suivante :

– nous définissons un modèle de politique de contrôle de flux d'information qui permet d'apporter une propriété de confidentialité à des données fournies par des utilisateurs à un service ou à une orchestration de services. Ce modèle est dérivé du modèle proposé par Myers et Liskov [ML97] et décrit au chapitre 3 ;
– nous enrichissons ce modèle de politique de contrôle de flux d'information afin de permettre une modification dynamique de la politique de contrôle de flux d'information. Ce nouveau modèle doit permettre à l'utilisateur de connaître les informations utilisées pour produire les données pour lesquelles une modification dynamique de la politique de flux est nécessaire ;
– nous étudions ensuite les flux d'information produits par le langage BPEL, un langage permettant les orchestrations de services. Cette étude nous permet de préparer l'implémentation d'un mécanisme de contrôle de flux d'information permettant de mettre en œuvre notre modèle de politique de contrôle de flux d'information ;
– nous étudions enfin l'implémentation des mécanismes de contrôle de flux d'information dans un orchestrateur BPEL.

Le mémoire est organisé de la façon suivante : les trois premiers chapitres présentent un état de l'art de la sécurité des architectures orientées services (chapitres 1 et 2) ainsi que des politiques de contrôle de flux d'information (chapitre 3). Les trois chapitres suivants proposent la mise en œuvre d'une politique de contrôle de flux d'information définie par les utilisateurs pour les orchestrations de services. Le chapitre 4 décrit ainsi un modèle de politique de contrôle de flux d'information pour des orchestrations de services. Le chapitre 5 s'intéresse à l'étude des flux d'information dans le langage BPEL. Enfin, avant de conclure, le chapitre 6 propose la mise en œuvre des mécanismes décrits dans les deux précédents chapitres dans un orchestrateur BPEL.

Chapitre 1

Sécurité des architectures orientées services

1.1 Architecture orientée service

1.1.1 Généralités

Dans une architecture orientée service (*Service Oriented Architecture* (SOA)) [LZV08, PH07, RVC07] les ressources logicielles sont organisées sous la forme de services. Les services sont des modules indépendants les uns des autres vis-à-vis de leur état ou de leur contexte d'exécution. Ils fournissent un ensemble de fonctions qui reçoivent des messages et restituent le résultat de leur traitement.

Une architecture orientée service est bâtie autour de trois composants principaux :

- un **fournisseur de services**, qui produit, maintient et propose un certain nombre de services ;
- un **consommateur de services**, soit un humain, soit un logiciel, qui utilise les services proposés par le fournisseur ;
- un **annuaire de services** qui permet de rechercher les services disponibles.

Ainsi le fournisseur de services publie ses services dans l'annuaire. Le consommateur de services peut ainsi rechercher les services disponibles dans l'annuaire puis se connecter au fournisseur de services afin d'utiliser le service recherché. La figure 1.1 reprend l'ensemble des composants d'une architecture orientée services ainsi que leurs interactions.

Dans le cadre des architectures orientées services dites statiques le consommateur recherche à la main dans l'annuaire les services qu'il va utiliser et les organise de manière statique afin de créer, par exemple, de nouveaux services. Dans ce cas l'annuaire n'est utilisé que dans la phase de développement.

Il existe deux manières différentes de permettre les interactions entre les services. Dans le cadre d'une chorégraphie les services connaissent les services avec lesquels ils interagissent afin de réaliser une tâche définie. Le schéma de la figure

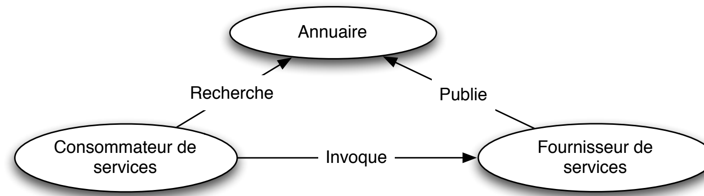

FIGURE 1.1 – Composants d'une architecture orientée services

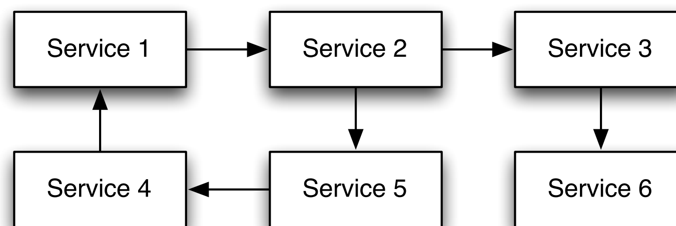

FIGURE 1.2 – Chorégraphie de services

1.2 présente une chorégraphie de services. Nous pouvons voir qu'il n'y a pas de service contrôlant l'ensemble de la chorégraphie.

A l'opposé l'orchestration suppose la présence d'un service central qui fait appel à différents services indépendants afin de réaliser sa tâche. Ce service central joue alors le rôle d'un chef d'orchestre. Le schéma de la figure 1.3 présente une orchestration.

Les architectures orientées services dites dynamiques reposent sur les orchestration et les chorégraphies. En effet, dans ce cas, les services qui interagissent entre eux ne sont pas liés physiquement. Lorsqu'un client appelle un service, le service appelé recherche dans l'annuaire les services dont il a besoin afin de s'exécuter.

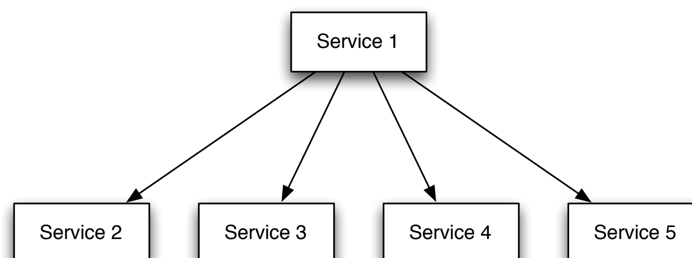

FIGURE 1.3 – Orchestration de services

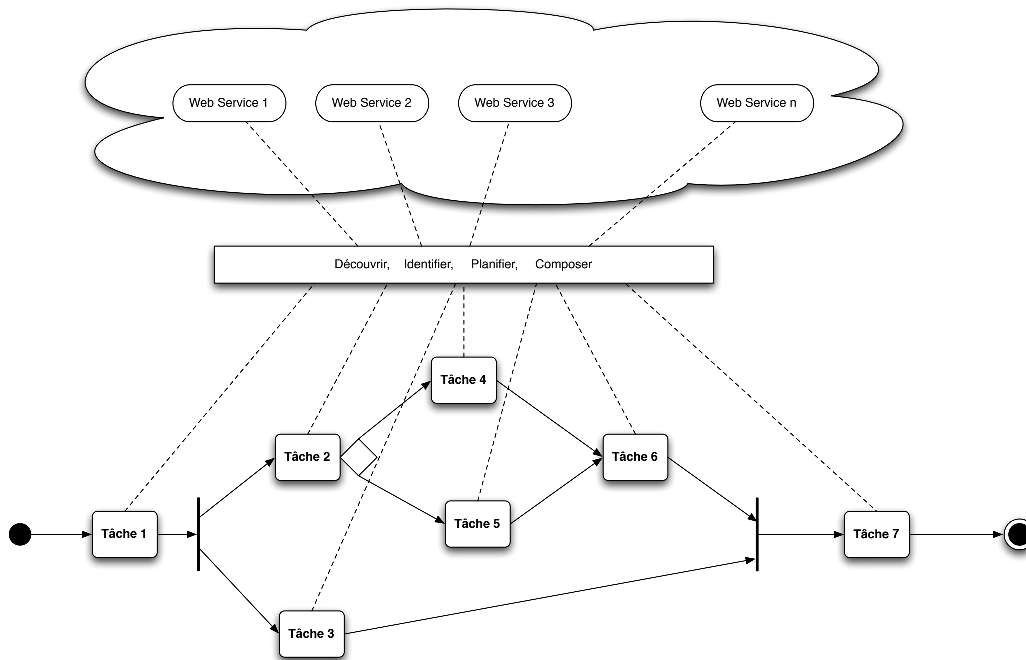

FIGURE 1.4 – Composition dynamique de services

Ainsi si un service n'est pas disponible il peut-être remplacé dynamiquement par un autre service. La figure 1.4 présente le fonctionnement d'un service dynamique qui recherche et compose dynamiquement les services nécessaires dans le cadre d'une orchestration.

Dans la pratique, selon [Jos07],

> une des promesses des SOA est que les applications seront conçues par des experts des processus métiers, plutôt que par des experts des nouvelles technologies et que leur tâche va être facilitée car ils vont assembler des applications à partir de services existants stables rendus disponibles dans un annuaire

.

1.1.2 Une implémentation des SOA : les *Web Services*

Les *Web Services* [CDK+02, Ma05, Lou06, YLBM08] sont une implémentation des architectures orientées services basée sur le langage XML. Ils sont basés sur trois piliers principaux :

- SOAP pour le transport des messages,
- WSDL pour la description des interfaces de service,
- UDDI pour la création des annuaires de services.

```
<SOAP:Envelope xmlns:SOAP=le
     'http://schemas.xmlsoap.org/soap/envelope/'>
 <SOAP:Header>
  <!-- content of header goes here -->
 </SOAP:Header>
 <SOAP:Body>
  <!-- content of body goes here -->
 </SOAP:Body>
</SOAP:Envelope>
```

FIGURE 1.5 – Structure d'un message SOAP [CDK+02]

Une des forces des *Web Services* repose sur l'utilisation des standards de l'Internet. En effet si SOAP est un langage basé sur XML, le protocole de transport utilisé pour véhiculer les messages est le plus souvent HTTP. Ainsi les *Web Services* ne définissent pas un réseau parallèle aux réseaux existants, mais s'appuient directement sur les infrastructures existantes.

Echange de messages

SOAP [1] [SOA07] est un protocole d'échange de messages basé sur XML. A l'origine SOAP a été créé afin de faire de l'appel de procédures distantes depuis HTTP. Cependant SOAP ne définit pas un nouveau protocole de transport, il s'appuie sur les protocoles de transport existants. SOAP est aujourd'hui disponible sur HTTP, SMTP, ... La norme ne définit pas seulement la structure des messages mais aussi les différentes manières de l'employer.

Un message SOAP est composé de 2 parties principales : un en-tête et un corps contenus dans une enveloppe globale (Figure 1.5). L'en-tête optionnel permet de rajouter des informations complémentaires relatives notamment à la sécurité. Le corps du message contient quant à lui l'ensemble des données utiles au service. Le message SOAP est au format XML. La seule contrainte est que la syntaxe du message doit-être valide. Le contenu du message n'est pas limité à des données XML. En effet tout type de contenu (en particulier des données binaires) peut-être inséré au sein d'un message SOAP.

Description des interfaces

WSDL (*Web Service Description Language*) [WSD01] est une norme basée sur XML qui permet de décrire l'interface qui permet d'accéder aux *Web Services*. WSDL peut-être vu comme un contrat qui permet de spécifier comment accéder au service. Un *Web Service* est décrit dans un fichier WSDL comme une collection

1. Autrefois appelé *Simple Object Access Protocol*, depuis la version 1.2 SOAP n'est plus un acronyme

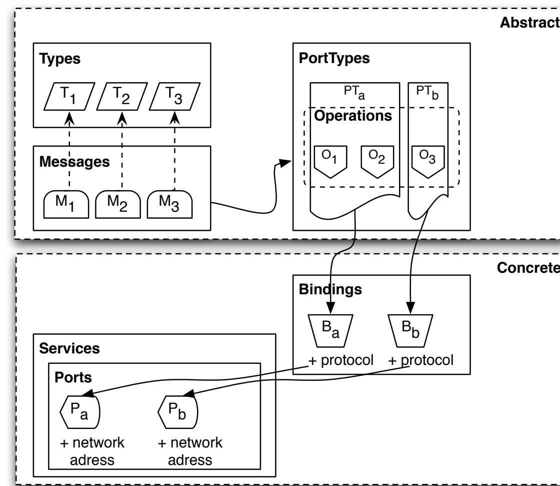

FIGURE 1.6 – Les principaux éléments d'un fichier WSDL

de ports de communication avec lesquels on peut échanger certains messages. La figure 1.6 [2] présente les principaux éléments d'un fichier WSDL.

Un fichier WSDL est décomposé en deux parties :

- une partie abstraite qui décrit l'interface fonctionnelle, c'est à dire les opérations supportées et les messages à utiliser afin de communiquer avec le service, elle est est composée des parties suivantes :
  - *Types*, qui permet de définir les types de données utilisés (en particulier en utilisant des schéma XSD),
  - *Messages*, qui permet de définir de manière abstraite les données échangées lors de la communication,
  - *Operation*, qui permet de définir de manière abstraite les actions offertes par le service ; ces opérations sont regroupées dans un *PortType* qui est supporté par un ou plusieurs point de connexion (*endpoint*) ;
- une partie concrète qui décrit les détails des protocoles à utiliser pour accéder au service, ainsi que l'adresse physique du service. Cette partie contient en particulier les éléments suivants :
  - *Binding*, un protocole concret pour un `PortType` particulier ;
  - *Port*, un point de connexion défini comme une combinaison entre un *binding* et une adresse réseau ;
  - *Service*, un ensemble de ports.

Annuaire des services

Il existe aujourd'hui deux spécifications principales permettant de réaliser des annuaires de services, UDDI proposé par l'OASIS et ebXML développé à l'initia-

2. D'après un schema d'Eric Le Merdy (Wikipedia)

```
<process>
 <partnerLinks> ? .. </partnerLinks>
 <partners> ? .. </partners>
 <variables> ? .. <variables>
 <correlationSets> ? .. </correlationSets>
 <faultHandlers> ? .. activity .. </faultHandlers>
 <compensationHandler> ? .. activity .. </
    compensationHandler>
 <eventHandlers> ? .. activity .. </eventHandlers>
        activity
</process>
```

FIGURE 1.7 – Structure d'un processus BPEL

tive de l'OASIS et de l'UN/CEFACT.

UDDI (*Universal Description Discovery and Integration*) [UDD04] est un annuaire de services basé sur XML. Celui-ci est utilisable comme un service Web. L'ensemble des communications avec un annuaire UDDI sont basées sur SOAP. Un des gros problèmes d'UDDI vient du fait qu'à l'origine il était destiné à mettre en place un annuaire public universel des services. De ce fait les problématiques de sécurité n'ont été prises en compte qu'à partir de la version 3.

ebXML (*Electronic Business XML*) est une structure qui fournit un ensemble de technologies, dont en particulier un annuaire, similaire à UDDI, mais qui offre des fonctionnalités additionnelles en particulier à destination des processus métiers.

1.1.3 BPEL : l'orchestration des *Web Services*

BPEL (*Business Process Execution Language*) [BPE07] est un langage permettant de décrire une orchestration de *Web Services*. Il est issu des langages WSFL (*Web Services Flow Language*) d'IBM et XLang de Microsoft. Il permet de décrire des processus métiers en XML. Un processus BPEL est ensuite exécuté à travers un *Web Service*. Une présentation plus complète de l'utilisation de BPEL dans le développement d'applications web peut-être trouvé dans [Pas05].

Un processus BPEL est composé de partenaires qui interagissent avec le processus à travers des *partner link*. Le processus global résultant est décrit par un fichier WSDL à l'instar de n'importe quel Web Service. La Figure 1.7 présente les principaux éléments de l'élément <*process*> d'un processus BPEL.

- L'élément < *partnerLink* > spécifie les différents acteurs interagissant avec le processus.
- La section < *partners* > est optionnelle. Elle permet de spécifier les capacités requises par les différents partenaires, par exemple qu'un service doit être fourni par ce partenaire.
- La section < *variables* > permet de spécifier le type des variables utilisées par le processus (par exemple message WSDL, types XML simples,

Activité basique	Utilisation
`assign`	copie de variables
`compensate`	compensation
`empty`	activité vide
`invoke`	appel d'un service web
`receive`	réception d'un message entrant
`reply`	réponse à un message entrant
`terminate`	demande explicite d'arrêt du processus
`throw`	déclenchement d'une exception
`wait`	attente pour une période donnée

TABLE 1.1 – Activités basiques de BPEL (d'après [RVC07])

Activité structurée	Utilisation
`flow`	organisation concurrente d'activités
`pick`	attente d'évènements
`scope`	contexte d'exécution
`sequence`	organisation séquentielle d'activités
`switch`	exécution conditionnelle d'activités
`while`	exécution en boucle d'activités

TABLE 1.2 – Activités structurées de BPEL (d'après [RVC07])

ou schéma XML).

- Le processus principal est illustré par le mot clé *activity* qui peut être par exemple :
  - une séquence (< *sequence* >) d'activités ordonnées ;
  - des activités exécutées en parallèle <*flow*> ;
  - une activité < *reply* > envoyant un message en réponse à un message reçu via l'activité < *receive*> ;
  - une activité < *wait* > permettant d'attendre pendant un certain temps ;

Les activités BPEL sont de deux types ; les activités structurées qui contiennent d'autres activités et les activités basiques. Les différentes activités sont présentées dans les tableaux 1.1 et 1.2.

- L'élément < *faultHandlers* > contient le gestionnaire d'erreurs qui définit les actions exécutées lorsqu'une erreur est détectée lors de l'invocation d'un service.
- L'élément < *correlationSets* > permet de lier des activités < *invoke* > et des activités < *receive* >. En effet lorsque plusieurs actions sont lancées en mode parallèle, il peut être indispensable de les relier entre elles.
- L'élément < *compensationHandler* > contient le gestionnaire de compensation qui définit les actions qui doivent être exécutées pour compenser d'autres activités.
- L'élément < *eventHandlers* > contient le gestionnaire d'évènement qui défi-

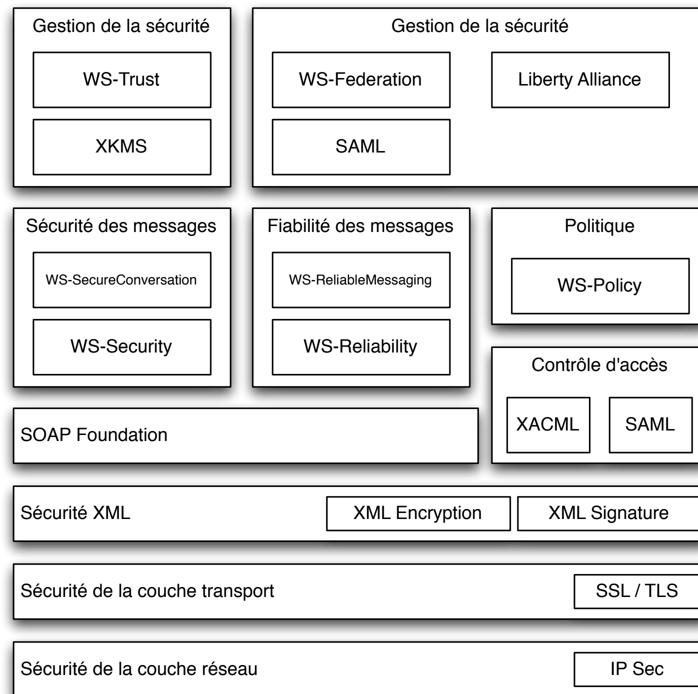

FIGURE 1.8 – Les briques de sécurité standards des *Web Services* (d'après [SWS07])

nit les actions qui doivent être exécutées en fonction de certains évènements en particulier la réception de messages.

1.2 Sécurité des Web-Services

1.2.1 Briques standards pour la sécurité des Web-Services

Les principaux standards permettant de mettre en place des fonctions de sécurité dans les architectures orientées services sont présentés figure 1.8. Une présentation plus complète de ces solutions standards peut-être trouvée dans [KCE+04] et dans [Nor09]. Leur utilisation en pratique est présentée dans [HFBK03] et dans un guide de sécurisation des *Web Services* ([SWS07]).

La sécurité des architectures orientées services repose généralement sur les mécanismes de sécurité de HTTP et en particulier les sessions TLS. Cependant SOAP est un protocole d'échange de messages basé sur XML indépendant du protocole de transport utilisé. C'est pourquoi des mécanismes permettant de mettre en place des sessions sécurisées qui assurent l'authentification des parties, l'intégrité et la confidentialité des messages, ont été développés. Ces standards sont basés sur les

mécanismes de sécurité des documents XML.

La protection des messages est assurée par *WS-Security* [WSS06], un protocole de communication qui permet de sécuriser l'échange entre le fournisseur de services et le consommateur. Il spécifie à cet effet des en-têtes SOAP dans lesquels sont placés les assertions de sécurité. Il permet de mettre en place des mécanismes d'authentification, de protection de la confidentialité et permet d'assurer l'intégrité des messages échangés. Les spécifications permettant d'utiliser des mécanismes cryptographiques au sein de documents XML sont basées sur les technologies existantes, en particulier les infrastructures de clés publiques (X.509, SPKI, PGP). La protection de la confidentialité est basée sur l'utilisation conjointe de XML-Encryption ([XML02]) qui permet le chiffrement de données XML et de jetons de sécurité. L'intégrité des messages repose sur *XML Signature* ([XML08]) et les jetons de sécurité. L'ensemble des mécanismes de sécurité mis en place dans *WS-Security* sont extensibles afin de pouvoir prendre en charge les nouvelles technologies de chiffrement et de signature.

Le principal inconvénient de *WS-Security* réside dans la multiplicité des mécanismes qu'il peut mettre en œuvre rendant ainsi difficile les communications entre deux services utilisant des mécanismes de sécurité différents. A cet effet *WS-Trust* [WST07] a été développé. *WS-Trust* repose sur l'utilisation d'un tiers de confiance et se base sur *WS-Security* afin de spécifier l'échange des jetons de sécurité. *WS-Trust* permet donc d'étendre l'interopérabilité permise par les *Web Services* aux mécanismes de sécurité. L'utilisation conjointe de *WS-Security* et *WS-Trust* a conduit à la spécification de *WS-SecureConversation* [WSC07] qui permet de mettre en place une session sécurisée entre un service et un consommateur de la même façon que TLS.

Afin de décrire les mécanismes de sécurité mis en œuvre par un *Web Service*, la spécification *WS-SecurityPolicy* [WSP07] a été développée. En se basant sur *WS-Policy* [WSP06], un framework permettant de décrire des politiques d'emploi de *Web Service*, elle permet à un *Web Service* de spécifier des renseignements de sécurité et de routage. *WS-SecurityPolicy* offre la possibilité aux développeurs de signer et de chiffrer un message sans code supplémentaire.

Afin de contrôler l'accès aux services, des mécanismes permettant l'authentification et de gérer les autorisations sont indispensables. A cet effet *Security Assertion Markup Language* (SAML) [SAM05] a été développé. C'est un standard OASIS, basé sur XML, qui définit un protocole d'échange d'assertions de sécurité. Ce protocole requête-réponse permet en particulier l'échange d'informations d'authentification et d'autorisation. L'interopérabilité entre les systèmes d'authentification et d'autorisation est assuré par le standard OASIS *eXtensible Access Control Markup Language* (XACML) [XAC05]. Il fournit à la fois un langage de description des politiques de sécurité, mais aussi un langage de requête-réponse qui permet de contrôler l'accès à des ressources Internet. Les règles décrites en XML permettent de définir les droits des utilisateurs, que ce soit des services ou des personnes sur des services ou des données.

Les mécanismes de sécurité proposés dans les *Web Services* se concentrent sur

la façon d'accéder aux services. Ils permettent ainsi de mettre en place une politique de sécurité régulant l'accès aux services via l'utilisation de *WS-Policy*. Ils permettent ensuite d'authentifier les consommateurs (*XACML*) et de gérer leurs droits (*SAML*). Enfin l'utilisation de *WS-Security* permet de protéger la confidentialité et l'intégrité des transactions.

1.2.2 Protection des *Web Services* et du contenu des messages

John Viega et Jeremy Epstein ([VE06]) montrent que l'application des standards de sécurité n'est pas suffisant. L'ensemble de ces mécanismes en ne controlant que l'accès aux services ne permet pas de contrôler les messages échangés. Du fait de ces limitations de nombreuses attaques sont possibles contre les *Web Services* en particulier lorsque n'importe quel utilisateur peut s'enregistrer afin d'utiliser un service. Le lecteur intéressé trouvera une description de différentes attaques possibles contre les *Web Services* dans [VE06, JGHL07, MH06].

Ces attaques peuvent être le fruit de failles dans les standards de sécurité. Ainsi les auteurs de [BFG04] proposent une sémantique formelle de *WS-SecurityPolicy* permettant ainsi de vérifier la cohérence des politiques produites avec cette spécification. Les auteurs de [BFG05] se sont intéressés à la spécification et à la vérification de protocoles cryptographiques pour XML, en particulier ceux utilisés par *WS-Security*. Cependant le plus souvent ces attaques sont dues à une mauvaise utilisation des standards de sécurité. Ainsi les auteurs de [BFGO05] ont construit un outil permettant de rechercher les vulnérabilités dans une architecture de Web-Services dont la sécurité est spécifiée à l'aide de *WS-SecurityPolicy* et de proposer le cas échéant des solutions permettant de palier ces vulnérabilités. Afin de faciliter la mise en place de *WS-Security*, les auteurs de [YCTU07] proposent une API facilitant l'utilisation de *WS-Security* par les programmeurs. Les auteurs de [SY07] proposent quant à eux un *framework* permettant de transformer des politiques *WS-SecurityPolicy* en spécification *WS-Security*. Enfin, les auteurs de [GJD07] proposent une passerelle implémentant les spécifications *WS-Security*, permettant ainsi de découpler les mécanismes de sécurité de l'implémentation des services.

Ainsi de nombreux travaux ont cherché à compléter les standards de sécurité des *Web Services*. Cependant, ces standards n'ont pas été prévus pour fonctionner directement avec les orchestrations de service. C'est pourquoi des travaux ont cherché à étendre les fonctionnalités de sécurité aux orchestrations de service et plus particulièrement aux problèmes de sécurité liés aux orchestrations faisant intervenir plusieurs entités différentes.

Ainsi les auteurs de [CFH05] proposent une approche permettant de concilier les exigences de sécurité des utilisateurs et fournisseurs de service. Ils définissent un langage permettant de spécifier ces exigences, en particulier en matière du choix des algorithmes de chiffrement ou de signature. Ces définitions permettent ensuite de construire des programmes BPEL respectant l'ensemble de ces contraintes. Les auteurs de [FMS07] proposent le même type d'approche, mais en exprimant direc-

tement ces contraintes dans les programmes BPEL à l'aide d'un langage spécifique appelé Secure BPEL. Les auteurs de [BDF06] proposent une approche formelle de ce problème en considérant un λ-calcul enrichi permettant de décrire des orchestrations de services. Une analyse statique permet ensuite de vérifier que les orchestrations répondent à ces exigences.

Ces travaux permettent d'assurer la protection des messages échangés par les *Web Services* en respectant les exigences des différentes parties en matière de confidentialité et d'intégrité. Cependant ces travaux de permettent pas de protéger les données au cours d'une orchestration de services lorsque celle-ci implique plusieurs partenaires. Ainsi par exemple au cours d'une vente en ligne rien ne permet de garantir que les coordonnées bancaires d'un client ne soient transmises qu'à la banque. C'est pourquoi les auteurs de [SP07] et de [JG09] proposent d'encapsuler des flux chiffrés à l'aide de *WS-Security* dans des programmes BPEL. Cette solution basée sur l'emploi de chiffrement symétrique ne permet néanmoins pas de gérer plusieurs destinataires différents pour une information donnée. Afin de permettre de multiples destinataires à une information les auteurs de [MJS11] encodent chaque donnée à protéger à l'aide d'une clé symétrique. Ils utilisent ensuite le chiffrement asymétrique afin de chiffrer cette clé avec la clé publique de chaque destinataire de l'information. La clé de chiffrement est donc dans ce cas envoyée chiffrée avec l'information.

1.3 Bilan

Les travaux présentés permettent de répondre ponctuellement à des problèmes de confidentialité et d'intégrité en se basant sur les spécifications standard des *Web Services*. Ils permettent ainsi de protéger les communications entre différents *Web Services* et de protéger la confidentialité des informations lorsque plusieurs entités sont impliquées. Cependant ces travaux seraient incomplets sans une politique de sécurité dédiée aux problématiques de sécurité des *Web Services*. C'est pourquoi dans le chapitre suivant nous présenterons les modèles classiques de politique de sécurité, leur adaptation aux architectures orientées services, ainsi que les modèles de politique de sécurité développés afin de répondre aux problématiques particulières des SOA.

Chapitre 2

Politique de sécurité pour les architectures orientées services

Nous avons montré dans le chapitre précédent que les standards de sécurité développés pour les Web-Services permettaient de protéger l'intégrité et la confidentialité des messages échangés. Cependant ces travaux ne se sont pas intéressés à la mise en place de politiques de sécurité dédiées aux architectures orientées services.

Dans le cadre de ce chapitre nous nous intéressons à la mise en place du contrôle d'accès dans les architectures orientées services. De nombreux modèles de contrôle d'accès ont été proposés. Les modèles traditionnels de sécurité peuvent être classés en deux grandes familles, les modèles d'accès discrétionnaires (DAC - *Discretionary Access Control*) et les modèles d'accès obligatoires (MAC - *Mandatory Access Control*). La section 2.1 présente l'utilisation de ces modèles dans les architectures orientées services. Plus récemment d'autres modèles plus flexibles ont été définis comme le modèle RBAC (*Role-Based Access Control*). Celui-ci est présenté dans la section 2.2. Enfin, dans la section 2.3, nous présentons des modèles de contrôle d'accès spécifiquement étudiés pour les architectures orientées services.

2.1 Modèles d'accès discrétionnaire et obligatoire

2.1.1 Le modèle d'accès discrétionnaire

Une politique de type discrétionnaire (DAC : *Discretionnary Access Control*) [Lam71] est définie par [Jor87] comme :

> A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are "discretionary" in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other

subject. [1]

Dans le modèle discrétionnaire, on peut définir un propriétaire pour chaque ressource du système et ensuite laisser ce propriétaire décider qui peut accéder à cette ressource, ainsi le contrôle d'accès est à la discrétion de chaque propriétaire. Le principal avantage des politiques de type discrétionnaire réside dans leur flexibilité. Cependant ces politiques ne confèrent pas un haut niveau de sécurité. En effet un politique de type discrétionnaire permet de copier le contenu d'un objet dans un autre et ainsi permettre à un utilisateur d'accéder à une donnée à laquelle il n'était pas sensé avoir accès.

2.1.2 Le modèle d'accès obligatoire

Une politique de type obligatoire (MAC : Mandatory Access Control) ([Bib77] et [BL76]) est défini par le *Trusted Computer System Evaluation Criteria* comme :

> A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity. [2]

Dans le modèle MAC, l'ensemble des accès autorisés est décidé au niveau du système. Ainsi l'utilisateur n'a aucune possibilité de modifier la politique de sécurité.

Les modèles de type MAC permettent de faciliter l'administration des systèmes. Ils permettent la mise en place de règles permettant de contrôler les flux d'information, par exemple en interdisant la lecture d'une information classifiée par un sujet non habilité à y accéder et en interdisant l'écriture à un sujet vers un objet de niveau de classification inférieur à son niveau d'habilitation. Le modèle de Bell et LaPadula définit une telle politique. Ils fournissent donc un haut niveau de sécurité et de confiance mais restent peu flexibles. De plus si le modèle reste simple à mettre en œuvre pour un nombre limité d'objets, il est complexe à mettre en œuvre pour un grand nombre d'objets et d'utilisateurs car l'ensemble de la définition des droits repose sur les seuls administrateurs.

2.1.3 Utilisation de ces modèles dans les architectures orientées services

Le modèle d'accès obligatoire a été mis en œuvre dans les systèmes distribués. En particulier, les auteurs de [KFE98] considèrent une architecture distribuée sé-

1. Un ensemble de restrictions d'accès à des objets basé sur l'identité d'un sujet (ou d'un groupe auquel il appartient). Ce contrôle est discretionnaire dans le sens où un sujet avec une certaine permission d'accès est capable de passer cette permission (parfois indirectement) à n'importe quel autre sujet.

2. Un ensemble de restrictions d'accès à des objets basé sur la classification de l'information (représentée par un label) contenue dans cet objet et sur l'autorisation formelle (c'est à dire l'habilitation) d'un sujet d'accéder à l'objet d'une telle classification.

parée en différents domaines. A chaque domaine est attribué un niveau de classification. Les auteurs étudient en particulier les communications entre les différents domaines. Ils cherchent en particulier à assurer qu'il existe une séparation stricte entre les différents domaines de classification et que le flux d'information entre les différents niveaux est contrôlé. Afin de pouvoir mettre en œuvre ces propriétés dans une architecture distribuée ils définissent trois fonctions principales :

– un service d'activation des serveurs permettant à des entités d'un domaine d'accéder à des serveurs d'autres domaines via la mise en place de proxy ;
– un service de coordination permettant à des domaines utilisant des protocoles différents de communiquer entre eux ;
– un service cryptographique garantissant l'authentification et la confidentialité des messages échangés entre des domaines différents.

Le contrôle des messages échangés entre les domaines de différents niveaux est effectué par un contrôleur placé aux bornes des domaines chargé de vérifier la légalité du flux d'information.

Les auteurs de [KFS$^+$99] et [KFEM00] proposent un modèle théorique s'appuyant sur les travaux de [KFE98]. Ils proposent une méthode permettant de transformer des workflows multi-niveaux en un ensemble de workflows d'un seul niveau de classification. Afin de gérer les interactions entre les niveaux ils s'appuient sur les entités définies dans [KFE98].

A la suite de ces travaux, les auteurs de [RS06] présentent un modèle conceptuel d'architecture multi-niveaux pour les SOA. Ils considèrent deux dimensions de classification : confidentialité et intégrité.

Leur modèle repose sur deux types d'entités : des entités de simple niveau et des entités multi-niveaux, à la manière des auteurs de [ND96]. Les entités de simple niveau ne peuvent manipuler que des données du même niveau que le leur et les données modifiées ou générées par cette entité héritent du niveau de cette entité.

Afin de permettre les échanges de données entre les niveaux, ils proposent le concept d'entités multi-niveaux qui peuvent générer des données de niveau inférieur ou égal à leur niveau en confidentialité ou de niveau supérieur ou égal à leur niveau d'intégrité. Afin de permettre des échanges de données impossibles avec ces entités, ils proposent de mettre en œuvre des entités de confiance chargées de permettre des déclassifications. Cependant ces travaux restent au stade de concept et aucune implémentation de leur modèle n'est proposé.

2.2 Modèle RBAC

Le modèle RBAC [FSG$^+$01] a été conçu pour être plus souple que les modèles présentés précédemment. Il est basé sur trois grandes notions : *utilisateur, rôle, et permission*. Dans ce modèle, chaque utilisateur est associé à un rôle et à chaque rôle est associé un ensemble de permissions. Ce modèle correspond à la structure des organisations et reste très flexible. Il est en particulier bien adapté aux environnements multi-domaines quand les politiques sont exprimées en utilisant la

hierarchie des rôles ainsi que les contraintes.

Dans [Wea04], les auteurs utilisent le modèle RBAC afin de contrôler l'accès à des Web-Services. Cependant l'essentiel de leurs travaux portent sur la mise en place d'une méthode d'authentification. Ils se basent pour cela sur l'utilisation de WS-Policy en y ajoutant une extension permettant de donner un niveau en fonction des algorithmes de chiffrement et de signature utilisés. Ce niveau permet de déterminer un contexte dans lequel un appel de Web Service est effectué. Ainsi, les règles RBAC appliquées diffèrent selon le contexte d'appel d'un Web-Service. Cette méthode permet également de gérer les méthodes d'authentification de manière hiérarchique. Ainsi, une méthode d'authentification plus forte est acceptée par un Web Service requérant une méthode d'authentification plus faible.

Les auteurs de [LC04] présentent un modèle RBAC étendu afin de prendre en compte certaines contraintes des Web-Services. Ils ajoutent aux traditionnels utilisateurs et rôles du modèle RBAC quatre notions spécifiques aux Web Services. Les notions de Web-Service, entreprise, processus métier et session sont ainsi ajoutés dans le modèle étendu. Les Web-Services correspondent aux objets à protéger, à chaque Web-Service est assigné un ou plusieurs rôles autorisés à les utiliser. Un processus métier permet l'appel organisé d'un ensemble de Web-Services. Pour chaque processus métier il est défini un certain nombre de rôles autorisés à appeler ce processus métier. Une session correspond à l'exécution d'un processus métier par un utilisateur. Enfin une entreprise est une organisation qui participe aux processus métiers et qui fournit des Web-Services. Dans ce modèle les contraintes gèrent les relations entre ces différentes notions. Les auteurs expriment ensuite ces contraintes par des politiques de sécurité décrites en WS-Policy.

Dans [BR06], les auteurs s'intéressent à la mise en place d'un modèle RBAC protégeant non l'accès aux services mais aux informations. Ainsi lors de l'utilisation d'un Web-Service, les informations renvoyées dépendent des autorisations associées à l'utilisateur exécutant ce service. Les auteurs proposent également une méthode permettant de convertir les rôles d'une organisation en ceux d'une autre organisation permettant ainsi la mise en place de processus métier inter-organisationnels.

Les auteurs de [BCP06] proposent une version étendue de BPEL appelée RBAC-WS-BPEL permettant en particulier d'ajouter des contraintes RBAC au langage BPEL. Ce langage présuppose que les organisations fournissant des programmes BPEL mettent en œuvre un système d'identification des utilisateurs leur attribuant un rôle au sein de l'organisation. Les auteurs proposent ainsi d'adosser au langage BPEL, un langage appelé BPCL (Business Process Constraint Language) permettant de spécifier pour chaque service appelé par le programme BPEL les contraintes associées. Ces contraintes sont de deux types. Tout d'abord les contraintes propres au modèle RBAC autorisant ou non un utilisateur à accéder à un service en fonction de son rôle. Ils définissent également des contraintes propres à l'exécution du programme BPEL permettant par exemple de définir des contraintes de séparation des obligations permettant de spécifier que deux actions doivent être exécutées par deux utilisateurs différents ou le même. Ces contraintes permettent aussi d'utili-

ser la hiérarchie des rôles qui permet d'ordonner les différents rôles au sein d'une organisation. L'ensemble des contraintes est spécifié dans un fichier différent du programme BPEL ce qui permet de différencier la politique de sécurité du programme mais aussi de modifier plus facilement celle-ci.

2.3 Modèles de sécurité pour les architectures orientées services

Afin de prendre en compte les spécificités du modèle d'architecture orientée services de nouveaux modèles de sécurité ont été proposés.

Les auteurs de [PLC09] proposent la mise en place du modèle TBAC (Task-Based Access Control) pour les programmes BPEL. Ce modèle, à l'origine, a été proposé dans [HK03] dans le cadre des processus métiers. Ce modèle est centré sur les tâches effectuées dans un processus métier. Il permet en particulier de prendre en compte l'exécution ou non d'une tâche avant d'en effectuer une autre. Il permet également de prendre en compte des autorisations différentes en fonction du temps. Dans [PLC09], les auteurs proposent une modification de BPEL afin de permettre l'ajout de contraintes TBAC. Les fichiers BPEL modifiés sont ensuite transformés afin de produire un fichier BPEL compatible avec les interpréteurs BPEL qui fera appel à un service externe chargé de vérifier et d'appliquer les contraintes TBAC.

Les auteurs de [FBFF07] proposent un modèle de politique de sécurité adapté aux architectures orientées services. Ce modèle est architecturé autour de serveurs de politique de sécurité qui définissent pour chaque service une politique d'emploi, c'est à dire l'usage que les utilisateurs peuvent faire de ces services. A cet effet ils définissent six classes de restriction d'emploi des services :

- services utilisables sans restrictions ;
- services dont l'emploi est interdit ;
- services avec restrictions en lecture, c'est à dire que les données renvoyées par ces services ne peuvent être utilisées en dehors d'un certain domaine ;
- services dont certains paramètres d'entrée ne peuvent être utilisés qu'avec certaines valeurs ;
- services dont certains paramètres d'entrée ne peuvent être utilisés qu'avec des informations provenant d'un certain domaine.

Afin de vérifier l'application de la politique d'emploi des services, ils proposent des analyses statiques permettant de vérifier que les orchestrations BPEL accédant à ces services ne violent pas la politique d'accès aux services.

L'intérêt de cette politique réside dans le fait qu'elle permet d'adresser précisément une politique d'emploi des services au sein d'une entreprise. Néanmoins celle-ci reste fortement centralisée et ne permet pas à un utilisateur de définir une politique d'emploi des données qu'il fournit à des orchestrations de services.

2.4 Bilan

Nous avons présenté dans ce chapitre les principaux modèles de contrôle d'accès et leurs applications aux architectures orientées services. Le contrôle d'accès permet d'indiquer si un sujet est autorisé ou non à accéder à un objet et à son contenu. Cependant ce modèle de politique de sécurité ne permet pas de contrôler la dissémination d'une information. En effet une fois qu'un sujet a accédé au contenu d'un objet, il n'y a plus aucun contrôle sur ce que le sujet fait du contenu de l'objet.

Les modèles de contrôle d'accès permettent de définir une politique d'emploi des services au sein d'une entreprise ou d'une organisation. Cependant ils ne permettent pas à un utilisateur de définir une politique d'emploi des informations qu'il fournit à une orchestration de services. Afin de pouvoir contrôler la dissémination d'informations au sein d'une orchestration de services nous avons étudié les modèles de politique de sécurité ainsi que les techniques permettant le contrôle des flux d'informations. Ces modèles sont présentés dans le chapitre suivant.

Chapitre 3

Contrôle de flux d'information

L'objectif du contrôle de flux d'information est de contrôler la diffusion de l'information au cours de l'exécution d'un programme ou d'un système. Une politique de flux d'information définit les flux d'information autorisés ou interdits. Une violation de la politique de flux est détectée lorsqu'un utilisateur accède à tout ou partie d'une information à laquelle il ne devrait pas accéder selon cette politique.

Dans ce chapitre, nous présentons les principes généraux du contrôle de flux d'information, nous nous intéressons aux différentes politiques de flux qui permettent sa mise en place, puis nous décrivons les différents modes d'implémentations utilisés (en particulier les approches statiques et dynamiques). Nous présentons enfin les principaux problèmes liés aux modifications des politiques de contrôle de flux.

3.1 Suivi des flux d'information

3.1.1 Flux d'information

Les flux d'information expriment une relation de dépendance entre des données sources et des données cibles. De manière générale il y a un flux d'information d'un objet a vers un objet b dès lors que l'information contenue dans b dépend de l'information contenue dans a. Certains des tous premiers travaux à s'intéresser à la notion de flux d'information sont ceux de Denning ([Den76, DD77]). L'auteur note $a \Rightarrow b$, le flux d'information d'un objet a vers un objet b.

Le suivi des flux d'information peut s'effectuer à plusieurs niveaux. Des analyses des flux d'information ont été proposées tant pour les programmes [DD77], qu'au niveau des systèmes d'exploitation [ZMB03, EKV$^+$05, KYB$^+$07] ou des architectures de services [HV06, RM09].

3.1.2 Flux d'information au sein des programmes

Denning [Den76, DD77] s'intéresse aux flux d'information au sein des programmes et en distingue en particulier deux types :

– *Les flux explicites*, un flux explicite $x \Rightarrow y$ est généré d'un objet x vers un objet y si ce flux est généré indépendamment de la valeur de x. Par exemple, les opérations d'affectation ou les entrées sorties sont à l'origine de flux explicites. Ainsi l'élément de programme suivant constitue un flux explicite des variables x et y vers z :

```
z:=x+y;
```

A la fin de l'exécution de ce programme, le contenu de la variable z dépend à la fois des informations de x et de celles de y et le flux $x, y \Rightarrow z$ est effectué quelle que soit la valeur de x et de y.

– *Les flux implicites*, c'est à dire les informations passées à travers les structures de contrôle d'un programme. Un flux $x \Rightarrow y$ est implicite si l'opération générant ce flux est conditionné par la valeur de x. Par exemple, suivant la valeur de x, une opération génère un flux explicite $z \Rightarrow y$, z pouvant être une constante. A l'issue de l'exécution de l'opération, la valeur de y dépend donc de la valeur de x et potentiellement de celle de z. L'élément de programme suivant constitue un flux implicite d'information $x \Rightarrow y$ si la variable y n'a pas été créée avant l'exécution de cet élément :

```
if x=0 then y:=1;
```

En effet une information sur x est apportée par y grâce à l'affectation $y := 1$. Ainsi si $y = 2$ alors on sait que $x \neq 0$. On distingue en particulier :

– *les flux implicites directs* créés lorsque la variable $y = 1$ ce qui nous indique que $x = 0$,

– *les flux implicites indirects* créés par y lorsque $x \neq 0$, en effet dans ce cas y n'existe pas, mais sa non existence nous indique que $x \neq 0$.

3.1.3 Flux d'information dans un système d'exploitation

Un système d'exploitation assure une interface entre les processus (c'est à dire l'ensemble des programmes en cours d'exécution sur celui-ci) et la couche matérielle. Le suivi des flux d'information au sein d'un système d'exploitation permet de suivre les flux d'information générés par l'ensemble des applications du système. Le rôle du noyau du système d'exploitation est de gérer les ressources matérielles et de fournir aux programmes une interface uniforme pour l'accès à ces ressources. L'accès aux ressources est fourni par des fonctions primitives appelées appels système. Les appels systèmes permettent en particulier de contrôler l'accès aux fichiers et à la mémoire, de contrôler les processus et les communications inter-processus. Le contrôle des flux d'information dans un système passe donc par l'étude des flux d'information créés par ces appels systèmes.

Les travaux de Bell et Lapadula [BL76] modélisent les flux d'information par les accès en lecture et en écriture. Par exemple, un flux d'information d'un conteneur A vers un conteneur B est considéré si un programme accède en lecture au conteneur A et en écriture au conteneur B. Une partie du contenu de B provient alors potentiellement de A.

Plusieurs travaux permettant de contrôler les flux d'information au niveau du système d'exploitation ont été proposés. Zimmerman [ZMB03] propose par exemple Blare, une surcouche du système Linux permettant de suivre les flux d'information intervenant lors des appels systèmes. Les auteurs de Flume [KYB+07] proposent une autre surcouche du système Linux permettant le suivi dynamique des flux d'information. D'autres travaux se sont intéressés à développer de nouveaux systèmes d'exploitation permettant nativement le suivi dynamique de flux d'information comme par exemple les systèmes Asbestos [EKV+05] ou Histar [ZBWKM06].

La plupart des modèles proposés ([ZMB03, EKV+05, KYB+07]) considèrent les processus comme des boîtes noires (toutes les sorties dépendent de toutes les entrées). Les flux d'information considérés sont ceux qui interviennent entre les processus et les conteneurs d'information (en particulier fichiers et espaces mémoire).

Cependant ce type d'approche considère des conteneurs d'information de forte granularité et adaptés au niveau du système d'exploitation (fichier, page mémoire, etc.). Le système n'a donc pas accès aux flux internes des applications entre conteneurs de faible granularité (variable, champs d'objets, etc.) et est amené à surapproximer les flux d'information. Certains travaux comme [Hie08] ont cherché à lier une analyse au niveau du système d'exploitation à une analyse au niveau des programmes afin de prendre en compte le maximum de flux d'information.

3.1.4 Flux d'information dans les orchestrations de service

Au niveau des architectures orientées services [HV06, RM09], les approches proposées sont à mi-chemin entre les approches système et les approches programme. Elles considèrent en effet le plus souvent les orchestrations de services. Elles considèrent les flux d'information au sein des orchestrations de services comme au sein d'un programme interagissant avec d'autres services qui sont alors vus comme des appels de fonction. Nous retrouvons alors les flux décrits par Denning au sein des programmes, c'est à dire les flux explicites (lors des appels de services ou des affectations de variables par exemple) et les flux implicites (produits par les boucles et les conditionnelles).

Les auteurs de [HV06] présentent deux problèmes spécifiques aux architectures orientées services.

Un premier problème est posé par la nature dynamique des architectures orientées services. En effet le lien vers le service effectivement exécuté peut n'être effectué qu'au cours de l'exécution d'une orchestration de services. Le service est d'abord recherché dans un annuaire avant d'être effectivement exécuté. L'information sur le flux d'information effectivement exécuté n'est alors disponible qu'au moment de l'exécution.

Le second problème est lié aux différentes entités impliquées dans une architecture orientée service. Le contrôle de flux d'information ne doit en effet pas seulement s'effectuer au sein de l'orchestration de services, mais doit aussi prendre en compte les flux générés par l'invocation de services externes. En effet, le résultat

d'une invocation de service dépend à la fois des données utilisées par l'orchestration pour accéder à ce service et des données internes au service.

3.2 Politiques de flux d'information

«La politique de sécurité d'un système spécifie l'ensemble des lois, règlements et pratiques qui régissent la façon de gérer, protéger et diffuser les informations et autres ressources sensibles au sein d'un système spécifique.» [ITS91]

Une politique de flux d'information spécifie l'ensemble des flux d'information légaux ou illégaux au sein d'un système ou d'un programme. Une violation de la confidentialité est caractérisée par une fuite d'information vers un utilisateur qui n'est pas autorisé à accéder à cette information.

Un système mettant en œuvre une politique de contrôle de flux d'information cherche à identifier l'ensemble des flux d'information se produisant dans le système et pour chaque flux d'information à indiquer si celui-ci est légal ou non vis-à-vis de la politique de flux d'information considérée.

De nombreux modèles de politiques de flux d'information ont été proposés dans la littérature ([VSI96, ML97, KYB$^+$07]). Nous présenterons successivement dans cette section le modèle en treillis inspiré par les travaux de Bell et LaPadula ([BL76]), les modèles décentralisés inspirés par Myers ([ML97]), et enfin le modèle du système Blare ([HTMM09]).

3.2.1 Politiques en treillis

Politique de flux d'information en treillis

Afin de contrôler les flux d'information entre les objets, Denning ([Den76]) propose d'associer à chaque objet une classe de sécurité, la comparaison entre les classes de sécurité des objets permettant de spécifier si le flux est légal ou non.

Dans [Den76], Denning définit formellement une politique de flux d'information par le triplet suivant $(CS, \rightarrow, \bigoplus)$, où CS est un ensemble de classes de sécurité, $\rightarrow$ la relation autorisant les flux d'information entre les classes de sécurité et $\bigoplus$ l'opérateur permettant de combiner les classes de sécurité entre elles. L'opérateur de jonction spécifie quelle est la classe de sécurité d'une information obtenue en combinant des informations de classes différentes.

Une forme très simple de politique de flux d'information peut-être définie en utilisant deux classes de sécurité, par exemple *secret* (S) et *public* (P). Tous les flux entre ces classes de sécurité sont autorisés sauf de *secret* vers *public*. Cette politique peut-être formulée de la manière suivante :

Classes de sécurité : $CS = \{S, P\}$

Relation d'autorisation des flux $\rightarrow = \{(S, S), (P, P), (P, S)\}$, (c'est-à-dire que $S \rightarrow S$, $P \rightarrow P$, $P \rightarrow S$ et $S \nrightarrow P$)

Opérateur de combinaison des classes $\bigoplus$ est défini de la manière suivante :
$S \bigoplus S = S$, $P \bigoplus S = S$, $S \bigoplus P = S$ et $P \bigoplus P = P$.

Cette politique est présentée sur la figure 3.1(a).

Parmi l'ensemble des politiques de flux d'information exprimables à l'aide de classes de sécurité, Denning distingue une catégorie spécifique qui peut s'exprimer sous la forme d'un treillis fini. Une politique de flux d'information $(CS, \rightarrow, \bigoplus)$ est exprimable sous la forme d'un treillis fini si elle vérifie les quatre axiomes suivants :

1. l'ensemble des classes de sécurité CS est fini ;
2. la relation de flux $\rightarrow$ définit un ordre partiel [1] sur CS ;
3. CS a une borne inférieure selon la relation $\rightarrow$;
4. l'opérateur de jointure $\bigoplus$ est totalement défini et possède une borne supérieure.

L'axiome 1 est une condition nécessaire à ce que le treillis soit fini. L'axiome 3 permet de modéliser les informations publiques. L'axiome 4 permet d'assurer que l'on peut combiner les informations de n'importe quelles classes.

L'exemple de la figure 3.1(a) constitue un treillis fini. En effet l'axiome 1 est respecté car le nombre de classe est fini (2). La relation $\rightarrow$ est une relation d'ordre partielle car elle est réflexive ($S \rightarrow S$ et $P \rightarrow P$), transitive (il n'y a que 2 classes de sécurité), et antisymétrique ($S \rightarrow S$, $P \rightarrow P$ et $S \nrightarrow P$) (axiome 2). La borne inférieure de CS est P (axiome 3). L'opérateur $\bigoplus$ est totalement défini et possède une borne supérieure S (axiome 4).

Les travaux de Denning sont exprimés en utilisant la relation de flux autorisé $\rightarrow$, cependant dans la plupart des travaux la relation de dominance, c'est-à-dire la relation inverse est utilisée. Elle est définie de la façon suivante :

– $A \geq B$ (A domine B) si et seulement si $B \rightarrow A$;
– $A > B$ si et seulement si $A \geq B$ et $A \neq B$;
– si $A > B$ alors $A \nrightarrow B$ mais $B \rightarrow A$.

Le treillis militaire

Un exemple de politique de flux d'information est une suite totalement ordonnée de classes de sécurité. L'exemple le plus courant est celui utilisé dans le milieu militaire. Il est composé de 4 niveaux de confidentialité (non classifié, diffusion restreinte, confidentiel, secret) totalement ordonnés. Une représentation en treillis de cette politique est présentée figure 3.1(b).

Cependant, avec un ordre de niveaux de sécurité, nous ne pouvons exprimer qu'un ensemble limité de politiques de sécurité. Il est en particulier difficile de gérer le besoin d'en connaître, c'est-à-dire le fait qu'une personne autorisée à accéder à une information confidentielle n'est pas autorisée à accéder à l'ensemble

1. Une relation $\circ$ est une relation d'ordre si et seulement si c'est une relation reflexive ($a \circ a \; \forall a$), transitive (si $a \circ b$ et $b \circ c$ alors $a \circ c$) et antisymétrique (si $a \circ b$ et $b \circ a$ alors $a = b$). Elle est partielle si la relation d'ordre n'est pas définie pour tous les éléments deux à deux.

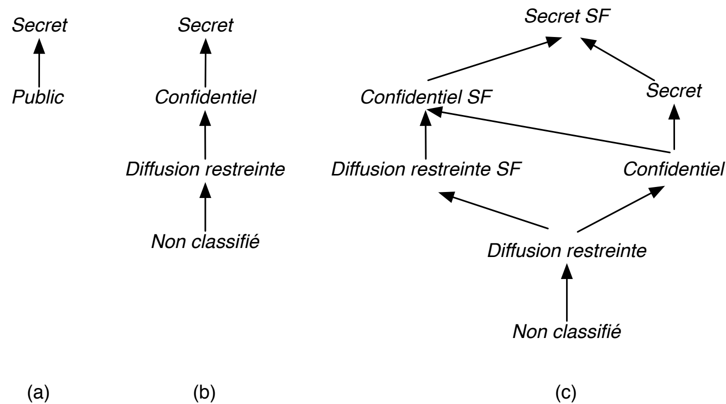

FIGURE 3.1 – Exemples de politiques de flux d'information

des informations confidentielles, mais uniquement aux informations confidentielles qu'elle a besoin de connaitre. Ainsi l'ensemble des personnes habilitées au niveau secret peuvent accéder à l'ensemble des informations des projets secrets et pas seulement aux projets sur lesquels elles travaillent. Afin de pallier ces limitations le modèle suivant de treillis a été introduit par [BL73].

Considérons H, un ensemble de classifications avec un ordre hiérachique $\leq_H$ et C un ensemble de catégories (par exemple nom d'un projet, d'une division d'une entreprise, d'un département académique,...). Une étiquette de sécurité est alors une paire (h, c) ou $h \in H$ est un niveau de sécurité et $c \subseteq C$ une catégorie. Un ordre partiel $\leq$ des étiquettes de sécurité est alors défini ainsi, $(h_1, c_1) \leq (h_2, c_2)$ si et seulement si $h_1 \leq_H h_2$ et $c_1 \subseteq c_2$. Au sein du ministère de la défense français une mention *Spécial France (SF)* est ainsi ajoutée afin de spécifier les documents qui ne peuvent être transmis qu'à des français. Une représentation en treillis de cette politique est présentée figure 3.1(c). Une étude plus complète des politiques de sécurité en treillis est présentée dans [San93].

Ce type de politique de sécurité est essentiellement utilisé dans le milieu étatique. Cependant celle-ci nécessite l'accréditation des utilisateurs avant que ceux-ci puissent accéder à une information. De plus elle ne permet pas de spécifier les destinataires d'une information (et gérer ainsi finement le besoin d'en connaître), mais uniquement des niveaux de sensibilité au sein d'un ensemble d'informations. Afin de pallier ces limitations de nombreux modèles ont été proposés en particulier le modèle décentralisé ([ML97]) ou le modèle du système Blare ([HTMM09]).

Modèle de Bell et LaPadula

Le modèle de contrôle proposé par Bell et LaPadula [BL76] vise à contrôler la confidentialité des données utilisées dans un système. Il est présenté sous la forme d'une machine à états finis. Ce modèle est une première approche du contrôle de flux d'information au niveau d'un système.

Le système de Bell et LaPadula est composé des éléments suivants :

- S un ensemble de sujets ;
- O un ensemble d'objets ;
- A les opérations d'accès sur les objets {`execute`, `read`, `write`, `append`} (nous simplifions les notations en considérant uniquement les accès en *lecture* (`read` et `write`) et les accès en *écriture* (`write` et `append`)) ;
- L un ensemble de classes de sécurité avec un ordre partiel $\leq$;
- un état est une matrice $(S \times O \to A)$ qui pour chaque sujet $s \in S$, chaque objet $o \in O$, décrit tout les accès autorisés $a \in A$.

A chaque sujet est attribué une habilitation $h(s) \in L$ et à chaque objet est associé une classification $c(o) \in L$. Ces habilitations et classifications sont fixes et n'évoluent pas lors des modifications de l'état du système.

Un état est considéré comme sûr si il respecte les deux propriétés suivantes :

Propriété 3.2.1 (Propriété simple)**.** *Si* $(s_i, o_j, lecture) \in (S \times O \times A)$ *alors* $c(o_j) \leq h(s_i)$.

Propriété 3.2.2 (Propriété ★)**.** *Si* $(s_i, o_j, lecture) \in (S \times O \times A)$ *et* $(s_i, o_k,$ *écriture)* $\in (S \times O \times A)$*, alors* $c(o_j) \leq c(o_k)$.

La propriété simple permet d'assurer qu'un sujet n'accède en lecture qu'à des objets dont le niveau de classification est inférieur à son niveau d'habilitation. Cette propriété permet d'assurer le contrôle d'accès à l'information. La propriété ★ quant à elle permet de contrôler le flux d'information en empêchant un sujet pouvant accéder à un objet d'un niveau de classification d'écrire le contenu de celui-ci dans un objet de classification inférieure.

Biba [Bib77] a de son côté montré que les problèmes d'intégrité étaient duaux à ceux de confidentialité en adaptant le modèle de Bell et Lapadula au contrôle de l'intégrité des données.

Modèle de Denning

Denning [Den76] propose d'associer à l'ensemble des objets du système une classe de sécurité indiquant un niveau de confidentialité. Lorsqu'un flux d'information se produit d'un objet o_1 vers un objet o_2 la comparaison des classes de o_1 et o_2 doit permettre de calculer la légalité ou non de ce flux.

Nous pouvons alors définir le modèle de contrôle de flux d'information proposé par Denning de la manière suivante. Nous considèrons alors :

- un ensemble L de classes de sécurité associé à une relation d'ordre $\leq$,
- un ensemble O d'objets,
- une fonction $f_o : O \to L$ qui attribue à chaque objet une classe de sécurité.

Les flux d'information légaux au regard de la politique de flux d'information sont alors les suivants : un flot d'information de l'objet o_1 vers l'objet o_2 est légal si $f_o(o_1) \leq f_o(o_2)$.

Ainsi le flux d'un objet o_1 *public* vers un objet o_2 *public* est légal car $f_o(o_1) = f_o(o_2) = public$. Le flux d'un objet o_1 *privé* vers un objet o_2 *public* est par contre illégal car $f_o(o_1)$ =*privé* $> f_o(o_2) = public$.

Denning propose d'appliquer ce modèle à des systèmes réels afin de garantir l'absence de flux d'information illégaux au sein de ces systèmes. La principale difficulté réside dans l'identification et le suivi des flux d'information. Deux solutions sont envisagées :

– une approche statique dans laquelle les classes de sécurité associées aux objets sont fixes au cours de l'exécution d'un programme ;
– une approche dynamique qui permet de faire varier les classes de sécurité associées aux objets au cours de l'exécution du programme.

Cependant Denning [DD77] ne propose qu'une formalisation d'un mécanisme de vérification statique, même si elle évoque la machine de Fenton [Fen74] qui permet de contrôler dynamiquement les flux d'information entre les variables d'un programme bien que de manière incomplète car ne prenant pas en compte les flux implicites indirects.

Composition de politiques de sécurité

Dans le cadre des architectures orientées services il est souvent difficile de mettre en œuvre les modèles présentés précédemment car ceux-ci peuvent impliquer plusieurs entités qui peuvent chacune avoir leur propre politique de sécurité. Il est donc nécessaire qu'un orchestrateur de service implémente non-seulement la politique de sécurité de celui qui invoque l'orchestration de service mais aussi celle de chaque service invoqué. A cet effet, les auteurs de [HV06] proposent donc un modèle permettant de composer les politiques de sécurité de différentes entités.

Ils considèrent une politique définie par le treillis $(SC, \leq)$ où SC est un ensemble fini de classes de sécurité partiellement ordonné par la relation $\leq$. Afin de représenter les politiques de sécurité d'objets issus de différents objets ayant des politiques de sécurité différentes ils proposent d'utiliser le modèle suivant. Ils combinent différentes politiques de flux d'information $(SC_1, \leq_1)$,..,$(SC_n, \leq_n)$ en une politique composée $(SC, \leq)$ où $SC = SC_1 \times ... \times SC_n$ et $\langle \tau_1, ..., \tau_n \rangle \leq \langle \tau'_1, ..., \tau'_n \rangle$ si et seulement si $\tau_i \leq \tau'_i$ pour tout $1 \leq i \leq n$. La borne supérieure (respectivement inférieure) de SC est calculée à l'aide de l'ensemble des bornes supérieures (respectivement inférieures) de chaque composant.

La politique de sécurité est alors exprimée ainsi : l'utilisateur affecte à chaque donnée envoyée à un service une classification et à chaque service une habilitation, σ_{ws} étant l'ensemble des habilitations associées aux web-services.

Comme l'ensemble des Web-Services utilisés n'est pas forcément connu au départ, l'utilisateur peut déléguer à certains Web-Service le droit de classifier les données qu'il manipule suivant la politique de sécurité qu'il découvre. Ainsi σ_{del} est l'ensemble des délégations de classification associées aux Web-Services. $\sigma_{del}(WS)$ est l'habilitation maximum qu'un Web-Service WS peut affecter à un service inconnu. Si $\perp$ est la borne inférieure de SC alors $\sigma_{del}(WS) = \perp$ signifie que WS

n'est pas autorisé à habiliter des services inconnus. Si $\top$ est la borne supérieure de SC alors $\sigma_{del}(WS) = \top$ signifie que WS est autorisé à habiliter l'ensemble des services inconnus. σ_{del} ne peut-être modifié durant l'exécution d'une orchestration.

3.2.2 Le modèle décentralisé

Le modèle de Myers

Myers ([ML97]) propose d'associer aux conteneurs d'information d'un système une étiquette de sécurité décrivant la politique de sécurité associée à ce conteneur.

Une étiquette de sécurité est d'abord formée par un ensemble d'utilisateurs, propriétaires du conteneur associé à cette étiquette. A chaque propriétaire est ensuite associé un certain nombre d'utilisateurs, appelés lecteurs, autorisés à accéder aux conteneurs associés à cette étiquette de sécurité. La politique de sécurité est interprétée de la façon suivante : pour qu'un utilisateur soit autorisé à accéder au conteneur associé à une étiquette de sécurité, il faut qu'il soit désigné comme lecteur de cette étiquette de sécurité par l'ensemble des propriétaires de cette étiquette de sécurité. Chaque propriétaire d'une donnée est autorisé à modifier l'étiquette de sécurité de cette donnée en lui ajoutant un certain nombre de lecteurs.

Afin de définir formellement les étiquettes de sécurité les éléments suivants sont définis :

Les utilisateurs : $U = \{u_1, u_2, ..., u_n\}$ est l'ensemble des utilisateurs du système.

Les propriétaires associés à une étiquette de sécurité e sont des utilisateurs notés $prop(e) \subseteq U$.

Les lecteurs associés à une étiquette de sécurité e sont des utilisateurs définis par les propriétaires de e notés $lec(e) \subseteq U$. Chaque propriétaire pouvant définir un ensemble de lecteurs, on note alors $lec(e, u) \subseteq U$ l'ensemble des lecteurs de e autorisés par u. On a alors :

$$lec(e) = \bigcap_{u \in prop(e)} lec(e, u) \tag{3.1}$$

Les étiquettes de sécurité associées aux données sont de la forme

$$e = \{\mathbf{p}_\alpha : l_{\alpha_1}, ..., l_{\alpha_m}; ...; \mathbf{p}_\beta : l_{\beta_1}, ..., l_{\beta_n}\}$$

où $prop(e) = \{p_\alpha\} \cup ... \cup \{p_\beta\}$, $lec(e, p_\alpha) = \{l_{\alpha_1}, ..., l_{\alpha_m}\}$ et $lec(e, p_\beta) = \{l_{\beta_1}, ..., l_{\beta_n}\}$.

Considérons par exemple l'étiquette $e_1 = \{p_1 : l_1, l_2;\ p_2 : l_2, l_3\}$. Nous avons alors $prop(e_1) = \{p_1, p_2\}$, $lec(e_1, p_1) = \{l_1, l_2\}$, $lec(e_1, p_2) = \{l_2, l_3\}$. Donc $lec(e_1) = lec(e_1, p_1) \cap lec(e_1, p_2) = l_2$. Les conteneurs associés à l'étiquette de sécurité e_1 ont donc deux propriétaires p_1 et p_2 et un lecteur l_2.

Relation d'ordre Une relation d'ordre est définie entre les étiquettes de sécurité. Considérons deux étiquettes de sécurité e_1 et e_2, e_1 est une restriction de e_2, notée $e_1 \sqsubseteq e_2$ si les propriétaires de e_1 sont inclus dans ceux de e_2 (c'est-à-dire que e_2 a plus de propriétaires que e_1, un propriétaire n'est pas un lecteur autorisé d'une donnée, pour pouvoir lire une donnée, il faut être autorisé par l'ensemble des propriétaires) et si pour chaque propriétaire de e_1, les lecteurs autorisés par ce propriétaire pour e_2 sont aussi autorisés par ce propriétaire pour e_1 (c'est à dire que e_2 a moins de lecteurs autorisés que e_1) :

Définition 3.2.1 ($e_1 \sqsubseteq e_2$)**.** *si et seulement si :*
$prop(e_1) \subseteq prop(e_2)$
$\forall p \in prop(e_1), lec(e_1, p) \supseteq lec(e_2, p)$

Etiquette de sécurité des conteneurs dérivés Lorsqu'un programme combine deux conteneurs dans un troisième, l'étiquette de sécurité associée au nouveau conteneur doit refléter la politique de sécurité associée à ces deux conteneurs. Les propriétaires du conteneur dérivé sont l'ensemble des propriétaires des deux conteneurs initiaux. Les lecteurs du conteneur dérivé sont l'intersection des lecteurs des deux conteneurs initiaux. La règle d'union de deux étiquettes de sécurité est la suivante :

Définition 3.2.2 ($e_1 \sqcup e_2$)**.** *est défini par :*
$prop(e_1 \sqcup e_2) = prop(e_1) \cup prop(e_2)$
$lec(e_1 \sqcup e_2, p) = lec(e_1, p) \cap lec(e_2, p)$

La politique de contrôle de flux d'information est donc définie par :
- E l'ensemble des étiquettes de sécurité que l'on peut exprimer à l'aide de l'ensemble U des utilisateurs du système ;
- $\sqsubseteq$ la relation d'ordre entre les classes de sécurité ;
- $\sqcup$ la relation d'union entre les classes de sécurité.

Le modèle d'étiquettes de sécurité proposé par Myers permet donc d'exprimer des politiques de sécurité complexes. Il a été mis en œuvre dans un compilateur Java [Mye99]. Des évolutions du modèle original ont été proposées permettant la mise en pratique d'une politique de contrôle de flux décentralisée au sein d'un système d'exploitation [EKV$^+$05, KYB$^+$07].

Néanmoins il reste un modèle bien adapté aux architectures orientées services. En effet un processus BPEL fait intervenir de nombreux services qui peuvent être vus comme des utilisateurs. Ce modèle a en particulier été utilisé par les auteurs de [ZA11] et [SARL12] qui proposent une analyse statique d'orchestrations de services afin de vérifier que celles-ci ne violent pas une politique de sécurité basée sur le modèle décentralisé de contrôle de flux d'information.

Les évolutions du modèle décentralisé

Le système Asbestos Afin de contrôler la confidentialité des données, les auteurs du système Asbestos ([EKV$^+$05]) utilisent 4 niveaux de sécurité à la manière du

modèle en treillis notés ainsi $[*, \mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}]$. $\mathbf{0}$ étant le niveau non confidentiel, $\mathbf{3}$ étant le niveau le plus protégé. Le symbole $*$ est utilisé pour gérer les déclassifications (c'est à dire les modifications de la politique de flux).

A la manière des utilisateurs chez Myers, les auteurs de Asbestos proposent de définir des catégories, une catégorie désignant une catégorie de données. A chaque utilisateur u est associée une catégorie marquée u_m.

Les classes de sécurité proposées dans le modèle de Asbestos consistent à associer à chaque catégorie un niveau de sécurité. Elles sont de la forme $\{c_1\mathbf{0}, c_2\mathbf{1}, \mathbf{2}\}$. Ainsi le niveau **0** est associé à la catégorie c_1, le niveau **1** est associé à la catégorie c_2 et le niveau par défaut **2** est associé à l'ensemble des autres catégories.

Une relation d'ordre $\sqsubseteq$ est définie entre les classes de sécurité en comparant chacune des composantes selon la définition suivante :

Définition 3.2.3 ($cl_1 \sqsubseteq cl_2$). *Soit cl_1 et cl_2 deux classes de sécurité, $cl_1 \sqsubseteq cl_2$ si et seulement si :*

$cl_1(c) \leq cl_2(c)$ *pour tout c appartenant à l'ensemble des catégories de données.*

La relation d'union entre deux classes de sécurité $\sqcup$ est définie par :

Définition 3.2.4 ($cl_1 \sqcup cl_2$). *Soit cl_1 et cl_2 deux classes de sécurité, $cl_1 \sqcup cl_2$ est défini par :*

$(cl_1 \sqcup cl_2)(c) = max(cl_1(c), cl_2(c))$ *pour tout c appartenant à l'ensemble des catégories de données.*

Cependant, décrire une politique de flux pour Asbestos est difficile, c'est pourquoi les auteurs de [EK08] proposent un langage dédié pour la spécification de politique de flux. Ils définissent à cet effet des compartiments qui représentent un ensemble d'objets ayant la même politique de sécurité. Les relations entre les compartiments sont alors décrites paire à paire. Ainsi il existe quatre possibilités de communication entre deux compartiments X et Y :

- $X!Y$, aucune communication entre X et Y n'est possible ;
- $X > Y$, X peut envoyer des messages à Y mais ne peut en recevoir de Y ;
- $X < Y$, X peut recevoir des messages de Y mais ne peut lui en envoyer ;
- $X <> Y$, X et Y peuvent communiquer librement.

Ils décrivent ensuite un algorithme permettant de transformer une telle politique de flux en classe de sécurité d'Asbestos.

Le système Flume Les auteurs de Flume [KYB$^+$07] ont proposé un autre modèle de sécurité pour les systèmes d'exploitation permettant le contrôle de flux d'information. Ce système permet de protéger à la fois l'intégrité et la confidentialité des données.

Ce système considère des processus c'est-à-dire soit des threads soit des objets (des conteneurs d'information).

Dans ce système, le contrôle de flux est assuré par des marques, chacune des marques est associée à un ou plusieurs processus. Une classe de sécurité est constituée d'un ensemble de marques. Chaque processus est associé à deux classes de sécurité S_p (confidentialité) et I_p (intégrité). Les classes de sécurité des objets ne sont pas modifiables alors que ceux des threads sont dynamiques.

A chaque thread est associé un ensemble de capacités, un ensemble de marques indiquant la capacité d'un thread à modifier ses classes de sécurité. Pour une marque t, la capacité t^+ indique la capacité d'un thread à ajouter la marque t à une de ses classes de sécurité, la capacité t^- indique la capacité d'un thread à enlever la marque t à une de ses classes de sécurité. Chaque thread a un ensemble de capacités O_p. Il existe aussi un ensemble de capacités O détenu par l'ensemble des threads.

Ces classes de sécurité permettent de gérer la confidentialité. Un thread ayant la capacité t^+ peut recevoir des données marquées par t si t est une marque de confidentialité. Si il a la capacité t^-, il peut alors déclassifier une donnée marquée par t. Un thread possédant les capacités t^+ et t^- est donc l'équivalent d'un propriétaire du modèle de Myers ou de Asbestos.

Ces classes de sécurité gèrent également l'intégrité. Si t est un marqueur d'intégrité alors tous les processus ont par défaut la capacité t^- d'enlever ce label t. L'autorité de certification possède quant à elle la capacité t^+ d'ajouter le tag t.

3.2.3 Le modèle du système Blare

Dans [Hie08], l'auteur propose de distinguer explicitement les conteneurs d'information (par exemple un fichier, une variable,...) des contenus, c'est à dire l'information contenue dans les conteneurs. Ils proposent alors la définition suivante des flux d'information, qui est une reprise de la définition proposée par [DD77] en distinguant les contenus des conteneurs :

> Il existe un flux d'information d'un contenu I, vers un conteneur c dès lors que ce contenu I est transféré vers c ou utilisé pour générer une information qui est elle-même transférée dans c.

Les auteurs du système Blare ([ZMB03, HTMM09]) ont proposé une politique de flux d'information en termes de liens autorisés entre contenus et conteneurs (CCAL : *Contents - Container Authorized Link*). Cette politique à grain très fin permet de prendre en compte l'ensemble des flux d'information pouvant intervenir dans un système ou un programme.

Le modèle distingue les informations du système, c'est à dire les contenus, des conteneurs d'information, c'est-à-dire les objets du système. L'ensemble des informations du système est noté $\mathbb{I} = \{I_1, ..., I_n\}$ et l'ensemble des conteneurs est noté $\mathbb{C} = \{C_1, ..., C_n\}$.

Politique de sécurité En considérant $I \in \mathcal{P}(\mathbb{I})$ un ensemble d'informations et $C \in \mathcal{P}(\mathbb{C})$ un ensemble de conteneurs, une politique de flux d'information est un

ensemble SP de paires (I, C), appelées $CCAL$. Une paire (I, C) signifie que le flux des informations inclus dans I vers les conteneurs inclus dans C est légal.

Une violation de la politique de sécurité SP est caractérisée par l'existence d'un ou plusieurs conteneurs dont le contenu courant n'est pas autorisé par la politique.

Considérons la politique autorisant les flux d'information suivants :

– les flux d'informations de i_1 et i_2 vers c_1 et c_2 ;
– les flux d'informations de i_2 vers c_2 ou c_3.

La politique de flux est alors modélisée par les deux $CCAL$ suivants :

– $CCAL_1 = (\{i_1, i_2\}, \{c_1, c_2\})$;
– $CCAL_2 = (\{i_2\}, \{c_2, c_3\})$

La politique de flux est alors : $SP = \{CCAL_1, CCAL_2\}$

Modèle de Blare étendu Les auteurs de [GHTVTT11] proposent un modèle de Blare étendu. Ils proposent en particulier de distinguer parmi l'ensemble $\mathcal{C}$ des conteneurs, l'ensemble $\mathcal{PC}$ des conteneurs persistants (les fichiers par exemple), l'ensemble $\mathcal{VC}$ des conteneurs volatiles (les plages mémoire par exemple) et l'ensemble $\mathcal{P}$ des processus (qui sont alors considérés comme des sujets actifs). Ils définissent également Π l'ensemble de toutes les classes de processus (deux processus exécutant un même programme appartiennent à la même classe) et $\mathcal{U}$ l'ensemble des utilisateurs. Ils notent enfin $\mathcal{I}$ et $\mathcal{X}$ l'ensemble des informations attachées respectivement à une donnée ou à un code exécuté. Une politique de flux d'information est alors définie par un triplet $\mathbb{P} = (\mathbb{P}_{\mathcal{PC}}, \mathbb{P}_{\mathcal{U}}, \mathbb{P}_{\Pi})$ où :

– une paire $(c, a) \in \mathbb{P}_{\mathcal{PC}}$ exprime que le conteneur c est autorisé à contenir n'importe quel sous-ensemble de a, $a \in \wp(\mathcal{I} \cup \mathcal{X})$;
– une paire $(u, a) \in \mathbb{P}_{\mathcal{U}}$ exprime que n'importe quel sous ensemble de a peut-être lu ou exécuté par l'utilisateur u ;
– une paire $(\pi, a) \in \mathbb{P}_{\Pi}$ exprime que n'importe quel sous-ensemble de a peut-être lu ou exécuté par une classe de processus π exécutant le même code.

Initialisation de la politique de sécurité Une des difficultés de la mise en œuvre du modèle de Blare réside dans la définition de la politique de flux. Il est en effet nécessaire de définir pour l'ensemble des conteneurs du système l'ensemble des informations que chaque conteneur est autorisé à contenir. Les auteurs de [HTMM09] ont proposé de dériver la politique de flux initiale à partir des permissions du contrôle d'accès discrétionnaire, qui peuvent être exprimées sous la forme d'une matrice des contrôles d'accès spécifiant les droits statiques liés aux conteneurs. Les auteurs de [GHTVTT11] ont quant à eux proposé de dériver la politique de flux d'une politique de contrôle d'accès obligatoire [2].

2. Mandatory Access Control (MAC)

3.2.4 Le modèle de la non-interférence

Les modèles de politique proposés se basent sur la description des flux d'information et la description pour chaque flux d'information de sa légalité ou non vis à vis de la politique de sécurité. Cependant ceux-ci ne prennent en compte que les flux d'information décrits dans le modèle et non l'ensemble des flux d'information possibles. Ce problème a été étudié par Goguen et Meseguer [GM82] qui définissent le concept de non-interférence de la façon suivante :

> Un groupe d'utilisateurs, utilisant un certain lot de commandes, n'interfère pas avec un autre groupe d'utilisateurs si ce que le premier groupe fait avec ses commandes n'a pas d'effet sur ce que le second groupe d'utilisateurs peut voir.

De manière plus formelle ils définissent les éléments suivants :
- S l'ensemble des états du système
- $s_0 \in S$ l'état initial
- U l'ensemble des utilisateurs du système
- C l'ensemble des commandes utilisables par les utilisateurs afin de modifier l'état du système
- Out l'ensemble des sorties visibles par les utilisateurs du système

Ils considèrent les fonctions do et out. La fonction $do : S \times U \times C \to S$ est la fonction de transition permettant de construire le nouvel état créé par un état, un utilisateur et une commande. La fonction $out : S \times U \to Out$ modélise la sortie visible à partir d'un état.

Ils notent $W = (U \times C)^*$ l'ensemble des séquences possibles de commandes utilisées par des utilisateurs. Une séquence $w \in W$ est écrite de la façon suivante : $w =< (u_0, c_0), (u_1, c_1), ..., (u_n, c_n) >$. On peut alors définir la fonction $do(s_0, w)$ qui consiste à appliquer les entrées les unes après les autres en séquence. La notation suivante est alors utilisée : $[\![w]\!] = do(s_0, w)$.

La sortie visible par un utilisateur u après l'exécution de w est notée : $[\![w]\!]_u = out([\![w]\!], u)$ Cette approche permet donc de modéliser les sorties visibles par l'utilisateur u.

Afin de pouvoir définir la non interférence les auteurs utilisent une fonction purge $P_G(w)$ ($G \subseteq U, w \in W$) qui est la sous-séquence de w où toutes les commandes issues des utilisateurs de G ont été enlevées.

La non interférence est alors définie de la façon suivante :

Définition 3.2.5 (Non interférence)**.** *Considérons le système (S, U, C, Out, do, out) et un ensemble $G \subseteq U$ d'utilisateurs. Les utilisateurs de G n'interfèrent pas avec les autres utilisateurs du système si et seulement si :*

$$\forall w \in W, \forall u \in U \backslash G, [\![w]\!]_u = [\![P_G(w)]\!]_u$$

Ce qui est alors noté $G : |U \backslash G$.

La non-interférence d'un groupe d'utilisateurs G avec les autres utilisateurs est donc définie par le fait que quelles que soient les commandes effectuées par G, la vision des autres utilisateurs du système n'est pas modifiée.

En considérant un ensemble de niveaux de confidentialité C totalement ordonné, les auteurs de [GM82] proposent une définition d'un système multi-niveaux sécurisé. A chaque utilisateur $u \in U$ est associé un niveau de confidentialité $x \in C$. La fonction $level : U \rightarrow C$, donne le niveau de confidentialité d'un utilisateur. Pour tout $x \in C$, on définit : $U[-, x] = \{u \in U : level(u) \leq x\}$ et $U[+, x] = \{u \in U : level(u) \geq x\}$. Ils proposent alors la défiintion suivante d'un système multi-niveaux sécurisé :

Définition 3.2.6 (Système multi-niveaux sécurisé [GM82])**.** *Un système est multi-niveaux sécurisé si et seulement si :*

$$\forall x, x' \in C : x < x' \Rightarrow U[x', +] : |U[-, x]$$

Dans ce cas un système est multi-niveaux sécurisé si et seulement si pour chaque utilisateur de niveau x, celui-ci est non-interférent avec l'ensemble des utilisateurs de niveau inférieur.

La formalisation du concept de non-interférence sous forme de propriété, permet de prendre en compte l'ensemble des flux d'information et en particulier les canaux cachés. Cependant cette propriété est généralement invérifiable, celle-ci nécessitant de prendre en compte l'ensemble des traces du système.

Cependant le principe d'une formulation d'une propriété de sécurité de bout en bout reste intéressante. C'est pourquoi de nombreux auteurs ont cherché à proposer d'autres propriétés de non-interférence dans lesquels les pouvoirs d'observation d'un attaquant sont limités et définis. Les auteurs de [VSI96] ont ainsi proposé une définition restreinte de la propriété de non-interférence ne prenant en compte que les flux d'information entre les variables publiques et privées d'un système.

3.3 Vérification statique des politiques de flux

Afin de mettre en oeuvre une politique de contrôle de flux d'information au sein d'un programme ou d'un système, il est nécessaire d'implémenter des mécanismes permettant de définir et de contrôler le respect de cette politique.

Il existe deux grandes familles d'analyse de programme : les analyses statiques et les analyses dynamiques. Les analyses statiques vérifient, au niveau du code source et avant l'exécution, qu'un programme respecte un certain nombre de règles. L'analyse dynamique vérifie qu'un programme respecte un certain nombre de propriétés au cours de son exécution. Elle permet, le cas échéant, de modifier l'exécution d'un programme afin d'assurer le respect de ces propriétés.

3.3.1 Principe du contrôle statique de flux d'information

Dans [DD77], les auteurs ont identifié les grands principes du contrôle statique de flux d'information :

- à chaque variable d'un programme est associée une classe de sécurité, cette classe de sécurité est fixe ;
- la classe de sécurité associée à une expression est calculée à l'aide de l'opérateur de jonction $\bigoplus$ à partir des classes de sécurité de l'ensemble des variables de l'expression ;
- la prévention des flux explicites s'effectue en interdisant à une expression dont la classe de sécurité est A d'être affectée à une variable dont la classe de sécurité est inférieure à A ;
- la prévention des flux implicites s'effectue en interdisant les affectations vers une variable de niveau A si celles-ci sont effectuées dans une boucle ou une conditionnelle dont l'expression conditionnelle est d'une classe de sécurité supérieure à A.

3.3.2 Contrôle de flux d'information par typage

En reprenant les principes énoncés par Denning, les auteurs de [VSI96] proposent une vérification statique de programmes permettant de garantir l'absence de flux d'information illégaux. Cette analyse statique se base sur la définition de type de sécurité. De la même façon que l'on définit un type pour chaque variable d'un programme (entier, booléen, flottant, chaine de caractère...), ils proposent de définir un type de sécurité pour chaque variable du programme. Ils utilisent pour ce faire les classes de sécurité du modèle en treillis.

Dans le cadre des analyses de programmes, Volpano Smith et Irvine [VSI96] ont proposé une définition restreinte de la propriété de non-interférence. Ils considèrent le modèle d'un attaquant ne pouvant accéder qu'à l'état des variables publiques au début et à la fin de l'exécution d'un programme. Ainsi, dans ce cas assurer une propriété de non interférence entre variables revient à s'assurer que l'exécution de deux instances d'un même programme ayant le même jeu de variables publiques au début de l'exécution produisent le même jeu de variables publiques à la fin de l'exécution.

Ainsi plus formellement pour l'exécution d'un programme P en notant $[\![P]\!]\, s_i$ l'état final obtenu en exécutant le programme P à partir de l'état initial s_i et $=_L$ la relation qui caractérise l'égalité des variables publiques.

Définition 3.3.1 (Non-interférence entre variables)**.** *Un programme P est dit non-interférent entre variables si et seulement si :*
$\forall s_1, s_2 \in S. s_1 =_L s_2 \Rightarrow [\![P]\!]\, s_1 =_L [\![P]\!]\, s_2$

Un programme P satisfait la propriété de non-interférence entre variables si et seulement si deux états initiaux s_1 et s_2 ayant des valeurs initiales d'entrées publiques *(low)* identiques ($s_1 =_L s_2$), produisent, après exécution de P, des états

```
A ::= x := e
    | skip
B ::= if e then S else S end
    | while e do S done
S ::= S ; S | B | A
```

FIGURE 3.2 – Grammaire du langage « While »

$$[E1-2] \quad \vdash exp : high \qquad \frac{h \notin Vars(exp)}{\vdash exp : low}$$

$$[C1-3] \quad [pc] \vdash \textbf{skip} \qquad [pc] \vdash h := exp \qquad \frac{\vdash exp : low}{[low] \vdash \ l := exp}$$

$$[C4-5] \quad \frac{[pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash C_1; C_2} \qquad \frac{\vdash exp : pc \qquad [pc] \vdash C}{[pc] \vdash \textbf{while}\ exp\ \textbf{do}\ C\ \textbf{done}}$$

$$[C6-7] \quad \frac{\vdash exp : pc \quad [pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash \textbf{if}\ exp\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{end}} \qquad \frac{[high] \vdash C}{[low] \vdash C}$$

FIGURE 3.3 – Système de type sécurisé

finaux équivalents au niveau des données publiques ($[\![P]\!]\, s_1 =_L [\![P]\!]\, s_2$). Cette équivalence caractérise le pouvoir d'observation d'un attaquant en spécifiant ce qu'il peut identifier entre deux exécutions différentes.

Nous présentons un système de type simplifié présenté par Sabelfeld et Myers [SM03] afin de présenter les principes du contrôle de flux d'information par typage. Le langage utilisé est de type « While » dont la grammaire est donnée figure 3.2.

Dans ce système le niveau de sécurité des variables est décrit par des labels statiques. Nous désignerons par h l'ensemble des variables de type *high* (c'est-à-dire privées) et par l l'ensemble des variables de type *low* (c'est-à-dire publiques).

Le système de type est défini par les règles présentées figure 3.3.

$\vdash exp : \tau$ signifie que exp est de type τ suivant la règle de typage. De même $[pc] \vdash C$ signifie que le programme C doit être exécuté dans le contexte d'exécution $[pc]$. Dans notre cas le contexte d'exécution est soit public (*low*), soit privé (*high*).

Les règles $[E1-2]$ permettent de garantir l'absence de flux explicite. En effet, si n'importe quelle expression peut-être de type $high$, seule une expression ne contenant que des variables publiques peut-être de type low. De plus la règle $[C3]$ permet de garantir qu'une variable publique ne puisse être affectée que par des expressions publiques.

Le contexte d'exécution permet quand à lui de garantir l'absence de flux implicites. Les règles $[C5]$ et $[C6]$ permettent de garantir que lorsque le contexte est influencé par une variable privée, alors celui-ci est $[high]$. Dans ce cas les affectations dans les variables publiques sont interdites (règle $[C3]$).

Ainsi le programme suivant :

```
h1:=l1+4; l1:=l1-5;
```

où $h1$ est une variable de type $high$ et $l1$ est une variable de type low, peut-être typable dans n'importe quel contexte ([$high$] ou [low]). En effet la règle [$C2$] nous indique que h1:=l1+4; est typable dans n'importe quel contexte et la règle [$C3$] nous permet de typer l1:=l1−5; dans le contexte [low]. Enfin d'après la règle [$C4$] le programme est typable dans n'importe quel contexte car il est composé de deux commandes typables dans n'importe quel contexte.

Par contre le programme suivant :

```
if h1=1 then l1:=1 else skip end
```

qui contient un flux implicite d'information n'est pas typable car il contredit la règle [$C6$]. En effet $h1$ est une variable privée et influence la valeur de $l1$ qui elle est publique.

3.3.3 Application du contrôle de flux par typage

A la suite des travaux de Volpano et al. [VSI96], Myers [Mye99] a proposé une extension de Java, le langage JIF (*Java Information Flow*)[3]. C'est un langage de programmation plus réaliste. Il permet d'utiliser des variables typées dynamiquement. Il implémente les concepts d'objet, l'héritage ainsi que les exceptions. Ce langage implémentant des types de sécurité permet d'étendre le langage Java en lui permettant de mettre en œuvre à la fois une politique de contrôle de flux et une politique de contrôle d'accès. Ces politiques sont vérifiées au moment de la compilation et au cours de l'exécution. Le contrôle statique de flux d'information mis en œuvre par JIF permet de contrôler à la fois la confidentialité et l'intégrité des données.

JIF étend le langage Java en ajoutant des labels aux données qui expriment des restrictions sur l'utilisation qui peut-être faite de ces données. Par exemple la déclaration de variable suivante indique non seulement que x est un entier mais aussi la politique de sécurité associée à l'information contenue dans x :

```
int {Alice -> Bob} x;
```

Dans ce cas la politique de sécurité indique que le propriétaire de x est $Alice$ et que celui-ci autorise Bob à lire cette information. Le label $\{Alice \leftarrow Bob\}$ permet de contrôler l'intégrité [CVM07] en indiquant que $Alice$ en est la propriétaire et que seul Bob est autorisé à modifier l'information contenue dans x. Ces labels de sécurité permettent un contrôle de flux d'information au moment de la compilation basé sur les règles de typage proposées par [VSI96]. Après avoir vérifié l'absence de flux d'information illégaux au sein du programme, celui-ci est transformé par le compilateur JIF en un programme Java qui est compilé et exécuté par une machine Java classique.

3. http://www.cs.cornell.edu/jif

```
final label{L} x;
 Channel {*x} c;
 int {H} y;
 switch label(y) {
    case (int{*x} z) c.send(z) ;
    else throw new UnsafeTransfer();
 }
```

FIGURE 3.4 – Exemple d'utilisation de label dynamique dans JIF

Les auteurs de [ZM07] ont ajouté à JIF le support de labels dynamiques permettant d'utiliser des variables dont le type est découvert à l'exécution (en particulier lors de l'appel de fonctions). Dans JIF les labels peuvent-être utilisés comme une classe à part entière et une variable de type `label` peut-être utilisée comme label pour une autre valeur. JIF fournit l'instruction `switch label` permettant d'effectuer des tests sur les labels au cours de l'exécution. Le code présenté figure 3.4, issu de [ZM07], permet d'envoyer une valeur à travers un canal de communication à l'aide d'un label dynamique [4].

L'opération d'envoi `send` est sécurisée seulement si `x` est une variable de type H, ce qui est déterminé au cours de l'exécution. La notation `*x` se réfère à la valeur du label de la variable `x`. Dans cet exemple l'intruction `switch label` exécute le premier des cas si le label associé à `x` est au moins aussi restrictif que celui de `y`. La valeur de `y` est alors affectée à la variable correspondante (dans l'exemple `z`). Dans cet exemple, le premier cas n'est exécuté que si $H \sqsubseteq$`*x`, garantissant que `c` est un canal de communication sécurisé.

L'intérêt d'une telle approche est qu'elle facilite l'utilisation des mécanismes de sécurité du langage Java. Le langage JIF permet de construire un programme respectant le modèle de politique décentralisée proposé par Myers (section 3.2.2). Les règles sont clairement établies et une analyse statique vérifie que le programme ne contrevient pas à cette politique.

Les auteurs de [CVM07] ont développé SIF (Servlet Information Flow) un framework basé sur JIF permettant de développer des applications web respectant des propriétés d'intégrité et de confidentialité de bout en bout. L'utilisation de SIF permet de prévenir les flux d'information illégaux. Cependant le langage SIF est orienté vers la protection des applications et des serveurs en permettant de contrôler les flux d'information confidentiels vers les clients et en contrôlant les flux d'information non intègres depuis les clients. Du fait de la vérification statique des flux d'information, SIF ne permet pas au client de spécifier sa propre politique de confidentialité vis à vis de ses données.

Les auteurs de [LGV$^+$09] ont proposé Fabric une plate-forme basée sur JIF mettant en œuvre des composants distribués. A chaque nœud du système est asso-

4. Les valeurs H (privée) et L (publique) sont utilisées pour les labels afin de faciliter la lecture de l'exemple, dans la pratique ce sont les labels de JIF qui sont utilisés.

cié un nom représenté par une URL et un label sur le modèle de JIF. Ils distinguent trois types de nœud :

– les nœuds de stockage qui fournissent leurs données à la demande ;
– les nœuds de travail qui exécutent un programme, ils utilisent à la fois leurs objets mais aussi des copies d'objets des nœuds de stockage ou des autres nœuds de travail ;
– les nœuds de dissémination qui fournissent des copies des objets.

De la même manière que dans JIF, la politique de sécurité (couvrant à la fois la confidentialité et l'intégrité) est vérifiée de manière statique (quand les labels sont statiques) ou dynamique (quand ceux-ci sont connus à l'exécution). Cependant, tout comme dans JIF, la politique de sécurité est définie par les programmeurs et ne peut-être définie par les utilisateurs.

Ce langage a été utilisé pour écrire deux applications réelles. Les auteurs de [HAM06] ont développé JPMail un client e-mail permettant de garantir de bout en bout la confidentialité des échanges. Les auteurs de [CCM08] ont quant à eux développé une plate-forme de vote électronique. Ces deux expériences ont permis de montrer l'application pratique d'un langage comme JIF. Cependant elles ont aussi montré la difficulté à écrire un programme mettant en œuvre une politique de flux d'information.

De leur côté Simonet et Pottier, [Sim03] ont proposé le langage FlowCaml [5] et une analyse statique pour ce langage fonctionnel dérivé de OCaml. Cependant, le système de labels utilisé est beaucoup plus simple que celui de Myers. Les labels sont une chaîne de caractères définissant une classe de sécurité. Les flux d'une variable de la classe `!Alice` ne sont autorisés que vers des variables de la même classe de sécurité. Ce type de politique permet de spécifier une isolation entre les données appartenant à des classes de sécurité différentes.

Face aux limitations imposées par les types de labels utilisés dans JIF et FlowCaml, les auteurs de [SCH08] et [CSH09] ont proposé un langage formel Fable et son implémentation SELinks permettant de mettre en œuvre différentes politiques de sécurité indépendamment du modèle de label utilisé.

Le système de type de Fable [SCH08] permet de mettre en œuvre de nombreux types de politiques de sécurité, en particulier le contrôle de flux d'information, le contrôle d'accès et le contrôle de provenance. Les dernières versions de SELinks [CSH09] se sont attachées au développement d'applications web n-tiers et en particulier la mise en place d'un contrôle d'accès aux données stockées dans les bases de données. Le modèle choisi par SELinks permet la mise en œuvre et la combinaison de différentes politiques de sécurité. Cependant, celles-ci doivent être définies entièrement par le programmeur. Pour une politique de contrôle de flux celui-ci doit définir les classes de sécurité, les opérateurs d'union et d'intersection des labels ainsi que la relation d'ordre partiel entre les labels. SELinks constitue donc un framework intéressant permettant la vérification statique de nombreuses politiques de flux d'information au sein d'un programme.

5. `http://www.normalesup.org/ simonet/soft/flowcaml/`

3.3.4 Au niveau des orchestrations de service

Les auteurs de [RM09] proposent un cadre formel à l'étude des orchestrations de service. Ils proposent un langage formel dérivé du λ_{calcul} prenant en compte les actions qui interviennent entre les sujets (en particulier les services) et les variables utilisées dans les orchestrations de services. Ils mettent en œuvre une politique de flux multi-niveau où chaque variable et sujet est associé à une classe de sécurité. Leurs principaux résultats théoriques concernent l'étude des possibilités de remplacer un service par un autre service en préservant les propriétés de sécurité (en particulier une propriété de non interférence entre variables proche de la définition 3.3.1) afin de garantir dans tous les cas l'absence de flux d'information illicite. Leur travail reste à un niveau formel et aucune application pratique de leurs résultats n'est proposée.

L'auteur de [Nak04] propose une version étendue de BPEL dans laquelle il propose d'associer à chaque variable du programme un label représentant la classe de sécurité associée aux informations contenues dans cette variable. Les classes de sécurité SC employées forment un treillis fini. L'auteur développe le modèle de treillis employé et la définition de son extension du language BPEL cependant il ne donne que peu d'information sur sa façon de contrôler les flux d'information illicites. Il propose uniquement d'utiliser le contrôleur de modèle SPIN [6] et de ce fait propose une translation de BPEL vers Promela le langage utilisé par SPIN.

Les auteurs de [HV06] ont quant à eux proposé une analyse statique basée sur le système de type proposé par les auteurs de [VSI96]. Les types de sécurité employés sont dérivés des classes de sécurité issues de la composition de politiques de sécurité exposé à la section 3.2.1. Leur principal apport à ce système est la prise en compte des appels de service. Ils rajoutent de ce fait à un langage procédural très simple l'instruction `call(WS,` e_1`,`e_2`)` qui permet d'appeler un Web Service WS avec l'expression e_1, le résultat de l'appel étant l'expression e_2. Soit τ_1 le type de sécurité associé à e_1, τ_2 le type de sécurité associé à e_2 et τ' l'habilitation associée à WS. La règle de typage associée à l'instruction `call` assure non seulement que $\tau_1 \leq \tau'$ et que $\tau_2 \leq \tau'$ ce qui permet d'assurer que les informations manipulées par WS ne sont pas supérieures à son habilitation mais aussi que $\tau_1 \leq \tau_2$ ce qui permet d'assurer que le résultat e_2 de l'exécution de WS prend en compte le type de sécurité τ_1 des données qui ont servi à produire ce résultat. Les auteurs de [HV06] proposent donc un mécanisme qui permet à la fois aux services externes d'associer un type de sécurité aux données qu'ils renvoient mais aussi de s'assurer que le type de sécurité renvoyé prend en compte la politique de flux initiale.

Les auteurs de [ZA11] proposent de générer à partir d'un programme BPEL le graphe des dépendances correspondant de la même manière que les auteurs de [HKN06] pour le langage JAVA. Le graphe des dépendances est généré en deux étapes. La première étape consiste à réaliser un graphe ou chaque nœud représente une activité BPEL et les arcs orientés les séquences possibles d'activités. La seconde étape consiste à effectuer l'analyse des dépendances proprement dite en

6. `http://spinroot.com/spin/whatispin.html`

attribuant les variables du système aux activités. Cette analyse est basée sur les relations suivantes : les couples (Variable - Définition), (Variable - Utilisation) et (Définition, Utilisation). Un couple (Variable - Définition) est une opération d'affection vers une variable v dans une activité n_d et est représenté par un couple $def(v, n_d)$. Un couple (Variable - Utilisation) correspond à l'utilisation sans modification d'une variable v dans une activité n_u. Ils distinguent les dépendances de données (partie origine d'une affectation par exemple) et les dépendance de contrôle (expression de contrôle d'une conditionelle ou d'une boucle). Un couple (Définition, Utilisation) est un couple ordonné n_d, n_u où l'activité n_d contient la définition d'une variable v qui est utilisée dans une activité n_u du programme. Afin de finir la construction du graphe des dépendances un arc orienté du nœud n_d au nœud n_u est ajouté pour chaque couple n_d, n_u.

Une fois le graphe des dépendances construit, un label est affecté à chaque variable d'entrée suivant le modèle de label décentralisé de Myers (section 3.2.2) où les utilisateurs sont remplacés par les services. Le graphe des dépendances permet ensuite de calculer le label des variables utilisées par les appels de services et de vérifier que ceux-ci sont autorisés par le label ainsi calculé.

3.4 Suivi dynamique des flux d'information

3.4.1 Au niveau des programmes

Les analyses dynamiques vérifient au cours de l'exécution d'un programme que celui-ci vérifie certaines propriétés. Certaines analyses vont modifier l'exécution d'un programme afin que l'exécution de celui-ci garantisse certaines propriétés, en particulier l'absence de flux d'information illégaux au regard de la politique de flux. D'autres analyses vont suivre les flux d'information afin de signaler tout flux d'information illégal mais sans modifier l'exécution du programme. Néanmoins les deux types d'analyse ont les mêmes fondements théoriques.

De nombreuses études cherchent à comparer les approches statiques et dynamiques pour le contrôle des flux d'information. Russo et Sabelfeld ont proposé un état de l'art des recherches dans ce domaine ([RS10]) distinguant en particulier les analyses dynamiques qui associent des étiquettes de sécurité fixes aux variables et celles qui associent des étiquettes de sécurité variables.

Les auteurs de [SR10] ont proposé une analyse dynamique inspirée des travaux de Fenton [Fen74] permettant de suivre les flux d'information à la manière de Denning [Den76]. Ils suivent les flux d'information à l'aide d'un moniteur qui peut autoriser, modifier ou interdire l'exécution de chaque instruction. Un label est associé à chaque variable du programme. Ils considèrent une politique de sécurité à deux niveaux public et secret[7]. Les labels associés à chaque variable sont statiques. Les flux directs de la forme l:=h sont prévenus en interdisant l'exécution des affectations de données provenant de variables secrètes vers des variables

7. Nous noterons dans la suite l (low) les variables publiques et h (high) les variables secrètes

publiques. Les flux implicites sont prévenus en interdisant l'exécution des affectations vers des variables publiques dans des boucles ou des conditionnelles dépendant de variables secrètes. Leur détection est effectuée à l'aide des états du moniteur. Ces états sont formés par une pile de niveaux de sécurité. A chaque ouverture d'une conditionnelle le label résultant de l'expression conditionnelle est ajouté à la pile. A la fermeture de la conditionnelle un label est ôté à la pile. Si la pile contient un label $secret$ alors les instructions exécutées dépendent d'une variable privée. Dans l'exemple suivant **if** h then l:=1 **else** l:=0 un label *secret* est ajouté à la pile lors du calcul de l'expression conditionnelle. Les affections vers des variables publiques étant supprimées les affectations l:=1 et l:=0 ne sont pas exécutées empêchant ainsi les flux implicites. Les flux indirects implicites de la forme **if** h then l:=1 **else** skip sont également prévenus. En effet quelle que soit la valeur de h aucune affectation n'est effectuée et dans les deux cas aucune opération n'est effectuée. Les auteurs de [SR10] ont montré que leur analyse est aussi précise qu'une analyse statique dans le style de Denning, c'est à dire que l'ensemble des programmes autorisés par l'analyse statique sont exécutés sans modification par leur moniteur. Les programmes violant la politique de flux sont, quant à eux, exécutés de façon à garantir l'absence de flux illicites ; l'exécution des commandes provoquant des flux illicites étant interdite.

Néanmoins les analyses dynamiques basées sur des labels statiques ne peuvent permettre à l'utilisateur d'un programme de définir sa propre politique de sécurité, celle-ci étant définie par le programmeur lors de l'écriture du programme. La définition d'une politique de sécurité par l'utilisateur d'un programme nécessite l'utilisation de labels dynamiques. Cependant une analyse dynamique ne peut contrôler les flux implicites indirects de la forme **if** h then l:=1 **else** skip si les labels sont dynamiques. En effet si h est différent de zéro alors la variable l devient secrète alors que sinon celle-ci n'est pas modifiée et de ce fait reste publique. L'observation du label de l permet donc de déduire si h est ou non différent de zéro. Il est alors impossible d'empêcher ce flux indirect implicite par une analyse purement dynamique utilisant des labels dynamiques comme l'ont montré les travaux de [Sch00].

Afin de résoudre ce problème de nombreux auteurs comme par exemple ceux de [SST07, GBJS06, NJK^{+}07, VXDS06] ont proposé des combinaisons d'analyses dynamiques et statiques. Ainsi les auteurs de [GBJS06] ont proposé une analyse dynamique associée à une analyse statique permettant de prendre en compte les flux indirects implicites en utilisant des labels dynamiques.

Le langage considéré est un langage « While » augmenté d'une sortie générique **output**(e). Les valeurs des variables ne sont jamais directement accessibles, même à la fin d'une exécution. Une valeur ne devient disponible - et donc publique - que via l'exécution d'une sortie. Assurer l'absence de flux d'information illégaux pour un tel programme revient donc à s'assurer que les sorties ne dépendent pas des variables privées initiales. L'automate de contrôle a comme responsabilité de traquer la variété (c'est à dire la dépendance d'une variable à une valeur secrète) et de prévenir la transmission de la variété à la séquence de sortie. Afin de traquer la

variété, les états de l'automate sont une paire (V, w). V est l'ensemble des variables ayant une variété, au départ $V = H$ (ensemble des variables secrètes). w est un mot de $(\bot, \top)^*$ qui traque la variété dans le contexte d'exécution. Celui-ci peut-être vu comme une pile, un élément étant ajouté au début de chaque conditionelle ($\top$ si la conditionelle dépend de H, $\bot$ sinon), et enlevé à la fin de l'exécution d'une branche.

Les flux explicites sont gérés à l'aide de l'ensemble V (ensemble des variables dépendant de H). Ils apparaissent lorsqu'une affectation ou une sortie de variable privée est effectuée dans un contexte public. A chaque fois qu'une affectation $x := e$ est effectuée, l'automate vérifie qu'aucune des variables de e n'appartient à V, dans le cas contraire x est ajouté à V. Les sorties **output**(e) sont autorisées si aucune variable de e n'appartient à V, sinon celles-ci sont remplacées par la sortie d'une valeur par défaut θ. Les flux implicites directs apparaissent lorsque le contexte d'exécution dépend d'une variable privée, c'est à dire si $\top$ appartient à w. Dans ce cas toutes les variables modifiées sont considérées comme privées et sont ajoutées à V. Les flux implicites indirects sont liés aux branches non exécutées d'une conditionnelle dépendant d'une variable privée. Afin d'éviter que les sorties ne révèlent le passage dans l'une ou l'autre branche, l'automate interdit l'exécution des sorties dans un contexte $\top$. De plus une analyse statique est rajoutée à la fin de l'exécution d'une conditionnelle dans un contexte $\top$ afin d'ajouter l'ensemble des variables modifiées dans la branche non exécutée à l'ensemble des variables privées, c'est à dire V. Ainsi dans l'exemple **if** h then l:=1 **else** skip la variable l est ajoutée à l'ensemble V quelle que soit la valeur de h.

Mise en application des analyses dynamiques de programme

Plusieurs classes de solutions ont été proposées afin de suivre dynamiquement les flux d'information au sein d'un programme. La figure 3.5 [8] présente trois grandes classes possibles d'analyseur :

- la première classe d'analyseur est implémentée sous la forme d'un **moniteur externe**. Celui-ci est implémenté au niveau de l'environnement d'exécution (système d'exploitation, machine virtuelle,...) et permet d'observer les accès aux conteneurs d'information. Le principal avantage de ce type d'analyseur est qu'il permet l'exécution des applications sans aucune modification ;
- l'**encapsulation** permet de suivre les flux d'information en modifiant les bibliothèques système en y ajoutant un ensemble de fonctions permettant le suivi des flux d'informations. Cette méthode ne nécessite aucune modification du système d'exploitation ou de la machine virtuelle. Elle permet de la même manière qu'un moniteur externe d'exécuter les programmes sans modification. Cependant les fonctions ajoutées aux bibliothèques système ne permettent pas de prendre en compte l'intégralité des flux d'informations, les bibliothèques systèmes ne constituant pas l'unique moyen d'échanger

8. Cette figure est inspirée de celle proposée dans la thèse de Guillaume Hiet [Hie08]

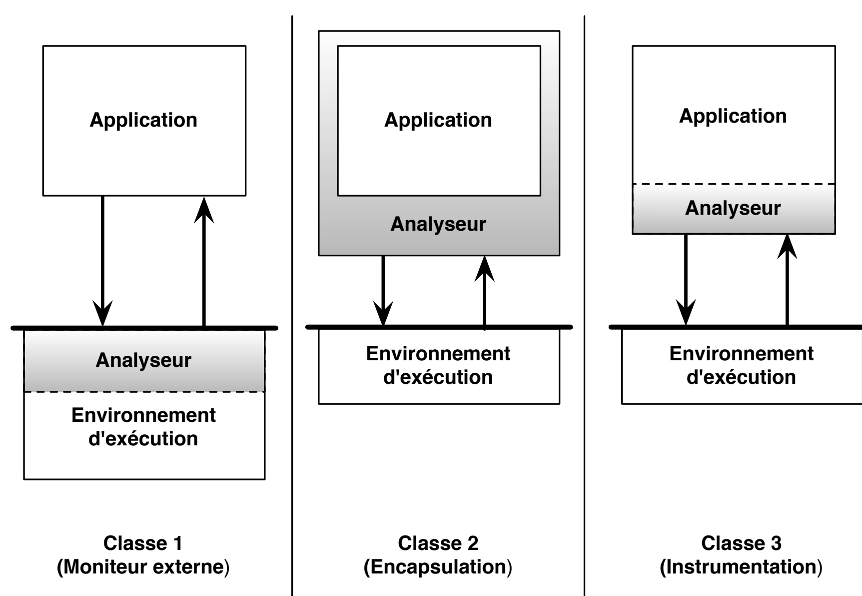

FIGURE 3.5 – Classes de solutions pour le suivi dynamique des flux d'informations

des informations entre deux conteneurs ;
- l'**instrumentation** consiste à ajouter des instructions permettant le suivi des flux d'informations. Ces instructions sont, soit ajoutées manuellement par le programmeur, soit tissées lors de la compilation. Cette analyse est réalisée soit au niveau du code source, soit au niveau du *bytecode*. Cette implémentation au plus près du code source permet en particulier de contrôler l'accès aux différentes variables du programme.

Instrumentation Les auteurs de [LC06] proposent une instrumentation du langage C permettant de suivre dynamiquement les flux d'information. Leur framework GIFT [9] permet au programmeur de spécifier les canaux à surveiller. Le suivi des flux s'effectue au niveau des variables auxquelles sont associées un label sous la forme d'un entier. Les canaux d'entrée qui permettent de recevoir des informations (typiquement les fonctions `read()`) sont remplacés au cours de la compilation par leur équivalent permettant d'initialiser les labels. Les canaux de sorties sont de la même manière remplacés au cours de la compilation par des appels permettant la vérification des labels et permettant d'interdire le cas échéant l'exécution de l'instruction. Le compilateur ajoute après chaque affectation l'appel d'une fonction permettant de modifier la valeur du label de la variable modifiée. Des fonctions supplémentaires permettent au programmeur de suivre les flux indirects en ajoutant le label des variables de l'expression conditionnelle dans une pile.

9. *(General Dynamic Information-Flow Tracking Framework)*

Les auteurs de [HCF05a] proposent, quant à eux, une instrumentation du *bytecode* Java permettant de suivre dynamiquement les flux d'information en labélisant les variables utilisées dans le *bytecode* Java. Cependant, leur analyse ne permet de suivre que les flux directs. Dans [HCF05b], les mêmes auteurs proposent d'utiliser un contrôle d'accès obligatoire sur les données afin d'initialiser la politique de flux. Dans [Fra06], Franz étend ce mécanisme afin de prendre en compte les flux indirects explicites. De la même façon que [HCF05a], les auteurs de [CF07] ont proposé une instrumentation du *bytecode* Java qui permet de prendre en compte les flux directs et indirects. Les labels sont des entiers de 32 bits. Leur analyse combine une analyse dynamique qui prend en compte les flux explicites et une analyse statique rajoutant des annotations permettant de prendre en compte les flux implicites. Ils instrumentent ainsi tous les objets afin d'ajouter un champ contenant le label de l'objet et l'ensemble des constructeurs afin d'initialiser les labels. L'implémentation proposée ne permet pas à l'utilisateur de spécifier lui-même les labels associés aux informations initiales, ceux-ci étant définis par le programmeur. Leur analyse repose sur un compteur de programme pc permettant de suivre les flux indirects. Lors des appels de méthodes, ils transmettent à la fois la valeur de pc ainsi que les labels des différents paramètres et transmettent les labels des valeurs de sorties à la fin de l'exécution de la méthode. Lors de l'initialisation d'une méthode, ils créent un label pour l'ensemble des variables locales et initialisent les labels de différents paramètres. Les affectations sont englobées dans une conditionnelle qui permet de vérifier si l'affectation est vérifiée par la politique de flux représentée par les labels.

Encapsulation Les auteurs de [SRMM11] proposent une bibliothèque de contrôle de flux d'information pour Haskell, un langage fonctionnel. Ils remplacent en particulier les instructions standard d'entrée/sortie par des instructions permettant la labelisation des données et le suivi des flux d'informations. Leur bibliothèque permet d'utiliser différents types de labels en définissant la sémantique des labels ainsi que les règles de fusion et d'intersection. Leur bibliothèque fournit en particulier les fonctions suivantes :

- la labélisation qui permet d'associer un label l à une valeur v ;
- l'opération inverse, la délabélisation, qui permet de renvoyer la valeur v sans le label ;
- l'opération qui permet de retourner le label d'une valeur ;
- et enfin une opération qui permet d'encapsuler le résultat de l'appel d'une fonction avec un label.

Ces différentes opérations permettent d'implémenter le suivi des flux d'information dans un programme Haskell et d'empêcher si nécessaire l'exécution d'une action afin de garantir le respect des propriétés de sécurité.

Moniteur externe Les auteurs de [VBC$^+$04] proposent RIFLE une proposition d'architecture pour un processeur permettant le suivi dynamique des flux d'information. Ce processeur utilise des registres mémoires modifiés permettant l'ajout

de label de sécurité à chacun des registres mémoires. Ce processeur permet ainsi de suivre les flux directs. Chaque instruction est modifiée afin de permettre la propagation des labels. Lorsqu'une exception à la politique de flux est détectée l'instruction n'est pas exécutée. Leur moniteur ne permettant de prendre en compte que les flux directs, ils proposent une analyse statique permettant de transformer un programme compilé en un autre programme compilé qui ne comporte que des flux d'informations directs.

Les auteurs de [NSCT08] proposent Trishul une machine virtuelle Java modifiée permettant de suivre dynamiquement les flux d'information. Leur analyse est effectuée au niveau du bytecode Java. Trishul permet de suivre à la fois les flux directs et les flux indirects en utilisant une combinaison d'analyses statiques et dynamiques sur le bytecode Java. Un label est associé à chaque variable. L'implémentation de ces labels est effectuée en modifiant la structure de la pile des variables. La pile d'exécution est modifiée afin de prendre en compte le contexte dans lequel est exécuté une instruction permettant ainsi la prise en compte des flux indirects. Une analyse statique permet de labéliser les variables modifiées dans les branches non exécutées. Leur analyse prend en compte également les exceptions spécifiées dans les programmes Java. Celles-ci sont traitées de la même manière que les conditionnelles. Afin de garantir le respect de la politique de flux, les instructions violant cette politique ne sont pas exécutées. Les auteurs de [NSCT08] décrivent l'implémentation de leur solution mais ne proposent pas de politique de flux particulière, celle-ci étant définie par le programmeur de la machine virtuelle.

Les auteurs de [YYW$^+$07] proposent une combinaison entre un moniteur implémenté dans une machine virtuelle Java et une instrumentation du *bytecode* Java permettant aux programmes Java de dialoguer avec le moniteur. Leur moniteur permet de contrôler les flux d'information dans une application web n-tiers. De ce fait ils proposent également une base de données modifiée permettant de labeliser l'ensemble des données contenues dans cette base. Leur approche de l'instrumentation est similaire aux travaux de [HCF05a, HCF05b, Fra06] cependant leur analyse couvre un nombre plus important d'instructions.

3.4.2 Au niveau des systèmes d'exploitation

Les premiers systèmes permettant de contrôler les flux d'informations au sein d'un système d'exploitation se sont inspirés du modèle de contrôle d'accès obligatoire de Bell et LaPadula présenté section 3.2.1. Les systèmes modernes SELinux [10], TrustedBSD [11] ou Solaris Trusted Extensions [12] implémentent ce modèle d'accès. Cependant afin de garantir un haut niveau de sécurité, ces systèmes utilisent des labels statiques afin de spécifier la politique de sécurité associée aux données manipulées. Ce sont donc des systèmes peu souples et de ce fait peu employés hors du monde militaire.

10. http ://www.nsa.gov/selinux/
11. http ://www.trustedbsd.org
12. http ://hub.opensolaris.org/bin/view/Community+Group+security/tx

Les travaux de Weissman [Wei69] ont été les premiers travaux à proposer un modèle de système d'exploitation utilisant des labels dynamiques. Néanmoins comme l'ont montré les auteurs de [MR92] et [FGQ96] ces labels dynamiques peuvent constituer un canal caché. Si les auteurs de [MR92] considèrent que cette fuite d'information est tolérable pour un usage non militaire, les auteurs de [FGQ96] proposent des restrictions d'emploi des labels permettant de limiter l'emploi de ce canal caché potentiel.

Les auteurs de [EKV+05] et [ZBWKM06] ont proposé Asbestos et Histar deux systèmes d'exploitation permettant de contrôler nativement les flux d'informations. Si Asbestos est constitué uniquement d'un micro-noyau, Histar est un système plus complet. Ces deux systèmes implémentent un même modèle de contrôle de flux issu d'une évolution du modèle décentralisé présenté dans la section 3.2.2. Ces systèmes implémentent nativement le contrôle de flux d'information et présentent de ce fait des performances intéressantes. Cela permet également de suivre au plus près les flux d'information. Cependant ces systèmes spécifiques nécessitent de re-compiler, voir réécrire l'ensemble des applications.

Les auteurs de [ZMB03] ont quand à eux proposé Blare, un moniteur de suivi des flux d'information pour un système Linux. Ce moniteur est basé sur le modèle présenté section 3.2.3. Les auteurs de [HTMM09] ont, quant à eux, présenté une extension de Blare permettant de suivre les flux d'informations à l'intérieur des applications Java permettant ainsi un suivi de bout en bout de la politique de sécurité et une réduction des sur-approximations. En effet le modèle original de Blare considère l'ensemble des processus comme des boîtes noires et de ce fait considère au même niveau de sensibilité l'ensemble des données manipulées par un processus.

Les auteurs de [KYB+07] ont proposé Flume un moniteur de suivi des flux d'informations pour un système Linux. Ce moniteur est basé sur le modèle de label décentralisé présenté section 3.2.2. Les auteurs de [RPB+09] ont proposé de suivre la même approche que les auteurs de [HTMM09] mais utilisent le modèle de contrôle de flux d'information de Flume.

3.5 Modification de la politique de sécurité

Dans sa définition la plus stricte, une politique de flux est souvent très restrictive. Ainsi un flux est le plus souvent soit autorisé soit interdit. Cependant il est parfois nécessaire d'autoriser certains flux d'informations afin de permettre le fonctionnement normal d'un programme. L'exemple le plus souvent cité dans ce cas est celui du mot de passe. Un mot de passe en lui-même est une information secrète. Cependant il est nécessaire pour assurer le bon fonctionnement d'une authentification d'indiquer si le mot de passe rentré par un utilisateur est le bon même si cette information est dérivée du mot de passe et devrait donc être secrète. Un autre exemple est celui d'une enquête statistique dans laquelle on peut désirer utiliser la moyenne des salaires des employés d'une entreprise sans pour autant révéler

les salaires individuels. Cependant cette moyenne est calculée à partir des salaires individuels. Il apparaît donc comme nécessaire de pouvoir spécifier des fuites de certaines informations afin de réaliser des applications réelles. Mais cet exemple nous montre que la mise en place de ces mécanismes doit s'accompagner d'une politique de sécurité. Dans ce cas on veut autoriser la divulgation uniquement de la moyenne, en s'assurant que les salaires individuels ne sont pas rendus publics.

Ce problème est appelé déclassification dans de nombreux travaux. Il consiste en effet à autoriser la modification d'une classe de sécurité d'un objet afin de rendre sa politique de sécurité moins restrictive. Le problème de la déclassification a fait l'objet de nombreux travaux que Sabelfeld et Sands [SS07] ont proposé de classer suivant quatre directions : quelles informations peut-on déclassifier, qui peut déclassifier, où peut-on déclassifier, quand peut-on déclassifier. De plus amples informations sur le problème de la déclassification peuvent être trouvées dans leur étude qui couvre de nombreux travaux. Nous ajouterons à notre étude une autre dimension souvent utilisée : comment sont effectuées les déclassifications.

3.5.1 Comment sont effectuées les déclassifications

Dans de nombreux travaux sur la déclassification, les opérations de déclassification sont spécifiées au niveau du code source par le programmeur. Ainsi les auteurs de JIF [Mye99] définissent par exemple une fonction de déclassification qui permet de modifier explicitement le label associé à une variable. Une analyse statique permet alors de garantir que le logiciel respecte bien la politique de sécurité mise en place.

Cependant l'utilisation d'une opération de déclassification nécessite d'avoir un langage de programmation dédié permettant la mise en place d'une politique de sécurité. La majorité des langages actuels de programmation devraient alors être réécrits selon l'exemple de JIF. La définition de la politique de sécurité est alors faite par le programmeur et vérifiée de manière statique avant l'exécution du programme.

Une autre approche utilisée par les auteurs de [YYW+07] et [HCF05a] consiste à définir un ensemble de fonctions dont le résultat peut-être rendu public.

3.5.2 Qui peut déclassifier ?

Si la question de savoir qui peut déclassifier une information est d'abord liée aux politiques de sécurité, la réponse à cette question influe néanmoins sur le choix des analyses employées afin de garantir la sécurité du logiciel.

Si les déclassifications sont autorisées par l'utilisateur du logiciel, une vérification dynamique peut-être envisagée. Dans ce cas les déclassifications peuvent être vérifiées au cours de l'exécution du logiciel.

Si le programmeur est directement à l'origine des déclassifications alors une analyse statique permet de garantir que le logiciel respecte bien la politique de sécurité mise en place par le programmeur.

Myers et Lyskov ont proposé dans [ML97], un modèle basé sur des labels de sécurité avec mention explicite du propriétaire de l'information. Cette approche a été mise en place dans le langage JIF [Mye99]. Le propriétaire d'une information est défini au niveau du code. Cette entité est également propriétaire des fonctions. Ainsi une variable ne peut être déclassifiée que via une fonction appartenant à son propriétaire. Dans ce cas déclassifier consiste à ajouter un lecteur autorisé à une information.

A la manière de JIF, les auteurs de Flume [KYB+07] et d'Histar [ZBWKM06] autorisent les propriétaires d'une information à ajouter un ou plusieurs lecteurs autorisés permettant ainsi de déclassifier certaines informations.

3.5.3 Quelles informations peut-on déclassifier ?

Afin de spécifier quelles informations peuvent être déclassifiées, plusieurs approches ont été proposées. Les premières approches se sont intéressées à la quantité d'information révélée. Ces approches se réfèrent souvent à la théorie de l'information. Ainsi par exemple la connaissance de la parité d'un nombre ne permet pas de remonter au nombre initial, il permet juste de diviser par deux le nombre de possibilités différentes. Une information de ce type peut donc être déclassifiée.

Les auteurs de [HCF05a] et [YYW+07] proposent une approche similaire de la déclassification. Ainsi les auteurs de [HCF05a] considèrent certaines procédures comme permettant de déclassifier comme par exemple le résultat d'un test avec une expression régulière. En effet le résultat du test avec une expression régulière renvoie un booléen, c'est à dire une valeur sur un bit permettant difficilement de remonter à l'information initiale. Ces procédures sont déterminées en fonction de la quantité d'information qu'elles laissent fuire à chaque exécution. Les auteurs de [YYW+07] quant à eux spécifient directement la politique de déclassification au niveau des API Java. Par exemple si il existe une méthode de l'API qui masque un numéro de carte bancaire à l'exception des quatre derniers chiffres, la politique de labélisation de cette méthode peut-être modifiée afin de rendre automatiquement public le résultat de cette méthode et permettre ainsi des déclassifications sans modification du code des applications.

Cependant cette approche par quantité d'information présente des limites. Considérons l'exemple suivant écrit en Java.

```
for (int i=0;i<32;i++)
        {l:=(h%2==1);
         System.out.println(l);
         h>>>1;}
```

A chaque étape seul un booléen est déclassifié. Cependant celui-ci ne porte que sur l'observation du bit de poids faible de h. En effectuant une opération de décalage à droite dans une boucle, on déclassifie ainsi la valeur complète de h uniquement à partir d'une opération autorisée (le résultat de la parité d'une variable). Dans cette approche une information doit être considérée comme déclassifiée complètement à partir d'un certain nombre de déclassification.

Une autre approche consiste à utiliser une fonction de déclassification. Cette approche permet au programmeur de spécifier directement quelles variables il souhaite déclassifier. L'analyse statique qui est ensuite mise en place vérifie que les variables déclassifiées ne violent pas la politique de sécurité. Par exemple dans [SM04], la *Delimited Release* vérifie que les variables déclassifiées ne dépendent pas d'autres variables privées qui elles ne sont pas déclassifiées. Li et Zdancevic [LZ05] proposent d'utiliser un treillis de sécurité défini par des fonctions. Ainsi ce ne sont plus des variables qui sont directement déclassifiées, mais le résultat de l'évaluation d'une variable par une fonction.

Les deux approches que nous avons présentées cherchent à quantifier ou spécifier les exceptions possibles à une politique de sécurité. Cependant ces propriétés sont définies avant l'exécution du programme et ne permettent pas de savoir explicitement les informations qui ont été déclassifiées au cours de l'exécution du programme et vers qui elles ont été envoyées. Elles permettent uniquement de savoir quelle quantité d'information a été déclassifiée par rapport à la taille de l'information initiale (approche par quantité d'information) ou quelles informations (ou expressions) ont été autorisées à être déclassifiées. Il n'y a aucun moyen de savoir si ces déclassifications ont été effectuées, vers qui les informations ont été envoyées et à quel moment les déclassifications ont été effectuées.

De plus la plupart du temps la déclassification est vue comme le passage d'un niveau haut de confidentialité vers un niveau plus faible. Elle ne permet pas de définir spécifiquement le destinataire d'une information confidentielle.

Askarov et Sabelfeld [AS07] ont montré les limites des définitions de la déclassification qui s'intéressent uniquement aux variables déclassifiées. En effet les deux exemples suivant ne sont pas différenciés dans ce type de définition.

```
l:=declassify(h);  l:=h;
l:=h;  l:=declassify(h);
```

En effet la commande **declassify**(h) implique que h devient publique. Donc le premier programme est considéré comme sécurisé. Le deuxième quand à lui n'est pas capturé par les définitions qui ne s'intéressent qu'à cette dimension. En effet même si la déclassification est spécifiée dans le code, la politique de sécurité « quoi » considère que h peut être déclassifiée et donc passer de privée à publique. Ces politiques comparent les états initiaux et finaux de la mémoire et non à l'état de la mémoire au cours de l'exécution. Il est donc nécessaire de s'intéresser également à « où » les déclassifications sont effectuées.

3.5.4 Où peut-on déclassifier ?

S'intéresser à "où" sont effectuées les déclassifications conduit d'abord à s'intéresser à l'endroit où est spécifiée la politique de déclassification. Sabelfeld et Sands [SS07] distinguent deux approches. Certaines déclassifications se font directement au niveau du système d'exploitation (cette approche permet un contrôle plus fin de la déclassification par le propriétaire de l'information), alors que d'autres se

font au niveau du code source (cette deuxième approche permet de développer plus facilement des programmes permettant de déclassifier certaines informations mais en contrepartie l'utilisateur contrôle plus difficilement quelles informations sont déclassifiées).

Lorsque les déclassifications sont spécifiées au niveau du code, une opération de déclassification est souvent ajoutée au langage, comme par exemple dans [MS04]. Ceci permet de rendre explicites ces opérations. Cependant plusieurs implémentations sont possibles :

- l:= **declassify** (h). l reçoit la valeur de la variable h qui reste privée
- **declassify** (h,low). la variable h devient publique (mais est-elle considérée comme publique dans tout le programme ou bien seulement à partir du moment où la fonction de déclassification a été effectuée ?)

La dimension « où » ne peut se suffire à elle-même car elle ne spécifie pas quelles informations peuvent être déclassifiées. Mantel et Reinhard [MR07] considèrent deux analyses statiques successives afin de prendre en compte les dimensions « quoi » et « où ». Dans [AS07] Askarov et Sabelfeld proposent une politique de sécurité qui combine à la fois les dimensions « quoi » et « où » de la déclassification. Leur propriété de sécurité, appelée *Delimited Localized Release*, garantit qu'une variable n'est considérée comme publique qu'après que celle-ci ait été déclassifiée, mais également que le programme respecte la *Delimited Release* [SM04].

3.5.5 Quand peut-on déclassifier ?

La dimension temporelle du problème est vue sous trois aspects par Sabelfeld et Sands [SS07] : la complexité du problème, la probabilité de perte d'information, ou la réalisation d'une condition.

Volpano et Smith [VS00, Vol00] ont considéré des fonctions non-réversibles. Dans ce cas un attaquant ne doit pas pouvoir lire la valeur complète d'un secret de taille n en un temps polynomial.

Dans le cadre d'exécutions parallèles, Smith propose une notion de non-interférence approximative [Smi07]. Une application est considérée comme sécurisée si la probabilité qu'un attaquant fasse la distinction entre deux exécutions d'une même application, ayant en entrée des variables sécurisées différentes, est plus petite qu'une constante ε.

La politique de sécurité de Chong et Myers ([CM04]) s'intéresse à « quand » l'information peut-être déclassifiée. Ceci est réalisé en annotant les variables avec des types de la forme $l_0 \overset{c_1}{\rightsquigarrow} ... \overset{c_k}{\rightsquigarrow} l_k$, ce qui signifie intuitivement qu'une variable avec de telles annotations peut-être déclassifiée successivement aux niveaux l_1,...,l_k si les conditions c_1,...,c_k sont vérifiées à l'exécution des différents points de déclassification. Cette politique peut-être utilisée dans le cadre du commerce en ligne, par exemple la clé d'un logiciel n'est délivrée qu'après la confirmation du paiement de celui-ci.

3.6 Bilan

L'objectif du contrôle de flux d'information est de contrôler la diffusion de l'information au cours de l'exécution d'un programme ou d'un système. Une politique de flux d'information définit les flux d'information autorisés ou interdits. Une violation de la politique de flux est détectée lorsqu'un utilisateur accède à tout ou partie d'une information à laquelle il ne devrait pas accéder selon cette politique.

Le modèle de politique de contrôle de flux d'information le plus utilisé est le modèle en treillis. Dans ce modèle les flux d'informations autorisés sont faciles à décrire. Cependant ce modèle ne permet pas de décrire finement une politique de flux d'information. Par exemple dans le cadre des architectures orientées services il ne permet pas pour une information donnée de décrire spécifiquement vers quel service celle-ci peut-être envoyée.

D'autres modèles de politique de contrôle de flux d'information ont été proposés comme par exemple le modèle du système Blare. Ces modèles permettent de décrire une politique de contrôle de flux d'information avec une granularité beaucoup plus fine. Cependant il est beaucoup plus difficile de décrire une politique de contrôle de flux d'information dans le cadre de ces modèles. En particulier l'initialisation de ces politiques nécessite de décrire un très grand nombre de flux. Pour pallier ces difficultés certaines méthodes ont été proposées comme par exemple l'initialisation d'une politique de contrôle de flux d'information à partir de la politique de contrôle d'accès.

Une politique de flux d'information non complète va créer des flux d'information illégitimes au regard de la politique de contrôle de flux d'information mais qui peuvent être légitimes du point de vue du propriétaire des informations ou nécessaires afin de permettre le bon fonctionnement du programme. Il est donc parfois nécessaire de modifier la politique de contrôle de flux d'information ou de spécifier des exceptions à cette politique. Ce problème est appelé déclassification dans de nombreux travaux et peut-être étudié suivant cinq axes : comment sont effectuées les déclassifications, qui peut déclassifier, quelles informations peut-on déclassifier, où peut-on déclassifier.

Dans le cadre de cette thèse nous proposons donc un modèle de politique de contrôle de flux d'information inspiré du modèle de politique décentralisé proposé par Myers qui permet de décrire pour l'ensemble des informations envoyées à une orchestration de service vers quels services ces informations peuvent être envoyées. Cette politique est spécifiée par l'utilisateur de l'orchestration de services. L'initialisation de cette politique n'étant pas toujours possible à l'invocation de l'orchestration de services nous avons en particulier étudié dans ce modèle la modification dynamique de la politique de contrôle de flux d'information à partir des cinq axes de la déclassification. Ce modèle de politique de contrôle de flux d'information pour les orchestrations de services est présenté dans le chapitre 4.

Deux grande techniques permettant l'analyse de programmes sont principalement utilisées afin de contrôler les flux d'information : les analyses statiques et les

analyses dynamiques.

L'analyse statique permet de certifier que l'ensemble des exécutions d'un programme respecte une politique de contrôle de flux d'information. Cependant l'analyse statique nécessite le plus souvent la réécriture des programmes afin que ceux-ci puissent être certifiés. Cette réécriture est même parfois impossible car l'exécution correcte d'un programme peut nécessiter des violations de la politique de contrôle de flux d'information.

L'utilisation de programmes non certifiés est donc souvent nécessaire. L'analyse dynamique permet de garantir qu'une exécution donnée ne produit pas de flux d'information non autorisé par la politique de contrôle de flux d'information.

Néanmoins une analyse dynamique ne permet pas de prendre en compte l'ensemble des flux d'informations possibles. En particulier les flux implicites indirects ne peuvent pas être pris en compte par les analyse purement dynamiques. Cette approche introduit également une surcharge lors de l'exécution du programme.

L'analyse dynamique se révèle cependant indispensable au contrôle des flux d'information en particulier si la politique est dynamique. Dans le cadre de cette thèse on distingue en particulier trois cas qui rendent l'analyse dynamique indispensable : la politique de contrôle de flux d'information est décidée par l'utilisateur, les services sont découverts dynamiquement ou la politique de contrôle de flux d'information est modifiée dynamiquement.

La politique de contrôle de flux ne peut dans la plupart des cas être décidée au moment de la programmation d'un logiciel. C'est l'utilisateur qui va décider du cadre d'emploi du logiciel et de ce fait de la politique de contrôle de flux d'information qu'il voudrait appliquer aux données qu'il utilise avec ce logiciel. Dans le cadre des architectures orientées services et en particulier des orchestrations de services cela est d'autant plus vrai que les services ont été conçus comme des briques logicielles librement disponibles et réutilisables. De ce fait les services peuvent être utilisés par des utilisateurs ayant des besoins en confidentialité différents.

Une autre contrainte s'ajoute au cadre des architectures orientées services : les services peuvent être découverts dynamiquement au moment de l'exécution. De ce fait le programmeur n'a aucune connaissance des services qui seront effectivement utilisés au cours de l'orchestration de services et ne peut de ce fait proposer une politique de contrôle de flux d'information. Celle-ci ne peut être décidée qu'au moment de l'exécution, imposant de ce fait un contrôle dynamique des flux d'information.

Enfin la découverte dynamique des services ne permet pas à l'utilisateur de définir la politique de contrôle de flux d'information à l'invocation du service. De ce fait une modification dynamique de la politique de contrôle de flux d'information est le plus souvent nécessaire afin de prendre en compte les services découverts au cours de l'exécution de l'orchestration.

Après avoir étudié les flux d'informations se produisant dans le langage BPEL au chapitre 5 nous proposons dans le chapitre 6 un contrôle dynamique des flux d'information pour le langage BPEL pemettant de mettre en œuvre la politique de contrôle de flux d'information présentée au chapitre 4. Cette implémentation per-

met notamment de mettre en œuvre une politique de contrôle de flux d'information définie par l'utilisateur. Cette implémentation prend en compte la découverte dynamique de services et permet notament de modifier dynamiquement la politique de contrôle de flux d'information.

Chapitre 4

Politique de flux d’information pour les orchestrations de service

4.1 Modèle du système

Nous considérons dans cette partie une architecture orientée service (Service Oriented Architecture (SOA)) basée sur des orchestrations. Les différentes composantes de l’architecture considérée sont présentées sur la figure 4.1.

Dans notre architecture les ressources logicielles sont organisées sous la forme de **services**. Les services sont des modules indépendants les uns des autres vis-à-vis de leur état ou de leur contexte d’exécution. Ils fournissent un ensemble de fonctions qui reçoivent des **messages** et fournissent des réponses après traitement. Les messages sont porteurs d’une ou plusieurs **informations**. Les services sont appelés par des **clients**. Les clients sont soit d’autres services, soit un logiciel, soit un humain.

Les différents services sont proposés par un ou plusieurs **fournisseurs de services**. Chaque fournisseur de service propose un ou plusieurs services.

Les clients accèdent aux services au travers de leurs **interfaces**. Celles-ci définissent l’ensemble des signatures des opérations publiques, en particulier la syntaxe des messages échangés. L’interface est un contrat entre le fournisseur du service et le client. Elle est séparée de l’implémentation et indépendante de la plateforme. Sa description fournit une base pour l’implémentation du service par le fournisseur et pour l’implémentation du client.

Un même service peut être fourni par plusieurs implémentations différentes. Ces implémentations présentent la même interface. Elles peuvent être fournies par plusieurs fournisseurs différents.

L’ensemble des services des différents fournisseurs sont référencés dans un **annuaire**. Ils permettent en particulier de rechercher l’ensemble des services fournissant la même fonctionnalité et présentant la même interface.

Le concept central de notre architecture est la composition de services en processus métier. Ce processus métier est mis en œuvre au sein d’une **orchestration**.

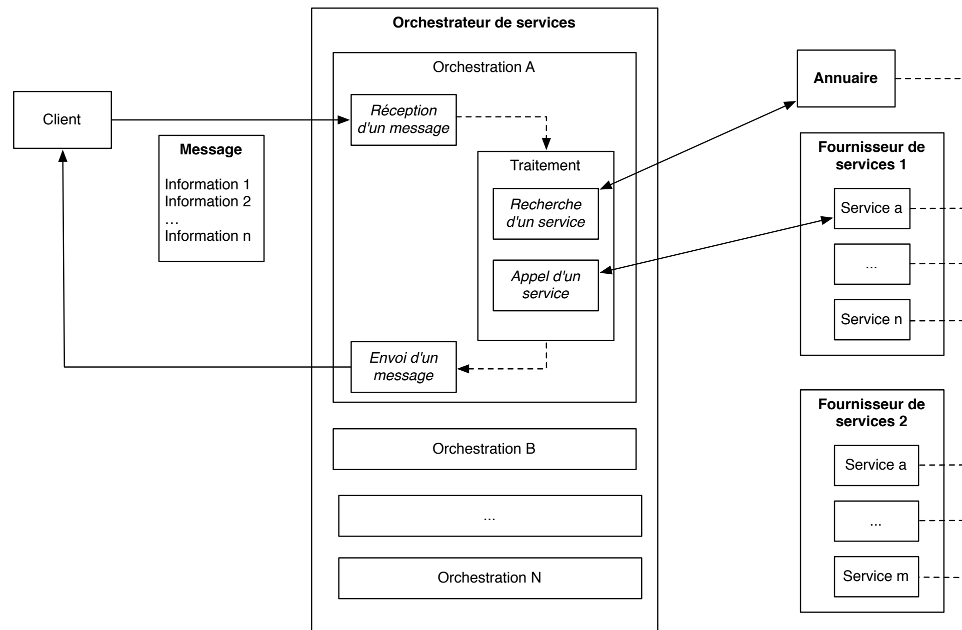

FIGURE 4.1 – Composants d'une architecture orientée services

Une orchestration est d'abord un service qui reçoit des messages et fournit des réponse après traitement. Un certain nombre d'opérations peuvent être effectuées au sein de l'orchestration. Cependant pour chaque fonctionnalité métier l'orchestration fait appel à un service externe qui peut être préalablement recherché dans un annuaire. Un **orchestrateur de services** fournit une ou plusieurs orchestrations.

Les services, et a fortiori les orchestrations, effectuent des traitements, des opérations sur des valeurs que nous appellerons informations. Parmi les informations, certaines dépendent les unes des autres. Pour un niveau de granularité défini, on peut considérer que d'autres informations ne sont, en revanche, dépendantes d'aucune autre, soit parce que l'on choisit d'ignorer ces dépendances, soit parce que les informations ont été directement injectées dans le système. Ces informations sont dites atomiques.

Dans le cas des orchestrations, les informations atomiques sont injectées soit par des services externes soit directement par un utilisateur. Une information atomique est la plus petite unité d'information que nous considèrerons. Toutes les informations manipulées dans le système sont soit atomiques, soit obtenues après des traitements (calculs) sur une ou plusieurs informations atomiques. Une information non atomique sera dite composée ou dérivée d'une ou plusieurs informations atomiques. Par exemple, si x, y sont des informations atomiques, $2 \times x, x+y, \ldots$ sont des informations composées, la première dérivant de x, la seconde dérivant de x et y.

De façon générale, les informations (atomiques ou composées) sont localisées

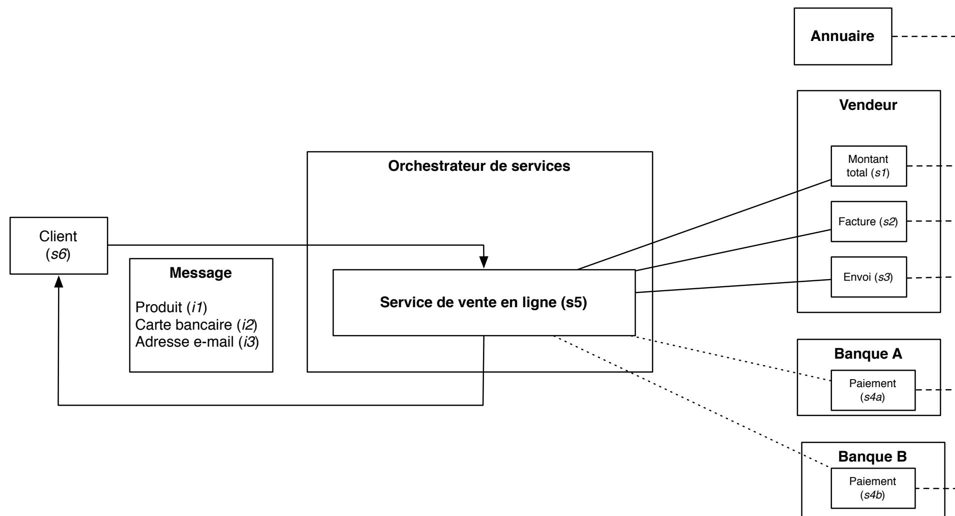

FIGURE 4.2 – Service de vente en ligne

dans des conteneurs d'information (physiques ou logiques) qui appartiennent à des services. Dans le cas des services, les principaux conteneurs d'informations sont les messages et les variables.

Les traitements, les opérations réalisées par les programmes (en particulier les services) vont engendrer des flux d'information c'est-à-dire que ces opérations vont faire transiter les informations par différents conteneurs d'information.

(Exemple 1)

Considérons par exemple un service de vente de livres en ligne présenté sur la figure 4.2. Ce service de vente de livres en ligne est fourni par une orchestration qui fait appel à différents services. Les différents acteurs de cette transaction sont :

- *le vendeur qui fournit trois services : calculer le montant total de la transaction afin de permettre le paiement par la banque* (s_1)*, émettre la facture* (s_2) *et finalement préparer la commande qui sera envoyée à l'adresse fournie par le client* (s_3) *;*
- *les deux banques A et B qui fournissent chacun un service de paiement* (s_{4a} *et* s_{4b}) *;*
- *le service de vente en ligne* (s_5) *qui interagit avec le client et les services du vendeur et des banques A et B ;*
- *le client qui peut-être vu comme un service* (s_6)*.*

Le message envoyé par le client au service de vente en ligne est constitué de trois informations atomiques :

- *le produit choisi* (i_1) *;*
- *le numéro de carte bancaire du client* (i_2) *;*

– *l'adresse électonique du client (i_3).*

4.2 Politique de flux d'informations

La politique de flux d'informations détermine pour chaque information atomique (et par composition pour l'ensemble des informations composées) vers quel service cette information peut-être envoyée. Pour cela, nous déterminons un propriétaire pour chaque information atomique (le service qui l'a injecté dans le système). Ce propriétaire a la charge de déterminer la politique de flux de l'ensemble de ses informations atomiques. La suite de la politique est déterminée par composition. Lorsqu'une information dérive de plusieurs informations atomiques, les propriétaires de cette information sont l'ensemble des propriétaires des informations atomiques dont elle dérive. Cette information n'est alors accessible que par les services qui pouvaient accéder à l'ensemble des informations atomiques dont celle-ci dérive. La politique de sécurité peut donc se voir comme une politique de flux d'information : un flux d'une information i (atomique ou composée) vers un conteneur appartenant à un service s est légal si et seulement si le service s à le droit d'accéder à l'information i.

Plus formellement nous utiliserons les notations suivantes :

Les informations : $I = \{i_1, ..., i_n, ...\}$ est l'ensemble, éventuellement infini, des informations atomiques du système. Une information dérivée de plusieurs informations atomiques $i_j, \ldots, i_k$ sera notée $i_j \oplus \ldots \oplus i_k$

Les services : $S = \{s_1, ..s_m\}$ est l'ensemble des services du système.

Les propriétaires d'une information i sont des services que nous noterons : $prop(i) \subseteq S$. Ils sont définis de la façon suivante :
- si i est une information atomique alors son propriétaire est le service qui l'a injecté dans le système ;
- si i est une information composée, c'est à dire que $i = i_j \oplus \ldots \oplus i_k$ alors

$$prop(i) = prop(i_j) \cup \ldots \cup prop(i_k) \tag{4.1}$$

(Exemple 2)

Revenons maintenant à l'exemple 1 présenté figure 4.2. Dans cet exemple trois informations initiales peuvent être identifiées :
- i_1 *est le produit choisi par le client ;*
- i_2 *est le numéro de carte bancaire du client ;*
- i_3 *est l'adresse électronique du client.*

Les informations atomiques du système sont fournies par le service appelant la commande, c'est à dire s_6. Dans tous les cas le propriétaire de ces informations atomiques est le service appelant la commande (appelé directement par le client), c'est à dire s_6.

Plus formellement dans cet exemple

Les informations *sont* $I = \{i_1, i_2, i_3\}$ *;*

Les services *sont* $S = \{s_1, \ldots, s_6\}$ *;*

Le propriétaire *de I,* $prop(I) = prop(i_1) = prop(i_2) = prop(i_3) = s_6$ *;*

Dans le modèle proposé par Myers et Liskov ([ML97]) présenté section 3.2.2, chaque variable est associée à un label de sécurité. Ce label indique quels sont les propriétaires de l'information contenue dans cette variable. Chaque propriétaire de cette information indique l'ensemble des utilisateurs qu'il autorise à accéder à l'information. Un utilisateur autorisé à accéder à une variable par l'ensemble des proprétaires est appelé un lecteur.

Nous proposons d'adapter cette politique de flux d'informations à notre modèle d'architecture orientée services. Nous proposons que, pour chaque information atomique, le propriétaire de cette information spécifie l'ensemble des services vers lesquels cette information peut-être envoyée. Ce modèle a été présenté dans [DTTT10].

A cet effet chaque service est associé à un id unique qui permet d'identifier ce service. La politique de flux associée aux informations atomiques définit l'ensemble des services lecteurs autorisés à accéder à cette information.

Les lecteurs d'une information i sont des services définis par le propriétaire de i que nous notons $lec(i) \subseteq S$. Nous considérons qu'un propriétaire d'une information i s'autorise toujours à être lecteur d'une information i même si il ne se spécifie pas dans la liste des lecteurs autorisés à accéder à cette information. Les lecteurs sont définis de la façon suivante :

- si i est une information atomique, les lecteurs de i sont les lecteurs autorisés par le service qui a injecté l'information dans le système ainsi que le service qui a injecté cette information dans le système ;
- si $i = i_j \oplus \ldots \oplus i_k$ alors

$$lec(i) = lec(i_j) \cap \ldots \cap lec(i_k) \tag{4.2}$$

La politique de flux d'informations définit les lecteurs autorisés de chaque information atomique. La règle de composition (4.2) définit, par composition, les lecteurs de chaque information composée. Cette politique est définie par le proriétaire de l'information. Ainsi, chaque propriétaire définit pour chacune de ses informations atomiques les lecteurs autorisés à lire cette information. Un appel vers un service qui nécessite une information est légal si et seulement si le service appelé est un *lecteur* de cette information.

(Exemple 3)

Considérons de nouveau l'exemple 1 présenté sur la figure 4.2. Dans cet exemple six services peuvent être distingués :

- s_1, s_2 *and* s_3 *sont les trois services fournis par le vendeur ;*

Information atomique	Propriétaires	Lecteurs
i_1	s_6	$\{s_1, s_2, s_3, s_6\}$
i_2	s_6	$\{s_{4a}, s_{4b}, s_6\}$
i_3	s_6	$\{s_1, s_2, s_3, s_6\}$

FIGURE 4.3 – Une politique de flux d'informations pour l'exemple 1

- s_{4a} *et* s_{4b} *sont les services de paiement fournis par les banques A et B ;*
- s_5 *est le service de vente en ligne fourni par une orchestration des services* $s_1, \ldots, s_{4b}$ *;*
- s_6 *est le service de l'utilisateur qui appelle l'orchestration* s_5 *afin d'effectuer une commande.*

Dans l'exemple, les informations atomiques du système sont fournies par le service appelant la commande, c'est à dire s_6*.* i_1 *correspond au produit choisi. Le client peut désirer que celui-ci ne soit accessible qu'au vendeur et qu'ainsi* $lec(i_1) = \{s_1, s_2, s_3\}$*, les services proposés par le vendeur.* i_2 *correspond aux coordonnées bancaire. Le client peut désirer que celles-ci ne soient accessibles qu'aux banques, on a alors* $lec(i_2) = \{s_{4a}, s_{4b}\}$ *les services de paiement proposés par les banques. De la même façon si le client désire que* i_3 *(le mail du client) ne soit accessible qu'au vendeur, on a* $lec(i_3) = \{s_1, s_2, s_3\}$*. Dans tous les cas le propriétaire de ces informations atomiques est le service appelant la commande (appelée directement par le client), c'est à dire* s_6*.*

Plus formellement dans cet exemple les lecteurs de I *sont définis de la façon suivante par le service* s_6 *:*

- $lec(i_1) = \{s_1, s_2, s_3\}$,
- $lec(i_2) = \{s_{4a}, s_{4b}\}$,
- $lec(i_3) = \{s_1, s_2, s_3\}$.

En d'autres termes, la politique de flux d'information est entièrement définie figure 4.3.

Si s_{4b} *n'est pas un service bancaire de confiance pour l'utilisateur, celui-ci peut alors ne pas l'inclure dans la liste des lecteurs de* i_2*. On a alors :* $lec(i_2) = \{s_{4a}\}$

4.3 Premier modèle de suivi de la politique de flux

4.3.1 Principe général

De façon générale, nous utilisons des meta-données (des labels) qui sont attachées à chaque conteneur d'informations et qui reflètent la politique de sécurité

attachée aux informations qu'il contient. Plus précisément, le label d'un conteneur liste l'ensemble des propriétaires et lecteurs de cette information. Lorsqu'un conteneur reçoit un flux d'information provenant de différents conteneurs, les meta-données attachées à ce conteneur sont modifiées afin de refléter la politique de sécurité des informations qu'il contient désormais. A chaque appel de service, la politique est vérifiée. Il est alors possible de décider si l'appel de service est légal au regard de la politique : l'appel est légal si et seulement si le service destinataire est un lecteur de la donnée envoyée pour cet appel.

4.3.2 Labels de sécurité

Un label est une meta-donnée attachée à chaque conteneur qui décrit l'ensemble des propriétaires et lecteurs de l'information contenue dans ce conteneur. Nous utiliserons la syntaxe suivante pour les labels. Si c est un conteneur, son label de sécurité est de la forme

$$L_c = \{\mathbf{s}_\alpha \rhd s_{\alpha_1}, ..., s_{\alpha_m}; ...; \mathbf{s}_\beta \rhd s_{\beta_1}, ..., s_{\beta_n}\}$$

Ce label signifie que l'information i contenue dans c a pour propriétaires les services définis par

$$prop(L_c) = \{\mathbf{s}_\alpha, ..., \mathbf{s}_\beta\}$$

.

Il signifie également que le service $\mathbf{s}_\alpha$ (respectivement $\mathbf{s}_\beta$) a autorisé les lecteurs $s_{\alpha_1}, ..., s_{\alpha_m}$ (respectivement $s_{\beta_1}, ..., s_{\beta_n}$).

On note $lec(L_c, p)$ les lecteurs autorisés par le propriétaire p à accéder à l'information contenue dans le conteneur c. Ainsi pour ce label,

$$lec(L_c, \mathbf{s}_\alpha) = \{s_{\alpha_1}, ..., s_{\alpha_m}\}$$

Si p n'est pas un propriétaire de L, alors p ne définit pas de politique de sécurité pour le label L. Donc p autorise implicitement l'ensemble des services S à accéder au contenu du conteneur labelisé par L. Ainsi de manière plus formelle :

Définition 4.3.1. $lec(L, p) = S \ si \ p \notin prop(L)$

Ce label nous permet de retrouver les lecteurs autorisés à accéder à l'information i. En effet un service est autorisé à accéder à l'information i si l'ensemble des propriétaires l'autorise à accéder à cette information. Par définition un propriétaire d'une information se définit lui-même comme lecteur de cette information. Nous avons alors :

Définition 4.3.2. $lec(L) = \bigcap_{p \in prop(L)} \{p\} \cup lec(L, p)$

C'est à dire pour le label L_c :

$$lec(L_c) = \{s_\alpha \cup lec(L_c, s_\alpha)\} \cap ... \cap \{s_\beta \cup lec(L_c, s_\beta)\}$$

```
<service>
<lecteurs>::=∅|<service>|<service>,<lecteurs>
<proprietaire>::=<service>
<label>::=∅|<proprietaire>▷<lecteurs>|
          <proprietaire>▷<lecteurs>;<label>
```

FIGURE 4.4 – Grammaire des labels

$$lec(L_c) = \{s_\alpha, s_{\alpha_1}, ..., s_{\alpha_m}\} \cap ... \cap \{s_\beta, s_{\beta_1}, ..., s_{\beta_n}\}$$

La grammaire des labels est présentée sur la figure 4.4. La description des services n'est pas définie, ils sont représentés par la balise <service>.

4.3.3 Propagation des labels

L'information contenue dans les conteneurs est modifiée au travers des affectations et des appels de fonction effectués par une orchestration ou par les services qu'elle invoque. Nous voulons que le label attaché à chaque conteneur décrive la politique de sécurité de l'information contenue dans ce conteneur. Pour cela les labels sont mis à jour à chaque opération sur un conteneur et donc à chaque observation d'un flux d'information.

La relation de restriction est définie par les auteurs de [ML97] de la façon suivante :

Définition 4.3.3 (Définition de $L_1 \sqsubseteq L_2$ d'après [ML97])**.** *Un label L_2 est une restriction d'un label L_1, ce que l'on note $L_1 \sqsubseteq L_2$ si et seulement si :*

$$prop(L_1) \subseteq prop(L_2) \tag{4.3}$$

$$\forall p \in prop(L_1), lec(L_1, p) \supseteq lec(L_2, p) \tag{4.4}$$

Vérifions maintenant que cette définition de la restriction permet bien de respecter la politique de flux d'informations que nous avons proposé dans la section 4.2, c'est à dire que si L_2 est une restriction de L_1 alors alors les lecteurs de L_2 sont inclus dans ceux de L_1.

Lemme 4.3.1. *Soit L_1 et L_2 deux labels : si $L_2 \sqsubseteq L_1$ alors $lec(L_2) \subseteq lec(L_1)$*

Démonstration. Soit L_1 et L_2 deux labels tels que $L_1 \sqsubseteq L_2$, montrons que $lec(L_2) \subseteq lec(L_1)$.

D'après la définition 4.3.2

$$lec(L_2) = \bigcap_{p \in prop(L_2)} \{p\} \cup lec(L_2, p)$$

Or d'après la définition 4.3.3 on a $prop(L_1) \subseteq prop(L_2)$, donc

$$prop(L_2) = prop(L_1) \cup (prop(L_2)$$

Nous avons donc :

$$\begin{aligned} lec(L_2) &= \bigcap_{p \in prop(L_1)} \{p\} \cup lec(L_2, p) \\ &\bigcap \bigcap_{p \in (prop(L_2) \setminus (prop(L_1) \cap prop(L_2)))} \{p\} \cup lec(L_2, p) \\ &\subseteq \bigcap_{p \in prop(L_1)} \{p\} \cup lec(L_2, p) \end{aligned}$$

Or d'après la définition 4.3.3 on a :

$$\forall p \in prop(L_1), lec(L_1, p) \supseteq lec(L_2, p)$$

Donc :

$$lec(L_2) \subseteq \bigcap_{p \in prop(L_1)} \{p\} \cup lec(L_1, p)$$

Donc d'après la définition 4.3.2

$$lec(L_2) \subseteq lec(L_1)$$

□

De façon générale si nous observons un flux d'information depuis les conteneurs $c_j, ..., c_k$ vers c, nous modifions alors le label de c qui devient la jointure des labels attachés à $c_j, ..., c_k$.

A cet effet, la relation $\oplus$ qui permet d'effectuer la jointure de deux labels L_1 et L_2 est définie de la même manière que les auteurs de [ML97]. La jointure de deux labels L_1 et L_2 doit produire un label qui est plus restrictif que L_1 et L_2 afin de permettre le respect de la politique de flux d'informations.

Définition 4.3.4 (Définition de $L_1 \oplus L_2$ d'après [ML97])**.** *Soit L_1 et L_2 deux labels, l'opération de jointure $L_1 \oplus L_2$ est alors définie de la manière suivante :*

$$prop(L_1 \oplus L_2) = prop(L_1) \cup prop(L_2) \tag{4.5}$$

$$\forall p \in prop(L_1 \oplus L_2), lec(L_1 \oplus L_2, p) = lec(L_1, p) \cap lec(L_2, p) \tag{4.6}$$

$$L_1 \oplus L_2 = \bigcup_{p \in prop(L_1 \oplus L_2)} \{p \rhd lec(L_1 \oplus L_2, p)\} \tag{4.7}$$

La relation 4.5 signifie que les propriétaires de $L_1 \oplus L_2$ sont l'union des propriétaires de L_1 et de L_2. L'ensemble des lecteurs de$L_1 \oplus L_2$ pour chaque propriétaire de L_1 et de L_2 est l'intersection des ensembles de lecteurs correspondants. Ceci est exprimé dans la relation 4.6. La relation 4.7 permet quand à elle de définir l'opération de jointure $\oplus$ à partir des relations 4.5 et 4.6.

Montrons maintenant que $L_1 \oplus L_2$ est une restriction de L_1 et de L_2.

Lemme 4.3.2. *Soit deux labels L_1 et L_2, alors on a :*
- $L_1 \sqsubseteq (L_1 \oplus L_2)$
- $L_2 \sqsubseteq (L_1 \oplus L_2)$

Démonstration. Considérons deux labels L_1 et L_2, montrons que $L_1 \sqsubseteq (L_1 \oplus L_2)$ et $L_2 \sqsubseteq (L_1 \oplus L_2)$.

D'après la définition 4.3.4 on a $prop(L_1 \oplus L_2) = prop(L_1) \cup prop(L_2)$ donc :

$$\begin{aligned} prop(L_1 \oplus L_2) &\subseteq prop(L_1) \\ prop(L_1 \oplus L_2) &\subseteq prop(L_2) \end{aligned}$$

Donc la première partie de la définition 4.3.3 est bien vérifiée.

D'après la définition 4.3.4 on a :

$$\forall p \in prop(L_1 \oplus L_2), lec(L_1 \oplus L_2, p) = lec(L_1, p) \cap lec(L_2, p)$$

Donc :

$$\begin{aligned} \forall p \in prop(L_1 \oplus L_2), lec(L_1 \oplus L_2, p) &\supseteq lec(L_1, p) \\ \forall p \in prop(L_1 \oplus L_2), lec(L_1 \oplus L_2, p) &\supseteq lec(L_2, p) \end{aligned}$$

Donc la seconde partie de la définition 4.3.3 est bien vérifiée.

Donc $L_1 \sqsubseteq (L_1 \oplus L_2)$ et $L_2 \sqsubseteq (L_1 \oplus L_2)$. □

(Exemple 4)

Afin d'illustrer la propagation des flux au sein d'une orchestration de services nous considérons les variables x, y et z. L'orchestration que nous considérons est en interaction avec 7 services s_i numérotés de 1 à 7. Nous considérerons que les variables x, y et z sont les conteneurs d'information de cette orchestration. Au début de l'éxécution, z n'est associée à aucun label et x et y sont associées aux labels suivants :
- L_x : $\{\mathbf{s_1} \rhd s_5, s_6; \mathbf{s_2} \rhd s_6, s_7\}$
- L_y : $\{\mathbf{s_1} \rhd s_6, s_7; \mathbf{s_3} \rhd s_6, s_7\}$

On a alors pour x :
- $prop(L_x) = \{s_1, s_2\}$

– $lec(L_x, s_1) = \{s_5, s_6\}$
– $lec(L_x, s_2) = \{s_6, s_7\}$
– $lec(L_x) = \{\{s_1 \cup lec(x, s1)\} \cap \{s_2 \cup lec(x, s2)\}\} = \{\{s_1, s_5, s_6\} \cap \{s_2, s_6, s_7\}\} = \{s_6\}$

et pour y :

– $prop(L_y) = \{s_1, s_3\}$
– $lec(L_y, s_1) = \{s_6, s_7\}$
– $lec(L_y, s_3) = \{s_6, s_7\}$
– $lec(L_y) = \{\{s_1 \cup lec(y, s1)\} \cap \{s_3 \cup lec(y, s3)\}\} = \{\{s_1, s_6, s_7\} \cap \{s_3, s_6, s_7\}\} = \{s_6, s_7\}$

Considérons l'affectation de l'addition du contenu des variables x et y dans z. On a donc un flux d'information direct de x et y vers z. Nous avons alors le remplacement de l'information contenue dans z par une information provenant de x et de y. Le label de sécurité associé à z à l'issue de l'exécution de cette instruction doit être la jointure des labels attachés à x et à y, c'est à dire que $L_z = L_x \oplus L_y$ a l'issue de l'exécution de cette instruction.

On a alors d'après la définition 4.3.4 :

– $prop(L_z) = prop(L_x) \cup prop(L_y) = \{s_1, s_2\} \cup \{s_1, s_3\} = \{s_1, s_2, s_3\}$
– $lec(L_z, s_1) = lec(L_x, s_1) \cap lec(L_y, s_1) = \{s_5, s_6\} \cap \{s_6, s_7\} = s_6$
– $lec(L_z, s_2) = lec(L_x, s_2) \cap lec(L_y, s_2) = \{s_6, s_7\} \cap S = \{s_6, s_7\}$
– $lec(L_z, s_3) = lec(L_x, s_3) \cap lec(L_x, s_3) = S \cap \{s_6, s_7\} = \{s_6, s_7\}$

Le nouveau label associé à z est donc :

$$L_z = \{\mathbf{s_1} \rhd s_6; \mathbf{s_2} \rhd s_6, s_7; \mathbf{s_3} \rhd s_6, s_7\}$$

On retrouve bien que $lec(L_z) = lec(L_x) \cap lec(L_y)$ car :

– $lec(L_x) \cap lec(L_y) = \{s_6\} \cap \{s_6, s_7\} = \{s_6\}$
– $lec(L_z) = \{s_1, lec(L_z, s_1)\} \cap \{s_2, lec(L_z, s_2)\} \cap \{s_3, lec(L_z, s_3)\} = \{s_1, s_6\} \cap \{s_2, s_6, s_7\} \cap \{s_3, s_6, s_7\} = \{s_6\}$

La propagation de la politique de sécurité s'effectue donc via la propagation des labels attachés aux conteneurs d'information. Si, dans la suite du programme, z est envoyé au service s_7, cette opération est refusée car s_7 ne fait plus partie des lecteurs autorisés de z.

4.3.4 Respect de la politique de flux par les labels

Montrons maintenant que les labels permettent effectivement de propager la politique de sécurité présentée dans la section 4.2. Il s'agit pour cela de montrer que la règle de relabélisation $\oplus$ respecte les règles 4.1 et 4.2. C'est à dire que la définition 4.3.4 respecte le lemme suivant :

Lemme 4.3.3. *Soit L_1 et L_2 deux labels, alors,*

$$prop(L_1 \oplus L_2) = prop(L_1) \cup prop(L_2) \tag{4.8}$$

$$lec(L_1 \oplus L_2) = lec(L_1) \cap lec(L_2) \tag{4.9}$$

Démonstration. La première propriété 4.8 est directement vérifiée par la propriété 4.5 de la définition de l'opérateur $\oplus$ (4.3.4).

Montrons maitenant la seconde propriété 4.9.

Soit L_1 et L_2 deux labels, montrons que :

$$lec(L_1 \oplus L_2) = lec(L_1) \cap lec(L_2)$$

D'après la définition 4.3.2 nous avons :

$$lec(L_1 \oplus L_2) = \bigcap_{p \in prop(L_1 \oplus L_2)} \{p\} \cup lec(L_1 \oplus L_2, p)$$

D'après les règles 4.5 et 4.6 de la définition 4.3.4 nous avons :

$$lec(L_1 \oplus L_2) = \bigcap_{p \in (prop(L_1) \cup prop(L_2))} \{p\} \cup (lec(L_1, p) \cap lec(L_2, p))$$

Nous avons alors :

$$\begin{aligned} lec(L_1 \oplus L_2) &= \bigcap_{p \in (prop(L_1) \cup prop(L_2))} (\{p\} \cup lec(L_1, p)) \cap (\{p\} \cup lec(L_2, p)) \\ &= \left(\bigcap_{p \in (prop(L_1) \cup prop(L_2))} (\{p\} \cup lec(L_1, p)) \right) \\ &\quad \bigcap \left(\bigcap_{p \in (prop(L_1) \cup prop(L_2))} (\{p\} \cup lec(L_2, p)) \right) \end{aligned}$$

Or nous avons :

$$\begin{aligned} \bigcap_{p \in (prop(L_1) \cup prop(L_2))} (\{p\} \cup lec(L_1, p)) = & \bigcap_{p \in prop(L_1)} (\{p\} \cup lec(L_1, p)) \\ & \bigcap \bigcap_{p \in prop(L_2) \setminus (prop(L_1) \cap prop(L_2))} (\{p\} \cup lec(L_1, p)) \end{aligned}$$

D'après la définition 4.3.1 nous avons :

$$\bigcap_{p \in prop(L_2) \setminus (prop(L_1) \cap prop(L_2))} (\{p\} \cup lec(L_1, p)) = S$$

Donc :

$$\bigcap_{p \in (prop(L_1) \cup prop(L_2))} (\{p\} \cup lec(L_1, p)) = \bigcap_{p \in prop(L_1)} (\{p\} \cup lec(L_1, p)) \bigcap S$$

C'est à dire que :

$$\bigcap_{p \in (prop(L_1) \cup prop(L_2))} (\{p\} \cup lec(L_1, p)) = \bigcap_{p \in prop(L_1)} (\{p\} \cup lec(L_1, p))$$

De la même manière on montrerait que :

$$\bigcap_{p \in (prop(L_1) \cup prop(L_2))} (\{p\} \cup lec(L_2, p)) = \bigcap_{p \in prop(L_2)} (\{p\} \cup lec(L_2, p))$$

Nous avons alors :

$$lec(L_1 \oplus L_2) = \bigcap_{p \in prop(L_1)} (\{p\} \cup lec(L_1, p)) \bigcap \bigcap_{p \in prop(L_2)} (\{p\} \cup lec(L_2, p))$$

Donc d'après la définition 4.3.2 on a :

$$lec(L_1 \oplus L_2) = lec(L_1) \cap lec(L_2)$$

□

4.3.5 Propagation de la politique de flux entre les services

Lorsqu'un service s fait appel à un orchestrateur o (ou un service) mettant en œuvre la politique de contrôle du flux d'information de Myers, il indique pour chacune des informations envoyées l'ensemble des services autorisés à lire ces informations. Le service s est considéré par o comme le propriétaire des informations qu'il a envoyé.

Lorsque o renvoie les informations qu'il a reçues de s vers un autre orchestrateur o' il commence par s'assurer que o' est un lecteur autorisé des informations envoyées par s. Si o' est un lecteur autorisé alors il spécifie à o' l'ensemble des lecteurs autorisés à accéder à ces informations, c'est à dire les lecteurs autorisés par s ou une restriction de ces lecteurs. o est alors le propriétaire des informations envoyées à o'. Si une modification de la politique de sécurité est nécessaire alors o' effectuera une demande auprès de o qui lui même répercutera cette demande auprès de s.

En ne propageant que la liste des lecteurs lors de la propagation de la politique de flux entre les services, les informations atomiques ne possèdent alors qu'un seul propriétaire, le service qui a envoyé l'information. Cela permet ainsi de justifier l'hypothèse d'un propriétaire par information atomique effectuée dans la section 4.2.

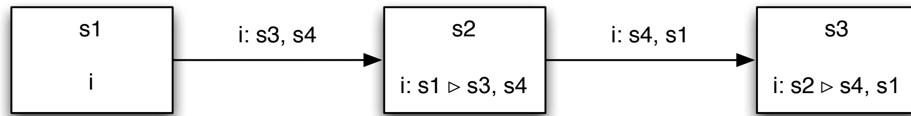

FIGURE 4.5 – Propagation d'une politique de flux entre des services

(Exemple 5)

Considérons l'exemple présenté sur la figure 4.5. Dans cet exemple le service s_1 envoie l'information i au service s_2. Il spécifie que l'information i peut être envoyée à s_3 et s_4. Le label associé à i par s_2 est donc $\{\mathbf{s_1} \triangleright s_3, s_4\}$. On a donc pour s_2, $prop(i) = s_1$ et $lec(i) = \{s_1, s_3, s_4\}$.

s_3 est un lecteur de i donc s_2 peut envoyer l'information i à s_3. Comme $lec(i) = \{s_1, s_3, s_4\}$ il spécifie à s_3 que les lecteurs de i sont s_1 et s_4. Le label associé à i par s_3 est donc $\{\mathbf{s_2} \triangleright s_1, s_4\}$. On a donc pour s_3, $prop(i) = s_2$ et $lec(i) = \{s_2, s_1, s_4\}$.

4.3.6 Les limites du modèle

L'application directe du modèle de Myers et Liskov permet de proposer un modèle simple de politique de flux pour les orchestrations de service. Ce modèle permet à chaque utilisateur d'une orchestration de services de spécifier pour chacune des données qu'il envoie à l'orchestration de services la politique de flux associée. Cette politique définit pour chaque information l'ensemble des services autorisés à y accéder. La propagation de cette politique au cours de l'orchestration permet d'assurer de bout en bout la confidentialité de ces informations.

Ce modèle présente néanmoins deux limitations importantes, la première est liée à la définition de la politique de sécurité, la seconde est liée à la modification dynamique de la politique de sécurité.

La définition de la politique de sécurité par le client est difficile dans ce modèle. En effet, le client doit spécifier pour chaque information l'ensemble des services qui sont autorisés à y accéder. La définition de cette politique peut-être fastidieuse et pas toujours possible avant l'exécution du programme. A cet effet, nous proposons de permettre à l'utilisateur de spécifier la politique de flux d'information au fur et à mesure de l'exécution de l'orchestration.

Il est néanmoins difficile de modifier au cours de l'exécution la politique de sécurité. En effet Myers et Liskov proposent afin de modifier la politique de sécurité que chaque propriétaire d'une information puisse ajouter un ou des lecteurs autorisés à lire cette information. Nous avons vu dans la section précédente (4.3.3) que l'ensemble des lecteurs autorisés par chaque propriétaire d'une information était conservé lors de la propagation des labels. Afin d'autoriser l'envoi d'une informa-

tion vers un service, il suffit donc de demander aux propriétaires qui n'autorisent pas l'envoi de cette information de modifier la politique de flux associée à cette donnée. Cependant le modèle de Myers permet de propager une politique de sécurité mais ne permet de savoir quelles informations ont été utilisées afin de produire l'information pour laquelle on veut modifier la politique de flux. De plus un propriétaire d'une information non atomique n'est pas forcément un lecteur de cette information. De ce fait il est parfois impossible de spécifier au propriétaire d'une information pour quelle information il est nécessaire de modifier la politique de flux. L'objet de la section suivante (4.4) est de proposer un mécanisme de suivi des flux d'informations permettant au propriétaire d'une information d'accéder à suffisamment d'éléments afin de lui permettre d'autoriser ou non une modification de la politique de flux en toute connaissance de cause.

Néanmoins cette décision reste difficile à prendre pour le propriétaire de l'information pour deux raisons. Il ne peut savoir quelle est l'information qu'il a initialement envoyée car l'information peut avoir été transférée du conteneur initial vers un autre. Il peut également ne pas être autorisé à accéder à l'information envoyée car il n'est pas forcément un lecteur autorisé de cette information. La résolution de ces problèmes est étudiée dans la section suivante (4.4).

4.4 Modification dynamique de la politique de flux d'information

4.4.1 Illustration des problèmes

Deux situations peuvent nécessiter de modifier dynamiquement la politique de contrôle de flux d'information au sein d'un orchestration de services. La première situation est liée à l'emploi d'un résultat provenant d'informations confidentielles. C'est la situation évoquée classiquement dans les travaux sur la déclassification. La deuxième situation est plus spécifique aux orchestrations de services et est liée au fait que les services employés dans une orchestration de services ne sont pas forcément connus à l'avance.

(Exemple 6)

Revenons à l'exemple d'un service de vente de livres en ligne présenté figure 4.2. Dans cet exemple la politique de contrôle de flux consiste à s'assurer que :

- *seul le vendeur a connaissance des produits qu'il a choisis,*
- *seule la banque a connaissance des informations bancaires fournies par le client.*

Cependant l'application stricte de cette politique ne permet pas le fonctionnement correct du service de vente en ligne. Considérons l'extrait de programme BPEL représenté figure 4.6. Dans cet exemple la politique de sécurité semble à priori respectée. En effet les informations initiales ne sont à priori transmises qu'aux destinataires légaux. Cependant la banque a besoin de connaitre le mon-

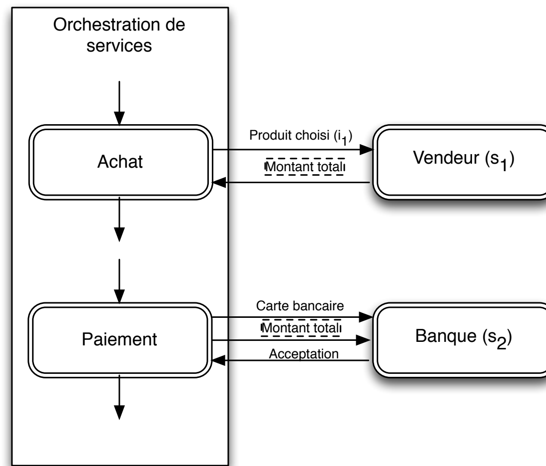

FIGURE 4.6 – Extrait d'une orchestration non conforme à la politique de flux

tant total de la commande afin de permettre le paiement de la commande par le client. C'est pourquoi le montant total est calculé par le vendeur pour être transmis à la banque. Celui-ci est dérivé des produits choisis que le client ne désire pas faire connaitre à la banque. Le montant total dépend du produit choisi, il doit donc lui être appliqué une politique de flux au moins aussi restrictive. Le montant total ne peut donc pas être transmis à la banque sans modification de la politique de flux même si on peut légitimement penser que l'information concernant le montant total peut difficilement permettre de retrouver le ou les produits choisis.

Nous constatons donc que même dans un exemple simple il peut-être difficile de permettre le fonctionnement nominal d'une orchestration en appliquant strictement la politique de flux. Il apparait donc nécessaire de proposer un mécanisme permettant de modifier ponctuellement la politique de flux afin de permettre l'exécution complète d'une orchestration.

Dans le cadre des architectures orientées services, les services peuvent être découverts au moment de l'exécution à partir d'une recherche dans un annuaire de services par exemple. Les fonctions invoquées par l'orchestration peuvent être découvertes dynamiquement à l'exécution : par exemple, le service de paiement par carte bancaire peut être celui de n'importe quelle banque en ligne. Cela signifie qu'il est impossible, lors de l'invocation de l'orchestration, de savoir vers quels services certains flux d'informations seront réalisés.

(Exemple 7)

On peut ainsi développer un service de vente en ligne qui invoque directement la banque à partir d'une requête à un annuaire selon le schéma de la

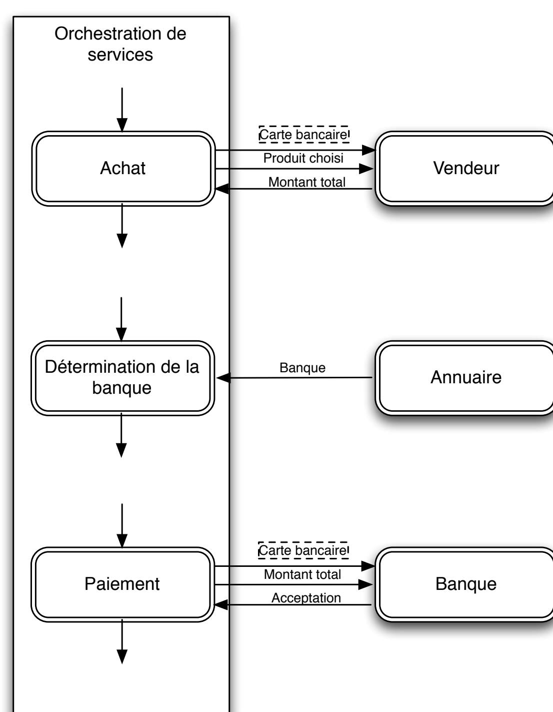

FIGURE 4.7 – Extrait d'une orchestration appelant dynamiquement un service de paiement

Figure 4.7. Le service de paiement n'est donc pas connu au moment de la définition de l'orchestration et celui-ci n'est découvert que lors de l'opération de paiement. Il est donc important que le client soit alors informé de la banque utilisée afin que celui-ci accepte ou non d'effectuer la transaction en fonction de la confiance qu'il accorde à l'établissement bancaire.

La mise en place d'un mécanisme permettant de modifier dynamiquement la politique de flux est rendu nécessaire par la nature dynamique des orchestrations de services. Les services pouvant être découverts lors de l'exécution de l'orchestration, il est donc nécessaire de pouvoir spécifier la politique au cours de l'exécution de l'orchestration.

Ces deux exemples nous montrent qu'il est nécessaire de spécifier des exceptions ponctuelles à la politique de sécurité afin de permettre l'exécution correcte du programme ou la mise en place de manière dynamique de la politique de flux. Ces exceptions ponctuelles s'apparentent à des déclassifications. Il s'agit d'autoriser un utilisateur à accéder de manière ponctuelle à tout ou partie d'une information à laquelle il n'a normalement pas accès. Il s'agit ici de modifier les lecteurs autorisés de certaines informations afin d'autoriser de manière ponctuelle ou définitive des services à avoir accès à tout ou partie d'une information à laquelle ils ne sont pas censés accéder selon la politique de flux.

4.4.2 Proposition d'une solution

Afin de mettre en œuvre un mécanisme permettant de modifier dynamiquement la politique de contrôle de flux d'information nous nous proposons d'explorer les quatre dimensions de la déclassification proposées dans [SS07] dans le cadre des orchestrations de services. C'est à dire répondre aux questions suivantes :

- Quand peut-on modifier la politique de contrôle de flux d'information ?
- Qui peut effectuer cette modification ?
- Pour quelles informations peut-on modifier la politique de contrôle de flux d'information ?
- Où peut-on effectuer ces modifications ?

Quand peut-on modifier la politique de contrôle de flux d'information ?

Le problème de la modification de la politique de contrôle de flux liée à une information se pose lorsqu'une information est envoyée par l'orchestration vers un service externe. Si l'invocation de ce service dépend d'une information que le service invoqué n'est pas autorisé à recevoir, il est alors nécessaire de demander aux propriétaires de l'information si ils autorisent ou non l'exécution de cette invocation afin de permettre de continuer l'exécution correcte de l'orchestration.

De la même façon si une information est utilisée afin de produire un des paramètres d'appel de cette commande, il est alors nécessaire que le service appelé soit un lecteur autorisé de cette information afin d'autoriser l'appel de ce service.

Dans la suite de l'exécution de l'orchestration deux modes d'exécution de la modification de la politique de contrôle de flux sont possibles :

- la modification est acceptée uniquement pour l'appel de service concerné. Dans ce cas l'appel de service est autorisé mais le label de la variable concerné n'est pas modifié ;
- la politique de sécurité liée à la variable est modifiée pendant la suite de l'exécution du programme BPEL. Dans ce cas le label de la variable est modifié et un lecteur supplémentaire est ajouté pour les propriétaires qui autorisent cette modification. Notons néanmoins qu'une modification de la politique est liée au contenu d'une variable et non à une information initiale. Si une information initiale a été utilisée pour produire le contenu d'une autre variable, une nouvelle modification de la politique devra être effectuée afin d'autoriser de nouveau l'appel du service. Ainsi, si les informations contenues dans c sont utilisées pour produire le contenu de d et e. Si d est utilisé comme paramètre d'appel d'un service s et que s n'est pas un lecteur de c alors s n'est pas un lecteur de d donc une modification de la politique de d est nécessaire. Si le label de d est modifié cela n'affectera ni le label de c ni celui de e. Si c ou e sont utilisés comme paramètre d'appel de s, il sera de nouveau nécessaire de modifier la politique de sécurité.

(Exemple 8)

Afin d'illustrer les problématiques liées à la dimension «quand», revenons à l'orchestration présentée sur la figure 4.6. Dans cet exemple, deux services sont appelés par l'orchestration : le service d'achat (s_1) et le service de paiement (s_2). Nous considérons que s_3 est le service qui appelle l'orchestration. Lors des appels de service la politique de sécurité associée aux informations est vérifiée. Le service d'achat s_1 est appelé avec la variable `produitChoisi` qui est associé au label suivant :

$\{\mathbf{s_3} \triangleright s_1\}$.

Comme s_1 est un lecteur de la variable passée en paramètre de l'invocation, l'appel de service est exécuté. L'information renvoyée contenue dans la variable `montantTotal` dérive de l'information contenue dans `produitChoisi`, elle est donc associée à un label au moins aussi restrictif.

Lors de l'appel du service de paiement s_2, la variable `montantTotal` est utilisée comme paramètre d'appel. s_2 n'est pas un lecteur autorisé de la variable `montantTotal`. Il est donc demandé au client si il désire modifier la politique de flux associée à la variable `montantTotal`. Le client a alors trois possibilités :

- *refuser la modification de la politique de sécurité, dans ce cas l'appel de service n'est pas exécuté ;*
- *accepter une modification ponctuelle de la politique de sécurité. Dans ce cas l'appel de service est exécuté, mais le label associé à la variable `montantTotal` n'est pas modifié ;*

– *accepter une modification de la politique de sécurité dans la suite de l'exécution de l'orchestration. Dans ce cas l'appel de service est exécuté et le label associé à la variable* `montantTotal` *est modifié et est maintenant :*
$\{\mathbf{s_3} \triangleright s_1, s_2\}$
Néanmoins le label associé à la variable `produitChoisi`*, dont le contenu de la variable* `montantTotal` *est issu n'est pas modifié.*

Qui peut effectuer une modification de la politique ?

L'intérêt de la politique de sécurité proposée par Myers et Liskov consiste en la spécification pour chaque information du ou des propriétaires de cette information. Cela permet de modifier simplement la politique de sécurité en permettant à chaque propriétaire d'ajouter un ou plusieurs lecteurs autorisés de cette information. Cependant celui-ci ne permet pas au propriétaire d'autoriser spécifiquement une déclassification. C'est le programmeur qui spécifie la déclassification et ensuite il est vérifié que celle-ci est bien effectuée dans un processus exécuté par le propriétaire de l'information.

L'utilisateur de l'orchestration étant celui qui a défini la politique de sécurité associée à chacune des informations qu'il a envoyé, il semble donc logique que ce soit celui-ci qui permette des modifications de la politique de sécurité. Si le contenu d'une variable a plusieurs propriétaires, il est nécessaire que l'ensemble des propriétaires autorisent la modification de la politique de sécurité afin que celle-ci soit réellement prise en compte.

(Exemple 9)
Considérons les variables x, y et z labelisées de la façon suivante :
– x : $\{\mathbf{s_1} \triangleright s_5, s_6; \mathbf{s_2} \triangleright s_6, s_7\}$
– y : $\{\mathbf{s_1} \triangleright s_6, s_7; \mathbf{s_3} \triangleright s_6, s_7\}$
Considérons l'appel du service s_7 avec la variable x. Dans ce cas l'appel est autorisé par s_2 et pas par s_1. Il est alors nécessaire de demander à s_1 si il accepte de modifier la politique de sécurité associée à x afin de pouvoir effectuer l'appel du service s_7.

Dans le cas de l'appel du service s_5 avec la variable y, il est nécessaire de demander à s_1 et à s_3 si ils acceptent de modifier la politique de sécurité associée à la variable y car aucun des deux services n'accepte l'envoi des informations contenues dans y vers s_5.

Pour quelles informations peut-on modifier la politique de contrôle de flux d'informations ?

Lorsque le contenu d'une variable est utilisé au cours d'un appel de service et que la politique de sécurité associée à celle-ci n'autorise pas l'envoi de ce contenu

au service (c'est-à-dire que le service appelé n'est pas un des lecteurs autorisés de l'information contenue dans la variable), il est alors nécessaire de modifier de façon ponctuelle la politique de sécurité associée à cette variable. Pour pouvoir effectuer cette opération en toute confiance l'utilisateur doit donc savoir quelle est l'information dont il s'apprête à modifier la politique de sécurité. Plusieurs problèmes se posent alors :

- l'information contenue dans la variable peut avoir plusieurs propriétaires qui ne s'autorisent pas mutuellement à accéder à cette information ;
- l'information contenue dans la variable peut être issue de calculs et de ce fait ne peut permettre de retrouver les informations initialement transmises à l'orchestration.

Considérons le premier problème. L'exemple 10 nous présente une variable ayant plusieurs propriétaires qui ne s'autorisent pas mutuellement à accéder à l'information contenue dans cette variable.

(Exemple 10)
Considérons la variable x associée au label suivant :
$\{\mathbf{s_1} \rhd s_5, s_6; \mathbf{s_2} \rhd s_6, s_7\}$

Considérons l'appel du service s_7 avec la variable x. Dans ce cas l'appel est autorisé par s_2 et pas par s_1. Il est alors nécessaire de demander à s_1 si il accepte de modifier la politique de sécurité associée à x afin de pouvoir effectuer l'appel du service s_7. Néanmoins s_1 ne peut accéder au contenu de la variable x car il n'est pas dans la liste des lecteurs autorisés de s_2. Dans ce cas on lui demande de modifier la politique de sécurité associée à cette variable dont il est l'un des propriétaires sans qu'il puisse avoir accès à son contenu.

Ce problème, en partie résolu par la dimension qui (les propriétaires n'auraient qu'à s'autoriser mutuellement à accéder à l'information afin de permettre à chacun de savoir quelle est l'information qu'ils s'apprêtent à déclassifier) empêche néanmoins dans l'ensemble des cas d'envoyer au propriétaire l'information pour laquelle on lui demande de modifier la politique. Cependant si une information a plusieurs propriétaires c'est qu'elle est issue d'informations atomiques qui n'ont qu'un seul propriétaire, le service qui les a envoyées à l'orchestration. On se ramène alors à la résolution du deuxième problème qui nécessite également d'avoir accès aux informations initiales qui ont conduit à la production du contenu de la variable servant à l'appel du service en question.

(Exemple 11)
Afin d'illustrer le deuxième problème, revenons à l'exemple à l'orchestration présentée sur la figure 4.6. Après l'exécution du service d'achat s_1, l'information renvoyée contenue dans la variable `montantTotal` dérive de l'information contenue dans `produitChoisi`, elle est donc associée à un label au moins aussi restrictif. Nous considérons ici que le service d'achat s_1 ne spécifie

aucune politique et qu'il n'existe pas d'autre flux d'information, elle est donc associée au même label :

$\{\mathbf{s_3} \triangleright s_1\}$

Le contenu de la variable `montantTotal` *est alors une valeur numérique (par exemple 10, pour 10 euros) qui est fondamentalement différente du contenu de la variable* `produitChoisi` *qui contenait une référence à un produit. Ainsi même si le propriétaire s_3 peut accéder au contenu de la variable* `montantTotal`*, il ne peut faire le lien entre le contenu de la variable et l'information qu'il a initialement envoyée, c'est à dire le produit choisi.*

Afin d'avoir accès aux informations initiales, nous proposons d'utiliser des labels qui permettent de suivre les flux d'information qui ont conduit à la production de l'information contenue dans la variable servant à l'appel du service. C'est ainsi qu'il est demandé à l'utilisateur la modification ponctuelle de la politique de flux liée à l'information qu'il a initialement transmise et si c'est possible il lui est également envoyé le contenu de la variable incriminée.

(Exemple 12)

Revenons à l'exemple d'orchestration présentée sur la figure 4.6. Lors de l'envoi de l'information du produit choisi (i_1) le propriétaire de l'information s_3 indique le lecteur autorisé s_1. L'orchestrateur associe donc à la variable `produitChoisi` *le label suivant : $\{i_1 : \mathbf{s_3} \triangleright s_1\}$. Ce label permet d'identifier non seulement les propriétaires et les lecteurs de la variable mais aussi l'information initiale dont elle est issue.*

Après l'exécution du service d'achat s_1, l'information renvoyée contenue dans la variable `montantTotal` *dérive de l'information contenue dans* `produitChoisi`*, elle est donc associée à un label au moins aussi restrictif. Nous considérons ici que le service d'achat s_1 ne spécifie aucune politique et qu'il n'existe pas d'autre flux d'informations, elle est donc associée au même label :*

$\{i_1 : \mathbf{s_3} \triangleright s_1\}$

Ainsi même si le contenu de la variable `montantTotal` *est alors une valeur numérique (par exemple 10, pour 10 euros) qui est fondamentalement différente du contenu de la variable* `produitChoisi` *qui contenait une référence à un produit, il est possible d'indiquer à s_3 que le contenu de la variable* `montantTotal` *provient de l'information i_1, c'est à dire le produit choisi.*

Une autre information liée à la modification de politique réellement effectuée peut aussi être envoyée au propriétaire de l'information. Il est également possible de lui indiquer si la variable est utilisée comme paramètre d'appel du service et dans ce cas l'information est envoyée explicitement au web service, ou bien si l'appel de service dépend de l'information mais que celle-ci n'est pas envoyée au service (par exemple l'appel du service envoyant la commande n'est effectué

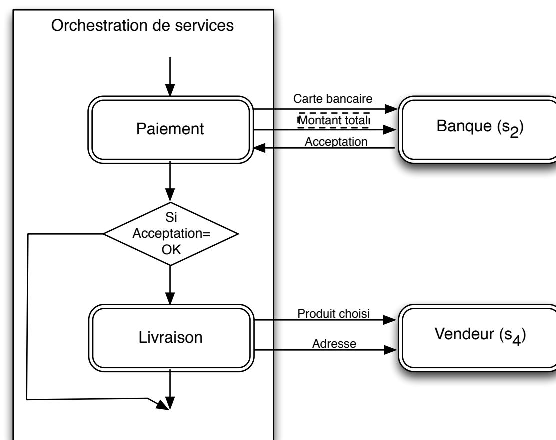

FIGURE 4.8 – Extrait d'une orchestration présentant un flux implicite d'informations

que si le service bancaire a envoyé la confirmation du paiement (qui dépend des coordonnées bancaires du client)).

Lors d'un appel de service non autorisé par la politique de sécurité l'orchestrateur va d'abord chercher à résoudre les conflits implicites (c'est-à-dire ceux provoqués par le fait que l'appel d'un service dépend d'une information) puis les conflits explicites (lorsqu'une information est envoyée directement au service appelé par les paramètres d'appel).

(Exemple 13)

Considérons la suite de l'orchestration de l'exemple 6 présenté sur la figure 4.8. Dans ce cas le service de livraison n'est exécuté que si le message d'acceptation de la banque est `OK`*. Le message d'acceptation a été produit en utilisant les données de la variable* `carteBancaire`*. Ainsi l'appel ou non du service de livraison est en lui-même une information produite à partir de la variable* `carteBancaire`*. Avant même d'autoriser des informations à être envoyées à un service, le client sera amené à décider si l'appel de service de livraison est licite vis à vis de sa politique de sécurité étant donné que le label associé à la variable* `acceptation` *est au moins aussi restrictif que le label associé à la variable* `carteBancaire`*, c'est à dire :* $\{\mathbf{s_3} \rhd s_2\}$*. Ainsi le client (*$s_3$*) n'autorise que le service de paiement de la banque (*s_2*) à accéder à cette information. L'appel du service de livraison* s_4 *n'est donc pas licite vis-à-vis de la politique de sécurité du client.*

On peut donc résumer ainsi les informations envoyées au propriétaire afin de savoir pour quelle information il autorise une modification de la politique :
– l'information initiale ;

– si possible l'information effectivement envoyée au service ;
– si l'envoi de l'information est effectué de manière explicite ou implicite.

Où peut-on effectuer des modifications de la politique ?

Nous nous plaçons dans le cadre des services pour lesquels les seules réponses qui sont renvoyées par le service appelé sont la réponse à l'appel de service ou un message d'erreur. Il n'est donc pas possible d'initier une transaction supplémentaire afin de permettre une modification de la politique de sécurité liée à une information ou au contenu d'une variable. La modification de la politique de flux ne peut donc être effectuée par le service appelant l'orchestration directement.

Nous proposons d'ajouter chez le client d'une orchestration un module externe lui permettant de modifier la politique de flux. Ce module externe est reconnu comme un service par l'orchestration et est vu comme une interface utilisateur par le client. Afin de rendre certaines déclassifications automatiques, il peut être envisagé de rajouter certaines règles automatiques à ce service de modification de la politique de contrôle de flux.

L'utilisation d'un module externe permet à l'utilisateur d'avoir la pleine maitrise de la politique de sécurité affectée à chacune des informations envoyées. Les différentes autorisations de modification de la politique de contrôle de flux sont ensuite utilisées par l'orchestrateur.

4.5 Flux d'informations et modification des labels

4.5.1 Définition des labels de suivi des flux d'informations

Nous avons vu dans la section 4.4.2 qu'au moment de modifier la politique de sécurité, il était nécessaire pour le propriétaire de l'information de savoir quelle était l'information initiale qu'il allait déclassifier. C'est pourquoi il est nécessaire de pouvoir suivre non seulement la politique de sécurité mais aussi la dépendance des contenus des variables aux informations initiales au cours de l'exécution de l'orchestration. Le modèle de labels que nous proposons afin de pouvoir résoudre ce problème a été présenté dans [DTTT11] et dans [DTTT12].

De la même façon que dans la section 4.3 nous proposons d'attacher des labels à chaque conteneur d'information. Ce label décrit l'ensemble des propriétaires et lecteurs de l'information contenue dans ce conteneur ainsi que les informations atomiques dont elle est issue. Nous utiliserons la syntaxe suivante pour les labels. Si c est un conteneur, son «nouveau» label de sécurité est de la forme

$$L_c = \{\mathbf{i_1} : \mathbf{s}_\alpha \rhd s_{\alpha_1}, ..., s_{\alpha_n}; ...; \mathbf{i_j} : \mathbf{s}_\beta \rhd s_{\beta_1}, ..., s_{\beta_m}\}$$

Le label L_c signifie que l'information i contenue dans c est issue des informations atomiques $\mathbf{i_1}, ..., \mathbf{i_j}$. Il a donc existé un flux des informations $\mathbf{i_1}, ..., \mathbf{i_j}$ vers c. Nous noterons dans la suite $inf(L)$ l'ensemble des informations initiales dont

```
<service>
<information>
<lecteurs>::=∅|<service>|<service>,<lecteurs>
<proprietaire>::=<service>
<label>::=∅|<information>:<proprietaire>▷<lecteurs>|
          <information>:<proprietaire>▷<lecteurs>;<label>
```

FIGURE 4.9 – Grammaire des labels

dépend l'information i contenue dans un conteneur dont le label est L. Ainsi pour ce label,

$$inf(L_c) = \{\mathbf{i_1}, ..., \mathbf{i_j}\}$$

L'information atomique $\mathbf{i_1}$ a pour propriétaire $\mathbf{s}_\alpha$ [1] qui autorise les lecteurs $s_{\alpha_1}, ..., s_{\alpha_n}$. Les propriétaires de L_c sont l'ensemble des propriétaires de chacune des informations atomiques. On a donc :

$$prop(L_c, \mathbf{i_1}) = \mathbf{s}_\alpha$$

Pour qu'un service puisse lire l'information contenue dans c, il faut qu'il soit autorisé par l'ensemble des propriétaires de c. En considérant qu'un propriétaire d'une information se définit lui-même comme lecteur de cette information, les lecteurs de L_c sont donc :

$$lec(L_c) = \{s_\alpha, s_{\alpha_1}, ..., s_{\alpha_n}\} \cap ... \cap \{s_\beta, s_{\beta_1}, ..., s_{\beta_m}\}$$

De manière plus formelle, la grammaire des labels est présenté sur la figure 4.9. Dans cette grammaire les services sont décrits à l'aide de la balise `<service>`.

Nous noterons $prop(L, i)$ le propriétaire de l'information atomique i dont est issu le contenu du conteneur dont le label est L et $lec(L, i)$ les lecteurs définis par le propriétaire pour cette information atomique.

Si l'information atomique i n'appartient pas à l'ensemble des informations atomiques de L, alors il n'existe pas de propriétaire de i pour L et l'ensemble de tous les services S est autorisé à accéder cette information atomique i par L. Ainsi par convention :

Définition 4.5.1. $prop(L, i) = \emptyset \ si \ i \notin inf(L)$

Définition 4.5.2. $lec(L, i) = S \ si \ i \notin inf(L)$

A partir de ce label nous pouvons retrouver l'ensemble des propriétaires. Il s'agit de l'union de l'ensemble des propriétaires des informations initiales. Nous noterons l'ensemble des propriétaires d'un label L, $prop(L)$:

1. Une information atomique a un seul propriétaire. Cette hypothèse a été justifiée dans la section 4.3.5.

	$inf(L)$	= $\varnothing$ si $L = \varnothing$ = i si $L = i$:<prop>$\triangleright$<lec> = $i \cup inf(L_2)$ si $L = i$:<prop>$\triangleright$<lec> ;L_2
(4.5.3)	$prop(L)$	= $\bigcup\limits_{i \in inf(L)} prop(L,i)$
(4.5.4)	$lec(L)$	= $\bigcap\limits_{i \in inf(L)} \{prop(L,i), lec(L,i)\}$
(4.5.1)	$prop(L,i)$	= $\emptyset$ si $i \notin inf(L)$ = p si $L = i : p$$\triangleright$<lec> ;$L_2$ = $prop(L_2, p)$ si $L = i_2$:<prop>$\triangleright$<lec> ;L_2 et $i \neq i_2$
(4.5.2)	$lec(L,i)$	= S si $i \notin inf(L)$ = ll si $L = i$:<prop>$\triangleright ll; L_2$ = $lec(L_2, i)$ si $L = i_2$:<prop>$\triangleright < lec >; L_2$ et $i \neq i_2$
(4.5.5)	$inf(L,p)$	= $\{i \in inf(L)/prop(L,i) = p\}$
(4.5.6)	$lec(L,p)$	= $\bigcap\limits_{i \in inf(L,p)} lec(L,i)$

TABLE 4.1 – Opérations sur les labels

Définition 4.5.3. $prop(L) = \bigcup\limits_{i \in inf(L)} prop(L,i)$

Ce label nous permet de retrouver les lecteurs autorisés à accéder à l'information i contenue dans un conteneur. En effet, un service est autorisé à accéder à l'information i, contenue dans ce conteneur, si l'ensemble des propriétaires l'autorise à accéder à cette information. Par définition un propriétaire d'une information se définit lui même comme lecteur de cette information. Nous avons la définition des lecteurs d'un label L

Définition 4.5.4. $lec(L) = \bigcap\limits_{i \in inf(L)} prop(L,i) \cup lec(L,i)$

Nous notons $inf(L,p)$ l'ensemble des informations atomiques de L dont le propriétaire est p. Nous avons alors :

Définition 4.5.5. $inf(L,p) = \{i \in inf(L)/prop(L,i) = p\}$

Nous pouvons alors déterminer les lecteurs autorisés par le propriétaire p pour le label L. C'est l'intersection des lecteurs autorisés par p pour chacune des informations atomiques dont il est le propriétaire. Nous avons alors :

Définition 4.5.6. $lec(L,p) = \bigcap\limits_{i \in inf(L,p)} lec(L,i)$

Le tableau 4.1 présente un résumé de l'ensemble des définitions des différents ensembles que nous pouvons calculer à partir d'un label L défini suivant la grammaire présentée sur la figure 4.9.

4.5.2 Initialisation et modification des labels de suivi des flux d'informations

Lorsqu'un service est appelé, celui effectue des opérations internes avant de renvoyer une réponse. Ces opérations internes produisent des flux d'information et modifient le contenu des conteneurs d'informations. Comme un label associé à un conteneur décrit les informations utilisées pour produire son contenu courant, il doit être modifié à chaque observation d'un flux d'information vers ce conteneur.

La propagation de la politique de flux entre différents services est effectuée exactement de la même manière que le mécanisme présenté dans la section 4.3.5.

Dans certains cas il n'est pas possible d'exprimer l'ensemble des flux d'informations vers un conteneur c. Lorsque l'on parle d'un flux d'information particulier vers un conteneur, la seule chose que l'on peut dire sur le label du conteneur à l'issue de l'exécution du flux c'est qu'il doit être au moins aussi restrictif que le label des informations utilisées dans le flux. Un label est plus restrictif si il permet moins d'accès que le label original, c'est à dire que le nouveau label autorise moins de lecteurs. Un label L_2 est une restriction d'un label L_1, ce que l'on note $L_1 \sqsubseteq L_2$ si L_1 a plus de lecteurs et est issu de moins d'informations atomiques que L_2.

Définition 4.5.7 (Définition de $L_1 \sqsubseteq L_2$)**.** *Soit L_1 et L_2 deux labels, nous dirons que L_2 est une restriction de L_1, ce que l'on note $L_1 \sqsubseteq L_2$ si et seulement si :*

$$inf(L_1) \subseteq inf(L_2) \tag{4.10}$$

$$\forall i \in inf(L_1), lec(L_1, i) \supseteq lec(L_2, i) \tag{4.11}$$

Nous remarquons que cette définition de la restriction ne prend pas en compte les propriétaires. En effet, chaque information atomique n'ayant qu'un seul et unique propriétaire [2], si $inf(L_1) \subseteq inf(L_2)$, alors $prop(L_1) \subseteq prop(L_2)$.

Vérifions maintenant que cette définition de la restriction permet bien de respecter la politique de flux d'informations que nous avons proposée dans la section 4.2, c'est à dire que si L_2 est une restriction de L_1 alors alors les lecteurs de L_2 sont inclus dans ceux de L_1.

Lemme 4.5.1. *Soit L_1 et L_2 deux labels tels que $L_1 \sqsubseteq L_2$ alors $lec(L_2) \subseteq lec(L_1)$*

Démonstration. Considérons deux labels L_1 et L_2 tels que $L_1 \sqsubseteq L_2$, montrons que $lec(L_2) \subseteq lec(L_1)$.

D'après la définition 4.5.4

$$lec(L_2) = \bigcap_{i \in inf(L_2)} (prop(L_2, i) \cup lec(L_2, i))$$

Or d'après la définition 4.5.7, $inf(L_1) \subseteq inf(L_2)$, donc

$$inf(L_2) = inf(L_1) \cup (inf(L_2) \backslash (inf(L_1) \cap inf(L_2)))$$

2. Cette hypothèse a été justifiée dans la section 4.3.5.

Nous avons donc :

$$\begin{aligned} lec(L_2) &= \bigcap_{i \in inf(L_1)} (prop(L_2, i) \cup lec(L_2, i)) \\ & \bigcap \bigcap_{p \in (inf(L_2) \setminus (inf(L_1) \cap inf(L_2)))} (prop(L_2, i) \cup lec(L_2, i)) \\ &\subseteq \bigcap_{i \in inf(L_1)} (prop(L_2, i) \cup lec(L_2, i)) \end{aligned}$$

Or d'après la définition 4.5.7 nous avons :

$$\forall i \in inf(L_1), lec(L_1, i) \supseteq lec(L_2, i)$$

Donc :

$$lec(L_2) \subseteq \bigcap_{i \in inf(L_1)} (prop(L_2, i) \cup lec(L_1, i))$$

Or le propriétaire d'une information atomique est unique (le service qui a envoyé cette information) et n'est jamais modifié et $inf(L_1) \subseteq inf(L_2)$ donc

$$\forall i \in inf(L_1), prop(L_2, i) = prop(L_1, i)$$

Donc :

$$lec(L_2) \subseteq \bigcap_{i \in inf(L_1)} (prop(L_1, i) \cup lec(L_1, i))$$

Donc d'après la définition 4.5.4

$$lec(L_2) \subseteq lec(L_1)$$

□

De façon générale si nous observons un flux d'information depuis les conteneurs $c_j, ..., c_k$ vers c, nous modifions alors le label de c qui devient la jointure des labels attachés à $c_j, ..., c_k$.

A cet effet nous définissons la relation $\oplus$ qui permet d'effectuer la jointure de deux labels L_1 et L_2. La relation $\oplus$ permet de construire une restriction de L_1 et de L_2 et permet ainsi de garantir la propagation de la politique de flux d'informations.

Définition 4.5.8 (Définition de $L_1 \oplus L_2$)**.** *Soit L_1 et L_2 deux labels, l'opération de jointure $L_1 \oplus L_2$ est alors définie de la manière suivante :*

$$inf(L_1 \oplus L_2) = inf(L_1) \cup inf(L_2) \tag{4.12}$$

$$\forall i \in inf(L_1 \oplus L_2), prop(L_1 \oplus L_2, i) = prop(L_1, i) \cup prop(L_2, i) \tag{4.13}$$

$$\forall i \in inf(L_1 \oplus L_2), lec(L_1 \oplus L_2, i) = lec(L_1, i) \cap lec(L_2, i) \tag{4.14}$$

$$L_1 \oplus L_2 = \bigcup_{i \in inf(L_1 \oplus L_2)} i : prop(L_1 \oplus L_2, i) \triangleright lec(L_1 \oplus L_2, i) \tag{4.15}$$

La relation 4.12 signifie que les informations atomiques qui ont produit le contenu de conteneur labelisé par $L_1 \oplus L_2$ sont l'union des informations atomiques de L_1 et de L_2. Le propriétaire d'une information atomique ne change jamais. Le résultat de la relation 4.13 est donc toujours le propriétaire de cette information atomique. L'ensemble des lecteurs de$L_1 \oplus L_2$ pour chaque information atomique de L_1 et de L_2 est l'intersection des ensembles de lecteurs correspondants. Ceci est exprimé dans la relation 4.14. La relation 4.15 permet quant à elle de définir l'opération de jointure $\oplus$ à partir des relations 4.12, 4.13 et 4.14.

Montrons que $L_1 \oplus L_2$ est une restriction de L_1 et de L_2.

Lemme 4.5.2. *Soit L_1 et L_2 deux labels, alors $L_1 \sqsubseteq (L_1 \oplus L_2)$ et $L_2 \sqsubseteq (L_1 \oplus L_2)$*

Démonstration. Soit L_1 et L_2 deux labels, montrons que $L_1 \sqsubseteq (L_1 \oplus L_2)$ et $L_2 \sqsubseteq (L_1 \oplus L_2)$.

D'après la définition 4.5.8 nous avons $inf(L_1 \oplus L_2) = inf(L_1) \cup inf(L_2)$, donc :

$$\begin{aligned} inf(L_1 \oplus L_2) &\supseteq inf(L_1) \\ inf(L_1 \oplus L_2) &\supseteq inf(L_2) \end{aligned}$$

Donc la première partie de la définition 4.5.7 est bien vérifiée.

D'après la définition 4.14 nous avons :

$$\forall i \in inf(L_1 \oplus L_2), lec(L_1 \oplus L_2, i) = lec(L_1, i) \cap lec(L_2, i)$$

Donc :

$$\begin{aligned} \forall i \in inf(L_1 \oplus L_2), lec(L_1 \oplus L_2, i) &\supseteq lec(L_1, i) \\ \forall i \in inf(L_1 \oplus L_2), lec(L_1 \oplus L_2, i) &\supseteq lec(L_2, i) \end{aligned}$$

Donc la seconde partie de la définition 4.5.7 est bien vérifiée.

Donc $L_1 \sqsubseteq (L_1 \oplus L_2)$ et $L_2 \sqsubseteq (L_1 \oplus L_2)$. □

(Exemple 14)

Afin d'illustrer la propagation des flux dans une orchestration nous considérons les variables x, y et z. L'orchestration que nous considérons est en interaction avec 7 services s_i numérotés de 1 à 7. Nous considèrerons que les variables x, y et z sont les conteneurs d'information de cette orchestration et que x et y sont associées aux labels suivants :

- $L_x = \{\mathbf{i_1} : \mathbf{s_1} \triangleright s_5, s_6; \mathbf{i_2} : \mathbf{s_1} \triangleright s_5, s_6\}$
- $L_y = \{\mathbf{i_1} : \mathbf{s_1} \triangleright s_5, s_6, s_7; \mathbf{i_3} : \mathbf{s_2} \triangleright s_6, s_7\}$

On a alors pour x :

– $inf(L_x) = \{\mathbf{i_1}, \mathbf{i_2}\}$
– $prop(L_x, i_1) = \mathbf{s_1}$, $lec(L_x, i_1) = \{s_5, s_6\}$
– $prop(L_x, i_2) = \mathbf{s_1}$, $lec(L_x, i_2) = \{s_5, s_6\}$

et pour y *:*

– $inf(L_y) = \{\mathbf{i_1}, \mathbf{i_3}\}$
– $prop(L_y, i_1) = \mathbf{s_1}$, $lec(L_y, i_1) = \{s_5, s_6, s_7\}$
– $prop(L_y, i_3) = \mathbf{s_2}$, $lec(L_y, i_3) = \{s_6, s_7\}$

Considérons un flux d'information explicite, provoqué par exemple par une affectation, de x *et* y *vers* z*. Nous avons alors le remplacement de l'information contenue dans* z *par une information provenant de* x *et de* y*. Le label de sécurité associé à* z *à l'issue de l'exécution de ce programme doit être la jointure des labels attachés à* x *et à* y*.*

D'après la définition 4.5.8 nous avons :

$$\begin{aligned} inf(L_z) &= inf(L_x \oplus L_y) \\ &= inf(L_x) \cup inf(L_y) \\ &= \{\mathbf{i_1}, \mathbf{i_2}\} \cup \{\mathbf{i_1}, \mathbf{i_3}\} \\ &= \{\mathbf{i_1}, \mathbf{i_2}, \mathbf{i_3}\} \end{aligned}$$

$$\begin{aligned} prop(L_z, i_1) &= prop(L_x \oplus L_y, i_1) \\ &= prop(L_x, i_1) \cup prop(L_y, i_1) \\ &= \mathbf{s_1} \end{aligned}$$

$$\begin{aligned} lec(L_z, i_1) &= lec(L_x \oplus L_y, i_1) \\ &= lec(L_x, i_1) \cap lec(L_y, i_1) \\ &= \{s_5, s_6\} \cap \{s_5, s_6, s_7\} \\ &= \{s_5, s_6\} \end{aligned}$$

$$\begin{aligned} prop(L_z, i_2) &= prop(L_x \oplus L_y, i_2) \\ &= prop(L_x, i_2) \cup prop(L_y, i_2) \\ &= \mathbf{s_1} \cup \emptyset \\ &= \mathbf{s_1} \end{aligned}$$

$$\begin{aligned} lec(L_z, i_2) &= lec(L_x \oplus L_y, i_2) \\ &= lec(L_x, i_2) \cap lec(L_y, i_2) \\ &= \{s_5, s_6\} \cap S \\ &= \{s_5, s_6\} \end{aligned}$$

$$\begin{aligned} prop(L_z, i_3) &= prop(L_x \oplus L_y, i_3) \\ &= prop(L_x, i_3) \cup prop(L_y, i_3) \\ &= \emptyset \cup \mathbf{s_2} \\ &= \mathbf{s_2} \end{aligned}$$

$$\begin{aligned} lec(L_z, i_3) &= lec(L_x \oplus L_y, i_3) \\ &= lec(L_x, i_3) \cap lec(L_y, i_3) \\ &= S \cap \{s_6, s_7\} \\ &= \{s_6, s_7\} \end{aligned}$$

Le nouveau label associé à z est donc :

$$L_z = \{\mathbf{i_1} : \mathbf{s_1} \rhd s_5, s_6; \mathbf{i_2} : \mathbf{s_1} \rhd s_5, s_6; \mathbf{i_3} : \mathbf{s_2} \rhd s_6, s_7\}$$

Nous avons alors :

$$prop(L_z) = prop(L_x) \cup prop(L_y) = \{s_1\} \cup \{s_1, s_2\} = \{s_1, s_2\}$$

$$lec(L_z) = lec(L_x) \cap lec(L_y) = \{s_5, s_6\} \cap \{s_5, s_6\} \cap \{s_5, s_6\} \cap \{s_6, s_7\} = \{s_6\}$$

$$inf(L_z) = inf(L_x) \cup inf(L_y) = \{i_1, i_2\} \cup \{i_1, i_3\} = \{i_1, i_2, i_3\}$$

La propagation de la politique de sécurité s'effectue via la propagation des labels attachés aux conteneurs d'information. De la même façon est propagée la dépendance aux informations initiales.

4.5.3 Lien avec le modèle de labels proposé par Myers

Montrons maintenant que le modèle de labels que nous proposons dans cette section est cohérent avec le modèle de labels proposé par Myers. Il s'agit pour cela de montrer que la définition de la restriction respecte les règles 4.3.3 et 4.4 et que la règle de jointure $\oplus$ respecte les règles 4.5 et 4.6.

Lemme 4.5.3. *Soit deux labels L_1 et L_2 tels que $L_1 \sqsubseteq L_2$, alors :*

$$prop(L_1) \subseteq prop(L_2) \tag{4.16}$$

$$\forall p \in prop(L_1), lec(L_1, p) \supseteq lec(L_2, p) \tag{4.17}$$

Démonstration. Considérons deux labels L_1 et L_2 tels que $L_1 \sqsubseteq L_2$, montrons alors que :

$$prop(L_1) \subseteq prop(L_2)$$

$$\forall p \in prop(L_1), lec(L_1, p) \supseteq lec(L_2, p)$$

Montrons d'abord la première propriété 4.16 :

Soit L_1 et L_2 deux labels tels que $L_1 \sqsubseteq L_2$, montrons que $prop(L_1) \subseteq prop(L_2)$.

D'après la définition 4.5.3 nous avons :

$$prop(L_1) = \bigcup_{i \in inf(L_1)} prop(L_1, i)$$

Or d'après la définition 4.5.7 nous avons $inf(L_1) \subseteq inf(L_2)$. De plus une information atomique a un unique propriétaire, le service qui a envoyé cette information. Donc :

$$\forall i \in inf(L_1), prop(L_1, i) = prop(L_2, i)$$

Donc nous avons :

$$prop(L_1) = \bigcup_{i \in inf(L_1)} prop(L_2, i)$$

Or d'après la définition 4.5.7 nous avons $inf(L_1) \subseteq inf(L_2)$ donc :

$$prop(L_1) \subseteq \bigcup_{i \in inf(L_2)} prop(L_2, i)$$

Donc d'après la définition 4.5.3 nous avons :

$$prop(L_1) \subseteq prop(L_2)$$

Montrons maintenant la seconde propriété 4.17 :

Soit L_1 et L_2 deux labels tels que $L_1 \sqsubseteq L_2$, montrons que $\forall p \in prop(L_1), lec(L_1, p) \supseteq lec(L_2, p)$.

D'après la définition 4.5.6 nous avons :

$$lec(L_2, p) = \bigcap_{i \in inf(L_2, p)} lec(L_2, i)$$

D'après la définition 4.5.5 nous avons

$$inf(L_2, p) = \{i \in inf(L_2) / prop(L_2, i) = p\}$$

Or d'après la définition 4.5.7 nous avons $inf(L_1) \subseteq inf(L_2)$ donc :

$$inf(L_2, p) = \{i \in inf(L_1)/prop(L_2, i) = p\} \bigcup \{i \in inf(L_2) \backslash inf(L_1)/prop(L_2, i) = p\}$$

De plus une information atomique a un unique propriétaire, le service qui a envoyé cette information. Donc :

$$\forall i \in inf(L_1), prop(L_1, i) = prop(L_2, i)$$

Donc nous avons :

$$\begin{aligned} inf(L_2, p) &= \{i \in inf(L_1)/prop(L_1, i) = p\} \\ &\bigcup \{i \in inf(L_2) \backslash inf(L_1)/prop(L_2, i) = p\} \end{aligned}$$

Donc nous avons :

$$\begin{aligned} lec(L_2, p) &= \bigcap_{\{i \in inf(L_1)/prop(L_1,i)=p\} \cup \{i \in inf(L_2) \backslash inf(L_1)/prop(L_2,i)=p\}} lec(L_2, i) \\ &= \bigcap_{\{i \in inf(L_1)/prop(L_1,i)=p\}} lec(L_2, i) \\ &\bigcap \bigcap_{\{i \in inf(L_2) \backslash inf(L_1)/prop(L_2,i)=p\}} lec(L_2, i) \\ &\subseteq \bigcap_{\{i \in inf(L_1)/prop(L_1,i)=p\}} lec(L_2, i) \end{aligned}$$

De plus d'après la définition 4.5.7 nous avons :

$$\forall i \in inf(L_1), lec(L_1, i) \supseteq lec(L_2, i)$$

Donc :

$$lec(L_2, p) \subseteq \bigcap_{\{i \in inf(L_1)/prop(L_1,i)=p\}} lec(L_1, i)$$

Donc d'après la définition 4.5.6 nous avons :

$$lec(L_2, p) \subseteq lec(L_1, p)$$

□

Lemme 4.5.4. *Soit L_1 et L_2 deux labels, alors $L_1 \oplus L_2$ respecte les propriétés suivantes :*

$$prop(L_1 \oplus L_2) = prop(L_1) \cup prop(L_2) \tag{4.18}$$

$$\forall p \in prop(L_1 \oplus L_2), lec(L_1 \oplus L_2, p) = lec(L_1, p) \cap lec(L_2, p) \tag{4.19}$$

Démonstration. Montrons d'abord la première propriété 4.18.

Soit L_1 et L_2 deux labels, montrons que $prop(L_1 \oplus L_2) = prop(L_1) \cup prop(L_2)$.

D'après la définition 4.5.3 nous avons :

$$prop(L_1 \oplus L_2) = \bigcup_{i \in inf(L_1 \oplus L_2)} prop(L_1 \oplus L_2, i)$$

D'après les règles 4.12 et 4.13 de la définition 4.5.8 nous avons :

$$\begin{aligned} prop(L_1 \oplus L_2) &= \bigcup_{i \in inf(L_1) \cup inf(L_2)} prop(L_1, i) \cup prop(L_2, i) \\ &= \bigcup_{i \in inf(L_1) \cup inf(L_2)} prop(L_1, i) \bigcup \bigcup_{i \in inf(L_1) \cup inf(L_2)} prop(L_2, i) \end{aligned}$$

Or nous avons :

$$\bigcup_{i \in inf(L_1) \cup inf(L_2)} prop(L_1, i) = \bigcup_{i \in inf(L_1)} prop(L_1, i) \bigcup \bigcup_{i \in inf(L_2) \setminus (inf(L_1) \cap inf(L_2))} prop(L_1, i)$$

D'après la définition 4.5.1 nous avons :

$$\bigcup_{i \in inf(L_2) \setminus (inf(L_1) \cap inf(L_2))} prop(L_1, i) = \emptyset$$

Donc :

$$\begin{aligned} \bigcup_{i \in inf(L_1) \cup inf(L_2)} prop(L_1, i) &= \bigcup_{i \in inf(L_1)} prop(L_1, i) \bigcup \emptyset \\ &= \bigcup_{i \in inf(L_1)} prop(L_1, i) \end{aligned}$$

De la même manière on montrerait que :

$$\bigcup_{i \in inf(L_1) \cup inf(L_2)} prop(L_2, i) = \bigcup_{i \in inf(L_2)} prop(L_2, i)$$

Nous avons donc :

$$prop(L_1 \oplus L_2) = \bigcup_{i \in inf(L_1)} prop(L_1, i) \bigcup \bigcup_{i \in inf(L_2)} prop(L_2, i)$$

Donc d'après la définition 4.5.3 nous avons :

$$prop(L_1 \oplus L_2) = prop(L_1) \cup prop(L_2)$$

Montrons maintenant la seconde propriété 4.19.

Soit L_1 et L_2 deux labels, montrons que $\forall p \in prop(L_1 \oplus L_2), lec(L_1 \oplus L_2, p) = lec(L_1, p) \cap lec(L_2, p)$.

D'après la définition 4.5.6 nous avons :

$$lec(L_1 \oplus L_2, p) = \bigcap_{i \in inf(L_1 \oplus L_2, p)} lec(L_1 \oplus L_2, i)$$

Or d'après la définition 4.5.5 nous avons :

$$inf(L_1 \oplus L_2, p) = \{i \in inf(L_1 \oplus L_2) / prop(L_1 \oplus L_2, i) = p\}$$

Donc d'après les règles 4.12 et 4.13 de la définition 4.5.8 nous avons :

$$\begin{aligned} inf(L_1 \oplus L_2, p) &= \{i \in (inf(L_1) \cup inf(L_2)) / (prop(L_1, i) \cup prop(L_2, i)) = p\} \\ &= \{i \in inf(L_1) / (prop(L_1, i) \cup prop(L_2, i)) = p\} \\ &\quad \bigcup \{i \in inf(L_2) / (prop(L_1, i) \cup prop(L_2, i)) = p\} \end{aligned}$$

Or le propriétaire d'une information atomique est unique (le service qui a envoyé cette information) et n'est jamais modifié donc :

$$\forall i \in (\inf(L_1) \cap inf(L_2)), prop(L_1, i) \cup prop(L_2, i) = prop(L_1, i) = prop(L_2, i)$$

De plus d'après la définition 4.5.1 nous avons :

$$\begin{aligned} \forall i \in (inf(L_1) \backslash (\inf(L_1) \cap inf(L_2))), prop(L_1, i) \cup prop(L_2, i) &= prop(L_1, i) \cup \emptyset \\ &= prop(L_1, i) \end{aligned}$$

Donc nous avons :

$$\forall i \in \inf(L_1), prop(L_1, i) \cup prop(L_2, i) = prop(L_1, i)$$

Et de la même manière nous pourrions montrer que :

$$\forall i \in \inf(L_2), prop(L_1, i) \cup prop(L_2, i) = prop(L_2, i)$$

Donc nous avons :

$$inf(L_1 \oplus L_2, p) = \{i \in inf(L_1) / prop(L_1, i) = p\} \bigcup \{i \in inf(L_2) / prop(L_2, i) = p\}$$

Donc d'après la définition 4.5.5 nous avons :

$$inf(L_1 \oplus L_2, p) = inf(L_1, p) \cup inf(L_2, p)$$

Nous avons alors :

$$lec(L_1 \oplus L_2, p) = \bigcap_{i \in inf(L_1,p) \cup inf(L_2,p)} lec(L_1 \oplus L_2, i)$$

Donc d'après la définition 4.14 nous avons :

$$\begin{aligned} lec(L_1 \oplus L_2, p) &= \bigcap_{i \in (inf(L_1,p) \cup inf(L_2,p))} lec(L_1, i) \cap lec(L_2, i) \\ &= \bigcap_{i \in (inf(L_1,p) \cup inf(L_2,p))} lec(L_1, i) \\ & \quad \bigcap \bigcap_{i \in (inf(L_1,p) \cup inf(L_2,p))} lec(L_2, i) \end{aligned}$$

Or nous avons :

$$\bigcap_{i \in (inf(L_1,p) \cup inf(L_2,p))} lec(L_1, i) = \bigcap_{i \in inf(L_1,p)} lec(L_1, i) \bigcap \bigcap_{i \in (inf(L_2,p) \setminus (inf(L_1,p) \cap inf(L_2,p))} lec(L_1, i)$$

Or d'après la définition 4.5.2 nous avons :

$$\bigcap_{i \in (inf(L_2,p) \setminus (inf(L_1,p) \cap inf(L_2,p))} lec(L_1, i) = S$$

Donc nous avons :

$$\begin{aligned} \bigcap_{i \in (inf(L_1,p) \cup inf(L_2,p))} lec(L_1, i) &= \bigcap_{i \in inf(L_1,p)} lec(L_1, i) \bigcap S \\ &= \bigcap_{i \in inf(L_1,p)} lec(L_1, i) \end{aligned}$$

De la même manière nous pourrions montrer que :

$$\bigcap_{i \in (inf(L_1,p) \cup inf(L_2,p))} lec(L_2, i) = \bigcap_{i \in inf(L_2,p)} lec(L_2, i)$$

Nous avons donc :

$$lec(L_1 \oplus L_2, p) = \bigcap_{i \in inf(L_1,p)} lec(L_1, i) \bigcap \bigcap_{i \in inf(L_2,p)} lec(L_2, i)$$

Donc d'après la définition 4.5.6 nous avons :

$$lec(L_1 \oplus L_2, p) = lec(L_1, p) \cap lec(L_2, p)$$

□

4.5.4 Modification de la politique à l'aide d'un service externe

Afin de permettre l'autorisation des demandes de modification de la politique de sécurité par le propriétaire d'une information, il est nécessaire de spécifier un protocole de communication entre le propriétaire de l'information et l'orchestration permettant de mettre en œuvre l'ensemble des fonctionnalités décrites dans la section 4.4.2.

Nous avons étudié l'ensemble des informations qu'il était nécessaire de transmettre au propriétaire de l'information afin qu'il puisse autoriser une modification de la politique de sécurité. Nous proposons donc que le message envoyé au propriétaire soit constitué des quatre champs suivants :

- le premier champ est constitué de l'information initiale que l'on veut déclassifier ;
- le deuxième champ, qui peut être vide, est constitué de l'information effectivement envoyée ;
- le troisième champ indique vers quel service l'information est envoyée ;
- le quatrième champ indique au propriétaire si l'information conditionne l'appel d'un service (flux implicite) ou si l'information est utilisée comme paramètre d'appel du service (flux explicite).

Nous avons également étudié la portée possible des opérations de déclassification dans la suite de l'exécution d'une orchestration. Nous proposons donc que la réponse du propriétaire soit constituée d'un seul champ permettant les trois réponses suivantes :

- le refus de la modification de la politique de sécurité. Dans ce cas l'appel de service n'est pas effectué ;
- l'acceptation de la modification de la politique de sécurité uniquement pour l'appel de service concerné. Dans ce cas il n'y a pas de modification de label de sécurité mais uniquement autorisation de l'appel du service ;
- l'acceptation de la modification de la politique de sécurité dans la suite du programme. Dans ce cas le label de la variable concernée est modifié. C'est à dire que le service appelé est ajouté à la liste des lecteurs de l'information initiale que l'on veut déclassifier. Notons cependant que cet ajout n'est effectué que pour le label de la variable utilisée afin d'appeler le service et que les labels associés aux autres variables qui dépendent de cette information initiale ne sont pas modifiés.

(Exemple 15)

Le principe du mécanisme de modification de la politique est présenté Figure 4.10. Au début de l'exécution de l'orchestration, le client envoie l'information i. A l'information i sont adjointes les informations relatives à la politique de sécurité. Ainsi le client spécifie le propriétaire de l'information, c'est-à-dire l'adresse du service de sécurité autorisé à déclassifier l'information si besoin. Il spécifie également la liste des services autorisés à accéder à cette information.

Au cours de l'exécution d'orchestration si l'information i est envoyée au

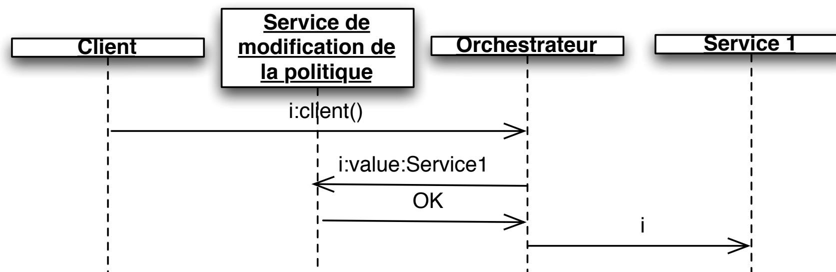

FIGURE 4.10 – Modification de la politique à l'aide d'un service externe

`service 1` qui n'est pas autorisé par la politique de sécurité liée à l'information i, alors un appel vers le service de sécurité du propriétaire de l'information i sera effectué. Si l'appel de service est autorisé alors l'appel de service est effectué en utilisant comme paramètre d'appel l'information i.

(Exemple 16)

Revenons à l'exemple 3, présenté sur la figure 4.2. Dans ce cas la politique de sécurité définie par le client consiste à s'assurer que seul le vendeur a connaissance des produits que le client a choisis et que seule la banque a connaissance des informations bancaires fournies par le client. Cela conduit donc à avoir les labels suivants :

- *Produit :* $\{\mathbf{i_1} : \mathbf{s_5} \triangleright s_1, s_2, s_3\}$
- *Coordonnées bancaires :* $\{\mathbf{i_2} : \mathbf{s_5} \triangleright s_4\}$

Dans le cas du produit choisi, l'information est d'abord transmise au service s_1 afin de calculer le montant total. Cette information étant envoyée de manière explicite au service s_1, le label associé à la réponse envoyée par s_1, le label associé à la réponse envoyée par s_1 est donc identique à celui associé au produit choisi [3]*, c'est à dire :* $\{\mathbf{i_1} : \mathbf{s_5} \triangleright s_1, s_2, s_3\}$.

Dans la suite de l'exécution du programme, cette information est utilisée pour appeler le service de paiement (s_4). Comme s_4 n'est pas autorisé par le label associé au montant total, il est donc nécessaire, afin de pouvoir continuer l'exécution du programme, d'autoriser l'envoi du montant total au service s_4.

L'orchestrateur envoie donc au service s_5, via son service d'autorisation, les informations suivantes afin de permettre l'autorisation ou non de la déclassification :

- *le produit choisi (l'information initiale pour laquelle on souhaite modifier la politique) ;*
- *le montant total (l'information effectivement envoyée), car s_5, en tant que propriétaire unique de l'information contenue dans montant total est un lecteur autorisé de cette information ;*

- s_4 *(le service de paiement de la banque vers lequel l'information est envoyée) ;*
- *flux explicite d'information (l'information est envoyée directement au service considéré).*

Le client sait donc que seul le montant total est envoyé au service bancaire. Or à partir du montant total il est difficile de revenir à l'article choisi, c'est pourquoi le client peut donc choisir d'autoriser la déclassification du montant total dans la suite de l'exécution du programme. L'information du montant total peut donc être envoyée au service s_4 *et le nouveau label associé au montant total est donc :* $\{\mathbf{i_1} : \mathbf{s_5} \triangleright s_1, s_2, s_3, s_4\}$.

Les informations du montant total et des coordonnées bancaires étant utilisées lors de l'appel de ce service, le label associé à la réponse du service s_4 *est donc le suivant :*

$$\{\mathbf{i_1} : \mathbf{s_5} \triangleright s_1, s_2, s_3, s_4; \mathbf{i_2} : \mathbf{s_5} \triangleright s_4\}$$

Après avoir considéré une modification de la politique de sécurité dans le cadre d'un flux d'information explicite, considérons désormais une déclassification dans le cadre d'un flux d'information implicite.

L'appel du service s_3 *qui entraine la préparation de la commande du client est conditionné par le fait que la banque, via le service* s_4 *ait autorisé la transaction. L'appel du service* s_3 *est donc exécuté dans une conditionnelle qui dépend de l'information renvoyée par* s_4*. Comme* s_3 *n'est pas un lecteur autorisé de cette information, il est donc demandé au service* s_5 *d'autoriser ou non la modification de la politique de contrôle de flux sur la base des informations suivantes :*

- *les coordonnées bancaires (l'information initiale que l'on souhaite déclassifier) ;*
- *le résultat de l'exécution du service bancaire (l'information effectivement utilisée) ;*
- s_3 *(le service de commande du vendeur dont l'exécution dépend l'information considérée) ;*
- *flux implicite d'information (l'exécution du service considéré dépend de l'information considérée).*

Etant donné qu'il s'agit d'un flux implicite d'information, le client peut donc autoriser l'exécution du service s_3*. Le résultat de l'exécution d'une conditionnelle ne permettant pas de remonter directement aux coordonnées bancaires. Cependant la modification de la politique n'est dans ce cas autorisée que pour l'appel de service considéré. Car l'exécution successive de flux implicites d'information permet de dévoiler à terme l'information complète.*

4.6 Bilan

Nous avons cherché dans ce chapitre à apporter une propriété de confidentialité à des données fournies par un utilisateur à un web service en lequel il n'a pas forcément confiance. Obtenir cette propriété passe, dans notre approche, par la définition d'une politique de sécurité sur les flux d'information qui sont réalisés à l'intérieur d'une orchestration (entre ses variables internes) et avec les services qu'il invoque. Le principe de l'approche est de suivre les flux d'information dans le système distribué afin de s'assurer que les données utilisateur ne seront fournies qu'aux services en lesquels il a confiance.

Cependant, nous pensons qu'une telle approche interdit souvent l'exécution correcte d'un programme. C'est pourquoi nous avons étudié le problème de la modification dynamique de la politique de sécurité, ce qui s'apparente à de la déclassification.

Nous avons ainsi, dans ce chapitre, proposé une utilisation possible du modèle de politique décentralisée proposé par Myers permettant à l'utilisateur d'une orchestration de spécifier une politique de flux d'information pour chacune des données envoyées à l'orchestration.

Nous avons enfin proposé une évolution de ce modèle permettant des modifications dynamiques de la politique de flux d'informations. La principale modification de ce modèle consiste à permettre à l'utilisateur de connaitre les informations utilisées pour produire les données pour lesquelles une modification dynamique de la politique de flux est nécessaire.

Afin de permettre l'implémentation de cette politique dans un orchestrateur de services nous avons étudié dans le chapitre 5 le suivi des flux d'informations dans BPEL *(Business Process Execution Language)*, un langage basé sur XML permettant de décrire des orchestrations de services.

L'implémentation de cette politique et des mécanismes de modification dynamique a été réalisée dans un orchestrateur BPEL écrit en Java. Cette implémentation est décrite dans le chapitre 6.

Chapitre 5

Suivi des flux d'information dans le langage BPEL

5.1 Introduction

5.1.1 Flux d'information dans le langage BPEL

Le langage BPEL [BPE07] est un langage relativement simple qui permet de décrire dans quel ordre les appels aux différents services nécessaires au bon fonctionnement du programme sont réalisés. Le programme BPEL peut recevoir des informations des utilisateurs, et utiliser ces données pour fournir des informations aux services qu'il invoque. Par conséquent, le programme BPEL réalise des flux d'information depuis les données de l'utilisateur vers les services qu'il utilise.

Les opérations principales de BPEL sont présentées dans le tableau 5.1. Nous classons les opérations en six catégories distinctes [1] :

- les opérations permettant de communiquer avec les services et qui de ce fait sont chargées d'initialiser ou de vérifier la politique de flux d'information ;
- les opérations qui ne produisent pas de flux d'information ;
- les opérations qui réalisent des flux d'information explicites ;
- les opérations qui produisent des flux d'information implicites ;
- les opérations permettant la gestion des erreurs ;
- les exécutions parallèles d'opérations.

L'information contenue dans les conteneurs est modifiée au travers des affectations et des appels de fonction effectués par le BPEL ou par les services invoqués par ce BPEL. Nous voulons que le label attaché à chaque conteneur décrive la politique de sécurité de l'information contenue dans ce conteneur ainsi que les informations initiales dont elle dépend. Pour cela les labels sont mis à jour à chaque opération sur un conteneur et donc à chaque observation d'un flux d'information.

De façon générale si nous observons un flux d'information depuis les conte-

1. Notre étude porte sur les quatres premières catégories. Les deux dernières catégories ne sont pas étudiées dans ce mémoire.

	Commande	Description	Politique de sécurité		Flux d'information	
			Entrée	Vérification	Explicite	Implicite
Affectations	Assign	Affectation			X	
Activité vide	Empty	Activité vide				
Communication avec les services	Invoke	Appel d'un Web Service	X	X	X	
	Receive	Réception d'un message entrant	X			
	Reply	Réponse à un message entrant		X		
Séquence	Sequence	Organisation séquentielle des activités				
Conditionnelle et boucle	Switch	Exécution conditionnelle d'activités				X
	While	Exécution en boucle d'activités				X
	ForEach	Boucle pouvant s'exécuter en parallèle				X
Exceptions	Compensate	Compensation				
	Terminate	Arrêt explicite d'un processus				X
	Throw	Levée d'une exception		X	X	

TABLE 5.1 – Les commandes principales du langage BPEL

```
<assign name="addition">
 <copy>
  <from expression="getVariableData(x)+getVariableData(y)" />
  <to variable="z" />
 </copy>
</assign>
```

FIGURE 5.1 – Extrait de programme BPEL additionnant les variables x et y dans z

neurs $c_j, ..., c_k$ vers c, nous modifions alors le label de c qui devient la jointure des labels attachés à $c_j, ..., c_k$.

5.1.2 Initialisation de la politique de flux d'informations

Labels des messages d'entrée

Un programme BPEL prend en entrée des messages provenant des autres services web. Comme l'ensemble des messages sont au format XML, nous modifions les entrées XML afin d'ajouter nos labels de sécurité. Nous modifions les types primitifs XML en ajoutant un attribut `readers` optionnel où les lecteurs autorisés sont représentés par des URI (l'adresse des Webs-Services) séparées par les points virgules. Si l'attribut `readers` est utilisé avec une chaine de caractères vide alors aucun service n'est autorisé à accéder à cette donnée. Si l'attribut `readers` n'est pas utilisé alors tous les services sont autorisés à accéder à cette donnée.

Afin de permettre les modifications dynamique de la politique de sécurité, chaque client héberge un service de sécurité lui permettant d'effectuer ces modifications. Ce service de sécurité est un simple web-service qui tourne sur la machine du client. Ce service reçoit toutes les requêtes destinées à modifier la politique de sécurité définie dans le programme BPEL. C'est pourquoi nous ajoutons également un attribut `owner` qui permet de spécifier l'adresse du service permettant de modifier dynamiquement la politique de sécurité.

Ces modifications sont effectuées en modifiant le fichier WSDL du programme BPEL dans lequel nous allons implémenter la politique de flux. Les fichiers WSDL

```
<xsd:complexType name="label:string">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="owner"
                              type="xsd:string" use="optional"/>
      <xsd:attribute name="readers"
                              type="xsd:string" use="optional"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

FIGURE 5.2 – Exemple d'un type entier XML labélisé

ont été présentés dans la section 1.1.2. Dans ce fichier sont modifiées les parties *Types* dans laquelle nous introduisons les types labélisés et les parties *Messages* dans laquelle les types primitifs sont remplacés par les types labélisés. La figure 5.2 présente la définition d'un entier XML labélisé.

Un type labélisé est une extension du type primitif pour lequel nous avons rajouté deux attributs `owner` et `readers` optionnels. En utilisant une extension du type primitif cela nous permet de rester compatible avec les messages SOAP originaux ne comprenant pas de labels. Si l'attribut `readers` n'est pas utilisé alors tous les services sont autorisés à accéder à cette donnée. Si l'attribut `readers` est utilisé avec une chaine de caractères vide alors aucun service n'est autorisé à accéder à cette donnée. Si l'attribut `owner` n'est pas utilisé alors le propriétaire des données ne pourra modifier la politique de flux et lors de l'envoi du message au programme BPEL un propriétaire par défaut sera associé à cette information.

Si le client est un autre service web qui n'est pas compatible avec notre système de labels, alors nous considérons le message qu'il envoie comme un ensemble d'informations atomiques sans labels (ce qui signifie que tous les services sont des lecteurs autorisés). L'application de cette propriété nous permet de rester compatible avec l'ensemble des web-services en particulier ceux qui ne mettent pas en œuvre notre mécanisme de protection.

> **(Exemple 17)**
> *Dans l'exemple présenté sur la figure 5.3 le produit choisi est accessible uniquement par les services du vendeur et de la banque représentés par leurs URI. Les coordonnées bancaires ne sont accessibles par personne et l'adresse est accessible par tout le monde.*
> *L'adresse du service permettant de modifier la politique de sécurité est spécifié par l'attribut* `owner`*.*

Jusqu'à la version 1.4 de BPEL, les messages étaient basés sur les messages WSDL dans lequel l'ensemble des informations sont présentées sous la forme de feuille d'un arbre XML. Ces feuilles sont appelées `part`. Dans ce cas, le schéma

```
<requete>
 <produitChoisi
     owner="http://client/"
     readers="http://monVendeur ; http://maBanque">
       monProduit
 </produitChoisi>
 <carteBancaire owner="http://client/" readers="">
   123456789
 </carteBancaire>
 <adresse> monAdresse </adresse>
</requete>
```

FIGURE 5.3 – Exemple d’un message XML labélisé

```
<message name="requete">
 <part name="produitChoisi" type="xsd:string">
 <part name="carteBancaire" type="xsd:string">
 <part name="adresse" type="xsd:string">
</message>
```

FIGURE 5.4 – Exemple d’un schema XML de message

XML correspondant à la requête précédente serait composé de trois parties présentées sur la figure 5.4. Dans ce cas les labels sont insérés exactement de la même façon que dans l’exemple présenté sur la figure 5.3.

5.1.3 Labels des variables de BPEL

Dans BPEL les variables peuvent être de trois types : soit un message WSDL, soit un élément XML Schema, soit une primitive XML Schema. Une variable est donc représentée par un arbre XML qui peut-être composé soit par une feuille (une simple valeur élémentaire dont le type est une primitive XML Schema), soit par des nœuds (variable complexe composée de pusieurs valeurs élémentaires, c’est à dire soit un message WSDL, soit un élément XML Schema). Les variables sont déclarées globalement au début du document de déclaration de processus ou localement pour leur donner une portée limitée. La figure 5.5 présente la syntaxe de déclaration des variables. Lors de la déclaration on doit préciser le nom et le type de la variable en spécifiant l’un des attributs suivants :

– `messageType` : correspondant à un message WSDL ;
– `element` : correspondant à un élément XML Schema ;
– `type` : correspondant à un type simple XML Schema.

Chaque valeur élémentaire étant porteuse d’une information, il est nécessaire qu’un label soit associé à chacune des feuilles de l’arbre XML afin de permettre l’expression d’une politique de sécurité pour chacune des informations élémen-

```
<variables>
   <variable name="ncname" messageType="qname"?
                  type="qname"? element="qname"?/>+
</variables>
```

FIGURE 5.5 – Structure d'une déclaration de variables

taires d'une variable XML. C'est pourquoi afin de stocker les labels associés à chaque variable, la structure de l'arbre est dupliquée et associe un label à chaque élément de la variable BPEL.

Néanmoins toutes les variables BPEL ne sont pas forcément associées à une politique de flux. Les variables BPEL qui ne sont pas associées à une politique de flux sont associées à un arbre des labels dont chacun des labels est vide. Une donnée associée à un label vide peut-être envoyée vers n'importe quel service.

(Exemple 18)

La figure 5.6 est un exemple d'une variable composée de deux parties : le montant total (`montant`) et le numéro de carte bancaire du client (`carteBancaire`). Cette variable est un élément XML Schema composée de deux primitives XML Schema. Chacun des deux éléments de la variable BPEL a son propre label dans l'arbre des labels correspondant. Après l'information initiale, la première URI du label représente la propriétaire de la donnée, et les URI suivantes, séparées par des points virgules représentent les lecteurs autorisés de cette donnée. Le `montant` est produit à partir de l'information initiale `produit`. Il a un propriétaire qui autorise deux lecteurs à accéder cette donnée. La `carteBancaire` a le même propriétaire qui autorise un seul lecteur.

5.2 Communication avec les services

Les communications avec les services se font de deux façons différentes soit sous la forme d'appels de services à l'aide de l'instruction `<invoke>` soit sous la forme de messages entrant et sortant avec les requêtes `<receive>` et `<reply>` permettant d'appeler un programme BPEL comme un *Web Service*.

Les primitives `<receive>` et `<reply>` permettent respectivement de recevoir une invocation de service et de répondre à une invocation. L'activité `<receive>` reçoit des données auxquelles sont attachées des labels de sécurité. L'activité `<reply>` fournit des informations en retour de l'appel de service. L'activité `<invoke>` permet quand à elle d'invoquer les opérations des autres services Web.

Les trois activités utilisent les mêmes attributs de base :

- `partnerLink` qui permet de spécifier le lien vers le service Web (partenaire) à utiliser ;
- `portType` qui spécifie le type de port utilisé ;

Exemple de déclaration d'une variable BPEL :

```
<variables>
   <variable name="paiement"
                   element="ins:paiement" />
</variables>
```

de son arbre XML associé :

```
<Paiement>
  <montant>12</montant>
  <carteBancaire>123123</carteBancaire>
</Paiement>
```

et l'arbre des labels correspondant :

```
|-Paiement
    |-montant : Label :
       {[produit : http://Client :
            http://monVendeur/ ; http://maBanque/]}
    |-carteBancaire : Label :
       {[carteBancaire : http://Client :
            http://maBanque/]}
```

FIGURE 5.6 – Exemple d'une variable BPEL et de l'arbre des labels associés

```
<EndpointReference xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
   <Address>ServiceURL</Address>
   <ReferenceProperties>...</ReferenceProperties> <!-- optionnel -->
   <PortType>PortTypeName</PortType> <!-- optionnel -->
   <ServiceName PortName="...">ServiceName</ServiceName> <!-- optionnel -->
</EndpointReference>
```

FIGURE 5.7 – Structure d'un élément `wsa:EndpointReference`

– `operation` permet de spécifier l'opération utilisée.

L'élément `partnerLink` nous permet d'accéder à l'URL du web-service qui nous sert à l'identifier. En effet la référence du point d'accès à un lien vers un partenaire est représentée en BPEL par l'élément XML `wsa:EndpointReference` défini par WS-Adressing. Cet élément possède la structure suivante présentée sur la figure 5.7. La référence du point d'accès `<Adress>` est le seul élément requis. Il est constitué de l'URL du service du lien vers le partenaire.

L'introduction de la politique de sécurité au sein d'un programme BPEL se fait au moment de la réception des messages lors des activités `<receive>` ou `<invoke>`. La vérification de la politique de sécurité est effectuée avant l'envoi de messages, c'est à dire lors des activités `<reply>` ou `<invoke>`.

```
<receive partnerLink="..."
         portType="..."
         operation="..."
         variable="..."
         createInstance="..." >
</receive>
```

FIGURE 5.8 – Structure d'une activité <receive>

5.2.1 Reception des messages

L'activité <receive> attend un message entrant (invocation d'opération), aussi bien pour l'initialisation d'un processus BPEL que lors de l'attente de la réponse d'un service lors d'un appel de service asynchrone. La figure 5.8 présente la structure type d'une activité <receive>. Afin de stocker le message entrant le processus utilise l'attribut `variable` qui permet de spécifier la variable de destination. L'attribut `createInstance` permet de spécifier au processus qu'il doit créer une nouvelle instance. Cela permet de créer une nouvelle instance du processus pour chaque client.

L'activité <receive> est un des points d'entrée de la politique de flux (avec l'activité <invoke>). Lors de l'envoi de message, le client du programme BPEL introduit également la politique de flux associée à chacune des informations qu'il envoie. Lors de l'exécution de l'activité <receive> la politique de flux insérée par le client dans le message est appliquée à la variable ainsi créée. La variable associée à l'activité <receive> est alors associée à l'arbre des labels créé à partir du message envoyé par le client.

> **(Exemple 19)**
> *Considérons une activité <receive> recevant le message présenté sur la figure 5.3. Cette activité receive copie ce message dans la variable `messageRecu`. Le contenu de la variable `messageRecu` est identique au message initial. La seule différence réside dans le fait que les attributs `owner` et `readers` ont été supprimés.*
> *A partir de ce message un arbre des labels a été créé. Le contenu de la variable ainsi que l'arbre des labels créé sont présentés sur la figure 5.9.*

5.2.2 Envoi des messages

L'activité <reply> est utilisée pour renvoyer une réponse dans les processus BPEL synchrones [2]. <reply> est toujours lié au <receive> initial par lequel le processus a été démarré. La figure 5.10 présente la structure d'une activité

2. Dans un processus BPEL synchrone, celui-ci est appelé comme un service Web. Il est appelé par l'activité <receive> et renvoie un message à l'aide de l'activité <reply>. Dans le cadre d'un processus BPEL asynchrone, celui-ci est appelé à l'aide de l'activité <receive>. Dans ce cas la

Variable créée à partir du message XML :

```
<requete>
 <produitChoisi > monProduit </produitChoisi>
 <carteBancaire> 123456789 </carteBancaire>
 <adresse> monAdresse </adresse>
</requete>
```

et l'arbre des labels correspondant :

```
|-requete
   |-produitChoisi : Label :
      {[produit : http://client :
          http://monVendeur/ ; http://maBanque/]}
   |-carteBancaire : Label :
      {[carteBancaire : http://Client : ]}
   |-adresse : Label :
```

FIGURE 5.9 – Variable créée à partir du message de la figure 5.3 et son arbre des labels associé

```
<reply partnerLink="..."
           portType="..."
           operation="..."
           variable="..." >
</reply>
```

FIGURE 5.10 – Structure d'une activité `<reply>`

```
<invoke partnerLink="..."
            operation="..."
            portType="..."
            inputVariable="..."
            outputVariable="..." >
</invoke>
```

FIGURE 5.11 – Structure d'une activité <invoke>

<reply>. Le seul attribut additionnel est le nom de la variable dans laquelle la réponse a été stockée.

L'activité <reply> produit un flux d'information de la variable spécifiée dans l'attribut `variable` vers le service spécifié dans l'attribut `partnerLink`. Il faut donc vérifier avant l'exécution de l'activité, que l'exécution de celle-ci est autorisée par la politique de flux. A cet effet le service défini dans l'attribut `partnerLink` doit être un des lecteurs autorisés des informations contenues dans la variable spécifiée dans l'attribut `variable`.

Si le service défini dans le `partnerLink` n'est pas un lecteur autorisé de la variable alors il est proposé aux propriétaires des informations contenues dans la variable de modifier leur politique de sécurité conformément au mécanisme proposé dans la section 4.5.4.

> **(Exemple 20)**
> *Considérons une activité <reply> qui utiliserait la variable `paiement` présentée sur la figure 5.6. Cette variable est composée de deux informations. L'information `montant` peut être envoyée au service à l'adresse `http://monVendeur` et au service à l'adresse `http://maBanque`. L'information `carteBancaire` peut-être envoyée uniquement au service à l'adresse `http://maBanque`. Donc cette activité <reply> ne pourrait adresser que le service à l'adresse `http://maBanque`. Donc le service défini dans le `partnerLink` doit avoir comme attribut <Adress> dans sa section `wsa:EndpointReference` l'adresse `http://maBanque` afin de pouvoir être exécutée. Dans le cas contraire il sera demandé au propriétaire de l'information (ici `http://Client`) si il accepte ou non de modifier la politique de sécurité.*

5.2.3 Appel de services

L'activité <invoke> permet d'invoquer les opérations des autres services web. La structure d'une activité <invoke> est présentée sur la figure 5.12. Lorsque

réponse est envoyée à un service Web spécifié par le client à l'aide de l'activité <invoke>.

le processus invoque un service web, il envoie un ensemble de paramètres. Ces paramètres sont modélisés comme des messages d'entrée de services web. Pour spécifier ces messages lors de l'invocation, l'attribut `inputVariable` est utilisé. Il permet de fournir une variable du type correspondant. Si une opération synchrone est invoquée, celle-ci renvoie un résultat. Celui-ci est stocké dans la variable spécifiée dans `outputVariable`.

L'activité `<invoke>` peut-être exprimée comme la succession d'une activité `<reply>` puis d'une activité `<receive>` avec le même service. Ainsi il est créé de la même manière que dans l'activité `<reply>` un flux d'information de la variable `inputVariable` vers le service désigné dans le `partnerLink`. Il est donc nécessaire de vérifier la politique de flux en suivant l'approche présentée dans la section 5.2.2.

Au niveau de la réception deux cas sont par contre à distinguer. Dans le premier cas le service appelé propage la politique de flux. Dans ce cas la politique de flux a été propagée à la variable renvoyée par le service appelé. Il suffit donc dans ce cas de récupérer la politique de flux associée au message renvoyé et de l'associer à la variable `outputVariable` de la même manière que pour l'activité `<receive>` présentée dans la section 5.2.1. Dans l'autre cas la politique de flux n'est pas propagée par le service appelé. Dans ce cas le service appelé est vu comme une boîte noire par le programme BPEL. On considère alors qu'il existe un flux d'informations de la variable `inputVariable` vers la variable `outputVariable`.

La politique de sécurité associée à la variable `outputVariable` à l'issue de l'exécution du service doit à la fois refléter la politique des informations renvoyées par le service appelé dans le `partnerLink` mais également que ce service a été appelé avec les informations contenues dans la variable `inputVariable`, dont il dépend directement. Le label de sécurité de `outputVariable` après l'exécution de l'appel du service est donc l'union des labels de sécurité de la variable `inputVariable` et des labels des informations atomiques renvoyées par le service. Cette union assure que, quelle que soit la politique de sécurité imposée par le service appelé, la politique imposée par le client du programme BPEL sera respectée.

(Exemple 21)

Considérons l'exemple présenté sur la figure 5.12. Le label associé à la variable d'entrée `inputVariable` *signifie que le contenu de la variable provient de l'information cb, que son propriétaire est* `http://Client` *et qu'il autorise le lecteur* `http://maBanque`*. Cette information peut donc être envoyée au service bancaire. Dans le message il est spécifié le propriétaire de l'information ainsi que les services autorisés à accéder à cette information.*

Le service bancaire renvoie la réponse rep. Le message spécifie que le propriétaire de l'information est le service à l'adresse `http://maBanque` *et que le service à l'adresse* `http://Client` *est autorisé à accéder à cette information.*

Néanmoins afin de produire l'information rep le service bancaire a utilisé

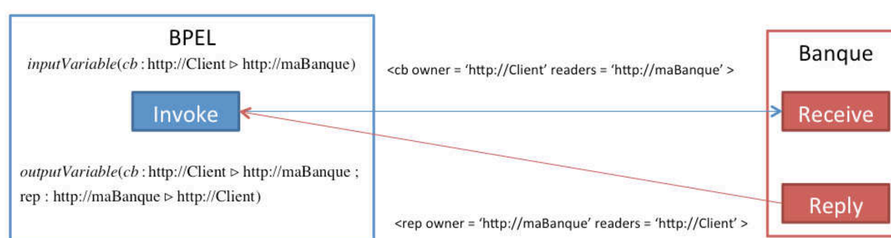

FIGURE 5.12 – Communication avec les services

```
<assign>
 <copy>
  <from ... />
  <to ... />
 </copy>
 <copy>
  <from ... />
  <to ... />
 </copy>
   ...
</assign>
```

FIGURE 5.13 – Syntaxe des affectations

l'information cb. Le label associé à `outputVariable` *doit refléter la politique de sécurité de cb et de rep. C'est donc la jonction des labels de cb et rep.*

5.3 Flux d'informations explicites

5.3.1 Les affectations

Les affectations permettent de transférer des données d'une expression, d'une variable ou d'un point d'accès vers un service [3], ou vers une variable, ou un point d'accès vers un service. L'activité `<assign>` permet d'englober une ou plusieurs commandes `<copy>`. Une commande `<copy>` est composée d'une partie source `<from>` et d'une partie destination `<to>`. La syntaxe d'une affectation est présentée sur la figure 5.13.

L'opération `<copy>` produit un flux d'informations de la source `<from>` vers la destination `<to>`. A la fin de l'opération `<copy>` le label associé à la destination `<to>` est plus restrictif que le label associé à la source `<from>` par définition de la propagation des labels.

3. Un point d'accès vers un service est l'adresse du service web definit dans un `partnerLink`

L'opération d'affectation permettant de modifier à la fois les variables et les points d'accès vers les services, il est donc nécessaire de propager la politique de sécurité à la fois aux variables mais aussi aux points d'accès vers les services afin d'éviter la création d'un canal caché.

Les sources `<from>` et destinations `<to>` doivent avoir les mêmes structures. Dans le cas de feuilles d'un arbre XML, le label associé à la feuille `<to>` doit être plus restrictif que le label associé à la feuille `<from>`. Par contre si la source `<from>` est un arbre XML, alors cette source possède un arbre des labels. En effet, dans ce cas, nous pouvons distinguer explicitement les différentes informations contenues dans les variables. A cet effet un label est associé à chacune des informations contenues dans la variable. Dans ce cas la destination `<to>` est aussi un arbre XML, et l'arbre des labels associé à la destination `<to>` doit être plus restrictif que l'arbre des labels associé à la source `<from>` à la fin de l'exécution de la fonction `<copy>`. C'est ainsi la politique de flux associée à chaque feuille qui est propagée et non la politique de flux globale associée à l'arbre XML. Cela permet ainsi de propager une politique de flux au niveau des informations quand cela est possible et non au niveau des variables.

Les formes possibles de la source `<from>` sont présentées sur la figure 5.14. La première forme représente soit une variable complète, soit une partie d'un message WSDL. La seconde forme est identique à la première tout en permettant l'exécution de requête sur la variable permettant d'extraire une partie de l'arbre XML d'une variable. Le langage de base utilisé pour spécifier les requêtes est XPath [4]. Dans ces deux cas le label associé à la source est un arbre des labels, celui associé à la variable ou une restriction de celui-ci. La troisième forme représente un point d'accès à un service. Dans ce cas le label est celui associé à ce point d'accès. La quatrième forme permet d'accéder à une partie de l'arbre XML d'une variable en fonction de propriétés. La cinquième forme est une expression. Il s'agit la plupart du temps d'expressions forgées à l'aide du langage de requêtes XPath. Dans ce cas le label associé est calculé à partir de l'union des arbres des labels associés à l'expression. Enfin la dernière forme permet d'utiliser des expressions littérales qui ne sont pas associées à des labels puisque ce sont des constantes qui ne varient pas au cours de l'exécution du programme BPEL.

Les formes possibles de la destination `<to>` sont présentées sur la figure 5.15. Les formes de la destination `<to>` sont une restriction des formes possibles de la source. Elles permettent de spécifier une variable, une partie d'une variable ou un lien vers un service.

(Exemple 22)

Revenons à l'exemple 18 présenté sur la figure 5.6. Considérons la requête `<from>` *suivante :*

```
<from variable="paiement"/>
```

4. XPath est le seul langage de requête étudié dans le cadre de cette thèse. D'autres langages de requêtes sont néanmoins utilisables avec BPEL.

```
<from variable="ncname" part="ncname"?/>
<from variable="ncname" part="ncname"? query="queryString"?/>
<!-- Processus executables uniquement -->
<from partnerLink="ncname" endpointReference="myRole|
    partnerRole"/>
<from variable="ncname" property="qname"/>
<from expression="general-expr"/>
<from> ... literal value ... </from>
```

FIGURE 5.14 – Formes possibles d'une source <from>

```
<to variable="ncname" part="ncname"?/>
<to variable="ncname" part="ncname"? query="queryString"?/>
<!-- Processus executables uniquement -->
<to partnerLink="ncname"/>
<to variable="ncname" property="qname"/>
```

FIGURE 5.15 – Formes possibles d'une destination <to>

Dans ce cas la variable "paiement" complète est copiée. La requête <from> *est donc associée à l'arbre des labels de la variable "paiement" présenté sur la figure 5.6.*

La destination <to> *correspondante est alors par exemple une copie vers une variable présentant la même structure XML. Dans ce cas c'est l'arbre des labels complet qui est transféré.*

Considérons maintenant la variable présentée sur la figure 5.9. Il s'agit d'un message XML suivant le schéma de la figure 5.4. Considérons maintenant la requête <from> *suivante :*

```
<from variable="requete" part="produitChoisi">
```

Dans ce cas seule une partie de la variable `requete` *est utilisée dans la requête* `from`*. C'est pourquoi seul le label associé à cette partie est transmis à la variable (ou partie de variable) associée à la requête* <to>*, c'est à dire :*

```
[produit : http://client :
          http://monVendeur/ ; http://maBanque/]
```

La navigation à l'intérieur d'un arbre XML se faisant le plus souvent à l'aide du langage XPath, des exemples présentant des affectations de parties de variables sont présentés dans la section 5.3.2.

```
bpws :getVariableData ('variable-name',
                 'part-name',          <!-- optionnel -->
                 'location-path')     <!-- optionnel -->
```

FIGURE 5.16 – Syntaxe de la fonction `getVariableData`

5.3.2 Les requêtes et expressions XPath

Xpath est un langage de requêtes permettant en particulier d'extraire une portion d'un arbre XML. C'est le langage de requêtes par défaut de BPEL. Une expression XPath est un chemin de localisation permettant d'accéder à tout ou partie d'un arbre XML. Le résultat de l'évaluation d'une expression XPath est une séquence contenant des nœuds et des valeurs atomiques (textes, booléens...).

En fonction de la nature (nombre, booléen, texte) des valeurs sélectionnées, XPath offre un certain nombre de fonctions. Ces fonctions sont limitées car elles sont plus destinées à être utilisées dans les prédicats que pour effectuer un traitement sur les données sélectionnées. Les fonctions les plus utilisées qui s'appliquent aux nombres sont : sum(), count() et les opérateurs arithmétiques. Les fonctions les plus utilisées qui s'appliquent aux chaines sont : subtring(), string-length(), concat().

Des variables sont le plus souvent utilisées dans les expressions XPath dans BPEL. BPEL fournit plusieurs extensions aux fonctions natives XPath. L'extension relative aux données d'une variable est la fonction `getVariableData`, qui permet d'extraire tout ou partie d'une variable BPEL. Cette fonction prend en entrée jusqu'à trois paramètres. Le premier, obligatoire, est le nom de la variable. Le second peut-être le nom d'une partie de message et le troisième la localisation du chemin. La syntaxe de cette fonction est présentée sur la figure 5.16.

(Exemple 23)

Revenons à l'exemple 18 présenté sur la figure 5.6. Dans cet exemple la variable possède deux éléments `<montant>` *et* `<carteBancaire>`*, chacun des éléments étant associé à un label. Considérons la requête* `<from>` *suivante :*

```
<from expression=
    "getVariableData('paiement', '/Paiement/montant')" />
```

Cette requête renvoie le contenu de la partie `<montant>` *de la variable "paiement". C'est donc un type simple XML Schema qui est alors renvoyé. Dans ce cas c'est le label associé au montant qui est renvoyé, c'est à dire :*

```
[produit : http://Client :
           http://monVendeur/ ; http://maBanque/]
```

```
<if>
   <condition>boolean−expression</condition>
   <!-- quelques activites -->
   <elseif>∗
      <condition>boolean−expression</condition>
      <!-- quelques activites -->
   </elseif>
   <else>?
      <!-- quelques activites -->
   </else>
</if>
```

FIGURE 5.17 – Syntaxe de l’activité <if>

5.4 Flux d’information implicites

Les flux d’information explicites sont les flux d’information les plus faciles à traiter. Cependant leur seule prise en compte ne suffit pas à propager l’ensemble de la politique de sécurité. Ainsi les conditionnelles ou les boucles génèrent des flux d’information implicites, c’est-à-dire qu’il n’existe pas de relation explicite au sein du programme entre les données mises en jeu. Dans ces cas particuliers, toute donnée manipulée à l’intérieur de la structure conditionnelle ou de la boucle dépend des variables utilisées dans la conditionnelle ou dans les conditions d’arrêt de la boucle.

5.4.1 Les conditions

En BPEL, les branchements conditionnels sont définis par l’activité <if>. Chaque activité <if> possède au moins une expression conditionelle <condition> qui détermine l’exécution ou non des activités qui suivent. Pour intégrer plusieurs branchements conditionels, un ou plusieurs branchements <elseif> optionnels peuvent-être imbriqués. Chaque branchement <elseif> est associé à sa propre expression conditionelle <condition>. L’activité <if> est terminée par une branche <else> optionnelle. La structure de l’activité <if> est présentée sur la figure 5.17. Les conditions booléennes des éléments <condition> sont exprimées dans le langage de requêtes selectionné, c’est à dire XPath.

Les opérations effectuées dans les structures <if> dépendent d’une ou plusieurs expressions conditionelles de cette structure. Il y a création d’un flux implicite d’information lors de l’exécution d’affectations ou d’appel de services dans une conditionnelle.

Dans le cas des affectations dans une conditionnelle, la variable <to> qui reçoit l’expression <from> dépend aussi de la valeur de l’expression conditionnelle de la structure <case>. On a donc un flux d’information de cette expression conditionnelle vers la variable <to>. Le label de <to> est donc l’union des labels de la

```
<if>
  <condition>"getVariableData('Acceptation')='OK' "</condition>
    <invoke partnerLink="Vendeur"
            operation="Livraison"
            portType="..."
            inputVariable="requete"
            outputVariable="reponse" >
    </invoke>
</if>
```

FIGURE 5.18 – Exemple d'une activité `<if>` produisant un flux implicite d'information

variable `<from>` et de l'expression conditionnelle `<condition>`.

Dans le cas des invocations de service dans une conditionnelle, si un appel de service est effectué dans une conditionnelle il y a un flux d'information de la condition vers l'appel de service puisque celui-ci est effectué en fonction de la valeur de l'expression conditionnelle. On doit donc au moment de l'appel de service s'assurer que celui-ci est également autorisé par la politique de sécurité associée à l'expression conditionnelle.

Dans le cas d'une structure `<if>` comportant plusieurs expressions conditionnelles `<condition>`, il y a également création d'un flux d'information depuis les expressions `<condition>` évaluées négativement. En effet, leur non-exécution implique que leur expression conditionnelle est fausse.

Enfin, il existe des flux d'informations implicites indirects vers les variables ou les appels de services modifiés dans les branchements conditionels non exécutés. Néanmoins, ils sont plus difficiles à traiter dans une analyse dynamique, c'est pourquoi l'étude de ces flux a été exclue de ce travail.

> **(Exemple 24)**
> *Considérons l'activité `<if>` présentée sur la figure 5.18. Cet exemple est la traduction dans le langage BPEL de l'exemple 13 présenté sur la figure 4.8. Dans cet exemple un appel de service est effectué au sein d'une conditionnelle. Un flux d'information est ainsi créé de la variable `Acceptation` vers l'appel de service. Ainsi l'exécution de l'appel du service de livraison du vendeur indique que la variable `Acceptation` est à la valeur `OK`. De la même façon le fait que la variable `reponse` ait été instanciée indique que le service a été exécuté et que donc la variable `Acceptation` est à la valeur `OK`. Le label associé à la variable `reponse` à l'issue de l'exécution du service de livraison doit donc être plus restrictif que le label associé à la variable `Acceptation`.*

```
<while condition="expression booleenne" >
   <!--Execute une activite ou un groupe d'activites encloses
       dans une activite <sequence>, <flow>, ou toute autre
       activite structuree -->
</while>
```

FIGURE 5.19 – Syntaxe de l'activité <while>

```
<while condition="getVariableData(i)<getVariableData(cb)">
 <assign name="incrementation">
  <copy>
   <from expression="getVariableData(i)+1" />
   <to variable="i" />
  </copy>
 </assign>
</while>
```

FIGURE 5.20 – Extrait d'une boucle <while> contenant un flux implicite d'information

5.4.2 Les boucles

BPEL supporte les boucles via l'activité <while>. Elle répète l'exécution des activités incluses jusqu'à ce que la condition booléenne ne soit plus vraie. Cette condition est exprimée par l'attribut `condition`, à l'aide du langage d'expression XPath par défaut. La syntaxe de l'activité <while> est présentée sur la figure 5.19.

Les opérations effectuées dans la structure <while> dépendent de la condition d'arrêt de cette structure. Il y a création d'un flux implicite d'information lors de l'exécution d'affectations ou d'appel de services dans une boucle.

Dans le cas des affectations dans une boucle, la variable <to> qui reçoit l'expression <from> dépend aussi de la valeur de la condition d'arrêt de la structure <while>. On a donc un flux d'information de cette condition d'arrêt vers la variable <to>. Le label de <to> est donc l'union des labels de la variable <from> et de la condition d'arrêt de la boucle <while>.

Dans le cas des invocations de service, si un appel de service est effectué dans une boucle il y a un flux d'information de la condition d'arrêt de la boucle vers l'appel de service puisque celui-ci est effectué en fonction de la valeur de l'expression de la condition d'arrêt. On doit donc au moment de l'appel de service s'assurer que celui-ci est également autorisé par la politique de sécurité associée à cette condition d'arrêt.

> **(Exemple 25)**
> *Considérons ainsi l'exemple présenté figure 5.20 où i est au départ une*

variable interne au programme BPEL sans politique de sécurité associée et cb *les coordonnées bancaires du client associées au label de sécurité suivant* $\{cb : \mathbf{client} \triangleright banque1\}$.

Dans ce cas l'affectation de i *ne dépend à priori que de* i *qui n'a pas de politique de sécurité associée. Donc si on ne prend en compte que les flux d'information directs comme dans l'exemple précédent à la fin de l'exécution de la boucle l'information contenue dans* i *n'est associée à aucune politique de sécurité et peut donc être envoyée à n'importe quel service. Cependant à la fin de l'exécution de cette boucle, la valeur de la variable* i *est égale à la valeur des coordonnées bancaires du client. On a donc création d'un flux d'information indirect des coordonnées bancaires vers la variable* i*. Il est donc nécessaire de prendre en compte les flux d'information indirects dans le cadre de la propagation des labels de sécurité. Dans ce cas comme* i *est modifiée dans une boucle qui dépend des coordonnées bancaires stockées dans* cb*, le label de sécurité associé à* i *doit être celui associé à* cb *à l'issue de la première exécution de l'affectation exécutée dans la boucle. Et donc à l'issue de l'exécution de la boucle le label de sécurité associé à* i *est* $\{cb : \mathbf{client} \triangleright banque1\}$ *et donc* i *qui contient désormais les coordonnés bancaires est associée à la même politique de sécurité que* cb.

5.5 Bilan

Nous avons étudié dans ce chapitre le suivi des principaux flux d'informations créés lors de l'exécution d'une orchestration BPEL. A cet effet nous avons proposé une méthode d'association de labels aux variables BPEL permettant d'associer une politique de sécurité à chacune des informations envoyées par le client d'une orchestration. Nous avons proposé une modification des messages utilisés pour échanger avec une orchestration BPEL permettant d'introduire une politique de sécurité propre à chaque information initiale.

Nous avons ensuite étudié les principales méthodes permettant la communication avec les services dans une orchestration BPEL. Pour chacune de ces méthodes nous avons spécifié comment vérifier la politique de flux et comment introduire de nouvelles politiques de flux lors de la réception de messages (c'est à dire d'informations externes).

Nous avons enfin étudié les principaux flux d'information explicites et implicites créés lors d'une orchestration. A cet effet nous avons plus spécialement étudié les affectations, les requêtes et expressions XPath, ainsi que les boucles et les conditionnelles.

Nous avons proposé dans le chapitre 6 une implémentation du modèle de politique de contrôle de flux d'information présenté dans le chapitre 4. Cette implémentation dans un orchestrateur BPEL écrit en Java est basée sur l'ensemble de l'étude proposée dans ce chapitre.

Chapitre 6

OrchestraFlow, un moniteur permettant le suivi des flux d'information dans un programme BPEL

6.1 Introduction

Dans le chapitre 4 nous avons présenté un modèle de politique de contrôle de flux d'information adapté aux architectures orientées services et en particulier aux orchestrations de services. Ce modèle permet de définir pour un ensemble d'informations atomiques un ensemble de services autorisés à y accéder. C'est un modèle de politique discrétionnaire qui permet à chaque utilisateur de définir la politique de contrôle de flux associée aux données qu'il produit. Dans le cadre de ce modèle nous avons en particulier présenté un mécanisme permettant la modification dynamique de la politique de contrôle de flux associée à des informations.

Le chapitre 5 a permis de présenter un langage d'orchestration de services, le langage BPEL [1] basé sur XML. Nous avons en particulier étudié les flux d'informations produits par les principales commandes de ce langage et proposé pour chacune de ces commandes un mécanisme permettant la propagation de la politique de flux associée aux informations manipulées.

L'objet du présent chapitre est de décrire une implémentation de la politique de contrôle de flux d'information et des mécanismes de contrôle de flux. A cet effet, nous proposons dans ce chapitre une implémentation, dans le cadre d'un orchestrateur BPEL écrit en JAVA, de mécanismes permettant le contrôle des flux d'informations pour le langage BPEL.

Nos objectifs étaient les suivants :

– l'interpréteur proposé doit être compatible avec l'ensemble des programmes

1. Business Process Orchestration Language

BPEL existants. Ainsi les mécanismes proposés ne doivent pas modifier la structure d'écriture des programmes BPEL et aucune instruction ne doit être rajoutée au langage ni modifiée dans sa syntaxe. Les seules modifications que nous avons apportées aux messages d'entrée et de sortie sont l'ajout d'un champ optionnel ;
- l'exécution d'un programme sans politique de flux ne doit pas être modifiée. Les seules modifications dans l'exécution d'un programme BPEL interviennent lorsqu'il y a violation de la politique de flux. En l'absence de politique de flux tous les flux doivent être autorisés et ainsi l'exécution du programme BPEL n'est pas modifiée ;
- l'insertion de la politique de flux d'informations par les utilisateurs, c'est à dire les services Web. Ils doivent également être en mesure de modifier dynamiquement la politique de contrôle de flux associée à leurs informations ;
- l'exécution d'un programme BPEL par l'interpréteur instrumenté ne doit pas entraîner de violation de la politique de flux définie par les utilisateurs. L'exécution de certaines instructions BPEL doit être modifiée afin de permettre le suivi des flux d'informations et la propagation de la politique de flux ainsi que le contrôle des flux d'informations.

Les sections 6.2 à 6.3 présentent le modèle d'architecture retenu pour l'implémentation du mécanisme de contrôle de flux dans Orchestra, un orchestrateur BPEL open-source écrit en JAVA et développé par Bull. Cet orchestrateur instrumenté a été baptisé OrchestraFlow.

6.2 Orchestra, un interpréteur BPEL

6.2.1 Choix d'Orchestra

Afin de pouvoir être compatible avec l'ensemble des programmes BPEL nous avons fait le choix de modifier un interpréteur BPEL existant afin d'y ajouter des mécanismes de contrôle de flux et de modification dynamique de la politique de contrôle de flux.

Il existe de nombreux interpréteurs BPEL open-source. Notre choix s'est porté sur Orchestra, un interpréteur BPEL produit par Bull pour deux principales raisons. Tout d'abord c'est un orchestrateur régulièrement mis à jour et qui prend en compte la dernière version de BPEL : BPEL 2.0. Mais surtout c'est l'architecture même d'Orchestra qui a motivé notre choix. En effet cet orchestrateur est basé sur la transformation du programme BPEL en arbre abstrait. Chaque instruction BPEL est transformée en une classe JAVA possédant une méthode `execute` qui décrit son exécution. La modification de l'interpréteur repose alors sur l'ajout de metadonnées dans ces classes BPEL (en particulier des labels) et sur la modification des méthodes `execute` associées.

Dans la suite de cette section nous décrivons le fonctionnement nominal d'Orchestra. Dans la section 6.3 nous décrivons les modifications que nous avons apportées à Orchestra afin de permettre le suivi et le contrôle des flux d'information.

BpelProcess
queryLanguage : String expressionLanguage : String suppressJoinFailure : boolean exitOnStandardFault : boolean targetNamespace : String wsdlInfos : WsdlsInfos resourcesRepository : ProcessResourcesRepository startElements : Map<OperationKey, InboundMessageElement> inboundMessageElements : List<InboundMessageElement> commonCorrelationSets : Collection<CorrelationSet>
newProcessInstance()

FIGURE 6.1 – Extrait de la définition de la classe `BpelProcess`

Scope
listOfVariableToBeInitialized : List<String> exitOnStandardFault : boolean variables : Map<String, Variable> isolated : boolean partnerLinks : Map<String, PartnerLink> messageExchanges : Set<String> correlationSets : Map<String, CorrelationSet> scopeNode : NodeImpl
executeActivity(BpelExecution) executeEventHandlers(BpelExecution) executeMainActivity(BpelExecution) signal(BpelExecution, String, Map<String, Object>) findCorrelationSet(String) findPartnerLink(String) findVariable(String)

FIGURE 6.2 – Extrait de la définition de la classe `Scope`

6.2.2 Exécution des programmes BPEL dans Orchestra

Transformation en arbres abstraits

Le package `org.ow2.orchestra.definition` contient les classes permettant la description d'un processus BPEL.

La racine de l'arbre d'un processus BPEL est l'élément `<process>`. Celui-ci est transformé en un objet de la classe `BpelProcess`. Un diagramme simplifié de la classe `BpelProcess` est présenté sur la figure 6.1. Cette classe reprend l'ensemble des déclarations d'une définition d'un processus BPEL sauf celles qui sont communes avec l'élément `<scope>`. Un objet de la classe `Scope` est en effet généré en même temps que la création d'un objet `BpelProcess`.

L'élément `<scope>` permet de créer un contexte de comportement pour les activités BPEL. Il permet en particulier la définition de gestionnaires de fautes, d'évènements et de compensations, de variables et d'ensembles de corrélation. L'ensemble de ces définitions se retrouvent dans la classe `Scope` dont un diagramme simplifié est présenté sur la figure 6.2.

Les éléments `<process>` et `<scope>` sont des conteneurs d'activités. Ils

```
<if>
  <condition>boolean-expression</condition>
  <!-- quelques activites -->
  <elseif>*
    <condition>boolean-expression</condition>
    <!-- quelques activites -->
  </elseif>
  <else>?
    <!-- quelques activites -->
  </else>
</if>
```

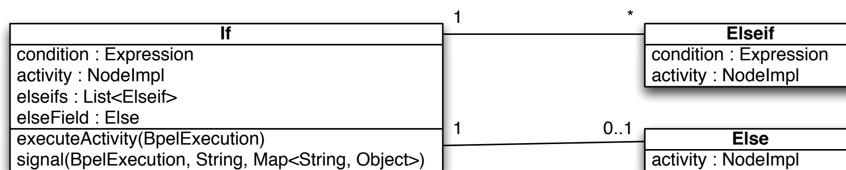

FIGURE 6.3 – Syntaxe de l'activité `<if>` et diagramme de classe correspondant

sont les racines de la structure d'arbre organisant les activités BPEL. Un programme BPEL est transformé en un arbre d'objets Java. Certains objets sont des activités BPEL et sont à ce titre destinés à être exécutés. Ils sont regroupés dans le sous package `org.ow2.orchestra.definition.activity`. Les autres objets sont des éléments du langage destinés pour la plupart à être instanciés au moment de l'exécution du processus BPEL comme par exemple la description des variables. Ces éléments sont regroupés dans le sous package `org.ow2.orchestra.definition.element`.

Chacun de ces objets représentant une activité BPEL et regroupé au sein du sous package `org.ow2.orchestra.definition.activity` contient une méthode `execute` qui contient l'ensemble des instructions qui vont permettre son exécution et d'appeler le nœud suivant. Certains objets contiennent en plus une méthode `signal` qui permet d'exécuter certaines instructions en fonction de l'arrivée d'un signal comme par exemple la réception d'un message.

Par exemple l'activité `<if>` est représentée par un objet de la classe `If`. La figure 6.3 présente la syntaxe de l'activité `<if>` ainsi que le diagramme de classe correspondant. Une classe est créée pour chacun des éléments de l'activité. Un objet de la classe `If` est constitué d'une expression conditionnelle, de l'arbre des activités effectuées si la condition est vraie, de la liste des éléments `Elseif` exécutés si jamais la condition est fausse, ainsi que d'un élément `Else`. Seul l'élément `<If>` est exécuté, les autres éléments étant uniquement des conteneurs d'activités éventuellement assortis d'une condition. Ainsi seul l'élément `<If>` possède les méthodes `signal` et `execute`.

La description des autres éléments du langage BPEL est regroupé dans le pa-

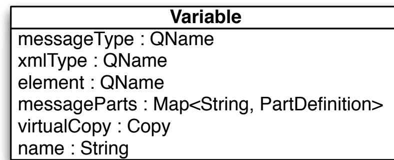

FIGURE 6.4 – Extrait de la définition de la classe `Variable`

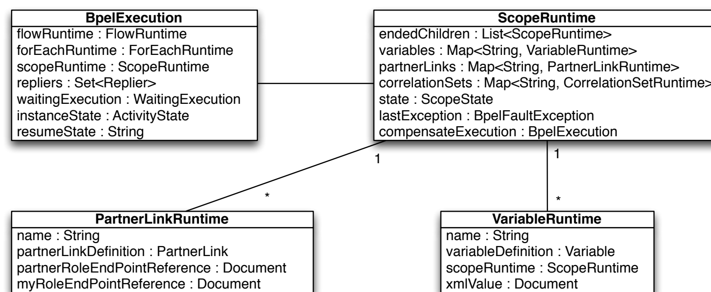

FIGURE 6.5 – Principales classes du package `org.ow2.orchestra.runtime`

ckage `org.ow2.orchestra.definition.element`. Les éléments du langage BPEL contenant l'information sont essentiellement les variables et les définitions des liens vers les partenaires. Au moment de la transformation en arbre d'objets Java seule la définition de ces éléments est considérée. La classe définissant les variables BPEL est présentée sur la figure 6.4. Ces éléments sont instanciés au moment de l'exécution du processus BPEL. Celle-ci est décrite dans la section 6.2.2.

Exécution des processus BPEL

Le package `org.ow2.orchestra.runtime` contient les principales classes permettant l'instantiation et l'exécution des processus BPEL. Les principales classes de ce package sont représentées sur la figure 6.5.

La classe `BpelExecution` permet l'exécution des processus BPEL. Les méthodes `execute` des objets représentant les activités BPEL prennent un objet de cette classe en paramètre d'entrée. Un objet de la classe `BpelExecution` représente l'état d'un processus BPEL en cours d'exécution. Elle contient en particulier l'état des exécutions parallèles, des boucles `ForEach` et du scope courant.

La classe `ScopeRuntime` représente l'état du scope courant. Elle contient en particulier la table contenant l'ensemble des variables BPEL instanciées ainsi que la table contenant l'instantiation de l'ensemble des liens vers les services.

La classe `PartnerLinkRuntime` permet d'instancier les liens vers les partenaires. Elle permet en particulier la modification des champs `partnerRole-`

Fonction	Package	Classes
Création et modification des labels	flowControl	label, treeLabel, labelNavigator
Ajout des labels aux conteneurs d'information	runtime	VariableRuntime, PartnerLinkRuntime
Introduction de la politique de flux	definition.activity	Invoke, Receive
Vérification de la politique de flux	definition.activity	Invoke, Reply
Suivi des flux explicites	definition.element definition.activity	Copy, From, To Invoke
Suivi des flux implicites	definition.activity runtime	If, RepeatUntil ScopeRuntime
Modification de la politique de flux	definition.activity runtime	Invoke, Reply BpelExecution

TABLE 6.1 – Principales modifications apportées à Orchestra par OrchestraFlow

EndPointReference, qui contient l'adresse du service appelé, et myRoleEndPointReference qui contient l'adresse de retour pour la réponse du service.

La classe VariableRuntime permet d'instancier les variables BPEL. Le contenu de la variable est un document XML contenu dans le champ xmlValue.

6.3 Implémentation d'OrchestraFlow

6.3.1 De Orchestra à OrchestraFlow

Les principales modifications apportées à Orchestra afin de permettre le suivi et le contrôle des flux d'information sont présentées sur la table 6.1.

Le package flowControl a été ajouté à Orchestra afin de permettre les opérations de créations et de modifications des labels. La classe label définit le modèle de label qui est associé à chaque conteneur d'information. La classe treeLabel permet d'associer un label à chacune des informations d'une variable BPEL. Enfin la classe labelNavigator permet de naviguer dans les arbres des labels. Les principales fonctions de création et de modification des labels sont présentées dans la section 6.3.2.

Les principaux conteneurs d'information de BPEL sont les variables et les liens vers les partenaires. Afin de permettre d'associer une politique de flux a chacun des conteneurs d'information les classes VariableRuntime et PartnerLinkRuntime du package runtime de Orchestra ont été modifiées. Ainsi un arbre des labels est ajouté à chacune des variables instanciées et un label est ajouté

à chacune des instances d'un lien vers un partenaire. Une description plus complète de ces modifications est présentée dans la section 6.3.2.

L'introduction de la politique de flux est effectuée par les services externes lors de l'envoi d'informations lors des activités `<receive>` et `Invoke`. C'est pourquoi les méthodes `execute` des classes `Invoke` et `Receive` du package `definition.activity` ont été modifiées afin de permettre l'instantiation des labels des variables recevant les informations de ces services. Une description plus complète de ces modifications est présentée dans la section 6.3.4.

La vérification de la politique de flux est effectuée par l'orchestrateur au moment de l'envoi d'informations vers les services externes, c'est à dire lors de l'exécution des activités `<receive>` et `<reply>`. A cet effet, les méthodes `execute` des classes `Receive` et `Reply` du package `definition.activity` ont été modifiées afin de permettre la vérification de la politique de flux avant l'envoi des informations vers les services externes. Une description plus complète de ces modifications est présentée dans la section 6.3.4.

Les principaux flux d'information explicites observables dans le langage BPEL sont les affectations effectuées lors de l'activité `<copy>`. A cet effet les classes `Copy`, `From` et `To` du package `definition.element` sont modifiées afin de permettre le suivi des flux directs d'information et la modification des labels des conteneurs recevant de l'information. Les services externes n'implémentant pas forcément un mécanisme de suivi des flux d'information, il est donc nécessaire lors de l'activité `<Invoke>` de suivre les flux d'informations créés depuis des variables d'envoi vers les variables de réception. A cet effet la méthode `execute` de la classe `invoke` du package `definition.activity` est modifiée. Une description plus complète de ces modifications est présentée dans la section 6.3.3.

Les principaux flux d'information implicites observables dans le langage BPEL sont issus des activités `<if>` et `<repeatuntil>`. Afin de pouvoir les suivre une pile de labels a été ajoutée à la classe `ScopeRuntime`. L'ajout et la suppression de labels de cette pile est effectuée en modifiant les méthodes `execute` des classes `If` et `RepeatUntil` du package `definition.activity`. Une description plus complète de ces modifications est présentée dans la section 6.3.3.

Lorsqu'un flux d'information illégal vers un service est détecté il est proposé aux propriétaires de l'information de modifier si besoin la politique de flux. Les classes `Invoke` et `Reply` sont modifiées afin de permettre la modification de la politique de flux. Une table des informations initiales est ajoutée à la classe `BpelExecution` afin de pouvoir spécifier aux propriétaires de l'information l'information initiale concernée. Une description plus complète de ces modifications est présentée dans la section 6.3.4.

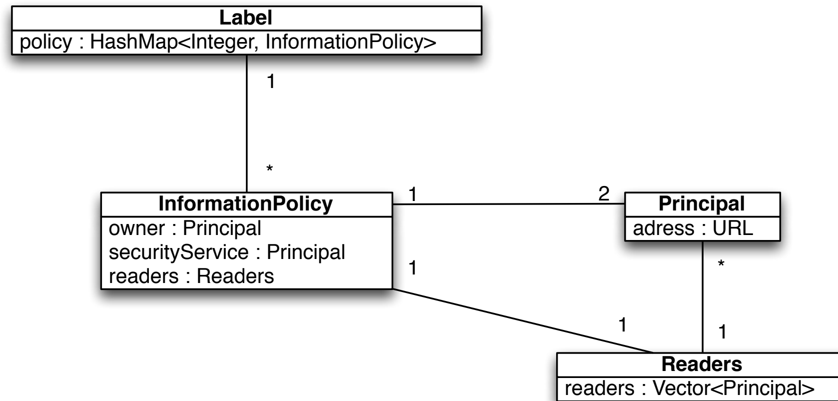

FIGURE 6.6 – Diagramme simplifié de la classe `Label` et des classes associées

6.3.2 Association de la politique de flux aux conteneurs

Création et modification des labels

Un label est associé à chaque conteneur d'information. Les labels ont été présentés dans le chapitre 4. Ils sont de la forme suivante :

$$L_c = \{\mathbf{i_1} : \mathbf{s}_\alpha \triangleright s_{\alpha_1}, ..., s_{\alpha_n}; ...; \mathbf{i_j} : \mathbf{s}_\beta \triangleright s_{\beta_1}, ..., s_{\beta_m}\}$$

Le label L_c indique que l'information contenue dans c est issue des informations atomiques $i_1, \ldots, i_j$. L'information atomique i_1 a pour propriétaire le service s_α. Le service s_α autorise les lecteurs $s_{\alpha_1}, \ldots, s_{\alpha_n}$ pour i_1.

Dans OrchestraFlow les labels sont représentés par un élément de la classe `Label`. Le diagramme de la classe `Label` est présenté sur la figure 6.6.

Un élément de la classe `Label` est un tableau regroupant les politiques de contrôle de flux d'information associées à chaque information atomique (représentée par un nombre entier) ayant servi à produire le contenu du conteneur d'information considéré. La politique de contrôle de flux d'information est un objet de la classe `InformationPolicy`, qui est composé d'un propriétaire désigné par une URL, l'adresse du service permettant au propriétaire de l'information de modifier si besoin la politique ainsi que d'une liste d'adresses de services autorisés à accéder à cette information.

L'entier désignant l'information initiale fait référence à la table des informations initiales ajoutée à la classe `BpelProcess`.

Association des labels aux conteneurs d'information

Nous avons vu au chapitre 5.1.3 qu'une variable BPEL contenait un arbre XML. Il est donc possible dans une variable BPEL d'identifier un ensemble d'in-

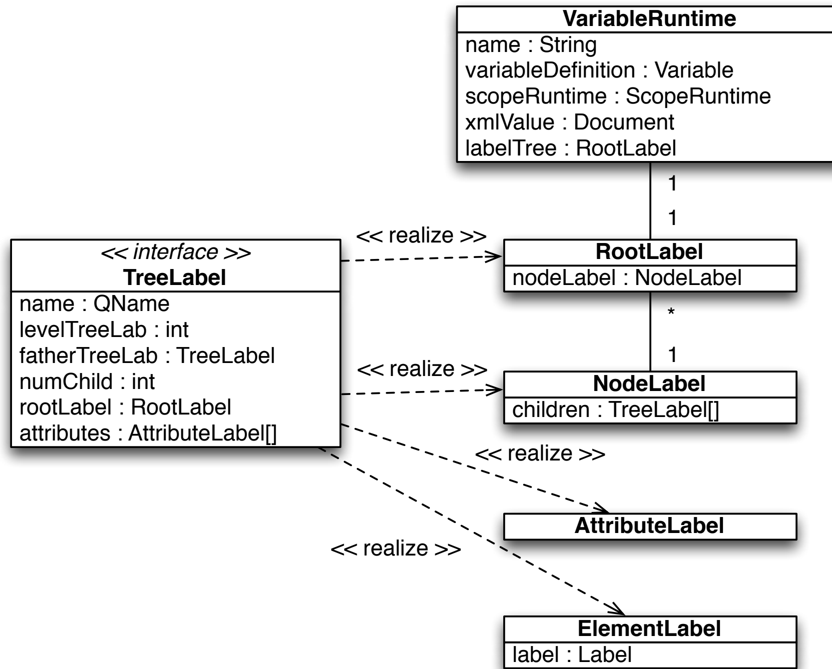

FIGURE 6.7 – Association d'un arbre des labels à une variable instanciée : diagramme de classe

formations correspondant aux feuilles de cet arbre. Nous avons donc choisi d'associer à chaque variable BPEL un arbre de labels correspondant à la structure de l'arbre XML afin d'associer un label à chacune des feuilles de cet arbre, c'est-à-dire à chaque information.

Dans OrchestraFlow un arbre des labels est associé à chaque variable instanciée de la classe `VariableRuntime`. Les classes associées à cet arbre des labels sont présentées sur le diagramme de classe présenté sur la figure 6.7. Ces classes permettent de reproduire un arbre XML. Un arbre des labels est un objet de la classe `TreeLabel`. La racine est un élément `RootLabel` et les feuilles sont des éléments `ElementLabel`. A chaque objet `ElementLabel` est associé un label de la classe `Label`.

Les variables ne sont pas les seuls conteneurs d'information que nous avons identifiés. Les liens vers les partenaires (`PartnerLink`) sont aussi modifiables, c'est pourquoi il est également nécessaire de leur associer un label. Cependant la seule information modifiable au cours de l'exécution du programme BPEL est l'adresse du service cible. Il est donc associé un label de la classe `Label` à l'instantiation de ces liens vers les partenaires, c'est à dire à la classe `PartnerLinkRuntime`.

6.3.3 Propagation de la politique de flux d'information

Suivi des flux implicites

Afin de permettre le suivi des flux implicites d'information une pile de labels est ajoutée à la classe `ScopeRuntime`. Cette pile est associée à la classe `Scope` car c'est dans cette classe que sont gérées les variables. A chaque création d'un processus BPEL à l'aide de la classe `BpelProcess`, une instance de la classe `ScopeRuntime` est créée.

L'instruction `If` ainsi que la classe associée est présentée sur la figure 6.3. Cette instruction permet l'ajout de conditions `ElseIf` contrairement à celle utilisée dans l'automate. Afin de pouvoir gérer ces conditions supplémentaires le fonctionnement de la pile est légèrement modifié. Lors du calcul de la première condition associée à l'intruction `If` le label associé à cette condition est associé à la pile. Lors du calcul des conditions associées aux instructions `ElseIf` le dernier label ajouté à la pile, c'est-à-dire celui ajouté lors du calcul de la condition `If`, est modifié afin que celui-ci soit la jonction du label préexistant et de celui associé à la condition `ElseIf`. Ainsi le dernier label de la pile prend en compte à la fois les dépendances liées à la condition `If` et celles liées aux conditions `ElseIf`. Le label ajouté à la pile au début de l'exécution de l'intruction `If` est dépilé à la fin de l'exécution de l'instruction. La figure 6.8 présente l'algorithme simplifié d'exécution de l'instruction `execute` ainsi que les modifications apportées afin de permettre le suivi des flux implicites d'information. Trois fonctions sont utilisées pour modifier la pile des labels :

- `setIFLabel` qui permet d'ajouter un label à la pile des labels ;
- `modifyIFLabel` qui permet de modifier le dernier label de la pile des labels en effectuant la jonction du label existant avec le label ajouté en paramètre de la fonction ;
- `removeIFLabel` qui permet de supprimer le dernier label de la pile des labels.

Dans le cadre de notre implémentation nous ne gérons que les flux implicites directs. Le suivi des flux implicites indirects nécessiterait de mettre en œuvre une analyse statique complémentaire afin de répercuter les dépendances aux conditions des variables modifiées dans les branches non exécutées.

Les boucles `While` sont gérées de la même façon que les conditionnelles. La figure 6.9 présente l'algorithme simplifié d'exécution de l'instruction `execute` ainsi que les modifications apportées afin de permettre le suivi des flux implicites d'information. Le label associé à la conditionnelle est ajouté à la pile des labels lors de l'initialisation de la boucle. Celui-ci est enlevé de la pile juste avant l'envoi du signal indiquant la fin de l'exécution de la pile.

Suivi des flux explicites

La fonction `copy` est la seule fonction à créer un flux d'information entre conteneurs d'information uniquement interne au programme. Les fonctions `invoke`

```
LabelXPath IfLabelXPath = new LabelXPath(this.condition.
    getText(), execution);
Label IfLabel = IfLabelXPath.getLabel(test.getRootExpr(),
    execution);
execution.setIFLabel(IfLabel);

final boolean ifResult = this.condition.getEvaluator().
    evaluateBoolean(execution);
if (ifResult) {
   execution.createExecution().execute(this.activity);
   execution.removeIFLabel();
} else {
   // manage elseifs
   for (final Elseif elseif : this.elseifs) {
      LabelXPath IfLabelXPath = new LabelXPath(this.
          condition.getText(), execution);
                              Label IfLabel = IfLabelXPath
          .getLabel(test.getRootExpr(), execution);
      execution.modifyIFLabel(IfLabel);

      final boolean elseifResult = elseif.getCondition().
          getEvaluator().evaluateBoolean(execution);
      if (elseifResult) {
         execution.createExecution().execute(elseif.
             getActivity());
         execution.removeIFLabel();
         return;
      }
}
// execute else
if (this.elseField != null) {
   execution.createExecution().execute(this.elseField.
       getActivity());
   execution.removeIFLabel();
} else {
   execution.removeIFLabel();
   this.afterRunned(execution);
}
}
```

FIGURE 6.8 – Méthode `execute` de la classe `If` modifiée afin de prendre en compte les flux implicites d'informations

```
public void executeActivity(final BpelExecution execution)
    {
final Node node = execution.getNode();
final Node firstNode = node.getNodes().get(0);

LabelXPath test = new LabelXPath(this.condition.getText(),
    execution);
Label monLabel = test.getLabel(test.getRootExpr(),
    execution);
execution.setIFLabel(monLabel);

// Execute loop on child execution
this.executeLoop((BpelExecution)execution.createExecution()
    , firstNode);
}

private void executeLoop(final BpelExecution execution,
    final Node node) {
final boolean whileResult = this.condition.getEvaluator().
    evaluateBoolean(execution);
if (whileResult) {
   execution.execute(node);
} else {
   execution.end(Execution.STATE_ENDED);

   final BpelExecution whileExecution = execution.getParent
       ();
   whileExecution.removeExecution(execution);

   execution.removeIFLabel();
   whileExecution.signal("finished");
   }
}
```

FIGURE 6.9 – Méthode `execute` de la classe `While` modifiée afin de prendre en compte les flux implicites d'informations

```
public void execute(final BpelExecution execution) {
    final Object fromValue = this.from.getValue(execution);
    Label fromLabel = this.from.getLabel(execution);

    this.to.setValue(execution, fromValue, this.
        keepSrcElementName);
    fromLabel = LabelUtil.intersectionLabel(execution.
        getIFLabel(),fromLabel);
    this.to.setLabel(execution, fromLabel, fromValue, this.
        keepSrcElementName);
}
```

FIGURE 6.10 – Méthode `execute` de la classe `Copy` modifiée afin de propager la politique de flux d'information

et `reply` qui créent des flux d'informations vers les services sont étudiées dans la section 6.3.4.

La méthode `execute` de la classe `Copy` est relativement simple. Elle se contente de recopier le contenu de l'élément `From` de la classe `Copy` vers l'élément `To`. La figure 6.10 présente la version modifiée de cette méthode permettant de propager la politique de flux d'information. A cet effet l'arbre des labels associé à l'élément `From` est calculé. La jointure entre cet arbre des labels et le label résultant de la pile des labels correspondant aux flux implicites est ensuite calculée. Ce label est ensuite associé à l'élément `To`. Ainsi lors d'une opération `Copy` sont propagés à la fois les flux explicites d'information mais aussi les flux implicites directs.

6.3.4 Communication avec les services

Introduction de la politique de flux

La politique de contrôle de flux d'information est introduite par les services communiquant avec le processus BPEL. Elle est introduite lors des activités de communication avec les services `<receive>` et `<invoke>`. Les services externes communiquent avec l'orchestration en utilisant une version modifiée des messages XML permettant l'introduction de la politique de flux présentée dans la section 5.1.2.

Le contenu du message envoyé par le service externe est transféré vers la variable spécifiée dans l'activité `<receive>`. L'arbre des labels associé à cette variable est ensuite calculé à partir du message envoyé par le service externe. Cet arbre des labels est construit selon le diagramme de classe présenté sur la figure 6.7 qui reprend les principaux éléments d'un arbre XML. L'algorithme utilisé pour calculer l'arbre des labels procède à un parcours par récurrence de l'arbre du message XML. Lorsque l'algorithme arrive à une feuille du message XML (où est attaché la politique de flux), il crée le label associé. L'information contenue dans cette feuille

est ajoutée dans la table des informations de la classe BpelExecution. Un objet Label est alors créé et associé à la feuille correspondante de l'arbre des labels ainsi créé. L'adresse du propriétaire est indiquée dans l'attribut owner et celle des lecteurs dans l'attribut readers. Ces informations sont utilisées afin de créer les attributs securityService et readers de l'objet InformationPolicy. Cet objet InformationPolicy est ajouté à la table policy de l'objet Label. L'entier servant d'index à l'InformationPolicy correspond à l'indication de l'information initiale correspondante. Dans notre cas il s'agit de l'index qui permet de retrouver l'information initiale dans la table des informations de la classe BpelExecution.

Une fois que l'arbre des labels est créé à partir de la politique de flux contenu dans le message envoyé, une opération de jonction est effectuée entre cet arbre des labels et le label résultant de la pile permettant de suivre les flux implicites d'information.

Dans le cadre de l'activité <invoke> l'introduction de la politique de contrôle de flux s'effectue de la même manière. Cependant, nous considérons que le service appelé ne propage pas la politique de flux. Nous propageons donc la dépendance entre le message envoyé au service et le message renvoyé par le service. Ainsi une opération de jonction est effectuée entre l'arbre des labels construit de la même manière que dans le cadre de l'activité <receive> et le label résultant du message envoyé au service.

Vérification et modification de la politique de flux

La politique de contrôle de flux associée aux informations est vérifiée lors de l'envoi d'informations aux services externes, c'est à dire lors des activités <invoke> et <reply>.

L'exécution d'une activité <invoke> ou <reply> s'effectue en quatre étapes :

– vérification de la politique de flux spécifiée dans la pile permettant de suivre les flux indirects et si nécessaire demande de modification ponctuelle de la politique aux services concernés. Si un des services refuse la modification ponctuelle de la politique alors l'activité n'est pas exécutée ;
– vérification de la politique associée au message envoyé et si nécessaire demande de modification de la politique aux services concernés. Si un des services refuse la modification de la politique alors l'activité n'est pas exécutée ;
– exécution de l'activité si celle-ci est autorisée par l'ensemble des services propriétaires ;
– dans le cadre d'une activité invoke progagation des différentes dépendances comme précisé précédemment.

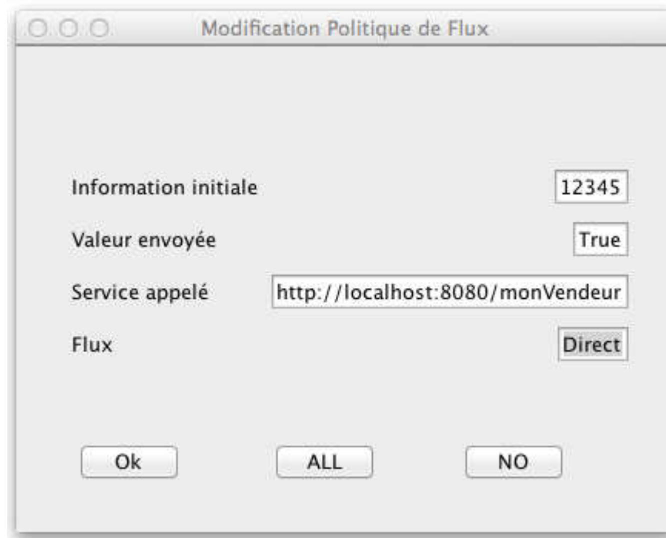

FIGURE 6.11 – Interface graphique permettant la modification de la politique de flux

Implémentation d'un service permettant de modifier une politique de flux

Nous avons fait le choix de dissocier les services dialoguant normalement avec l'orchestration qui introduisent la politique de flux et les services permettant de modifier la politique de sécurité. Ainsi à chaque service dialoguant avec l'orchestration est associé un service permettant de modifier la politique de sécurité. C'est l'adresse de ce service qui est spécifiée comme propriétaire lors de l'envoi d'un message vers l'orchestration. En l'absence de spécification de politique de flux, aucun service n'est propriétaire de l'information (cette information n'est pas nécessaire puisque aucune modification de la politique de flux n'aura à être effectuée).

Ce service se présente sous la forme d'une interface graphique présentée sur la figure 6.11. Celle-ci est ouverte lors de l'envoi d'un message par l'orchestration demandant la modification de la politique de flux. Le message envoyé est présenté dans l'interface graphique. Lorsque le flux est direct comme sur la figure 6.11 trois boutons sont présentés à l'utilisateur, lorsque le flux est indirect le bouton `ALL` n'est pas présenté. L'activation d'un des boutons produit la fermeture de l'interface graphique et l'envoi d'un des messages à l'orchestration.

Seule l'interface WSDL du service permettant de modifier la politique de flux est importante. Le message d'entrée du service doit en effet permettre l'envoi de l'information initiale, de la valeur effectivement envoyée, du service appelé ainsi que de la nature du flux. Elle doit également renvoyer un message. Un extrait du fichier WSDL de notre service de modification de la politique de flux est présenté sur la figure 6.12. D'autres implémentations du service sont possibles si elles respectent cette interface. Il est en effet tout à fait possible d'imaginer des services de

```
<definitions . . . >
  . . .
  <types>
    <xsd:schema>
      <xs:element name="natureflux" type="tns:natureflux"/>
      <xs:element name="modifpolsec" type="tns:modifpolsec"/>
      <xs:element name="modifpolsecResponse" type="
          tns:modifpolsecResponse"/>

   <xsd:simpleType name="natureflux">
     <xsd:restriction base="xsd:string">
       <xsd:enumeration value="Direct"/>
       <xsd:enumeration value="Indirect"/>
     </xsd:restriction>
   </xsd:simpleType>

  <xs:complexType name="modifpolsec">
    <xs:sequence>
      <xs:element name="Information" type="xs:string" minOccurs
          ="0"/>
      <xs:element name="ValeurEnvoyee" type="xs:string"
          minOccurs="0"/>
      <xs:element name="ServiceAppele" type="xs:anyURI"
          minOccurs="0"/>
      <xs:element name="NatureFlux" type="tns:natureflux"
          minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

   <xsd:simpleType name="modifpolsecResponse">
     <xsd:restriction base="xsd:string">
       <xsd:enumeration value="OK"/>
       <xsd:enumeration value="ALL"/>
       <xsd:enumeration value="NO"/>
     </xsd:restriction>
   </xsd:simpleType>
    </xsd:schema>
  </types>
  <message name="modifpolsec">
    <part name="parameters" element="tns:modifpolsec"/>
  </message>
  <message name="modifpolsecResponse">
    <part name="parameters" element="tns:modifpolsecResponse"/>
  </message>
  <portType name="SecurityServiceAppli">
    <operation name="modifpolsec">
      <input message="tns:modifpolsec"/>
      <output message="tns:modifpolsecResponse"/>
    </operation>
  </portType>
  <binding . . .>
  <service . . .>
</definitions>
```

FIGURE 6.12 – Extrait du fichier WSDL du service de modification de la politique de flux

	Exemple 1 Flux direct	Exemple 2 Boucle while	Exemple 3 Appel de services
Orchestra	29,33 ms	35,95ms	45,63 ms
OrchestraFlow	34,12 ms	45,62 ms	62,12 ms
Surcoût d'exécution	16%	27%	36%

TABLE 6.2 – Surcoût d'exécution

modification de la politique de flux entièrement automatisés réagissant suivant des critères définis par l'utilisateur.

6.3.5 Performances d'OrchestraFlow

Nous avons cherché à évaluer la performance d'OrchestraFlow par rapport à l'interpréteur Orchestra original. A cet effet nous avons exécuté le même programme sur Orchestra et OrchestraFlow en faisant varier le nombre d'instructions BPEL des programmes, le nombre de données initiales ainsi que la taille des labels introduits en entrée.

La mesure du temps d'exécution avec et sans les mécanismes de contrôle de flux d'informations a été effectuée en utilisant la fonction `nanoTime` de Java. Nous avons mesuré uniquement le temps d'exécution lié à l'interpréteur BPEL sans prendre en compte les communications avec les services (qui ne sont pas constant d'une invocation à l'autre).

Les résultats sont présentés sur la table 6.2. Ils montrent un surcoût d'exécution de l'ordre de 26%. Ce surcoût prend en compte à la fois la propagation et le calcul des labels ainsi que les mécanismes de contrôle des flux d'informations. Comme les orchestrations utilisées dans nos exemples sont simples, le surcoût à l'exécution est important. Cela est dû au fait que les mécanismes de sécurité ne sont pas négligeables comparé aux calculs effectués par l'orchestration.

6.4 Bilan

Nous avons étudié dans ce chapitre une implémentation de la politique de contrôle de flux d'information présentée dans le chapitre 4. Cette implémentation est présentée pour le langage d'orchestration de services BPEL. Les mécanismes permettant de suivre les flux d'information sont basés sur l'étude des flux d'information du langage BPEL présentée dans le chapitre 5.

Afin de permettre une prise en charge complète du langage BPEL, cette implémentation est basée sur l'orchestrateur BPEL open-source Orchestra proposé par Bull qui est écrit en Java. Nous avons présenté les principales modifications apportées à cet orchestrateur afin de mettre en œuvre le suivi et le contrôle des flux d'information.

Cette instrumentation d'Orchestra intègre en particulier un mécanisme permettant de vérifier la politique de contrôle de flux lors de l'appel de service. Lorsqu'une violation de la politique de flux est détectée un mécanisme de modification dynamique de la politique de contrôle de flux d'information est proposé. Ce mécanisme implémente le modèle de suivi des flux d'information proposé au chapitre 4 permettant de fournir au propriétaire d'une information les informations lui permettant de décider de modifier ou non la politique de contrôle de flux associée à cette information.

Conclusion

Nous avons présenté dans cette thèse un modèle de politique de contrôle de flux d'information pour les architectures orientées services et en particulier les orchestrations de services. L'objectif de ce modèle est de permettre à un utilisateur de spécifier une propriété de confidentialité pour chacune des informations qu'il envoie à une orchestration de services en restreignant les services qui peuvent avoir accès à ses informations.

Bilan

Les architectures orientées services proposent un modèle d'architecture distribuée basé sur la notion de service. Ces services peuvent être découverts dynamiquement au cours de l'exécution d'orchestrations de services. De ce fait, il est le plus souvent impossible de proposer une politique de contrôle de flux d'information complète au moment de l'invocation d'une orchestration de services. Il est donc nécessaire de pouvoir modifier cette politique de contrôle de flux d'information au cours de l'exécution d'une orchestration de services.

C'est pourquoi nous avons présenté un modèle de politique de contrôle de flux d'information dérivé du modèle décentralisé de Myers et Liskov [ML97] qui prend en compte l'ensemble de ces contraintes. Le modèle de politique proposé permet de contrôler les flux d'information à l'intérieur d'une orchestration et vers les services qu'elle invoque.

Le problème principal lié à la modification dynamique de la politique de contrôle de flux d'information est de permettre à l'utilisateur de connaitre les informations utilisées pour produire les données pour lesquelles une modification dynamique de la politique de contrôle de flux est nécessaire. A cet effet nous avons proposé la définition d'un modèle de label associé à des conteneurs d'information qui décrit à la fois les informations utilisées pour produire le contenu d'un conteneur mais aussi la politique de contrôle de flux d'information associée à ce conteneur.

BPEL est le langage le plus utilisé pour les orchestrations de services. C'est pourquoi nous avons proposé une implémentation de notre politique de contrôle de flux d'information pour un orchestrateur BPEL. Afin de réaliser cet orchestrateur BPEL permettant le contrôle des flux d'information nous avons étudié les flux d'information dans le langage BPEL. Puis nous avons implémenté les mécanismes

permettant la vérification et la modification dynamique de ce modèle de politique de contrôle de flux d'information au sein d'un orchestrateur BPEL écrit en Java.

Dans le cadre de l'étude des flux d'information pour le langage BPEL nous avons proposé successivement :

– une méthode d'association d'une politique de contrôle de flux aux conteneurs d'information du langage BPEL en particulier les variables ;
– une modification des messages SOAP utilisés dans le cadre des communications entre des Web Services afin de permettre l'introduction et la propagation d'une politique de contrôle de flux d'information ;
– une étude de l'introduction, de la vérification et de la transmission de la politique de contrôle de flux d'information lors des communications avec les services lors d'une orchestration BPEL ;
– une étude des principaux flux explicites et implicites se produisant dans le langage BPEL.

L'implémentation du mécanisme de contrôle de flux d'information pour un orchestrateur BPEL présentée dans cette thèse intègre en particulier un mécanisme permettant de vérifier la politique de contrôle de flux lors de l'appel de service. Lorsqu'une violation de la politique de flux est détectée un mécanisme de modification dynamique de la politique de contrôle de flux d'information est proposé. Ce mécanisme implémente le modèle de suivi des flux d'information permettant de fournir au propriétaire d'une information les informations lui permettant de décider de modifier ou non la politique de contrôle de flux associée à cette information. Afin de permettre une prise en charge complète du langage BPEL notre implémentation, appelée OrchestraFlow, est basée sur l'orchestrateur BPEL open-source Orchestra proposé par Bull.

Perspectives

Nous avons présenté dans cette thèse successivement un modèle de politique de contrôle de flux d'information, une étude des flux d'information dans le langage BPEL et une implémentation d'un mécanisme de contrôle de flux d'information pour un orchestrateur BPEL. C'est pourquoi nous proposons des perspectives dans le cadre de ces trois domaines.

Politique de contrôle de flux d'information

Le modèle de politique de contrôle de flux d'information proposé dans le cadre de cette thèse permet de décrire finement les lecteurs autorisés à lire une information. Cela permet de décrire de manière très précise une politique de contrôle de flux d'information. Cependant il est le plus souvent difficile de décrire l'ensemble des services qui peuvent être autorisés à lire une information. A cet effet il serait intéressant par exemple de pouvoir décrire des groupes des services qui pourraient être lecteurs d'une information. Ces groupes pourraient être définis par des critères

permettant de décrire les services autorisés, par exemple l'ensemble des services bancaires français. Ce nouveau modèle de politique pourrait à la fois être basé sur un langage de description de politique de contrôle de flux d'information de plus haut niveau, mais aussi par un ensemble de meta-données dans la description des services permettant ensuite de spécifier une politique de contrôle de flux d'information.

Etude des flux d'information

L'étude des flux d'information proposée dans le cadre de cette thèse ne couvre qu'une partie du langage BPEL. Il serait nécessaire d'étendre cette étude afin de couvrir l'ensemble du langage BPEL et en particulier la gestion des exécutions parallèles.

De plus notre étude ne couvre que le cas du langage BPEL, il serait intéressant de coupler cette étude avec celles d'autres langages comme par exemple JAVA afin de pouvoir proposer une solution complète de propagation d'une politique de contrôle de flux d'information au sein d'une orchestration et des Web Services.

Orchestrateur BPEL permettant le suivi des flux d'information

L'orchestrateur BPEL proposé dans le cadre de cette thèse est actuellement au stade de prototype. Il est nécessaire de poursuivre le développement de l'orchestrateur BPEL afin d'implémenter dans l'orchestrateur l'ensemble des mécanismes décrit dans l'automate formel permettant le contrôle des flux d'information ainsi que la modification de la politique.

Les performances actuelles des mécanismes de contrôle de flux d'information intégrés à notre orchestrateur BPEL montrent une surcharge processeur de l'ordre de 30%. Il est donc nécessaire d'améliorer les performances des mécanismes de contrôle de flux d'information à la fois en modifiant l'architecture actuelle des mécanisme de contrôle de flux d'information mais aussi en intégrant par exemple un ensemble d'analyses statiques permettant de déterminer par avance certains flux d'information.

Enfin il est nécessaire d'améliorer la sécurité intrinsèque de l'orchestrateur BPEL. Il est en effet nécessaire d'intégrer des mécanismes permettant de protéger la politique de contrôle de flux d'information en particulier contre sa modification non autorisé par des tiers. Il est enfin nécessaire de protéger les communications entre les services et l'orchestration afin d'empêcher les attaques par le milieu. A cet effet il serait nécessaire d'intégrer les mécanismes de sécurité des Web Services en particulier WS-Security qui permet de protéger la confidentialité et l'intégrité des messages échangés entre les Web Services.

Bibliographie

[AS07] A. Askarov and A. Sabelfeld. Localized delimited release : combining the what and where dimensions of information release. In *PLAS '07 : Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 53–60, New York, NY, USA, 2007. ACM.

[BCP06] Elisa Bertino, Jason Crampton, and Federica Paci. Access control and authorization constraints for ws-bpel. In *Proceedings of the IEEE International Conference on Web Services*, pages 275–284, Washington, DC, USA, 2006. IEEE Computer Society.

[BDF06] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Security issues in service composition. In *In Proceedings of FMOODS 2006, 8th IFIP Internat. Conf. on Formal Methods for Open Object-Based Distributed Systems, volume 4037 of LNCS*, pages 1–16. Springer, 2006.

[BFG04] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Verifying policy-based security for web services. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 268–277, New York, NY, USA, 2004. ACM.

[BFG05] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. A semantics for web services authentication. *Theor. Comput. Sci.*, 340 :102–153, June 2005.

[BFGO05] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Greg O'Shea. An advisor for web services security policies. In *Proceedings of the 2005 workshop on Secure web services*, SWS '05, pages 1–9, New York, NY, USA, 2005. ACM.

[Bib77] K. J. Biba. Integrity considerations for secure computers systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division , Bedford, 1977.

[BL73] D. Bell and L. J. LaPadula. Secure computer systems : Mathematical foundations. Technical Report 2547, MITRE, 1973.

[BL76] D. E. Bell and L. J. LaPadula. Secure computer system : Unified exposition and multics interpretation. Mtr-2997 (esd-tr-75-306), MITRE Corp., 1976.

[BPE07] Web services business process execution language version 2.0. OASIS Standard, April 2007.

[BR06] P.H. Bammigatti and P.R. Rao. Genericwa-rbac : Role based access control model for web applications. In *Information Technology, 2006. ICIT '06. 9th International Conference on*, pages 237 –240, dec. 2006.

[CCM08] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas : Toward a secure voting system. *Security and Privacy, IEEE Symposium on*, 0 :354–368, 2008.

[CDK+02] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web : An introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 6 :86–93, March 2002.

[Cer02] Ethan Cerami. *Web Services Essentials*. O'Reilly and Associates, Inc., Sebastopol, CA, USA, 2002.

[CF07] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 463 –475, dec. 2007.

[CFH05] B. Carminati, E. Ferrari, and P.C.K. Hung. Web service composition : A security perspective. In *Web Information Retrieval and Integration, 2005. WIRI '05. Proceedings. International Workshop on Challenges in*, pages 248 – 253, april 2005.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 198–209, New York, NY, USA, 2004. ACM.

[CSH09] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, labeld-based secuirty enforcement for web applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, JUNE 2009. To appear.

[CVM07] Stephen Chong, K. Vikram, and Andrew C. Myers. Sif : enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1 :1–1 :16, Berkeley, CA, USA, 2007. USENIX Association.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7) :504–513, 1977.

[Den76] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5) :236–243, 1976.

[DTTT10] Thomas Demongeot, Eric Totel, Valérie Viet Triem Tong, and Yves Le Traon. Protection des données utilisateurs dans une orchestration de web-services. In *SARSSI 2010*, 2010.

[DTTT11] Thomas Demongeot, Eric Totel, Valérie Viet Triem Tong, and Yves Le Traon. Preventing data leakage in service orchestration. In *Information Assurance and Security (IAS), 2011 7th International Conference on*, pages 122 –127, dec. 2011.

[DTTT12] Thomas Demongeot, Eric Totel, Valérie Viet Triem Tong, and Yves Le Traon. User data confidentiality in an orchestration of web services. *International Journal of Information Assurance and Security*, 7, 2012.

[EK08] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 301–313, New York, NY, USA, 2008. ACM.

[EKV+05] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39 :17–30, October 2005.

[FBFF07] Klaus-Peter Fischer, Udo Bleimann, Woldemar Fuhrmann, and Steven M. Furnell. Security policy enforcement in bpel-defined collaborative business processes. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 685–694, Washington, DC, USA, 2007. IEEE Computer Society.

[Fen74] N. S. Fenton. Memoryless subsystems. *The computer Journal*, 17(2) :143 – 147, 1974.

[FGQ96] Simon N. Foley, Li Gong, and Xiaolei Qian. A security model of dynamic labeling providing a tiered approach to verification. In *Proceedings of the 1996 IEEE conference on Security and privacy*, SP'96, pages 142–153, Washington, DC, USA, 1996. IEEE Computer Society.

[FMS07] G. Frankova, F. Massacci, and M. Seguran. From early requirements analysis towards secure workflows. In *Trust Management : Proceedings of IFIPTM 2007 : Joint iTrust and PST Conferences on Privacy, Trust Management and Security*, 2007.

[Fra06] Michael Franz. Moving trust out of application programs : A software architecture based on multi-level security virtual machines. Technical Report TR. 06-10, University of California, 2006.

[FSG+01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-

based access control. *ACM Trans. Inf. Syst. Secur.*, 4 :224–274, August 2001.

[GBJS06] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *In ASIAN'06 : the 11th Asian Computing Science Conference on Secure Software*, 2006.

[GBS06] G. Le Guernic, A. Banerjee, and D. Schmidt. Automaton-based non-interference monitoring. Technical Report 2006-1, Kansas State University, 2006.

[GHTVTT11] Stéphane Geller, Christophe Hauser, Frédéric Tronel, and Valérie Viet Triem Tong. Information flow control for intrusion detection derived from mac policy. In *Proceedings of the 2011 IEEE International Conference on Communications (ICC)*, page 6 p., Kyoto, Japon, Jun 2011.

[GJD07] Nils Gruschka, Meiko Jensen, and Torben Dziuk. Event-based application of ws-security policy on soap messages. In *Proceedings of the 2007 ACM workshop on Secure web services*, SWS '07, pages 1–8, New York, NY, USA, 2007. ACM.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. *Proc. IEEE Symp. on Security and Privacy*, pages 75 – 86, April 1982.

[HAM06] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. From languages to systems : Understanding practical application development in security-typed languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 153–164, Washington, DC, USA, 2006. IEEE Computer Society.

[HCF05a] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[HCF05b] Vivek Haldar, Deepak Chandra, and Michael Franz. Practical, dynamic information-flow for virtual machines. In *Programming Language Interference and Dependence (PLID'05)*, 2005.

[HFBK03] Bret Hartman, Donald J. Flinn, Konstantin Beznosov, and Shirley Kawamoto. *Mastering Web Services Security*. Wiley Publishing, 2003.

[Hie08] Guillaume Hiet. *Détection d'intrusion paramétrée par la politique de sécurité grâce au contrôle collaboratif des flux d'informations au sein d'un système d'exploitation et des applications : mise en oeuvre sous Linux pour les programmes Java*. PhD thesis, Supélec, 2008.

[HK03] Patrick C. K. Hung and Kamalakar Karlapalem. A secure workflow model. In *Proceedings of the Australasian information security workshop conference on ACSW frontiers 2003 - Volume 21*, ACSW Frontiers '03, pages 33–41, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[HKN06] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 119 –128, nov. 2006.

[HTMM09] Guillaume Hiet, Valerie Viet Triem Tong, Ludovic Me, and Benjamin Morin. Policy-based intrusion detection in web applications by monitoring java information flows. *Int. J. Inf. Comput. Secur.*, 3 :265–279, January 2009.

[HV06] Dieter Hutter and Melanie Volkamer. Information flow control to secure dynamic web service composition. In *Security in Pervasive Computing*, volume 3934 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006.

[ITS91] Critères d'évaluation de la sécurité des systèmes informatiques (itsec). Office des publications officielles des Communautés européennes, Juin 1991.

[JG09] Meiko Jensen and Nils Gruschka. Privacy against the business partner : Issues for realizing end-to-end confidentiality in web service compositions. In *20th International Workshop on Database and Expert Systems Application*, 2009.

[JGHL07] Meiko Jensen, Nils Gruschka, Ralph Herkenhoner, and Norbert Luttenberger. SOA and web services : New technologies, new standards - new attacks. In *Web Services, 2007. ECOWS '07. Fifth European Conference on*, pages 35–44, Halle, Germany, november 2007.

[Jor87] C.S. Jordan. A guide to understanding discretionary access control in trusted systems. Technical Report Library S-228, National Computer Security Center (NCSC), Fort George G. Meade, Maryland, 1987.

[Jos07] N. Josuttis. *SOA in practice : The Art of Distributed System Design*. O'Reilly, 2007.

[KCE+04] P. Kearney, J. Chapman, N. Edwards, M. Gifford, and L. He. An overview of web services security. *BT Technology Journal*, 22(1) :27–42, January 2004.

[KFE98] M. H. Kang, J. N. Froscher, and B. J. Eppinger. Towards an infrastructure for mls distributed computing. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 91–, Washington, DC, USA, 1998. IEEE Computer Society.

[KFEM00] Myong H. Kang, Judith N. Froscher, Brian J. Eppinger, and Ira S. Moskowitz. A strategy for an mls workflow management system. In *Proceedings of the IFIP WG 11.3 Thirteenth International Conference on Database Security : Research Advances in Database and Information Systems Security*, pages 161–174, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[KFS+99] Myong H. Kang, Judith N. Froscher, Amit P. Sheth, Krys Kochut, and John A. Miller. A multilevel secure workflow management system. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering*, CAiSE '99, pages 271–285, London, UK, 1999. Springer-Verlag.

[KYB+07] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[Lam71] Butler W. Lampson. Protection. In *Proc. 5th Princeton Conf. on Information Sciences and Systems*, 1971.

[LC04] Peng Liu and Zhong Chen. An access control model for web services in business process. In *Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '04, pages 292–298, Washington, DC, USA, 2004. IEEE Computer Society.

[LC06] Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[LGV+09] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric : a platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 321–334, New York, NY, USA, 2009. ACM.

[Lou06] P. Louridas. Soap and web services. *Software, IEEE*, 23(6) :62 –67, nov.-dec. 2006.

[LZ05] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. *SIGPLAN Not.*, 40(1) :158–170, 2005.

[LZV08] P.A. Laplante, Jia Zhang, and J. Voas. What's in a name ? distinguishing between saas and soa. *IT Professional*, 10(3) :46 –50, may-june 2008.

[Ma05] Kevin J. Ma. Web services : What's real and what's not ? *IT Professional*, 7 :14–21, March 2005.

[MH06] Esmiralda Moradian and Anne Hakansson. Possible attacks on xml web services. *International Journal of Computer Science and Network Security*, 6(1B), January 2006.

[MJS11] Filip Majernik, Meiko Jensen, and Jörg Schwenk. Marv - data level confidentiality protection in bpel-based web service compositions. In *SARSSI 2011 - Conference Proceedings*, 2011.

[ML97] A. C. Myers and B. Liskov. A decentralized model for information flow control. *Proc. ACM Symp. on Operating System Principles*, pages 129 – 142, October 1997.

[MR92] M. D. McIlroy and J. A. Reeds. Multilevel security in the unix tradition. *Softw. Pract. Exper.*, 22(8) :673–694, aug 1992.

[MR07] H. Mantel and A. Reinhard. Controlling the what and where of declassification in langage-based security. In R. De Nicolas, editor, *ESOP 2007*, volume 4421, pages 141–156. LNCS Springer Verlag, 2007.

[MS04] H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. In *Proc. Asian Symp. on Programming Langages and Systems*, volume 3302, pages 129–145. LNCS Springer Verlag, November 2004.

[Mye99] Andrew C. Myers. Jflow : Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 228–241, 1999.

[Nak04] Shin Nakajima. Model-checking of safety and security aspects in web service flows. In *Web Engineering*, volume 3140 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004.

[ND96] Vincent Nicomette and Yves Deswarte. A multilevel security model for distributed object systems. In *Computer Security — ESORICS 96*, volume 1146 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996.

[NJK+07] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07*, 2007.

[NN92] H. Riis Nielson and F. Nielson. *Semantics With Applications*. John Wiley and Sons, 1992.

[Nor09] N.A. Nordbotten. Xml and web services security standards. *Communications Surveys Tutorials, IEEE*, 11(3) :4 –21, quarter 2009.

[NSCT08] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow

control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1) :3–16, feb 2008.

[Pas05] J. Pasley. How bpel and soa are changing web services development. *Internet Computing, IEEE*, 9(3) :60 – 67, may-june 2005.

[PH07] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures : approaches, technologies and research issues. *The VLDB Journal*, 16(3) :389–415, 2007.

[PLC09] A.L. Petrescu, C. Leordeanu, and V. Cristea. Secure workflow execution in grid environments. In *New Technologies, Mobility and Security (NTMS), 2009 3rd International Conference on*, pages 1 –5, dec. 2009.

[RM09] Sabina Rossi and Damiano Macedonio. Information flow security for service compositions. In *ICUMT*, pages 1–8. IEEE, 2009.

[RPB+09] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar : practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 63–74, New York, NY, USA, 2009. ACM.

[RS06] HariGovind V. Ramasamy and Matthias Schunter. Multi-level security for service-oriented architectures. In *Proceedings of the 2006 IEEE conference on Military communications*, MILCOM'06, pages 760–766, Piscataway, NJ, USA, 2006. IEEE Press.

[RS10] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 186 –199, july 2010.

[RVC07] José Rouillard, Thomas Vantroys, and Vincent Chevrin. *Architectures orientées services. Une approche pragmatique des SOA*. Vuibert, 2007.

[SAM05] Assertions and protocols for the oasis security assertion markup language (saml) v2.0. OASIS Standard, March 2005.

[San93] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11) :9–19, nov 1993.

[SARL12] Lilia Sfaxi, Takoua Abdellatif, Riadh Robbana, and Yassine Lakhnech. Information flow control of component-based distributed systems. *Concurrency and Computation : Practice and Experience*, pages n/a–n/a, 2012.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3 :30–50, February 2000.

[SCH08] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable : A language for enforcing user-defined security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, MAY 2008.

[Sim03] Vincent Simonet. The Flow Caml System : documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), jul 2003. ©INRIA.

[SM03] A. Sabelfeld and A. Meyers. Language-based information-flow security. *IEEE J. on selected areas in communications*, 21(1) :5 – 19, January 2003.

[SM04] A. Sabelfeld and A. C. Myers. A model for delimited information release. *Proc. International Symp. on Software Security (ISSS'03)*, 3233 of LNCS :174 – 191, October 2004.

[Smi07] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 297–307. Springer-Verlag, 2007.

[SOA07] Soap version 1.2 part 1 : Messaging framework (second edition). W3C Recommendation, April 2007.

[SP07] L. Singaravelu and C. Pu. Fine-grain, end-to-end security for web service compositions. In *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pages 212 –219, july 2007.

[SR10] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back : riding the roller coaster of information-flow control research. In *Proceedings of the 7th international Andrei Ershov Memorial conference on Perspectives of Systems Informatics*, PSI'09, pages 352–365, Berlin, Heidelberg, 2010. Springer-Verlag.

[SRMM11] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.

[SS07] A. Sabelfeld and D. Sands. Declassification : Dimensions and principles. *Journal of Computer Security*, 2007.

[SST07] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 203–217, Washington, DC, USA, 2007. IEEE Computer Society.

[SWS07] A. Singhal, T. Winograd, and K. Scarfone. Guide to secure web services. recommendations of the national institute of standards and technology. Technical report, NIST, August 2007.

[SY07] F. Satoh and Y. Yamaguchi. Generic security policy transformation framework for ws-security. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 513 –520, july 2007.

[UDD04] Uddi version 3.0.2. UDDI Spec Technical Committee Draft, OASIS, 2004.

[VBC+04] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Blo Jason A., George A. Reis, Manish Vachharajani, and David I. August. Rifle : An architectural framework for user-centric information-flow security. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[VE06] J. Viega and J. Epstein. Why applying standards to web services is not enough. *Security Privacy, IEEE*, 4(4) :25 –31, july-aug. 2006.

[Vol00] Dennis Volpano. Secure introduction of one-way functions. In *Proceedings of the 13th IEEE workshop on Computer Security Foundations*, pages 246–, Washington, DC, USA, 2000. IEEE Computer Society.

[VS00] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 268–276, New York, NY, USA, 2000. ACM.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3) :167 – 187, 1996.

[VXDS06] V. N. Venkatakrishnan, Wei Xu, Daniel C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *ICICS*, pages 332–351, 2006.

[Wea04] A.C. Weaver. Enforcing distributed data security via web services. In *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, pages 397 – 402, sept. 2004.

[Wei69] C. Weissman. Security controls in the adept-50 time-sharing system. In *Proceedings of the November 18-20, 1969, fall joint computer conference*, AFIPS '69 (Fall), pages 119–133, New York, NY, USA, 1969. ACM.

[WSC07] Ws-secureconversation 1.3. OASIS Standard, March 2007.

[WSD01] Web services description language (wsdl) 1.1. W3C Note, March 2001.

[WSP06] Web services policy 1.2 - framework (ws-policy). W3C, April 2006.

[WSP07] Ws-securitypolicy 1.2. OASIS Standard, July 2007.

[WSS06] Web services security : Soap message security 1.1. OASIS Standard Specification,, Feb 2006.

[WST07] Ws-trust 1.3. OASIS Standard, March 2007.

[XAC05] extensible access control markup language (xacml) version 2.0. OASIS Standard, Feb 2005.

[XML02] Xml encryption syntax and processing. W3C Recommendation, December 2002.

[XML08] Xml signature syntax and processing (second edition). W3C Recommendation, June 2008.

[YCTU07] Yumi Yamaguchi, Hyen-Vui Chung, Masayoshi Teraguchi, and Naohiko Uramoto. Easy-to-use programming model for web services security. In *Proceedings of the The 2nd IEEE Asia-Pacific Service Computing Conference*, APSCC '07, pages 275–282, Washington, DC, USA, 2007. IEEE Computer Society.

[YLBM08] Qi Yu, Xumin Liu, Athman Bouguettaya, and Brahim Medjahed. Deploying and managing web services : issues, solutions, and directions. *The VLDB Journal*, 17 :537–572, May 2008.

[YYW+07] Sachiko Yoshihama, Takeo Yoshizawa, Yuji Watanabe, Michiharu Kudoh, and Kazuko Oyanagi. Dynamic information flow control architecture for web applications. In *Computer Security – ESORICS 2007*, volume 4734 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

[ZA11] H. Zorgati and T. Abdellatif. Sewsec : A secure web service composer using information flow control. In *Risk and Security of Internet and Systems (CRiSIS), 2011 6th International Conference on*, pages 1 –8, sept. 2011.

[ZBWKM06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[ZM07] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Secur.*, 6 :67–84, March 2007.

[ZMB03] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Experimenting with a policy-based hids based on an information flow control model. In *In proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2003.