HAL
open science

# COORDINATION ADAPTATIVE DE SERVICES A BASE DE CONTRATS

Tan Hanh

## To cite this version:

Tan Hanh. COORDINATION ADAPTATIVE DE SERVICES A BASE DE CONTRATS. Base de données [cs.DB]. Université de Grenoble, 2009. Français. NNT : . tel-01011472

**HAL Id: tel-01011472**

**https://theses.hal.science/tel-01011472v1**

Submitted on 27 Jun 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.

INSTITUT POLYTECHNIQUE DE GRENOBLE

N°attribué par la bibliothèque

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

**THÈSE**

pour obtenir le grade de

**DOCTEUR DE L'Institut polytechnique de Grenoble**

**Spécialité : « Informatique : Systèmes et Logiciels »**

préparée au laboratoire LABORATOIRE **d'**INFORMATIQUE **de** GRENOBLE **(LIG)**

dans le cadre de l'École Doctorale

**« Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII) »**

présentée et soutenue publiquement par

**TAN HANH**

Le 26 06- 2009

**Titre :**

# Coordination adaptative de services à base de contrats

**Directeurs de thèse :**

Mme. Christine COLLET et Mme Genoveva VARGAS-SOLAR

**JURY**

| | |
|---|---|
| Mme. Corine CAUVET, | Président |
| M. Omar BOUCELMA, | Rapporteur |
| M. Frédérique LAFOREST, | Rapporteur |
| Mme. Christine COLLET, | Directeur de thèse |
| Mme. Genoveva VAGAR-SOLAR, | Co-directeur de thèse |
| Mme. Corine CAUVET, | Examinateur |

# Table des matières

**1 Introduction** **1**

1.1 Contexte et motivation . . . . . 1

1.2 Contribution . . . . . 3

1.2.1 Coordination de services . . . . . 4

1.2.2 Classification de méthodes de service en famille . . . . . 5

1.2.3 Notion de contrat d'adaptabilité . . . . . 5

1.3 Organisation du document . . . . . 6

**2 Etat de l'art : Composition adaptative de services** **9**

2.1 Taxonomie de composition de services . . . . . 10

2.1.1 Composition statique de services . . . . . 11

2.1.2 Composition dynamique de services . . . . . 13

2.2 Adaptation de l'exécution d'une composition de services . . . . . 19

2.2.1 Substitution de services . . . . . 21

2.2.2 Adaptation par changement du plan d'exécution . . . . . 27

2.2.3 Rediriger l'activité vers une activité de traitement d'exceptions . . 28

2.3 Conclusion . . . . . 30

**3 Coordination adaptative de services** **33**

3.1 Approche . . . . . 33

3.1.1 Exemple d'une coordination de services . . . . . 34

3.1.2 Propositions . . . . . 35

3.2 Coordination de services . . . . . 39

3.2.1 Service . . . . . 39
3.2.2 Coordination . . . . . 42
3.2.3 Famille de méthodes de service . . . . . 47
3.2.4 Equivalence de familles de méthodes . . . . . 53
3.3 Opérations d'adaptation . . . . . 57
3.3.1 Substitution de méthodes de service . . . . . 58
3.3.2 Substitution d'activités . . . . . 60
3.4 Journal d'exécution . . . . . 69
3.5 Conclusions . . . . . 72

**4 Contrat d'adaptabilité 75**
4.1 Classe AdaptabilityContract . . . . . 78
4.2 Critère de QoS . . . . . 80
4.3 Réaction . . . . . 82
4.3.1 Exceptions . . . . . 83
4.3.2 Action d'adaptation . . . . . 85
4.4 Exemple de l'instanciation d'un contrat d'adaptabilité . . . . . 93
4.5 Conclusion . . . . . 97

**5 Evaluation de contrats d'adaptabilité 99**
5.1 Architecture générale . . . . . 99
5.1.1 Gestionnaire d'événements . . . . . 100
5.1.2 Exécuteur de contrats . . . . . 101
5.2 Evaluation d'un contrat d'adaptabilité . . . . . 103
5.3 Exécution d'une réaction d'un contrat . . . . . 105
5.3.1 Recherche de méthodes équivalentes . . . . . 105
5.3.2 Evaluation des critères de QoS . . . . . 107
5.3.3 Exécution d'une action d'adaptation . . . . . 108
5.3.4 Exécution de réactions déclenchées en cascade . . . . . 111
5.4 Calculs des mesures de QoS d'une méthode de service . . . . . 112

5.5 Implantation . . . . . 113
5.5.1 SEBAS . . . . . 113
5.5.2 Actions d'adaptation . . . . . 114
5.5.3 La substitution de méthodes . . . . . 115
5.6 Conclusion . . . . . 116

**6 Conclusion et perspective 117**
6.1 Résumé du travail effectué . . . . . 117
6.2 Perspectives . . . . . 119

**A Pré-requis 127**
A.0.1 Domaine . . . . . 127
A.0.2 Type $\bigtriangledown$ . . . . . 127
A.0.3 Classe . . . . . 129
A.0.4 Predicat . . . . . 134

**B Contraintes de flots de contrôle 137**

**C Ontologies 139**
C.1 Règles de transformation . . . . . 139
C.2 Ontologie de familles de méthodes de service . . . . . 140
C.3 Ontologie d'exceptions . . . . . 144

**D Calcul des mesures de QoS 147**
D.1 Temps d'exécution d'une famille composite . . . . . 147
D.1.1 Temps d'exécution du flot de contrôle *Sequence* . . . . . 147
D.1.2 Temps d'exécution du flot de contrôle *AndSplit*/*AndJoin* . . . . . 148
D.1.3 Temps d'exécution d'une méthode de service . . . . . 150
D.2 Fiabilité d'une famille composite . . . . . 150
D.2.1 Fiabilité du flot de contrôle textitSequence . . . . . 150
D.2.2 Fiabilité du flot de contrôle *AndSplit*/*AndJoin* . . . . . 151
D.2.3 Fiabilité du flot de contrôle *OrSplit*/*OrJoin* . . . . . 151

D.2.4 Fiabilité d'une méthode de service . . . . . 152
D.3 Disponibilité d'une famille composite . . . . . 152
D.3.1 Disponibilité d'une méthode de service . . . . . 152
D.4 Mise à jour de QoS de méthodes . . . . . 152

**E API du moteur de worflow (*XFLow*) 155**

# Table des figures


1.1 Application de recherche de vols à base de services . . . . . 2

1.2 Modèle de coordination adaptative de services à base de contrats d'adaptabilité . . . . . 4

1.3 Coordination de services de recherche de vols associée aux contrats d'adaptabilité . . . . . 6

2.1 Taxonomie de composition de services . . . . . 10

2.2 Orchestration de services . . . . . 12

2.3 Chorégaphie de services . . . . . 13

2.4 Protocole d'interactions de services . . . . . 15

2.5 Mapping entre les descriptions de services en OWL-S et d'actions en PDDL 16

2.6 Décomposition de la méthode $M$ . . . . . 18

2.7 Représentation de composition de services à base de fonctionnalité . . . . 22

2.8 Ontologie de service de AgFlow . . . . . 24

2.9 Plan de processus et plan d'exécution . . . . . 28

2.10 Types de traitement d'exceptions . . . . . 29

3.1 La coordination de services de recherche de vols . . . . . 34

3.2 Vision de la coordination de services adaptative . . . . . 36

3.3 Interface de service . . . . . 39

3.4 Graphe du plan d'exécution de la coordination de recherche de vols . . . 46

3.5 Familles de méthodes de consultation de météo et de recherche de vols . . 50

3.6 Une composition de familles pour la convertion de devises des tarifs de vols cherchés . . . . . 52

3.7 Equivalences entre les familles . . . . . 56

3.8 Replannification du plan d'exécution de Co à Co1 . . . . . 61

3.9 Opération ReplaceSeqActivity . . . . . 63

3.10 Opération ReplaceAndActivity . . . . . 65

3.11 Opération ReplaceOrActivity . . . . . 66

3.12 Substitution de l'activité A par une sous-composition . . . . . 68

4.1 Modèle du contrat d'adaptabilité . . . . . 75

4.2 Le contrat d'adaptabilité "C3" associé aux activités "GetFlight1" et "GetFlight2" . . . . . 77

4.3 L'instanciation des instances du contrat "C3" . . . . . 79

4.4 Hiérarchie de types d'exceptions . . . . . 84

4.5 Hiérarchie de types d'action d'adaptation . . . . . 86

4.6 L'instanciation de l'action de réexécution *Retry1* . . . . . 87

4.7 L'instanciation de l'action de notification Notify1 . . . . . 88

4.8 L'initialisation de l'instance *ReplaceMethod1* . . . . . 90

4.9 L'initialisation de l'objet *ReplaceActivity1* au moment de l'exécution . . 92

4.10 Contrat C1 de la coordination de recherche de vols . . . . . 94

4.11 L'instanciation des instances du contrat "C1" . . . . . 95

5.1 Architecture de SEBAS . . . . . 100

5.2 Interactions entre le moteur de coordination et le gestionnaire d'événements 100

5.3 Interactions entre le ContractExecutor et ses partenaires . . . . . 101

5.4 Structure générale d'un contrat d'adaptabilité . . . . . 103

5.5 Processus d'évaluation d'un contrat d'adaptabilité . . . . . 103

5.6 Sélection d'une réaction de la liste Reactions . . . . . 104

5.7 L'instance *ReplaceMethod1* de l'action *ReplaceMethodAction* . . . . . 106

5.8 Les mesures de QoS de la méthode *GFKLM* appelé par l'action *Retry1* . 107

5.9 Interactions entre le ContractExecutor et ses partenaires pour exécuter l'action de réexécution . . . . . 114

5.10 Interactions entre le ContractExecutor et ses partenaires pour exécuter l'action de notification . . . . . 115

5.11 Interactions entre le ContractExecutor et ses partenaires pour exécuter l'action de substitution de méthodes . . . . . 116

C.1 Ontologie de familles de méthodes . . . . . 140

C.2 Ontologie d'exception . . . . . 144

D.1 Une séquence de familles de méthodes . . . . . 148

D.2 And/Split-join de Familles . . . . . 148

D.3 Or/split-join de Familles . . . . . 149

# Liste des tableaux

4.1 Classes d'exceptions . . . . . 85

5.1 Caculs des mesures de QoS d'une méthode de service . . . . . 112

5.2 Les outils utilisés dans SEBAS . . . . . 113

D.1 Tableau 1 des mesures de QoS . . . . . 148

D.2 Tableau 2 des mesures de QoS des familles . . . . . 149

# Chapitre 1

# Introduction

## 1.1 Contexte et motivation

Récemment, nous avons vu divers services de commerce se développer pour fournir des services adaptés aux besoins des clients. La tendance courante montre une augmentation exponnentielle du nombre de services qui seraient accessibles au-delà de l'Internet. Ces services sont nécessairement des services miniatures et autonomes offrant une fonctionnalité spécifique basée sur certaines entrées. Les exemples de tels services incluent des services dans le domaine financier ou des prévisions météorologiques basées sur le code postal, etc.

Même si les services simples répondent à certain besoins, il est aussi nécessaire de les composer pour réaliser un processus métier. Deux standards de composition de services BPEL4WS [ACD+03] et WSCI [KBR+05] ont été développés respectivement par OASIS et W3C pour répondre à ce besoin. Les langages de ces standards sont conçus pour fournir l'interopérabilité entre les applications (services). Bien que ces plateformes indépendantes aux interfaces de services permettent de l'intégration de systèmes hétérogènes, elles souffrent de l'incapacité de composer dynamiquement les fonctionnalités fournies par les services existants afin de donner plus d'informations aux clients. Cette incapacité vient du manque de représentation sémantique des services disponibles sur l'Internet comme la représentation de leurs fonctionnalités. Pour répondre à cette limitation, plusieurs solutions ont été proposées par la communauté du sémantique web comme OWL-S [BHL+04]. Par ailleurs, la capacité de s'adapter dynamiquement aux changements de l'environnement donne au système de composition de services plus de fiabilité et d'autonomie.

La figure 1.1 illustre une application de recherche de vols à base de services. Cette application est conçue par une coordination de services. Chaque activité de la coordination réalise un appel à une méthode exportée par un service. Les services comme *KLM*, *AirFrance* sont les services fournis par les compagnies aériennes tandis que le service *TravelAgency* est le service interne de l'agence de tourisme.

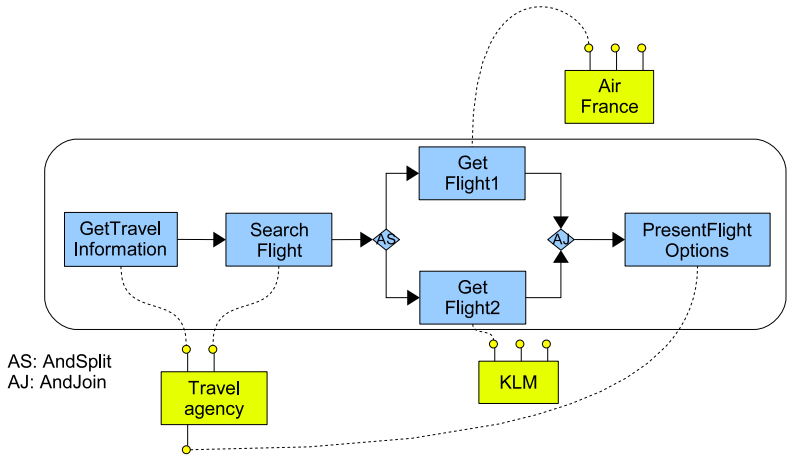


FIG. 1.1 – Application de recherche de vols à base de services

Cependant, avoir une coordination de services en le composant dynamiquement ou statiquement, ne garantit pas que son exécution déroule normalement et se termine avec succès car son exécution fait des appels aux services externes qui sont hors de portée de contrôle de la coordination comme les services *KLM* et *AirFrance* dans l'exemple présenté. Cela nécessite d'avoir une adaptation pour s'adapter aux exceptions produites au moment de l'exécution.

L'adaptation d'une coordination de services permet de garantir sa continuation et sa terminaison avec succès quand une exception se produit et bloque son exécution. Elle est réalisée au moment de l'exécution. L'adaptation peut être réalisée manuellement par l'intervention d'un utilisateur ou dynamiquement par les opérations d'adaptation prédéfinies.

L'adaptation d'une coordination de services est nécessaire pour :

1. **s'adapter à l'évolution des processus d'entreprise**
   L'économie globale étant volatile et dynamique, les organismes changent constamment aussi bien leurs politiques de commerce, d'organisation et que de gestion. Par conséquent, des processus métier et de gestion correspondants doivent être modifiés pour s'adapter aux changements.

   Par ailleurs, la plupart des services composés représentant des chaînes complexes d'activités ou processus ont une exécution de longue durée. Les organismes qui utilisent de tels services composés peuvent avoir besoin de changer leurs processus métiers pour s'adapter aux changements des conditions d'application, des technologies, des politiques commerciales. En conséquence, la modification d'exécution de composition de services est nécessaire pour autoriser l'adaptation aux changements.

2. **S'adapter dynamiquement pour améliorer la performance de l'exécution du processus**
   Dans les moteurs de coordination, une modification d'exécution est une procédure manuelle, qui est longue et coûteuse. Afin de réduire le coût et de fournir des ré-

ponses rapides à ces changements, il est nécessaire d'automatiser la modification d'exécution d'instances d'une coordination de services en cours d'exécution et également d'autoriser la modification de son schéma d'exécution.

Les standards de coordination de services comme BPEL, WSCI ne permettent pas de s'adapter dynamiquement aux changements de l'environnement. La plupart des travaux récents dans le domaine de la coordination de services concernent plutôt la composition dynamique que l'adaptation [ZBN+04, TBFM06, YL05, CDK+06]. Nous avons constaté que la complexité (le temps d'exécution) d'une composition dynamique de services est coûteuse. Ceci ne permet pas d'appliquer les techniques de composition dynamique pour s'adapter à une coordination de services.

L'adaptation d'une coordination de service proposée dans [SH03] est basée sur l'adaptation d'un workflow [vdABV+00]. Elle propose les opérations d'adaptation limitées comme Retry, Ignore, Resume. Ces opérations ne peuvent pas répondre aux fautes permanentes.

Dans ce contexte, l'objectif de la thèse est de proposer une coordination adaptative de services qui permet de

- réagir dynamiquement aux exceptions produites lors d'un appel à une méthode de service pour garantir sa terminaison.

- séparer les propriétés fonctionnelles comme la logique applicative et la propriété non-fonctionnelle (adaptabilité) qui permette de spécifier explicitement et modifier les comportements d'adaptation sans influencer la définition de la coordination de services.

## 1.2 Contribution

La principale contribution de cette thèse est le modèle de coordination adaptative de services à base de contrat d'adaptabilité qui fournit :

- les concepts de coordination de services sous forme d'un workflow,

- la classification de méthodes en familles qui permet aux concepteurs de domaines d'application de définir les méthodes équivalentes et les mappings entre leurs entrées et sorties pour résoudre ses incompatibilités et,

- la notion de contrat d'adaptabilité qui permet aux concepteurs d'application de spécifier explicitement les réactions qui spécifient les actions d'adaptation réagissant aux exceptions spécifiées en garantissant les critères de QoS exigée.

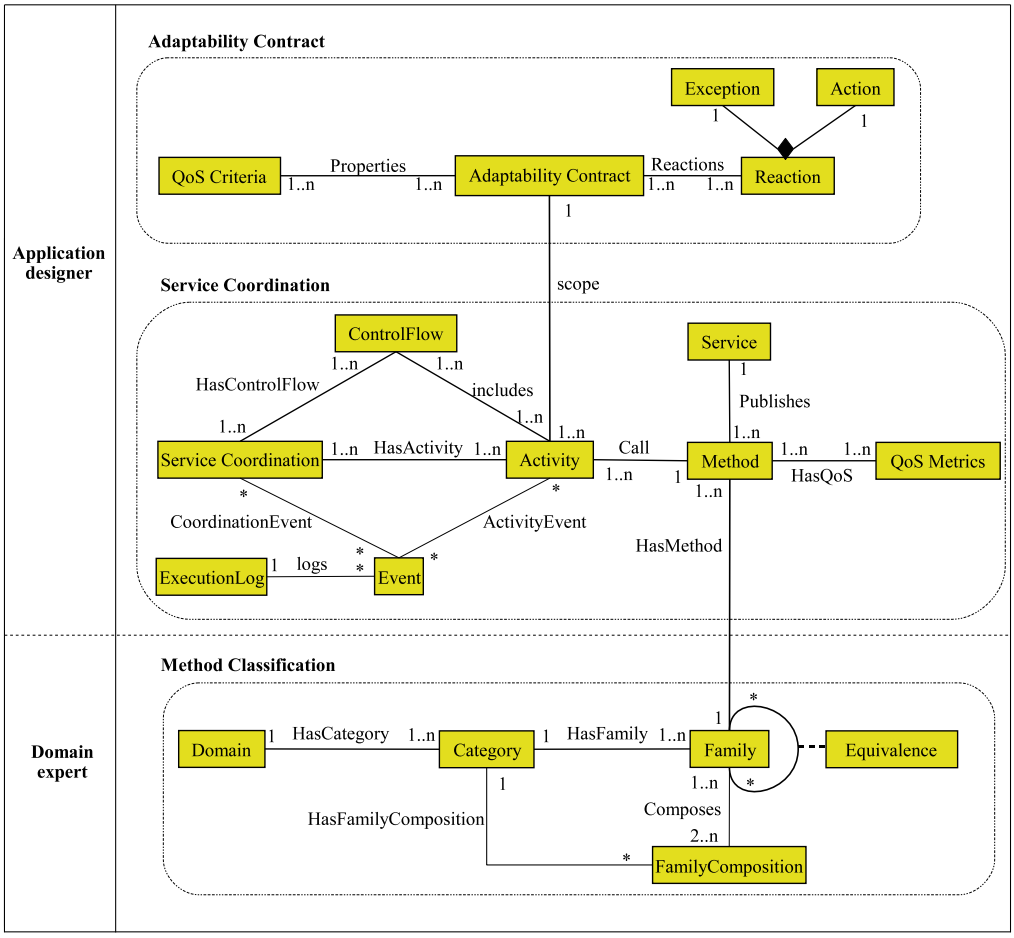


FIG. 1.2 – Modèle de coordination adaptative de services à base de contrats d'adaptabilité

La figure 1.2 illustre notre modèle de coordination adaptative de services à base de contrats d'adaptabilité.

### 1.2.1 Coordination de services

La définition d'une coordination de services est faite par les concepteurs d'applications. Une coordination se compose d'activités ordonnées. Chaque activité effectue un appel à une méthode de service spécifiée. Dans notre modèle nous avons proposé trois opérations d'adaptation : la réexécution, la substitution de méthodes de service et la substitution d'activités qui permettent de réagir aux exceptions causées par les fautes permanentes. Ces opérations se base sur :

– la classication de méthodes de service afin de trouver des méthodes équivalentes et

– la QoS de méthode de service pour choisir la plus appropriée aux critères de QoS exigés.

L'exécution d'une coordination de services est représentée par des événements de coordination et d'activité qui sont stockés dans un journal d'exécution. Ces événements sont utilisés pour :

– calculer les mesures de QoS de méthodes,

– contrôler et observer l'exécution d'une coordination de services, et

– localiser l'activité remplacée qui produit une exception lors de son exécution.

### 1.2.2 Classification de méthodes de service en famille

La classification de méthodes de service est définie par les experts de domaines d'application. Cette classification fournit la taxonomie des méthodes par une hiérarchie de domaines, de catégories et de familles qui représentent des besoins de l'utilisateur. Elle fournit aussi les équivalences entre les familles de méthodes et les mappings entre les signatures des méthodes de service équivalentes. Un besoin peut aussi être défini par une composition de familles de méthodes abstraite qui sera concrétisée par une composition de méthodes de service au moment de l'exécution en remplaçant chaque famille dans la composition abstraite par la méthode la plus appropriée aux critères de qualité de service de sa famille.

Cette classification est représentée par une base de connaissances grâce à laquelle les méthodes équivalentes peuvent être cherchées et remplacées l'une par l'autre.

### 1.2.3 Notion de contrat d'adaptabilité

Un contrat d'adaptabilité permet aux concepteurs d'application de définir explicitement les comportements d'adaptations réagissant aux exceptions produite lors de l'exécution de l'activité d'une coordination de services.

Un contrat d'adaptabilité qui est associé à une ou plusieurs activités d'une coordination spécifie les critères de QoS de méthodes de service et les réactions qui spécifient les actions d'adaptation réagissant aux événements/exceptions spécifiés.

La figure 1.3 illustre la coordination de services de recherche de vols associée aux contrats d'adaptabilité. Chaque activité de la coordination est associée à un contrat d'adaptabilité. Par exemple, le contrat "C1" associé à l'activité "Get Travel Information" spécifie explicitement les réactions réagissant aux exceptions spécifiées qui se produisent lors de l'exécution de cette activité.

Un contrat d'adaptabilité fournit la flexibilité de l'adaptation car il peut être modifié au moment de l'exécution de la coordination pour s'adapter à un changement survenu.

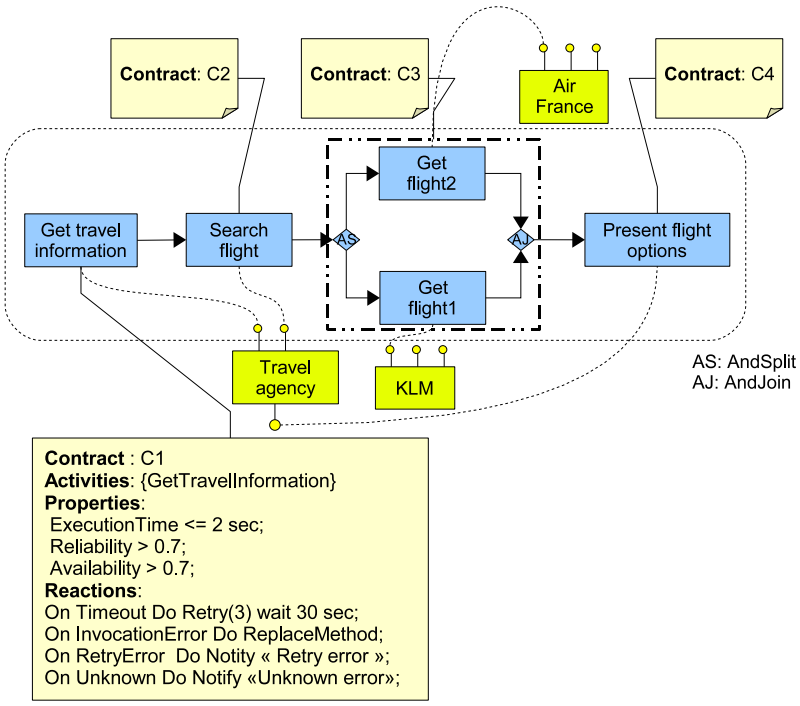


FIG. 1.3 – Coordination de services de recherche de vols associée aux contrats d'adaptabilité

Nous avons développé un moteur d'évaluation de contrats appelé SEBAS qui met en œuvre notre approche. SEBAS est un système à composants dans lequel l'exécuteur de contrats est le composant principal qui est responsable de l'évaluation des contrats d'adaptabilité.

## 1.3 Organisation du document

Le chapitre 2 présente l'état de l'art de la composition adaptative de services. Nous introduisons d'abord une taxonomie de compositions de services qui classifie les deux types d'approches de composition de services : statique et dynamique. Cela nous donne une vue globale sur les approches existantes dans le domaine de composition de services. Nous présentons ensuite les approches d'adaptation de l'exécution de la coordination de services.

Le chapitre 3 décrit notre modèle de coordination adaptative de services. Il définit les concepts de base pour une coordination de services et présente la classification de méthodes de service en familles. La notion d'équivalence de familles est introduite afin d'enrichir la possibilité de trouver les méthodes équivalentes. L'exécution d'une coordination de services est représentée par des types d'événements. Les événements d'exécution de coordination qui sont stockés dans un journal d'exécution sont les éléments utilisés pour le calcul des mesures de QoS d'une méthode de service.

Le chapitre 4 présente les concepts de contrat d'adaptabilité permettant de spécifier les actions d'adaptation réagissant à un type donné d'exceptions (événements) et définissant les critères de QoS pour les méthodes de service.

Le chapitre 5 présente le processus d'évaluation de contrat d'adaptabilité. Ce chapitre introduit d'abord l'architecture générale de l'évaluateur de contrats d'adaptabilité appelé SEBAS. Ensuite, il présente le processus d'évaluation d'un contrat d'adaptabilité. L'évaluation d'un contrat consiste à exécuter une réaction définie dans le contrat. Des calculs des mesures de QoS de méthodes de service sont également définis.

Nos conclusions et perspectives pour la suite de notre travail sont présentées dans la chapitre 6.

# Chapitre 2

# Etat de l'art : Composition adaptative de services

Une composition de services appelée service composite est une combinaison de fonctionnalités de services existants [ACKM04b]. Le processus de développer d'un service composite s'appelle une composition de services. La composition de services concerne donc les techniques de composition arbitraire des services existants pour avoir un service composite qui implante une logique applicative spécifique.

Généralement, une composition de services est réalisée selon deux phases suivantes :

Phase de conception : Dans cette phase, une composition est définie de manière statique ou dynamique. La définition d'une composition de services est décrite par un processus, par exemple un workflow. Un processus se compose d'activités et de l'ordre d'exécution de ses activités.

Phase d'exécution : Dans cette phase, une composition est instanciée. Le processus décrivant cette instance est transformé en un plan d'exécution. L'exécution de cette instance est réalisé selon son plan d'exécution par un moteur de coordination.

Cependant, il faut garantir que son exécution se déroule correctement et qu'elle se termine avec succès. La participation de services autonomes à une composition rend cet objectif difficile à atteindre car des exceptions peuvent se produire lors d'appel à leurs méthodes et lors de la communication de messages entre un service et une des activités de la composition. L'exécution d'une composition doit traiter et tolérer ces exceptions afin d'assurer une bonne qualité de service. Cela nécessite d'avoir une stratégie de récupération pour s'adapter aux exceptions produites au moment de l'exécution.

Le but de l'adaptation d'une composition de services est de garantir sa continuation et sa terminaison avec succès quand une exception se produit. L'adaptation d'une composition de services est réalisée au moment de l'exécution. L'adaptation peut être réalisée manuellement par l'intervention d'un utilisateur ou dynamiquement par des opérations d'adaptation prédéfinies. Dans cet état de l'art, nous nous considérons seulement les

travaux sur l'adaptation dynamique pour une composition de services. Pour définir explicitement les comportements d'adaptation nous remarquons que la plupart des travaux adoptent l'approche par des règles.

Dans cet état de l'art, nous étudions les travaux sur l'adaptation dynamique pour une composition de services.

La section 2.1 qui propose une taxonomie de compositions de services. Cela nous donne une vue globale des approches existantes dans le domaine de composition de services. Ensuite la section 2.2 décrit les approches d'adaptation de l'exécution d'une coordination de services. Finalement, la section 2.3 résume et conclut ce chapitre.

## 2.1 Taxonomie de composition de services

Selon [tBBG06, BG06], la taxonomie de composition de services se base sur :

- la manière de définir une composition de services : statique (manuel) ou dynamique (automatique) et
- l'ordre de l'exécution d'une composition de services : stricte ou dynamique.

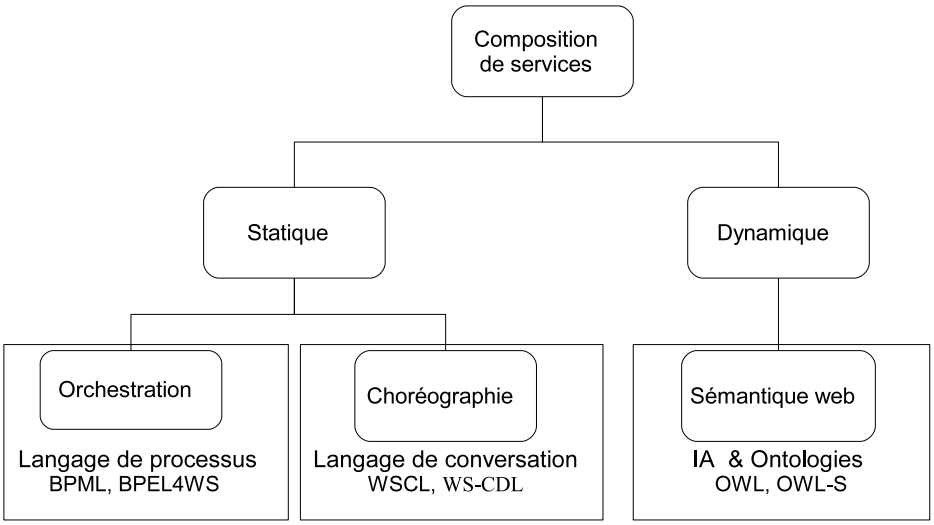


FIG. 2.1 – Taxonomie de composition de services

La figure 2.1 illustre cette taxonomie de composition de services . Un processus d'une composition de services peut être défini statiquement ou dynamiquement. Dans la composition statique, un processus d'une composition de services est défini à la phase de conception tandis que dans la composition dynamique, il est généré dynamiquement à la phase d'exécution.

Dans le cas statique, une composition de services est définie manuellement lors de la phase de conception par un concepteur d'application en utilisant un langage de proces-

sus comme BPEL4WS [ACD+03], BPEL [Ark02] ou un langage de conversation comme WSCI, WSCL [KBR+05, BBB+02]. Dans le milieu académique, les langages de processus comme $\pi$-Calculus [MPW92], Petri Net [GHB+06, Hoh06] sont utilisés pour définir et vérifier si un processus est correct par rapport à la logique applicative désirée [DvdAtH05].

Une composition dynamique de services est définie automatiquement. Son processus est créé dynamiquement en se basant sur les descriptions des fonctionnalités des services existants. Les outils falicitant ce type de composition sont les langages de planification de domaines comme PDDL ou de description de fonctionnalités comme RDF, OWL [ABH+01, BvHH+04], OWL-S [BHL+04], et les techniques de planification intelligences artificielle comme les machines à états [RHNT08], PDDL (Planning Domain Definition Language)[GNA+98], HTN (Hierarchical Task Network) [WSH+03], ou la démonstration de théorèmes [Wal01, MT01].

### 2.1.1 Composition statique de services

La composition statique de services est classifiée en se basant sur les interactions entre les services. Deux approches connues pour la composition statique de services sont les suivantes :

L'"*orchestration* de services (Web)", combine des services existants en ajoutant un coordinateur central (un orchestrateur) qui est responsable d'appeler et de faire interagir des services. L'exécution d'une orchestration de services est réalisée et contrôlée par le coordinateur selon son plan d'exécution.

La figure 2.2 illustre une orchestration de services décrivant deux flots de contrôle : séquence et parallèle. L'exécution des activités dans l'orchestration est ordonnée. Les interactions entre les services et le coordinateur sont réalisées en appelant les méthodes exportées par ces services.

La "*chorégraphie* de services (Web)" définit des tâches complexes par l'intermédiaire de la définition de conversations qui devraient être effectuées par chaque service participant. Une conversation décrit l'échange de messages ordonnée entre deux services selon un protocole. La définition d'une chorégraphie de services est décrite par un schéma des interactions pair à pair entre les services.

La figure 2.3 illustre une chorégraphie entre quatre services. Chaque service interagit avec les autres services en échangeant les messages selon un ordre prédéfini.

**BPEL4WS (Business Process Execution Language for Web Services) [ACD+03]** définit un modèle et une grammaire qui décrivent les interactions entre le processus et leurs partenaires en utilisant les interfaces des services web. Elle définit aussi les états, la logique de composition et la manière de traiter les conditions exceptionnelles. Un document BPEL4WS utilise XML pour décrire des aspects d'un processus métier comme des parte-

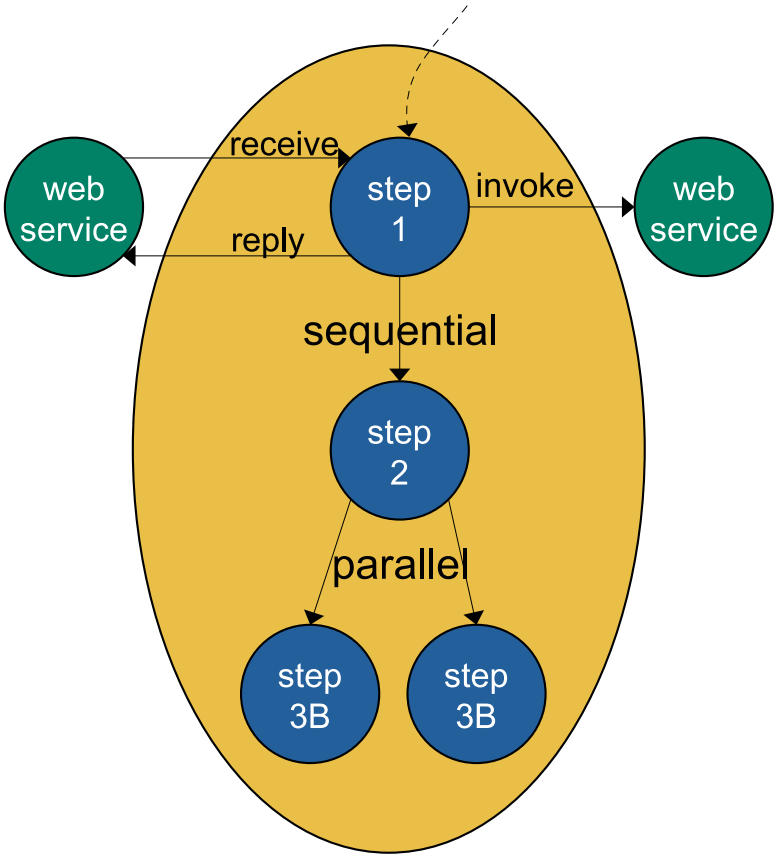


FIG. 2.2 – Orchestration de services

naires, des variables, des actions atomiques, des structures de flot de contrôle. Le résultat de l'utilisation de BPEL4WS pour modéliser un processus métier exécutable est un nouveau service Web composé par des services existants.

**BPML (Business Process Modeling Language) [Ark02]** s'appuie sur un modèle abstrait et une grammaire pour décrire des processus métier abstraits et exécutables. Dans BPML, un processus est une composition d'activités qui exécutent des fonctions spécifiques. Il peut être défini comme un type d'activité complexe. Une composition en BPEL interagit avec un ensemble deservices Web appelés *partners* pour réaliser une tâche donnée. Chacun des services est décrit par une interface WSDL.

**Web Service Choreography Interface (WSCI)[KBR+05]** est un langage de chorégraphie qui décrit les messages échangés entre les services web participant à un processus métier collaboratif. Il définit le flot de messages échangés par un service web décrivant son comportement observable. En spécifiant les dépendances temporelles et logiques parmi les échanges de messages, on peut décrire un service de telle façon que d'autres services Web peuvent l'appeler.

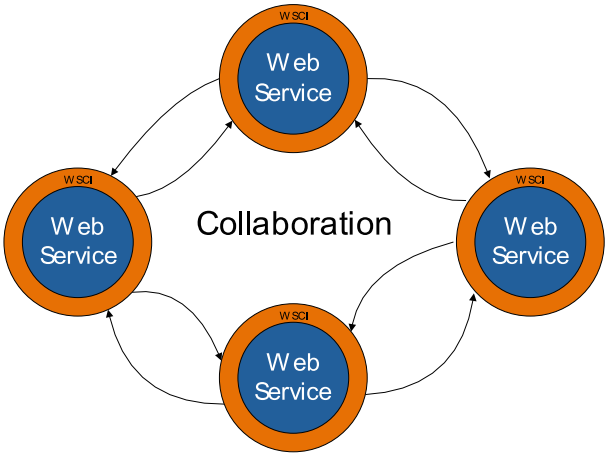


FIG. 2.3 – Chorégaphie de services

Tandis que plusieurs propositions existent pour les langages d'orchestration (par exemple BPML [Ark02] et BPEL4WS [ACD+03], des langages de chorégraphie sont dans une étape préliminaire de définition. Une proposition WS-CDL [KBR+05] a été publiée par le World Wide Web Consortium (W3C) en novembre 2005.

#### 2.1.1.1 Discussion

Les langages de processus ou de conversation permettent de définir correctement une composition de services. Bien que la composition statique de services soit bien formée et vérifiée, elle ne correspond pas à un environnement dynamique dans lequel les services peuvent être indisponibles ou modifiés. Les systèmes de composition statique de services basés sur BPEL ou WSCI manquent d'outils pour gérer l'évolution et l'adaptation.

L'adaptation d'une composition de services se réalise suite à des exceptions produites par un appel échoué, par une attente d'une réponse trop longue ou une réponse non désirée. Le problème de l'évolution concerne le changement d'un service dans une composition de services quand le fournisseur change ce service par une nouvelle version. Dans ce cas les incompatibilités entre les signatures des méthodes des deux services causeront un conflit lorsqu'on appelle une méthode de la nouvelle version du service en utilisant la signature de la méthode de la version précédente.

Bien que les approches de composition statiques comme BPEL4WS, WSCI, fournissent des mécanismes de traitement d'exceptions qui laissent aux concepteurs et aux programmeurs le choix et la programmation des procédures de traitement d'exceptions.

### 2.1.2 Composition dynamique de services

Une composition dynamique de services a pour but de produire un plan d'exécution en se basant sur des services existants. Le défi de la composition dynamique est de composer

automatiquement des services pour atteindre un but. En particulier, quand une fonctionnalité requise ne peut pas être réalisée par un service appelé, des services existants peuvent être composés pour accomplir la demande. La composition dynamique de services exige la localisation des services basée sur leurs fonctionnalités et l'identification de compatibilité opérationnelle de services pour créer une composition correcte.

L'hypothèse sur laquelle se base ce genre de techniques est que chaque service est considéré comme une action et décrit par sa signature de méthode, ses pré-conditions, ses post-conditions et ses effets. Cette hypothèse se base sur les raisons suivantes :

- un service est un logiciel autonome qui prend des données d'entrée et produit des données de sortie. Ainsi les pré-conditions et les post-conditions sont également évaluées pour déclencher son exécution et valider sa terminaison.

- un service change également l'état de la composition après son exécution. Ainsi l'état demandé pour l'exécution d'un service est la pré-condition, et les nouveaux états produits après l'exécution sont les effets de son exécution.

#### 2.1.2.1 Composition dynamique d'un protocole d'interactions

**Ramy Ragab Hassen et al [RHNT08]** utilisent une machine à états finis pour représenter les échanges chorégraphiques de messages, c'est-à-dire le protocole d'interactions de services web. Les états représentent les différentes phases par lesquelles un service peut passer durant son interaction avec un demandeur (service). Les transitions sont déclenchées par des messages envoyés par un demandeur au fournisseur (service). Chaque transition est étiquetée par un nom de message.

Les auteurs supposent qu'il existe un annuaire de services disponibles, par exemple $S_1$ et $S_2$, qui sont décrits par leurs protocoles $P_1$ et $P_2$ dans la figure 2.4 (a) et (b). Le développement d'un nouveau service $S_T$ décrit par le protocole désiré $P_T$ est montré par la figure 2.4 (c). Une question intéressante est que est-il possible d'implanter un service $S_T$ demandé en composant les services disponibles ($S_1$,$S_2$). $P_T$ est le protocole désiré tandis que $P_1$ et $P_2$ sont appelés les protocoles composants.

Le problème de synthèse d'une composition de services décrite par un protocole de conversations est considéré comme un problème de génération d'un délégué du service demandé en utilisant les services disponibles. Un délégué est un machine à états finis FSM (Finite State Machine). Par exemple, la figure 2.4(d) illustre un délégué qui permet de composer le protocole $P_T$ en utilisant les protocoles existants $P_1$ et $P_2$. Ce délégué spécifie que l'activité *selectVehicle* du protocole désiré est déléguée au protocole $P_1$ tandis que l'activité *estimatePayment* est déléguée au protocole $P_2$.

Le problème de synthèse d'une composition est considéré comme un problème de recherche d'un délégué "correct" pour un protocole demandé en utilisant un ensemble de protocoles disponibles. Une étape cruciale concernant à ce problème est de déterminer le nombre de protocoles (services) disponibles pouvant être utilisé dans la composition.

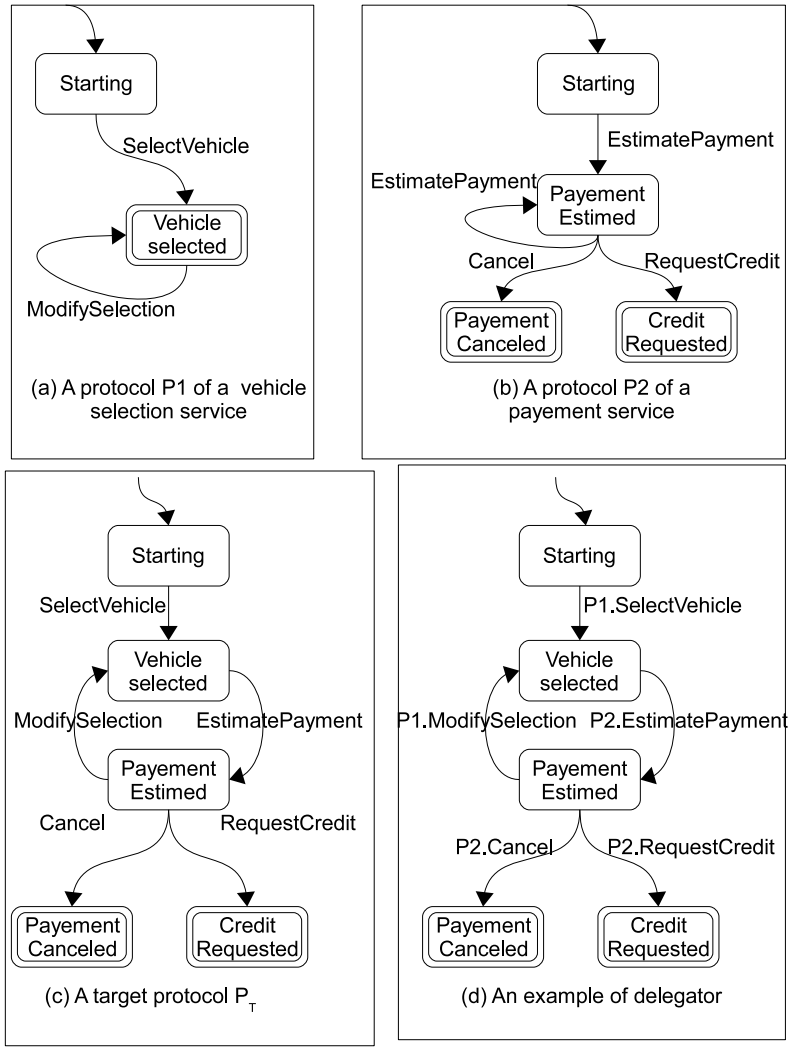


FIG. 2.4 – Protocole d'interactions de services

Les auteurs ont étendu le problème de synthèse d'un protocole désiré utilisant K instances de protocoles (The Bounded Case) à la synthèse d'un protocole avec K instances non spécifiées (The Unbounded Case). La complexité (le temps d'exécution) du problème "The Bounded Case" qui est moins compliqué que le problème "The Unbounded Case" appartient à la classe ExpTime [MW08].

#### 2.1.2.2 Composition dynamique de services à base de techniques d'intelligence artificielle

Une composition dynamique de services en se basant sur les techniques d'intelligence artificielle consiste à planifier (semi)automatiquement un plan d'exécution de services. En principe, la planification se base sur :

– une description de l'état initial qui représente l'état du monde par lequel le planifi-

cateur[1] commence ;

– une description du but que le planificateur doit atteindre et

– un ensemble d'actions/opérations possibles qui est utilisé par le planificateur pour atteindre le but à partir de l'état initial. Chaque action spécifie généralement des pré-conditions qui doivent être évaluées pour que cette action puisse être exécutée, et des post-conditions (effets) qui changent l'état actuel.

Pour la description d'un problème de planification, des langages de planification basés sur STRIPS[2] ont été proposés comme ADL (Action Description Language) et PDDL (Planning Domain Definition Language) [GNA+98]. Ensuite pour décrire la sémantique des actions d'un problème de planification non seulement pour des utilisateurs mais aussi pour les machines, le langage d'ontologie OWL-S [BHL+04] a été utilisé.

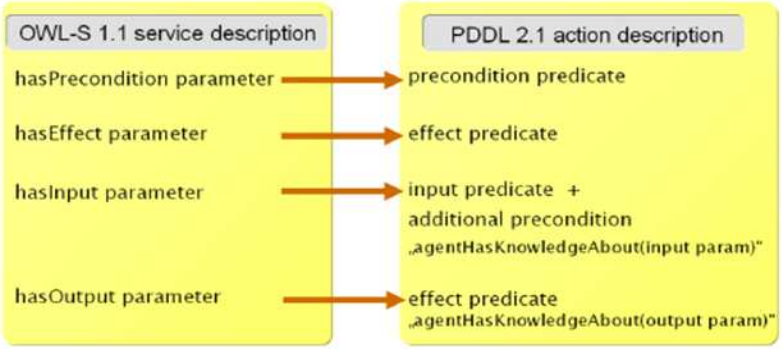


FIG. 2.5 – Mapping entre les descriptions de services en OWL-S et d'actions en PDDL

La similarité entre les représentations de OWL-S et de PDDL permet d'intégrer la sémantique de services dans la planification basée sur PDDL. La conversion de descriptions sémantiques de services en OWL-S en descriptions d'actions en PDDL est illustrée par la figure 2.5. PDDL est largement utilisé comme une entrée normalisée pour les planificateurs.

La suite de cette section présente quelques travaux de composition dynamique de services à base de techniques d'intelligence artificielle.

**Composition dynamique de services en utilisant le réseau de tâches HTN :** Un réseau hiérarchique de tâches se compose des tâches primitives et non primitives. Une tâche primitive est exécutable tandis qu'une tâche non primitive ne peut pas être exécutée directement parce qu'elle est composée par des tâches primitives et non primitives.

[1]Les logiciels de planification qui incorporent les algorithmes de planification se nomment planificateurs.

[2]STRIPS ( STanford Research Institute Problem Solver) est un algorithme de Planification classique conçu par Richard Fikes et Nils Nilsson en 1971

La planification de HTN consiste à réaliser partiellement ou entièrement une liste de tâches ordonnées (réseau de tâches). Un planificateur de HTN prend un problème qui est défini par un réseau de tâches comme une entrée et décompose les tâches non primitives en tâches primitives pour les exécuter.

[WSH+03] a utilisé le planificateur SHOP2 pour composer dynamiquement des services Web dont les signatures, les pré-conditions et les effets sont décrits en OWL-S. On peut constater que la décomposition de tâches dans la planification de HTN est très semblable à la décomposition de processus composites dans l'ontologies de processus OWL-S. Cependant peu de détails sont donnés sur le processus de traduction de OWL-S à SHOP2.

**Composition dynamique de services en utilisant PDDL :** Le langage de PDDL permet de décrire un domaine d'application et un problème à résoudre.

Un domaine d'application se compose des actions paramétrées qui caractérisent des comportements d'un objet spécifique et de prédicats décrits par des propositions logiques. Une action est caractérisée par un nom, une pré-condition et un effet. Une pré-condition et un effet sont décrits par des prédicats.

Un problème est caractérisé par des objets, un état initial et un but. Un état initial est décrit par une liste de prédicats qui ont la valeur vraie et un but est aussi décrit par une proposition logique formulée par des prédicats comme une pré-condition.

Un problème de planification est donc défini par le couplage d'une description de domaine et d'une description de problème. La planification est réalisée par un planificateur qui commence par l'état initial et applique des actions convenables pour atteindre le but.

**Matthias Klusch et al** [KG05] ont présenté un système de composition de services OWL-S, appelé OWLS-Xplan qui permet de composer des services. OWLS-Xplan convertit des descriptions de service en OWL-S en des descriptions de problème et de domaine équivalentes qui sont décrites par le langage de description de domaine de planification PDDL[3] (voir la figure 2.5). OWLS-Xplan utilise un planificateur efficace Xplan à base de techniques d'intelligence artificielle pour produire un plan de composition de services qui satisfait un but donné.

Xplan est un planificateur à base de graphes avec une fonctionnalité additionnelle qui permet d'exécuter la décomposition d'activité (méthode) comme un planificateur de HTN. La figure 2.6 montre un exemple de l'utilisation de Xplan pour décomposer une méthode donnée *M* en *WS1*. Ensuite, XPlan compose *WS1* et *WS3* pour atteindre le but donné.

[3]PDDL est un langage d'actions basé sur STRIPS (*STanford Research Institute Problem Solver*. PDDL décrit un domaine de planification et des problèmes associés au domaine.

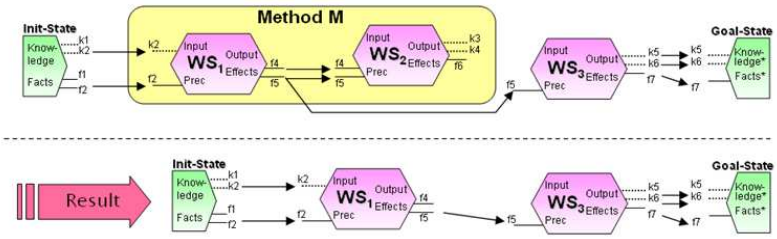


FIG. 2.6 – Décomposition de la méthode *M*

#### 2.1.2.3 Discussion

Le principal avantage de la composition dynamique est de composer automatiquement ou semi automatiquement un plan d'exécution en réduisant au maximum l'intervention humaine.

En général, la complexité de la composition dynamique de services vient des raisons suivantes [RS05] :

- Le nombre de services disponibles sur le Web a rapidement augmenté dans les années récentes et on aura un énorme dépôt de services web pour la recherche.

- Les services peuvent être créés et modifiés "en ligne", le système de composition nécessite de détecter ces modifications et les s'adapter.

- Les services sont développés par différentes organisations qui décrivent différemment les services. Pour l'instant, il n'existe pas un langage unique pour définir et évaluer les services dans une sémantique identique.

L'automatisation complète de la composition dynamique est toujours un sujet ouvert. Son problème principal est la différence entre les concepts que les utilisateurs utilisent et les données que les ordinateurs interprètent. Cet obstacle peut être surmonté en employant des technologies de sémantique Web [MBE03] comme OWL [ABH+01, BvHH+04], OWL-S [BHL+04].

Nous résumons les inconvénients suivants de la composition dynamique de services :

- Le fait que construire une base de connaissances des fonctionnalités de service est coûteux.

- Les planificateurs comme XPlan, Shop2 se basent sur l'hypothèse d'un ensemble d'actions fermé.
- Une latence longue surtout pour la génération d'un plan d'exécution approprié peut ne pas être acceptable pour les utilisateurs.

- La génération d'un nouveau plan peut ne pas totalement être conforme aux besoins exprimés. Les structures d'un plan complexe comme celui décrivant un *choix* et une *séquence non ordonnée* ne sont pas réalisables par une synthèse de plans créés par Xplan ou Shop2.

- Gestion des instances d'une composition de services
  Le changement dynamique d'une composition, c'est à dire changer son plan d'exécution, influence les instances déjà créées au moment de l'exécution car l'ancien plan de ces instances peut être incohérent avec le nouveau plan. Par ailleurs dans certains cas, ces instances peuvent ne pas respecter les nouvelles règles exigées par la composition modifiée. Trois solutions ont été proposées par [vdABV+99] pour résoudre cette incohérence :

  - Annulation de toutes les instances en cours de la composition modifiée : cette solution n'est pas acceptée par certaines applications, par exemple l'activité de payement dans un processus métier déjà réalisé.

  - Continuité de l'exécution des instances en cours sans considérer la modification de la composition. Cette solution ne peut seulement appliquer qu'aux instances où la continuité du reste de ses plans d'exécution ne souffre pas de ces changements.

  - Modification du plan d'exécution des instances en cours d'exécution : cette solution est impossible pour la plupart des moteurs de composition de services. Quelques systèmes de workflow par exemple eFlow [CIJ+00], Bonita [ET07], permettent de modifier manuellement le plan d'exécution au moment d'exécution des instances par l'intervention des experts.

- Traitement d'exceptions : la composition des services emploie des services externes qui sont fournis par des propriétaires de service. Le système de gestion de compositions de services doit traiter d'exceptions pendant l'exécution d'appels au cas où les services externes ne répondraient pas. Par ailleurs, les processus métiers sont habituellement des longs processus qui peuvent prendre des heures ou des semaines pour se réaliser, et donc la capacité de contrôler des compensations des appels de service est critique pour une composition de services.

## 2.2 Adaptation de l'exécution d'une composition de services

Une adaptation de l'exécution d'une composition de services consiste à réagir à l'exception produite lors de son exécution pour garantir sa continuation avec succès.

Bien que la composition dynamique de services puisse être utilisée pour s'adapter

à l'exception ou au changement de l'environnement, elle est difficilement d'applicable dans un système en cours d'exécution car son temps d'exécution est long. La complexité du temps d'exécution est normalement exponentielle.

Trois approches distinctes d'adaptation dynamique sont les suivantes :

– Adaptation par substitution de services : Quand l'exécution de la composition appelle un service défectueux, une exception se produit et la substitution de services sera réalisée pour remplacer ce service par un autre service ayant la même fonctionnalité.

  Pour réaliser ce type d'adaptation, il est nécessaire d'avoir une classification de services selon leurs fonctionnalités afin de trouver un service ayant la même fonctionnalité que le service défectueux et de le remplacer. Cependant deux services ayant la même fonctionnalité ne sont pas pour autant sûrs de pouvoir être remplacés correctement éventuellement à cause d'incompatibilités de leurs signatures. Il faut donc résoudre ces incompatibilités pour que la substitution soit correcte.

– Adaptation par changement du plan d'exécution : Dans cette approche, une composition de services est représentée par un ou plusieurs de processus qui se composent d'activités ordonnées. Ces plans seront transformés en plan d'exécution. Un plan d'exécution d'une composition est normalement présenté en graphe. L'exécution d'une composition se déroule selon un chemin principal de ce plan d'exécution, c'est à dire un processus. Normalement, plusieurs chemins peuvent être identifiés. Le choix du chemin principal se base sur les critères de QoS. Quand l'exécution d'une activité dans ce chemin échoue, un autre chemin sera localisé et commuté pour éviter l'activité défectueuse.

  Comme dans l'approche précédente, cette approche nécessite également de prédéfinir et stocker les plans de processus qui réalisent un processus métier dans un dépôt de plans de processus.

– Adaptation par transition vers une activité de traitement d'exceptions : Cette approche consiste à rediriger une activité échouée vers une activité de traitement d'exceptions qui traite les exceptions produites lors de l'exécution de son activité associée par des actions spécifiées.

La suite de cette section présente ces trois approches d'adaptation pour une composition de services : la substitution de services, le changement de chemin d'exécution et la transition vers l'activité de traitement d'exception.

### 2.2.1 Substitution de services

La substitution de services consiste à remplacer un service défectueux par un autre ayant la même fonctionnalité. Cette solution permet de s'adapter à cette situation sans replanifier le plan d'exécution de la composition de services. Ce type d'approches [SBDM02, BSD03, CDK+06, YL05, TMPE04, TPE+02, TBFM06, AVMM04] s'appuie sur les raisons suivantes :

- il existe de multiples services réalisant une même fonctionnalité dans le monde de services web. On peut donc trouver un service pour remplacer un autre.

- la substitution de services est évidemment moins coûteuse que la replanification dynamique de services.

Pour réaliser une substitution de service, il fait avoir une étape de conception pour préparer cette opération. L'étape de conception consiste à :

- construire une classe de services ayant la même fonctionnalité afin de pouvoir trouver un service remplaçant et

- résoudre les conflits des signatures d'opération de service, par exemple les conflits de types de données.

Pour l'instant, cette étape est réalisée manuellement par les experts du domaine d'application.

#### 2.2.1.1 Représentation d'une classe de services

La représentation d'une classe de services fournissant une même fonctionnalité illustrée par la figure 2.7 a été proposée par les travaux [SBDM02, BSD03, CDK+06, YL05, TMPE04, TPE+02, TBFM06, AVMM04]. La construction de classes de services est faite à la phase de conception par les experts de domaine d'application. On peut alors distinguer deux sortes de services : services abstraits et services concrets

**Service abstrait** est une classe de services concrets réalisant une même fonctionnalité. Grâce aux services abstraits, on peut trouver et remplacer un service concret par un autre équivalent à la fonctionnalité désirée. Cependant les services d'une classe peuvent avoir différentes signatures de méthodes. Cela engendre des conflits d'incompatibilités entre leurs signatures. A partir des services abstraits, on peut construire des compositions de services abstraits. Une composition de services abstraits est considérée comme un template de composition de services. Ce template sera concrétisé par une composition de services concrets au moment de l'exécution en remplaçant un service abstrait par un de

ses services concrets.

**Service concret :** Des services concrets sont des applications accessibles sur l'Internet. Ces services sont fournis publiquement par des fournisseurs au travers des annuaires de services comme des UDDIs de services web. Les services concrets peuvent être composés pour répondre à un besoin. On parle donc d'un service composite ou une composition de services.

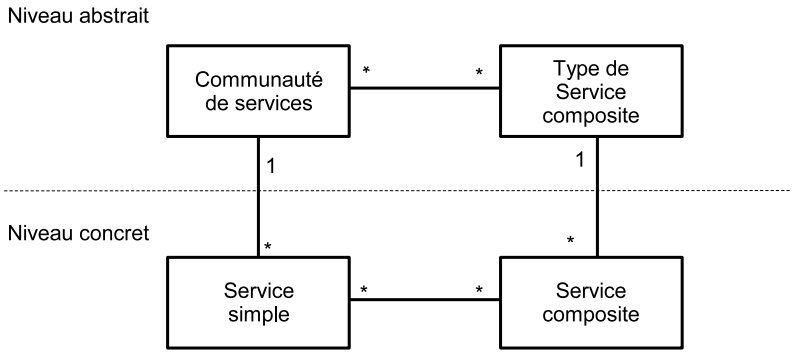


FIG. 2.7 – Représentation de composition de services à base de fonctionnalité

La suite présente quelques exemples de travaux existants [SBDM02, BSD03, CDK+06, YL05, TMPE04, TPE+02, TBFM06, AVMM04] qui adoptent cette représentation de service.

***SELF-SERV*** [SBDM02, BSD03] propose trois types de services : services élémentaires, services composites, et communautés de services.

- Un service élémentaire est un service Web accessible qui ne se fonde pas explicitement sur un autre service Web.

- Des services composites sont composés statiquement et sont exécutés dans un environnement pair à pair. Un service composite intègre de multiples services Web qui forment ses services composants. Un service composite est défini par un diagramme d'état. L'information liée à chaque état ou transition est l'identificateur d'état, le nom d'état, les paramètres d'entrée-sortie du service Web lié à l'état, la règle ECA décrivant la transition.

- Les communautés de services sont essentiellement des entrepôts de services alternatifs qui ont la même fonctionnalité. Elles fournissent des descriptions des services désirés (par exemple, fournir les interfaces de réservation de vol) sans adresser un fournisseur particulier (par exemple, le service Web de réservation de vol UA). Au moment de l'exécution, quand une communauté reçoit une demande d'exécution d'une opération, elle la délègue à un de ses membres courants. Le choix du membre à qui est délégué l'opération est basé sur les paramètres de la demande, les caractéristiques des membres, l'histoire des exécutions passées et l'état d'exécution.

SELF-SERV exploite le concept de communauté de services afin de répondre à la composition dynamique d'un grand nombre potentiel de services Web.

***A-WSCE*** [CDK+06] différencie les types de services Web qui sont des groupements de services Web semblables en termes de fonctionnalité, et les instances réelles de services Web qui peuvent être appelées. A-WSCE propose deux types de services : services primitifs et services composites. Un service composite est composé par des services primitifs. Ces deux types de services servent à créer des patrons de workflow selon leur fonctionnalité. Disposant d'une description d'une fonctionnalité d'application et les types de services, A-WSCE peut générer plusieurs patrons de workflow. Ayant un patron de workflow, A-WSCE crée alors une instance de workflow exécutable optimisée en choisissant les instances de services les plus appropriées par rapport aux propriétés non fonctionnelles comme des mesures de QoS (coût, temps de réponse, disponibilité de services)

La représentation séparant les types de services et leurs instances aide à les manipuler différemment, et permet d'avoir efficacement une grande collection de services. Les types et les instances peuvent être publiés dans un annuaire comme UDDI.

***QCWS*** [YL05] spécifie une classe de services qui est une collection de différents services Web ayant une fonctionnalité commune mais différentes propriétés non fonctionnelles. Une composition de services est décrite par un graphe dirigé acyclique DAG dans lequel les noeuds sont les services web individuels. Un processus métier peut avoir ainsi plusieurs plans de processus ou graphes. Un plan d'exécution d'un processus métier est une combinaison des plans de processus choisis par l'utilisateur. Au moment de l'exécution, un plan de processus dans le plan d'exécution sera choisi par l'algorithme de sélection qui identifie le meilleur chemin par rapport aux mesures de QoS.

**[TBFM06]** définit un service Web abstrait qui fournit une interface à travers laquelle une communauté de services est accessible. Cette interface décrit la fonctionnalité de la communauté par un ensemble d'opérations abstraites. Cette interface est employée par des réalisateurs pour développer l'accès de clients à une communauté. Le service Web est qualifié en tant qu'abstrait pour les trois raisons suivantes :

– représentation d'une communauté,

– faute d'implémentation de l'interface qu'il fournit, et

– aucune participation à une composition. La participation d'un service Web abstrait dans une composition se fait par des services Web concrets.

Les fournisseurs de services Web concrets joignent la communauté du service Web abstrait afin d'offrir une même interface pour ces services. En revanche, un service Web

concret doit implémenter sa propre interface en définissant ses opérations. On comprend qu'une communauté de services Web a un contenu dynamique : de nouveaux services Web peuvent intégrer la communauté, d'autres services Web peuvent en sortir, certains services peuvent devenir temporairement indisponibles, certains services peuvent reprendre une opération après suspension, etc.

***METEOR-S*** [AVMM04] définit les ontologies basées sur le langage OWL pour représenter des services. Il fournit un canevas représentatif pour intégrer la sémantique des données, la sémantique fonctionnelle et la sémantique de qualité de service des opérations de services web.

METEOR-S permet de définir une composition de services par un processus abstrait qui spécifie le flot d'opérations. METEOR-S a choisi BPEL4WS [ACD+03] comme langage de spécification de ces processus abstraits car c'est le standard industriel fournissant un ensemble riche de constructeurs pour la modélisation de patrons de workflows [vdATKB03]. Au moment de l'exécution, le moteur d'exécution lie un ensemble de services au processus abstrait et produit un processus exécutable.

***AgFlow*** [ZBN+04, ZBD+03, ZBN01] est un intergiciel qui permet de créer des compositions de services web dirigées par la qualité de service.

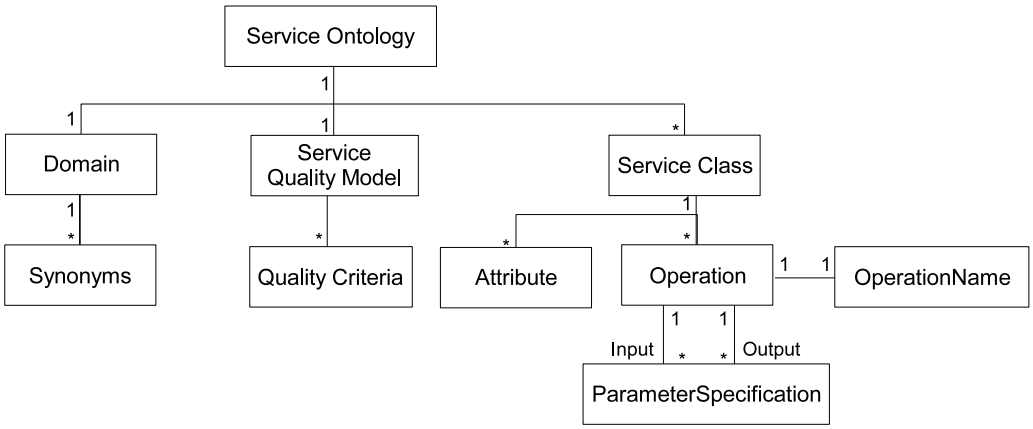


FIG. 2.8 – Ontologie de service de AgFlow

La représentation de services est exprimée par une ontologie de services (voir Figure 2.8). Une classe de services est encore spécifiée par ses attributs et opérations. Par exemple, les attributs d'une classe de services peuvent inclure l'information d'accès telle que l'URL.

Chaque opération est spécifiée par son nom et ses paramètres d'entrée et de sortie. L'ontologie de services spécifie également un modèle de qualité de service qui est employé pour décrire les propriétés non fonctionnelles des services, par exemple une durée d'exécution d'une opération. Le modèle de qualité de service se compose d'un ensemble de dimensions de qualité (ou de critères). Pour chaque critère de qualité, il y a trois éléments de base : la définition de QoS, les éléments de service (par exemple, services ou

opérations) auquels elle est liée, et à la façon de calculer ou de mesurer la valeur d'une mesure de QoS.

Un service composite est spécifié comme une collection de services génériques décrites en termes d'ontologies de service et combinées selon un ensemble de flot de contrôle et de dépendances des flot de données. AgFlow emploie les diagrammes d'état qui se compose des états et des transitions [HN96] pour représenter ces dépendances. Des transitions d'états sont étiquetées par des événements, des conditions et des opérations. Les états peuvent être basiques ou composés. Un état basique est étiqueté par un nom d'opération d'une classe de services qui est définie dans l'ontologie de service. Intuitivement, quand l'état basique est atteint, l'opération étiquetée sera appelée. Les états composés fournissent des moyens pour structurer le diagramme à l'aide de deux états spéciaux OR-State et And-State qui correspondent aux opérateurs OR-Split/join et And-Split/join de workflow.

#### 2.2.1.2 Résolution d'incompatibilités des signatures de méthodes/opérations

Pour résoudre les conflits causés par les incompatibilités entre les signatures de méthodes de service, l'idée générale est de construire des mappings (correspondances) entre elle (signatures). Ces mappings sont définis statiquement dans la phase de conception. Ils peuvent être définis en utilisant des règles de mapping [TBFM06] ou des règles d'association [SB06, Sam08] que nous examinerons dans la suite de cette section.

**Règles Mapping de services** [TBFM06] propose qu'une adaptation d'interfaces de service soit une activité qui se compose de deux phases : statiques (conception) et dynamique (exécution). La phase statique consiste à définir manuellement des fonctions d'adaptations entre l'interface d'un service abstrait et l'interface d'un service concret. En utilisant ces fonctions d'adaptation, la phase dynamique consiste à générer automatiquement des adaptateurs qui réalisent les adaptations désirées.

Les fonctions d'adaptation entre ces interfaces sont définies en utilisant un langage de mapping qui permet de décrire comment des paramètres actuels d'un appel peuvent être transformés au moment de l'exécution. Les règles de mappings sont utilisées pour décrire des correspondances entre les paramètres d'entrée/sortie d'opérations de ces services.

Un concepteur spécifie les règles de mapping en utilisant un template mapping entre des paramètres d'entrée/sortie d'une opération d'un service concret (C $op_i$) et d'une opération d'un service abstrait au travers de la fonction GAF (Generic Abstract Fonction) comme suit :

```
   Begin
For each Input parameter I paramL of C Opi
I paramL ← GAF (ID, Abstract Param List)
For each Output parameter O paramk of A Opj
O Paramk ← GAF (ID, Concrete Param List)
```

```
End
```

GAF (Generic Adaptation Function) est une fonction d'adaptation générale qui associe un paramètre de l'opération d'un service abstrait à un paramètre de l'opération d'un service concret.

**Règles d'association** [SB06, Sam08] définit le concept de service Web personnalisable comme tout service ayant la particularité de pouvoir être offert sous plusieurs variantes, ciblant la satisfaction de différentes catégories de clients de services.

Pour définir la personnalisation de service, ces auteurs ont proposé une structure utilisée à la fois par les demandeurs et par les fournisseurs de services. Cette structure est composée de deux sous-systèmes : le Système Structurel Typé, et le Système à Contraintes

Le système structurel typé représente la partie d'un service qui sert à sa découverte. Il est défini par le triplet (C, I, O) où

- C est le contexte de la spécification qui est défini par un mot-clé ayant trait au domaine du service spécifié.
- I est la description des variables d'entrée et de leurs types de données abstraits dans l'offre ou dans la demande de services.
- O est la description des variables de sortie et de leurs types de données abstraits dans l'offre ou la demande de services.

Les mots-clés des triplets (C, I, O) peuvent être annotés par des concepts formels définis dans une ontologie partagée entre les utilisateurs de services du domaine.

Les concepts sémantiques peuvent être utilisés pour la désignation des unités de mesure des entrées/sorties des servies Web. Par exemple, les types des entrées/sorties ne sont pas désignés par les types de données abstraits (Integer, Real, etc), mais par les unités de mesure utilisées pour exprimer les valeurs des entées/sorties dans l'ontologie du domaine.

Un système à contraintes permet de vérifier la cohérence des services découverts pendant l'appariement des systèmes structurels typés. En d'autres termes, la sélection des services qui peuvent certainement satisfaire la requête du client. En effet, la compatibilité des systèmes structurels typés entre les spécifications d'une offre et d'une demande de services ne suffit pas pour conclure qu'il y a similarité de services. Il faut aussi que leurs contraintes ne soient pas contradictoires. Le système à contraintes permet la représentation de deux sortes de contraintes : les contraintes sur les valeurs des entrées/sorties et les contraintes sur leurs types dans l'ontologie du domaine.

L'intérêt principal étant la personnalisation de services par transformation des offres de services en fonction des préférences des clients, les auteurs insistent sur les contraintes de typage qui permettent de préciser l'unité de mesure de la valeur d'une entrée/sortie dans la spécification d'une demande ou d'une offre de services.

Les contraintes sur les types des entrées/sorties des services peuvent tout simplement être considérées comme des spécialisations des concepts utilisés lors de l'annotation sémantique des systèmes structurels typés. Il s'agit de préciser par une seule unité de mesure chaque entrée/sortie annotée par un ensemble d'unités (concept extensionnel) dans le Système structurel typé.

Pour résoudre un conflit de types des entrées/sorties, une règle d'association de contexte est définie. Une règle d'association de contexte est un symbole de prédicat à deux arguments ConflictResolution(Concept,Context). "Concept" est une variable qui représente des concepts définis en extension dans une ontologie du domaine, et appartenant à la tête d'un axiome de couverture. "Context" est une variable destinée à recevoir le contexte du service de résolution de conflits.

Par exemple une règle relie le concept Money au mot clé (context) du service de résolution du conflit sur la monnaie : ConversionMoney.

### 2.2.2 Adaptation par changement du plan d'exécution

L'adaptation par changement du plan d'exécution consiste à trouver un autre chemin d'exécution qui contourne le service défectueux et atteint le but du processus. Après avoir trouvé ce chemin, l'exécution de l'instance de la composition se déroule sur le nouveau chemin identifié.

La suite de cette section présente les approches QCWS, AgFlow qui adoptent ce type d'adaptation.

Dans QCWS, les plans d'un processus sont créés par les utilisateurs. Il existe deux types de plans de processus :

- Les plans de processus qui sont définis en utilisant les classes de services et leurs relations d'ordonnancement. Les plans de processus créés sont stockés dans un dépôt de plans de processus.

- Le plan d'exécution est défini en sélectionnant un ou plusieurs plans de processus. Si un seul plan de processus est choisi, le plan d'exécution est ce même plan. Au cas contraire, si plusieurs plans sont choisis, le plan d'exécution est la combinaison de tous ces plans. Le chemin optimal dans ce plan d'exécution selon la QoS de processus désirée par l'utilisateur sera localisé dynamiquement par un algorithme proposé pour un processus métier demandé.

La figure 2.9 illustre un plan d'exécution composé par les deux plans de processus. Il est représenté en graphe direct.

Dans ce plan d'exécution, le chemin d'exécution optimal est le plan de processus 1 (de S1 à S7).

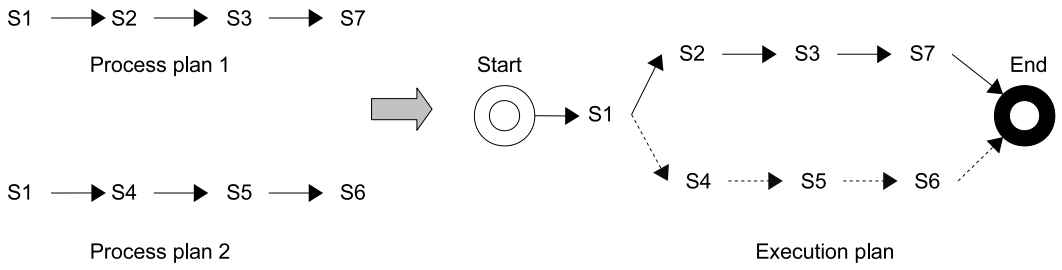


FIG. 2.9 – Plan de processus et plan d'exécution

Ces plans sont des plans abstraits car ils se base sur les classes de service, c'est-à-dire, aucun service concret n'est identifié dans ces plans. Les services concrets ne sont identifiés qu'au moment de localisation du chemin optimal dans le plan d'exécution.

Au moment de l'exécution d'un processus métier, si un service concret est défectueux, un mécanisme d'adaptation est nécessaire pour garantir que processus en cours ne soit pas interrompu et pour que le service défectueux puisse être replacé. Un chemin secondaire à partir du service en cours jusqu'à la fin du processus métier sera localisé. Un service dans le chemin peut donc être automatiquement commuté au chemin secondaire à chaque fois que son successeur devient indisponible.

### 2.2.3 Rediriger l'activité vers une activité de traitement d'exceptions

L'approche de transition d'activités par événement [SH03] permet au concepteur de processus de spécifier des situations exceptionnelles dans les activités d'un processus et une transition vers une activité de traitement d'exceptions quand une exception se produit au moment de l'exécution de cette activité.

Dans cette approche, le concepteur d'une composition de services sous forme d'un workflow définit des activités de traitement d'exceptions appelées traiteurs d'exceptions. Le concepteur devrait organiser un processus métiers en attachant les traiteurs d'exceptions à ce processus et à ses activités. Les traiteurs liés à un processus se situent à un niveau supérieur que les traiteurs liés aux activités. Ces traiteurs forment une hiérarchie de traiteurs d'exceptions. Chaque traiteur est responsable dans une sphère d'un processus, appelé un bloc. Chaque bloc a donc un traiteur. Lorsqu'une exception se produit lors de l'exécution du processus dans une sphère, le traiteur d'exceptions associé sera exécuté. Si ce traiteur ne peut pas traiter cette exception, cette exception sera propagée au bloc supérieur. Cette propagation continue de cette manière jusqu'au niveau de processus afin de trouver un traiteur.

Si l'exécution d'un traiteur d'exceptions associé à une activité se termine avec succès, alors l'exécution de cette activité est considérée comme une exécution réussie. Dans le cas contraire, le traiteur d'exceptions peut engendre une autre exception qui sera propagée vers un traiteur d'exceptions supérieur dans son hiérarchie.

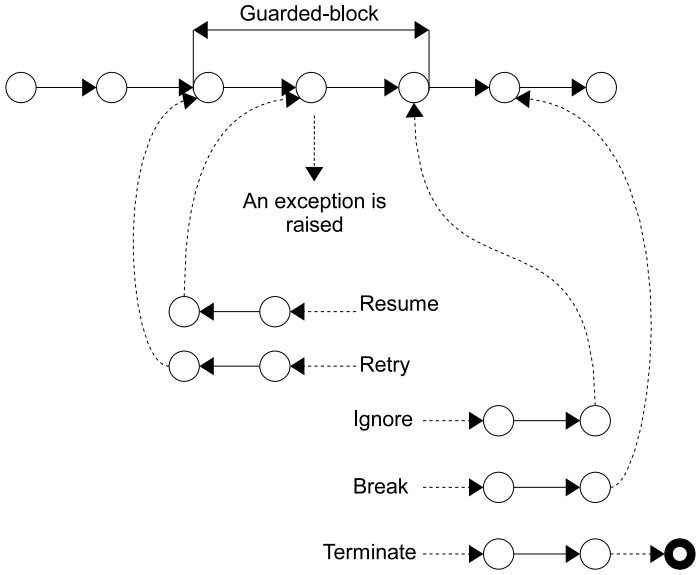


FIG. 2.10 – Types de traitement d'exceptions

Cette approche propose 5 types de traitement d'exceptions et le concepteur de processus peut choisir une parmi les types suivantes : resume, retry, ignore, break et terminate. La figure 2.10 montre que les types resume, retry et ignore sont exécutés lorsqu'une exception se produit dans le bloc spécifié.

**Discussion**

Cette section compare les approches d'adaptation en se basant sur les critères suivants :

1. Impact sur les instances d'une composition de services :
   Les approches d'adaptation par substitution de services et par transition d'activité n'influencent pas les instances d'une composition de services en cours d'exécution car elles ne changent pas son plan d'exécution. Dans l'approche par changement du plan d'exécution, il faut gérer les instances en cours d'exécution de la composition ayant changé son plan d'exécution.

2. Evolution :
   Comme le monde des services évolue, de nouveaux services ayant la même fonctionnalité sont créés. L'approche par substitution de services ayant la classification de services profite ce phénomène pour enrichir la base de services sera enrichie en stockant les nouveaux services. Cela augmente la possibilité de trouver un service remplaçant.

   L'approche par changement du plan d'exécution peut aussi évoluer la base de plans de processus mais un travail est alors nécessaire pour spécifier les nouveaux plans.

L'approche de transition d'activité est le moins flexible car les transitions correspondant aux exceptions dans le plan de processus sont prédéfinies. D'ailleurs, les activités de traitement d'exceptions doivent être couplées aux activités du processus. Cela rend cette approche peu flexible et difficile à faire évoluer.

3. Besoin d'une classification de services :
   Les deux approches d'adaptation par substitution de services et par changement de plan d'exécution ont besoin d'une classification de services. Ces approches ont donc besoin d'une base de connaissances de services ou d'une base de plans de processus. La conception et le développement de ces bases sont coûteux. Par ailleurs, l'approche d'adaptation par substitution de services doit résoudre les incompatibilités entre les signatures des services remplaçants et remplacés.

   Bien que l'approche d'adaptation par transition d'activité n'ait pas besoin d'une base de connaissances, la spécification de transition par événement devient compliquée quand il existe des exceptions multiples qui doivent être prises en compte à l'exécution d'une activité. Ces spécifications des exceptions multiples à une activité rendent le modèle de processus plus complexe.

## 2.3 Conclusion

Dans ce chapitre, nous avons présenté la taxonomie de composition de services en deux approches principales : composition statique et composition dynamique. Les modèles de composition de services statique sont basés sur le modèle de processus tandis que les modèles de composition de services dynamique s'appuient sur la représentation sémantique de services et les techniques de planification d'intelligence artificielle. Ces deux approches ont le même but de définir manuellement ou automatiquement une définition de composition de services ainsi qu'un plan d'exécution. Bien que l'approche de composition dynamique de services permette de générer automatiquement un plan d'exécution, cette approche est encore loin de pouvoir se mettre en place dans un système d'adaptation en temps d'exécution à cause de la complexité, du temps d'exécution et de la correction du plan de processus généré par rapport au but poursuivi.

Trois types d'approches sur l'adaptation d'une composition de services sont présentés :

- L'adaptation par substitution de services consiste à remplacer un service défectueux par un autre service ayant la même fonctionnalité. Cette approche propose également la résolution de conflits causés par les incompatibilités entre les signatures des services remplaçants et remplacés pour que la substitution soit correcte.

- L'adaptation par changement du chemin d'exécution consiste à localiser un autre itinéraire qui contourne le service défectueux et atteint le but désiré.

– L'adaptation par transition vers une activité de traitement d'exceptions redirige l'exécution d'une activité vers un traiteur d'exceptions qui est responsable de manipuler des exceptions produites dans sa sphère de surveillance.

Nous constatons que l'adaptation par substitution de services répond mieux à l'évolution de services que les deux autres approches car elle se base sur une base de connaissances de services qui permet de mettre à jour les nouveaux services ainsi que les changements des services.

# Chapitre 3

# Coordination adaptative de services

Ce chapitre présente les concepts du modèle de coordination de services adaptative que nous proposons. La section 3.1 donne une vision globale de notre modèle de coordination de services adaptative. La section 3.2 définit les concepts de base d'une coordination de services adaptative (service, méthode de service, activité, flot de contrôle et coordination). Cette section introduit également la notion de famille de méthodes nécessaire à la définition des équivalences entre les méthodes de service. La section 3.3 présente les opérations de : *substitution de méthodes de service* et *substitution d'activités* pour une exécution adaptative de coordination de services. La section 3.4 décrit le journal d'exécution de coordination de services qui stocke les événements (traces) de l'exécution de coordinations de services permettant d'observer son exécution et le calcul des mesures de QoS de méthodes de service. La section 3.5 conclut ce chapitre.

## 3.1 Approche

Une coordination de services pouvant être représentée sous forme d'un workflow se compose d'activités ordonnées par des flots de contrôle. Chaque activité effectue un appel à une méthode exportée par un service. Le schéma (définition) d'une coordination de services doit être pré-défini lors de la phase de conception. Ensuite, à la phase d'exécution, une instance de ce schéma sera créée et exécutée. L'exécution de cette instance se déroule strictement selon le schéma défini. Lorsqu'une exception se produit lors d'un appel d'une méthode de service d'une activité, son exécution peut être annulée ou compensée. Dans les systèmes de workflow récents, il est difficile de traiter ce type d'exception par manque de

- flexibilité pour pouvoir changer dynamiquement le schéma d'un workflow,

- connaissances des fonctionnalités des méthodes de service pour pouvoir trouver une autre méthode ayant une fonctionnalité équivalente et une signature compatible afin de remplacer la méthode échouée,

– opérations d'adaptation qui réagissent aux exceptions afin de continuer correctement l'exécution de la coordination de services.

### 3.1.1 Exemple d'une coordination de services

Pour illustrer notre approche, nous utilisons un exemple d'une application de recherche de vols représentée par une coordination de services sous forme d'un workflow (voir la figure 3.1).

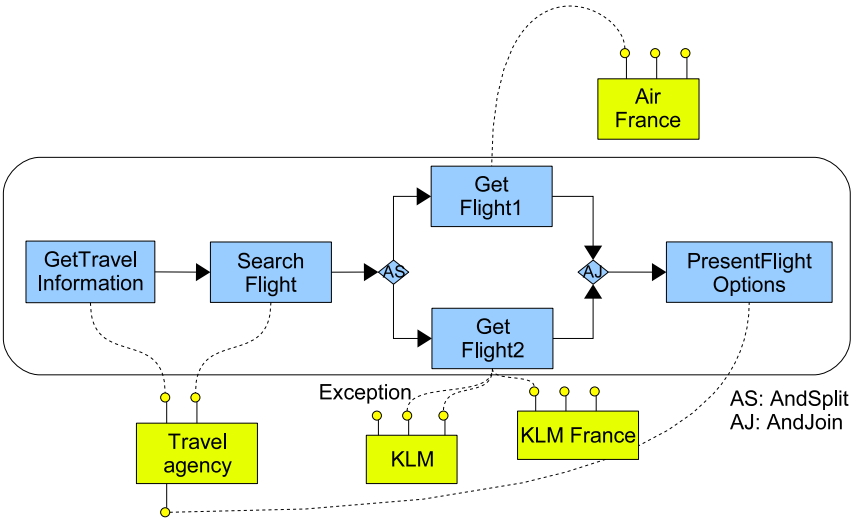


FIG. 3.1 – La coordination de services de recherche de vols

Un utilisateur qui veut consulter des vols envoie une recherche à l'activité "*GetTravelInformation*" en fournissant des informations comme la ville de départ, la ville de destination, les dates de départ et retour.

L'activité "*GetTravelInformation*" reçoit une demande de vols et vérifie que les champs obligatoires sont remplis. Si les champs obligatoires sont remplis, le processus passe ces données à l'activité "*SearchFlight*". Dans le cas contraire, une notification de manque de données sera envoyée à l'utilisateur.

L'activité "*SearchFlight*" vérifie la correction des données reçues dans un catalogue de services de réservation de vol. Si les données sont complètes et correctes, le processus passe ces données aux activités "*GetFlight1*" et "GetFlight2". Dans le cas contraire, une notification d'erreur sera envoyée à l'utilisateur.

Les activités "*GetFlight1*" et "*GetFlight2*" sont ensuite exécutées simultanément en appelant les méthodes de service choisies dans le catalogue, par exemple les services de recherche de vol de *Air France* et *KLM*.

L'activité "*PresentFlightOptions*" affiche les résultats de ces deux activités selon l'option de tri sur les données, par exemple sur les tarifs de vols, choisie par l'utilisateur.

Remarquons que dans cet exemple, les activités *GetFlight1* et *GetFlight2* appellent les méthodes de service externes qui sont fournies par les compagnies aériennes. L'exécution de ces méthodes est donc hors de portée du contrôle de la coordination. Au cas où l'appel à la méthode du service *KLM* fait par l'activité *Getflight2* échoue, le processus d'exécution de la coordination sera normalement arrêté avec une notification d'exception à l'utilisateur.

### 3.1.2 Propositions

Notre approche est de proposer une coordination de services adaptative aux exceptions d'appel de méthode de service pour garantir la continuation correcte de son exécution.

Par exemple, dans le cas où l'appel à la méthode de service *KLM* échoue et retourne une exception, afin de s'adapter à cette situation et continuer l'exécution de la coordination, nous proposons les trois types d'opérations suivants :

- ré-exécuter l'appel à cette méthode avec un nombre limité de fois ,
- substituer l'appel à cette méthode par l'appel à une autre méthode équivalente par exemple *GFKLMFrance*,
- substituer l'activité par une autre activité équivalente.

Proposer ces trois types d'opérations permet de pouvoir réagir aux types de fautes ponctuelles, intermittentes ou permanentes. La réexécution correspond au mieux aux fautes ponctuelles, par exemple l'indisponibilité temporaire d'un service, tandis que les deux autres types correspondent aux fautes permanentes, par exemple l'incompatibilité de signature d'une méthode de service. Pour une faute intermittente, l'opération d'adaptation choisie est basée sur la qualité de services de la méthode appelée.

Pour pouvoir fournir ces types d'opérations d'adaptation, les problèmes à résoudre sont les suivants :

- Comment trouver une méthode/activité équivalente et la remplacer ? Comment classifier les méthodes de service ?
- Comment choisir une méthode parmi un ensemble de méthodes équivalentes ?
- Comment adapter aux incompatibilités entre les signatures des méthodes de service équivalentes ?
- Comment remplacer une méthode par une autre méthode équivalente ?
- Comment remplacer une activité par une autre ou une sous-composition d'activités équivalente ? Comment modifier le schéma de la coordination ? Cela implique

de résoudre le problème de cohérence entre les instances créées et son schéma de workflow modifié.

– Quelle action d'adaptation réaliser quand une exception se produit ? Cela implique le problème de détecter une exception causée par quelle faute.

Ces questions sont au centre de notre travail et y apporter les réponses constituant l'objectif de cette thèse.

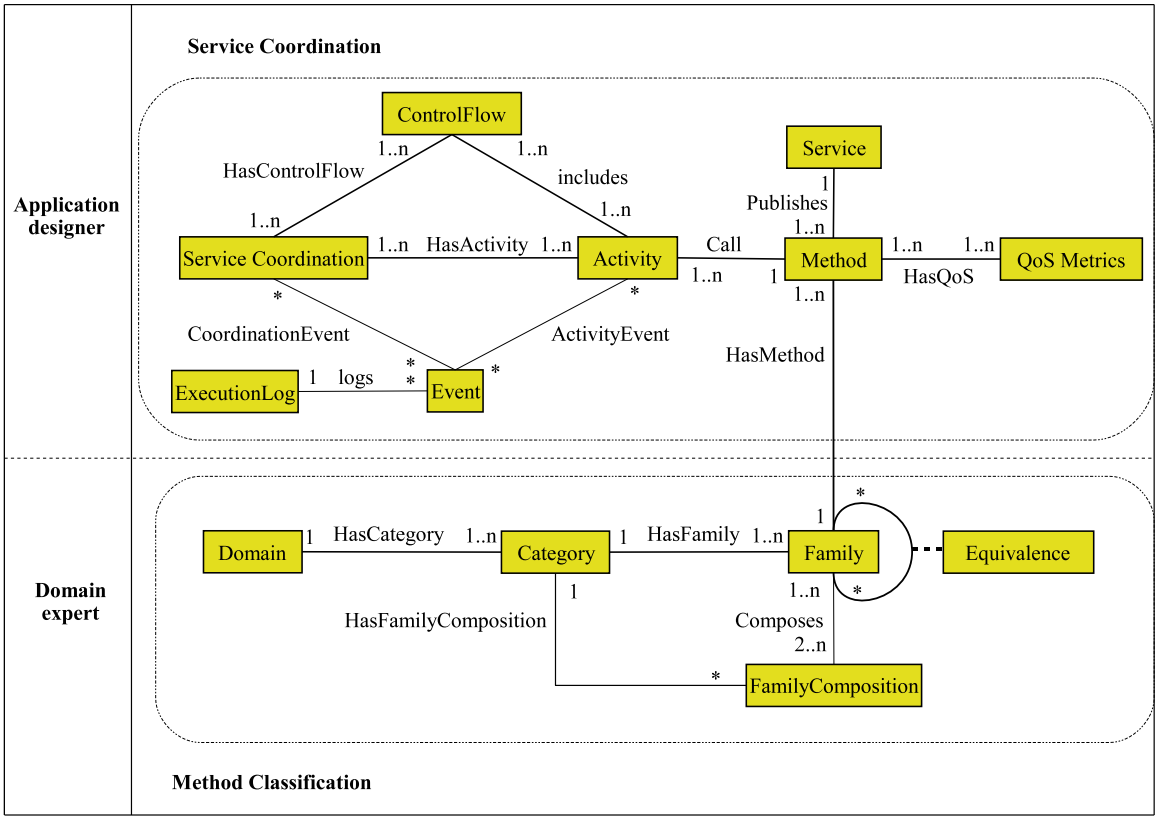


FIG. 3.2 – Vision de la coordination de services adaptative

La vision de notre approche pour une coordination de services adaptative est illustrée par la figure 3.2 qui introduit le diagramme de classes UML de notre modèle selon deux niveaux.

**Coordination de services :** Le niveau de coordination de services est réservé aux concepteurs qui définissent leurs applications à base de services. Ce niveau fournit d'abord les concepts de base pour définir une coordination de services sous forme de workflow : service, méthode, activité et flot de contrôle.

Les classes *Service* et *Method* représentent un service qui exporte des méthodes fournies par un fournisseur de services. Les instances de la classe *Method* sont utilisées pour effectuer les appels exécutés dans l'exécution des activités d'une coordination de services.

Les classes *Activity*, *ControlFlow* et *ServiceCoordination* sont utilisées pour définir une application à base de services qui est une instance du concept *ServiceCoordination*.

Ce niveau fournit également les classes : journal d'exécution (*ExecutionLog*), événement (*Event*) et qualité de service de méthode (*MethodQoS*).

Afin d'observer l'exécution de coordinations de service, une instance de la classe *ExecutionJournal* (journal d'exécution) est définie pour stocker des événements produits lors de ses exécutions. Les types d'événements sont définis non seulement pour observer l'état d'exécution mais aussi pour aider le calcul de la qualité de service de méthodes de service. La qualité de service (instance de *MethodQoS*) est représentée par le temps d'exécution, la fiabilité et la disponibilité.

La classe *MethodQoS* qui représente la qualité de service d'une méthode de service est utilisée pour :

– sélectionner la méthode la plus appropriée parmi des méthodes de service équivalentes,

– déterminer qu'une faute intermittente causée par un appel à une méthode de service est une faute permanente si la qualité de service de la méthode de service ne satisfait pas le critère de QoS désiré de l'utilisateur. Les critères de QoS d'une méthode de service seront présentés dans le chapitre suivant.

**Classification de méthodes de service :** Le niveau de classification de méthodes de service est réservé aux experts/concepteurs de domaine d'application qui définissent l'équivalence entre les méthodes de service par la classification de méthodes de service. Ce niveau fournit les concepts de classification de méthodes de service en famille par des besoins (de fonctionnalité). Un besoin est classifié par un domaine appliqué, une catégorie et une famille de méthodes de service. La classification de méthodes de service est utile pour la recherche de méthodes de service équivalentes.

La classe *Family* représente une classe de méthodes ayant les mêmes besoins et des signatures semblables. Les méthodes appartenant à une famille de méthodes sont donc équivalentes au besoin et compatibles à leurs signatures. Elles peuvent être remplacées correctement l'une par l'autre.

La classe *Equivalence* qui représente une équivalence entre deux familles de méthodes enrichit la possibilité de trouver des méthodes équivalentes. La notion d'équivalence de deux familles de méthodes *F1* à *F2* permet également de définir que les méthodes de la famille *F1* peuvent remplacer celles de la famille *F2*. Cependant cela peut engendre un problème d'incompatibilité entre leurs signatures de méthode qui doit être considéré pour que la substitution de méthodes soit correcte. Pour résoudre ce problème, des expressions de mappings entre les paramètres d'entrée/sortie de deux familles équivalentes sont définies explicitement dans l'objet de la classe *Equivalence* qui représente la relation équivalente de ces deux familles.

Une famille peut être définie par une composition de familles de méthodes (*Family-Composition*). Cela aide à trouver une solution quand aucune famille de méthodes existante ne satisfait un besoin demandé. Une composition de familles de méthodes est un *template* d'une composition de méthodes de service à laquelle une composition de méthodes concrètes correspondante sera créée dynamiquement au moment d'exécution.

Par exemple, reprenons l'exemple de la recherche de vols, supposons que l'opération correspondante à l'exception produite lors de l'exécution de l'activité *GetFlight2* est la substitution de méthodes. La recherche d'une méthode équivalente à la méthode *GFKLM* sera lancée en cherchant d'abord sur la famille *SearchFlight* contenant la méthode *KLM* et ensuite sur les familles équivalentes à *SearchFlight* si elles existent. La méthode la plus appropriée selon les critères de QoS désirés sera choisie, par exemple *GFKLM-France*, pour la remplacer.

Dans le cas où la qualité de service de la méthode *GFKLM* est "inacceptable" selon l'utilisateur, la substitution d'activités est choisie pour remplacer l'activité *GetFlight2* par une nouvelle activité *NewGetFlight2* qui appelle la méthode *GFKLM-France*. Dans ce cas, le schéma de la coordination est changé, mais l'intéret de cette solution est d'éliminer les fautes permanentes causée par l'appel à la méthode *KLM* lors de l'exécution de l'activité *GetFlight2*.

Un autre contexte peut être envisagé. Supposons que la méthode *GFKLM* retourne une liste de vols avec les tarifs en USD, les méthodes dans la famille *SearchFlight* retournent aussi une liste de vols mais avec les tarifs en Euro et aucune famille de méthodes de cette catégorie satisfait ce besoin. Donc il est nécessaire de convertir les tarifs en euros en tarifs en USD. Une composition de familles est créée pour répondre à ce besoin en composant séquentiellement la famille *SearchFlight* avec la famille *FlightDeviseConverter* qui contient les méthodes de service ayant la fonctionnalité de conversion des devises pour des tarifs de vols de Euro en USD.

Les classes de la coordination de service adaptative sont basées sur les notions classiques du modèle d'objets : type, domaine, classe, objet et prédicat qui sont présentées dans l'annexe A.

Pour la description des objets, nous utilisons le langage *Object Interface Format* (*OIF*) de *ODMG* [BEJ$^{+}$00].

Un fichier *OIF* contient des définitions d'objet. Chaque définition d'objet spécifie des types, des valeurs d'attributs et des relations avec d'autres objets définis.

Un identificateur d'objet est spécifié avec un nom d'étiquette d'objet unique qui est visible dans l'ensemble des fichiers *OIF*. Un exemple simple suivant illustre une définition d'un objet :

*Jack* **Person** { }

Dans cette définition, un objet de la classe *Person* est créé. Les valeurs des attributs de

cet objet ne sont pas initialisées. L'étiquette d'objet *Jack* est utilisée pour référencer à cet objet défini dans l'ensemble entier des fichiers *OIF*.

La suite de ce chapitre présente les éléments du modèle. La section 3.2 détaille les classes de ces deux niveaux de notre modèle. La section 3.3 décrit les opérations pour les adaptations de coordination. La section 3.4 présente l'observation d'exécution de la coordination adaptative par des événements de types définis.

## 3.2 Coordination de services

### 3.2.1 Service

Un service est un logiciel autonome qui fournit un ensemble de méthodes (opérations) accessible à travers un réseau (e.g. WWW, PSTN ou WiFi) au moyen d'une interface de programmation d'application (*API*) et garde le contrôle sur son exécution [ACKM04a]. Il est donc une ressource identifiée par une adresse *URI* (*Uniform Resource Identifier*) possédant un nom et qui exporte une interface *API* (*Application Programming Interface*). L'interface est définie par des signatures de méthodes. La signature d'une méthode spécifie le nom de la méthode (*M1*) et ses paramètres d'entrée (*I1*, *I2*) et de sortie (*O1*)(voir la figure 3.3).

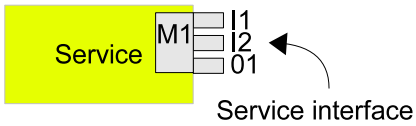


FIG. 3.3 – Interface de service

Tout service sera défini dans notre modèle comme une instance de la classe

**class Service** {
*ServiceName* String,
*Address* URI,
*Interface* Set(Method)}

Les objets de la classe *Service* sont construits en donnant les valeurs aux attributs : *ServiceName*, *Address*, *Interface*.

Une méthode d'un service réalise une fonction spécifique. Elle est caractérisée par un nom unique (*MethodName*), l'instance du service qui l'exporte, une liste de paramètres d'entrée(*Inputs*) et un paramètre de sortie (*Output*).

Toute méthode de service est un objet de la classe *Method* qui est définie par

**class Method**{
*MethodName* String,
*ServiceObject* Service,
*Inputs* List(Parameter),
*Output* Parameter)}

Le type *Parameter* $\in \bigtriangledown$ (cf Annexe A) caractérise les paramètres d'entrée/sortie d'une méthode de service. Il est défini par

**Parameter** :-tuple(
*paramName* String,
*paramValue* T)

où T est un type du domaines des types $\bigtriangledown$ (cf Annexe A)

Un paramètre est donc caractérisé par un nom et sa valeur est de type T.

Par exemple, le paramètre *Country* de type *String* est définie par

*Parameter* :-tuple(
*paramName* Country,
*paramValue* String)

Pour donner une valeur à l'instance du paramètre, on dénote simplement : {*paramName* Paramvalue}.
Par exemple, *Parameter* {*Country "France"*}.

**Exemple d'un service**

Un service appelé *GlobalWeather* produit par *Generic Objects Technologies Ltd* fournit des méthodes de consultation de météo d'une ville dans le monde entier à l'adresse *www.webservicex.net*. Il exporte des méthodes *GetCitiesByCountry*, *GetWeather* et *GetWeatherByZipCode*. La première fournit les noms des villes d'un pays demandé, la deuxième fournit les informations du jour de la météo d'une ville d'un pays et la troisième fournit aussi les informations du jour de la météo d'une ville aux Etats-Unis par son code postale.

Le service *GlobalWeather* est un objet de la classe *Service*. Il est défini par

*GWSid* **Service** {
*ServiceName* "*GlobalWeather*",
*Address www.webservicex.net*,
*Interface* {*GCCMid*, *GWMid*, *GetWeatherByZipCode*}
}

*GWSid* est l'étiquette d'objet qui référence l'objet de la classe *Service* représentant le service *GlobalWeather*.

Les objets de la classe *Method* correspondant au service *GlobalWeather* sont définis par

*GCCMid* **Method** {
*MethodName* "*GetCitiesByCountry*",
*ServiceObject GWSid*,
*Inputs* {[0] {*CountryName* String}},
*Output* {*Cities* String}}
}

L'étiquette d'objet *GCCMid* référence l'objet de la classe Method qui représente la méthode *GetCitiesByCountry* du service *GlobalWeather* étiqueté par *GWSid*.

*GWMid* **Method** {
*MethodName* "*GetWeather*",
*ServiceObject GWSid*,
*Inputs* {[0] {*CityName* String}},
*Output* {*Wheater* String}}
}

*GetWeatherByZipCode* **Method** {
*MethodName* "*GetWeatherByZipCode*",
*ServiceObject GWSid*,
*Inputs* {[0] {*ZipCode* String}},
*Output* {*Wheater* String}}
}

Les deux objets de la classe *Method* étiquetés respectivement par *GWMid* et *GetWeatherByZipCode* représentent les méthodes de consultation de la météo par le nom d'une ville *GetWeather* et par le code postal *GetWeatherByZipCode* du service *GlobalWeather* étiquetté par *GWSid*.

**QoS d'une méthode** La qualité de service (QoS) d'une méthode est représentée par les mesures suivantes :

- son temps d'exécution qui est le temps moyen entre l'appel de la méthode et la fin de son exécution (retour à l'appelant),
- sa fiabilité qui est la proportion entre le nombre d'appels réussis (résultat pertinent) et le nombre total d'appels,
- sa disponibilité qui est la proportion entre le nombre d'appels disponibles et le nombre total d'appels.

Toute qualité de service d'une méthode est un objet de la classe *MethodQoS* définie par

**class MethodQoS** {
*MethodID* Method,
*ExecutionTime* Float,
*Reliability* Float,
*Availabity* Float
}

– *MethodID* est l'étiquette d'une méthode de service,

– *ExecutionTime* est le temps moyen d'exécution de la méthode (en milliseconde),

– *Reliability* est la mesure de la fiabilité de la méthode,

– *Availabity* est la mesure de la disponibilité de la méthode.

Les mesures de QoS d'une méthode de service sont calculées en se basant sur les événements de l'exécution de coordinations de services stockés dans le journal d'exécution (voir la section 3.4). Les calculs des mesures de QoS sont détaillés dans la section 5.4.

Par exemple, la QoS de la méthode de service *GetWeatherByZipCode* est représentée par l'objet *QoSGetWeatherByZipCode* de la classe *MethodQoS* suivante :

*QoSGetWeatherByZipCode* **MethodQoS** {
*MethodID* GetWeatherByZipCode,
*ExecutionTime* 123,
*Reliability* 0.82,
*Availabity* 0.89
}

Son temps d'exécution est 123 (msec), sa fiabilité est 0.82 et sa disponibilité est 0.89.

### 3.2.2 Coordination

Une coordination de services spécifie une logique applicative. Elle est caractérisée par des activités, des opérateurs de contrôle basiques de workflow comme *Sequence*, *And/Or-split*, *And/Or-join* [50902, vdATKB03] et des flots de contrôles qui spécifient l'ordre d'exécution des activités dans la coordination.

**Activité :** Une activité d'une coordination de services représente une tâche à exécuter. Dans notre approche, une tâche concerne un appel à une méthode exportée par un service. Elle est caractérisée par un nom (*ActivityName*), des entrées (*Inputs*), une sortie (*Output*),

une pré-condition et une postcondition qui sont des prédicats et une instance de la classe méthode de service *InvocationMethod* à appeler.

Toute activité est un objet de la classe *Activity* définie par

**class Activity** {
*ActivityName* String,
*PreCondition* Predicate,
*PostCondition* Predicate,
*Inputs* List(Parameter),
*Output* T,
*InvocationMethod* Method,
*Call*()(T) }

où T est un type du domaine des types $\nabla$ (cf Annexe A).

En dehors des opérations de base que possède toute classe (voir l'annexe A), la classe *Activity* possède aussi l'opération *Call*()(T) qui réalise un appel à la méthode de service *InvocationMethod* permettant de réaliser la fonctionnalité de l'activité.

Par exemple, l'activité "*Getflight1*" est définie par

*GF1AID* **Activity** {
*ActivityName* "GetFlight1",
*PreCondition*
Inputs[0].From not NULL And
Inputs[1].To not NULL And
Inputs[2].DepartureDate not NULL And
Inputs[3].ReturnDate not NULL And
Inputs[4].Adults $>$ 0 And
Inputs[5].Children $\geq$ 0
*PostCondition* Output not NULL,
*Inputs* {
[0]{*From* String},
[1]{*To* String},
[2]{*DepartureDate* Date},
[3]{*ReturnDate* Date},
[4]{*Adults* Integer},
[5]{*Children* Integer}
}
*Output* File,
*InvocationMethod* GFAirFrance
}

Dans l'activité "*GetFlight1*" étiquetée par *GF1AID*, la pré-condition exige que les paramètres d'entrée ne soient pas Null et la post-condition exige que le résultat n'est pas

vide. Les paramètres d'entrées sont la ville de départ (*DepartingFrom*), la ville de destination (*TravellingTo*), la date de départ (*DepartureDate*), la date de retour (*ReturnDate*), le nombre d'adultes (*Adults*) et d'enfants (*Children*). Le paramètre de sortie est un fichier qui enregistre une liste de vols correspondants à la demande. La méthode d'appel est l'instance de la méthode *GFAirFrance* exportée par le service *Consultation* de la compagnie aérienne *Air France*.

**Opérateur de contrôle** décrit le comportement de l'exécution d'un flot de contrôle. Tout opérateur de contrôle est une instance du type *ControlOperator* défini par :

**ControlOperator** :-tuple(
*OperatorName* String,
*Condition* Predicate)

- `OperatorName` est le nom d'un opérateur de contrôle.
- `Condition` est associé seulement aux opérateurs *OrSplit* et *OrJoin*. Pour les autres opérateurs, la condition est vraie.

Les opérateurs de base de workflow : *Sequence*, *AndSplit*, *AndJoin*, *OrSplit*, *OrJoin* [50902, vdATKB03] sont décrits comme des instances de *ControlOperator*.

**Flot de contrôle** spécifie un ordre d'exécution d'activités par un opérateur de contrôle. Tout flot de contrôle est donc une instance du type *ControlFlow* défini par

**ControlFlow** :-tuple(
*Operator* ControlOperator,
*Input* Set(choice(Activity,ControlFlow)),
*Output* Set(choice(Activity,ControlFlow))

- `Operator` est un opérateur de contrôle.
- `Input` est l'ensemble des activités et/ou des flots de contrôle qui représentent l'entrée de l'opérateur de contrôle.
- `Output` est l'ensemble des activités et/ou des flots de contrôle qui représentent la sortie de l'opérateur de contrôle.

Par exemple, le flot de contrôle qui spécifie l'exécution séquentielle entre les activités "GetTravelInformation" et "SearchFlight" est défini par

**ControlFlow** {
*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* { GetTravelInformation_ID},
*Ouput* {SearchFlight_ID}
}

Dans ce flot de contrôle, son opérateur de contrôle est la séquence, son entrée/sortie est le nom d'étiquette d'objet *GetTravelInformation_ID* qui référence l'activité *GetTra-*

*velInformation/SearchFlight*.

Un flot de contrôle bien formé doit satisfaire des contraintes de cardinalité des entrées et des sorties des opérateurs de contrôle et des contraintes de sa structure (voir l'annexe B).

**Coordination de services :** toute coordination de services est un objet de la classe *ServiceCoordination* définie par

**class ServiceCoordination**{
*ServiceCoordinationName* String,
*StartActivity* Activity,
*EndActivity* Activity,
*ActivitySet* Set(Activity),
*OperatorSet* Set(String),
*ControlFlowList* List(ControlFlow)
*execute*()()}

- `ServiceCoordinationName` est le nom de la coordination de services,
- `StartActivity` est l'activité qui démarre la coordination de services,
- `EndActivity` est l'activité qui termine la coordination de services,
- `ActivitySet` est l'ensemble des activités dans la coordination de services,
- `OperatorSet` est l'ensemble des opérateurs de contrôle dans la coordination de services,
- `ControlFlowList` est la liste ordonnée des flots de contrôle dans la coordination de services.

Une opération suplémentaire de la classe *ServiceCoordination* est l'opération *execute*()() qui réalise l'exécution de la coordination.

La coordination de services *Recherche de vols*(*SFCoordination*) est définie par

*SFCoordination_ID* **ServiceCoordination** {
*ServiceCoordinationName* "SFCoordination",
*StartActivity* Gettravelinformation_ID,
*EndActivity* Presentflightoptions_ID,
*ActivitySet* {Gettravelinformation_ID, SearchFlight_ID,
Getflight1_ID, Getflight2_ID, Presentflightoptions_ID},
*OperatorSet* {"Sequence", "AndSplit", "AndJoin"},
*ControlFlowList* {
[0]{*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {GetTravelInformation_ID},
*Ouput* {SeachFlight_ID}},
[1]{*Operator* {*OperatorName* "AndSplit", *Condition* True},
*Input* {SearchFlight_ID},
*Ouput* {GetFlight1_ID, GetFlight2_ID}},

[2]{*Operator*{*OperatorName* "AndJoin", *Condition* True},
*Input* {GetFlight1_ID, GetFlight2_ID},
*Ouput* {PresentFlightOption_ID}}
}

La coordination *SFCoordination* démarre par l'activité *GetTravelInformation* et se termine par l'activité *PresentFlightOptions*. Elle se compose de cinq activités *GetTravelInformation*, *SearchFlight*, *Getflight1*, *Getflight2*, *PresentFlightOptions* qui sont ordonnées par une liste de trois flots de contrôle de *Sequence*, *AndSplit* et *AndJoin*. Chaque flot de contrôle contient les noms d'étiquette de ses activités d'entrée et de sortie.

**Graphe d'exécution d'une coordination de service :** Au moment de l'exécution (d'une définition) d'une coordination, celle-ci est vu comme un graphe d'exécution orienté : G($S$, $A$) où $S$ est l'ensemble de sommets. $A$ est l'ensemble d'arcs qui connectent des sommets. Un sommet est une instance d'activité. Un arc est représenté par une relation v($s1$, $s2$). On dit que l'arc $v$ part du sommet $s1$ et arrive au sommet $s2$. Les sommets sont décrits par des cercles et les arcs par des flèches.

Notons qu'une définition d'une coordination de services peut créer plusieurs graphes différents mais à un moment donné un seul graphe est exécuté.

Par exemple, le plan d'exécution de la coordination de recherche de vols *SFCoordination* est décrit par le graphe orienté *SearchFlightGraph*($S$,$A$) suivant :

$S$ = {*Gettravelinformation*, *SearchFlight*, *Getflight1*, *Getflight2*, *Presentflightoptions*}

$A$ = {(*Gettravelinformation*, *SearchFlight*), (*SearchFlight*, *Getflight1*), (*SearchFlight*, *Getflight2*), (*Getflight1*, *Presentflightoptions*), (*Getflight1*, *Presentflightoptions*)}

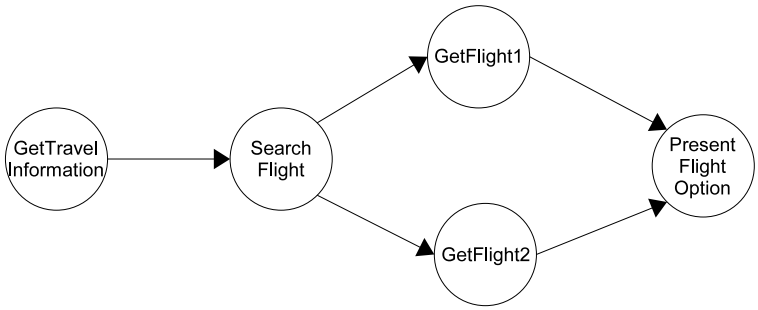


FIG. 3.4 – Graphe du plan d'exécution de la coordination de recherche de vols

La figure 3.4 montre le graphe d'exécution de la coordination de recherche de vols.

### 3.2.3 Famille de méthodes de service

Les sections précédentes ont présentées les classes du niveau de coordination de services dont les objets sont définis par le concepteur d'application. Les sections suivantes présentent les classes du niveau de classification de méthodes de service qui aident à classifier et à définir la relation d'équivalence entre les méthodes. Les objets de ces classes sont définis par les experts du domaine d'application. Comme présenté dans notre approche, la classe cœur de ce niveau est la famille de méthodes.

Une famille de méthodes de service rassemble des méthodes répondant à un même besoin et ayant les mêmes types de paramètres d'entrée et de sortie. Les classes *Domain*, *Category* et *Family* sont utilisés pour classifier un besoin. Un domaine d'application peut avoir des catégories et une catégorie possède des familles.

Un domaine qui est caractérisé par un nom et un ensemble de catégories est un objet de la classe *Domain* définie par

```
class Domain {
        DomainName String,
        CategorySet Set(Category)
}
```

Par exemple, le domaine *Utility* qui a des catégories comme la météo, la convertion de devises est définie par l'objet *UtilityID* comme suit :

```
UtilityID Domain {
    DomainName "Utility",
    CategorySet {MeteoID, DeviseConverterID}
}
```

Une catégorie, qui est caractérisée par un nom, le nom du domaine auquel elle appartient et un ensemble de familles de méthodes, est un objet de la classe *Category* définie par

```
class Category {
        CategoryName String,
        Domain Domain,
        FamilySet Set(Family)
}
```

Par exemple, l'objet *MeteoID* de la catégorie *Meteo* qui possède des familles comme *GeneralMeteoConsultation*, *GeneralMeteoConsultation* est définie par

```
MeteoID Category {
    CategoryName "Meteo",
```

```
        Domain UtilityID,
        FamilySet {GeneralMeteoConsultationID, USAMeteoByZipCodeID}
}
```

Une famille de méthodes est caractérisée par un nom *FamilyName*, le nom de la catégorie à laquelle elle appartient, une signature qui se compose d'une pré-condition *Precondition*, d'une postcondition *PostCondition*, d'une liste de paramètres d'entrée *Inputs* et d'un paramètre de sortie *Output*, et un ensemble d'instances de méthode qui correspond au besoin.

Toute famille de méthodes est un objet de la classe *Family* qui est définie par

```
class Family {
        FamilyName String,
        CategoryName String
        Signature tuple(
                Precondition Predicate,
                PostCondition Predicate,
                Inputs List(Parameter),
                Output Parameter),
        ServiceMethods Set(Method)
        }
```

La liste de paramètres d'entrée *Inputs* d'une famille représente exactement la liste de paramètres d'entrée d'instances de méthodes de service. Cette correspondance est nécessaire à l'utilisation des instances des méthodes.

**Exemples des familles de méthodes**

Une famille de méthodes, nommée *SearchFlight*, qui répond au besoin de recherche de vols contient des méthodes comme *GFOpodo*, *GFTerminalA*, *GFKLM* et *GFAirFrance* exportées respectivement par les services *Opodo*, *TerminalA*, *KLM* et *AirFrance* (voir la figure 3.5). Elle est définie par

```
SearchFlight_ID Family {
    FamilyName "SearchFlight",
    CategoryName "FlightService",
    Signature {
            Precondition
             Inputs[0].From not NULL And
             Inputs[1].To not NULL And
             Inputs[2].DepartureDate not NULL And
             Inputs[3].ReturnDate not NULL And
             Inputs[4].Adults > 0 And
```

Inputs[5].Children $\geq 0$,
*PostCondition* Output not NULL,
*Inputs* {
[0]{*From* String},
[1]{*To* String},
[2]{*DepartureDate* Date},
[3]{*ReturnDate* Date},
[4]{*Adults* Integer},
[5]{*Children* Integer}
},
*Output* File
},
*ServiceMethods* {GFOpodo, GFTerminalA, GFAirFrance, GFKLM}
}

Une famille de méthodes, nommée *USAMeteoConsultationbyZipCode*, qui répond au besoin de consultation de la météo aux Etat-Unis contient les méthodes *GetWeatherBy-ZipCode* et *GetWeatherInfo* exportées respectivement par les services *Global Wheather* et *Weather Service* (voir la figure 3.5). Elle est définie par

*USAMeteoByZipCodeID* **Family** {
*FamilyName* "*USAMeteoConsultationbyZipCode*",
*CategoryName* "Meteo",
*Signature* {
*Precondition* ServiceMethods[0].ZipCode not null,
*PostCondition* True,
*Inputs* {[0] {*ZipCode* String}},
*Output* {*Meteo* String}
},
*ServiceMethods* {GetWeatherByZipCode, GetWeatherInfo}
}

**Composition de familles de méthodes :** Une famille peut être composée par des familles de méthodes. On parle alors de famille composite qui répond à un besoin non satisfait par aucune autre famille de méthodes. La composition des familles de méthodes est définie de la même manière que la coordination de services dans laquelle une activité est remplacée par une famille et les flots de contrôle d'activités deviennent les flots de contrôle de familles. Cette composition est définie explicitement par un expert du domaine d'application.

Toute composition de familles de méthodes est un objet de la classe *FamilyComposition* définie ci-dessous

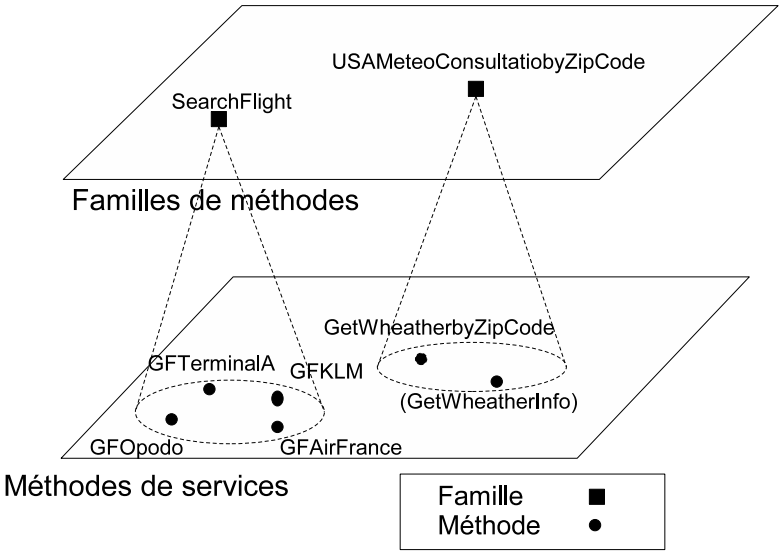


FIG. 3.5 – Familles de méthodes de consultation de météo et de recherche de vols

**class FamilyComposition** {
*FamilyCompositionName* String,
*CategoryName* String,
*Inputs* List(Parameter),
*Ouput* T,
*StartingFamily* Family,
*EndingFamily* Family,
*FamilySet* Set(Family),
*OperatorSet* Set(Operator),
*CompositionFlow* List(FamilyFlow),
*Execute*()()}

où T est un type du domaine des types $\bigtriangledown$ (cf Annexe A).

- `FamilyCompositionName` est le nom de la composition,
- `CategoryName` est le nom d'une catégorie de besoins,
- `Inputs` est la liste des paramètres d'entrée de la composition,
- `Output` est le paramètre de sortie de la composition dont la valeur est donnée par l'opération execute()(),
- `StartingFamily` spécifie l'instance de la famille de méthodes qui démarre la composition,
- `EndingFamily` spécifie l'identificateur de la famille de méthodes qui termine la composition,
- `FamiySet`spécifie l'ensemble des instances des familles dans la composition,
- `OperatorSet` spécifie l'ensemble des opérateurs de contrôle dans la composition,
- `CompositionFlow` spécifie une liste des instances du type *FamilyFlow* qui représentent le flot de familles.

– *execute*()() est l'opération suplémentaire de la classe *FamilyComposition*qui réalise l'exécution de cette composition.

Un flot de contrôle de familles (flot de familles) qui représente l'ordre d'exécution de familles est caractérisé par un opérateur de flot de contrôle et les ensembles des familles d'entrée et de sortie de l'opérateur. Ces instances sont définies par un type n-uplets qui est caractérisé par le nom d'étiquette d'une famille de méthodes, une liste de valeurs pour ses paramètres d'entrée et de sortie.

Tout flot de familles est une instance du type *FamilyFlow* qui est définie par

**FamilyFlow** :-tuple(
    *Operator* ControlOperator,
    *InputFamily* List(,
        tuple(
        *FamilyObject* Family,
        *InputValues* List(Parameter),
        *OutputValue* T)),
    *OutputFamily* List(
        tuple(
        *FamilyObject* Family,
        *InputValues* List(Parameter),
        *OutputValue* T)))

où T est un type du domaine de types $\bigtriangledown$ (cf Annexe A).

La classe de flot de familles qui est plus complexe que la classe de flot de contrôle d'activités a pour but de pouvoir définir l'ordre d'exécution de ses familles. Les paramètres d'entrée et de sortie des familles de méthodes dans un flot de familles doivent être affectés au moment de conception. Au moment donné de l'exécution, un flot de familles sera transformé en flot de méthodes. Chaque famille est remplacée par une de ses méthodes. La méthode choisie est la méthode la plus appropriée aux critères de QoS.

**Exemple :** on suppose qu'il existe une famille de méthodes *FlightDeviseConverter* qui répond au besoin de convertir des devises pour des tarifs de vols. Une composition séquentielle des deux familles *SearchFlight* et *FlightDeviseConverter* est construite pour convertir les tarifs des vols correspondants à la recherche de l'utilisateur en devise désirée (voir la figure 3.6). Cette composition prend pour entrée les paramètres d'entrée de la famille *SearchFlight* et les deux paramètres d'entrée qui représentent les types de devises à convertir. La sortie de cette composition est donnée par la valeur de la sortie de la dernière instance du flot de famille. Cette composition correspond à l'objet défini suivant :

*FDC_composition* **FamilyComposition** {
    *FamilyCompositionName* "FlightDeviseConverter_Composition",
    *CategoryName* "FlightService",
    *Inputs* {
        [0]{*From* String},

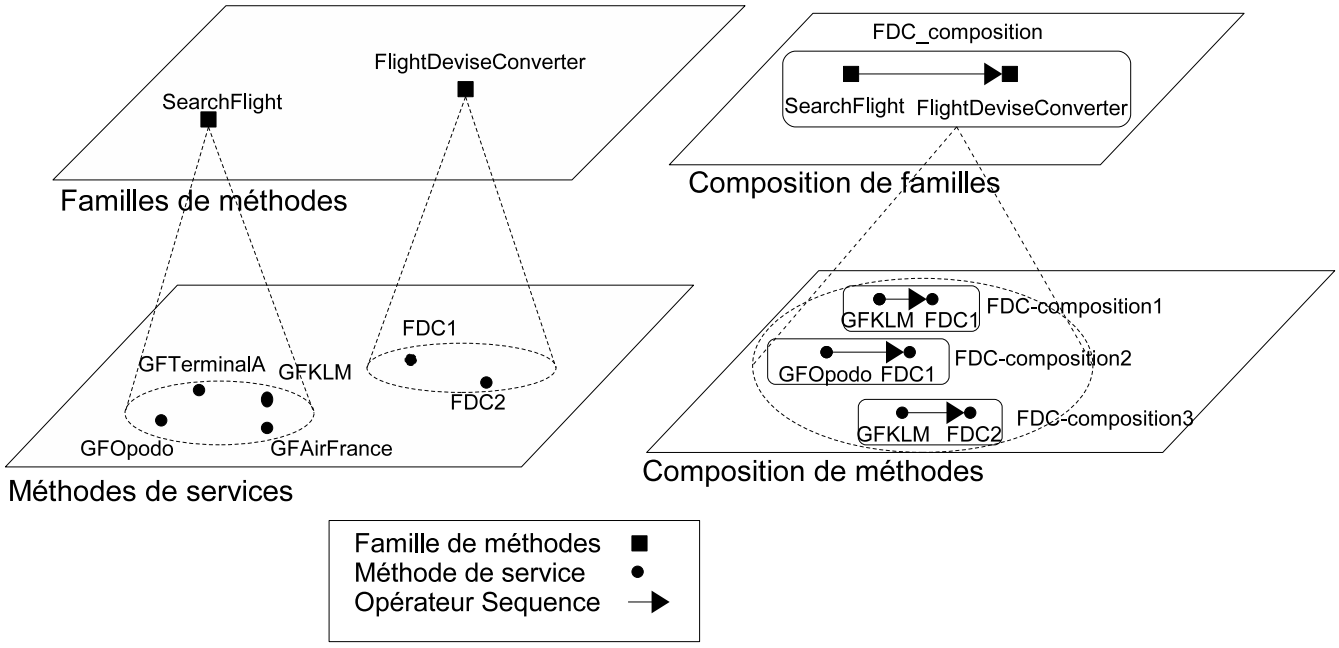


FIG. 3.6 – Une composition de familles pour la convertion de devises des tarifs de vols cherchés

```
            [1]{To String},
            [2]{DepartureDate Date},
            [3]{ReturnDate Date},
            [4]{Adults Integer},
            [5]{Children Integer}
            [6]{DeviseFrom String}
            [7]{DeviseTo String}
            }
Output CompositionFlow[0].output[0].outputvalue,
StartingFamily SearchFlight_ID,
EndingFamily FlightDeviseConverter_ID,
FamilySet {SearchFlight_ID, FlightDeviseConverter_ID},
OperatorSet {"Sequence"},
CompositionFlow {
        [0]{
        Operator {OperatorName "Sequence", Condition True},
        InputFamily {
            [0]{SearchFlight_ID,
                InputValues {
                    [0]{From Inputs[0].From},
                    [1]{To Inputs[1].To},
                    [2]{DepartureDate Inputs[2].DepartureDate},
                    [3]{ReturnDate Inputs[3].ReturnDate},
                    [4]{Adults Inputs[4].Adults},
                    [5]{Children Inputs[5].Children }
```

```
                }
            OutputValue SearchFlight.Output}
OutputFamily {
        [0]{FamilyObject DeviseConvertor_ID,
            InputValues {
                [0]{Flights CompositionFlow[0].InputValue.OutputValue},
                [1]{DeviseFrom Inputs[6].DeviseFrom },
                [2]{DeviseTo Inputs[7].DeviseTo },
                },
            OutputValue FlightDeviseConverter.Output}}
}
```

Dans le flot de familles, l'entrée de l'opérateur de contrôle *Sequence* est spécifiée statiquement par le nom d'étiquette de la famille de méthodes *SearchFlight_ID*. Ses paramètres d'entrée prennent pour valeur celles des éléménts de la liste des entrées de la composition *FDC_Composition_ID* indexées de 0 à 5. Son paramètre de sortie prend pour valeur celle de la sortie de la famille *SearchFlight_ID*. Pour la sortie de l'opérateur *Sequence*, la famille de méthodes est *FlightDeviseConverter_ID*. Ses paramètres d'entrée *Flights*, *DeviseFrom*, *DeviseTo* sont affectés respectivement les valeurs de l'attribut *OutputValue* de l'instance d'entrée de l'opérateur *Sequence* et des valeurs des éléménts de la liste d'entrée de la composition *FDC_Composition* indexés de 6 à 7. Son paramètre de sortie pour valeur la valeur de la sortie de la famille *FlightDeviseConverter*.

**Composition de méthodes :** Une composition de familles est considérée comme un patron de compositions de méthodes de service. Une composition de méthodes est créée à partir d'une composition de familles dans laquelle chaque famille est remplacée par une méthode de ses méthodes. Une composition de familles peut donc avoir plusieurs compositions de méthodes. Une composition de méthodes sera créée et exécutée automatiquement au moment de l'exécution en se basant sur la QoS des méthodes de service. La méthode choisie parmi les méthodes d'une famille est la méthode la plus appropriée par rapport à la qualité de service.

Par exemple, dans la figure 3.6, la composition de familles *FDC_Composition* peut avoir les compositions de méthodes comme *FDC_Composition1*, *FDC_Composition2* ou *FDC_Composition3*. Supposons qu'au moment de l'exécution, la composition de méthodes créée et exécutée est *FDC_Composition1*.

### 3.2.4 Equivalence de familles de méthodes

Comme présenté dans notre approche 3.1, la notion d'équivalence entre deux familles de méthodes *F1* à *F2* permet de définir que les méthodes de la famille *F1* peuvent remplacer celles de la famille *F2*. Cela enrichit la possibilité de trouver des méthodes équivalentes. Cependant cette équivalence est partielle car *F1* est équivalente à *F2* n'implique pas *F2* est équivalente à *F1*, c'est à dire qu'il n'est pas sûre que les méthodes de *F2*

peuvent remplacer correctement celles de *F1*.

Notons que les équivalences entre des familles de méthodes sont définies explicitement par des experts du domaine d'application. Dans une équivalence entre deux familles de méthode, l'expert définit également les expressions de mappings entre les paramètres d'entrée/sortie des deux familles équivalentes pour régler le problème d'incompatibilité entre leurs signatures de méthode. Cela garantit que la substitution entre les méthodes de ces deux familles est correcte.

Toute équivalence entre deux familles de méthodes est représentée par une instance de la classe association *Equivalence* entre deux classes familles de méthodes. La classe *Equivalence* est définie comme suit.

**class Equivalence** {
*FamilySource* Family,
*EquivalentTo* Family,
*InputMapping* String,
*OutputMapping* String
}

- *FamilySource* spécifie l'instance d'une famille de méthodes qui est la source de la relation d'équivalence,
- *EquivalentTo* spécifie l'instance d'une famille de méthodes qui est la destination de la relation d'équivalence,
- *InputMapping* est une expression qui définit les mappings entre les paramètres d'entrée des familles FamilySource et EquivalentTo,
- *OutputMapping* est une expression qui définit les mappings entre les paramètres de sortie des familles FamilySource et EquivalentTo,

Pour résoudre les incompatibilités des signatures des méthodes équivalentes, nous définissons le langage *Mapping* suivant :

```
MappingExpression ::= OperandMapping [ ";" OperandMapping ]*
OperandMapping ::= Operand = Operand | OperandExpression
Operand ::= Function.Input|Output. Name
OperandExpression ::= Operand Op OperandExpression | Literal
| (Type Operand)
Literal ::= Number | Character*
Type ::= Float | Integer | String
Op ::= "+" | "-" | "×" | "/"
```

Une expression mapping *MappingExpression* spécifie les mappings *OperandMapping*

entre les paramètres d'entrée/sortie entre de deux familles de méthode. Chaque mapping est une expression décrivant un opérande *Operand* qui est affecté à un autre opérande ou à une expression d'opérandes *OperandExpression*. Un opérande est un paramètre d'entrée (ou sortie) d'une famille. Une expression d'opérandes peut être un littéral, une conversion de type de donnée d'un opérande, ou une expression arithmétique entre les opérandes.

Soit les deux familles de méthodes ayant la fonctionnalité de vérification de carte de bancaire suivantes :

- la famille de méthodes *CCCbyCardNumber* qui vérifie la validité d'une carte bancaire par son numéro est définie par :

*CCCbyNumberCard_ID* **Family** {
*FamilyName* "CCCbyCardNumber",
*CategoryName* "CheckCreditCard",
*Signature* {
*Precondition* Inputs[0].CardNumber not NULL,
*PostCondition* True,
*Inputs*{
[0]{*CardNumber* String}
},
*Output* {*Valid* Boolean}},
*ServiceMethods* {CCCbyNumberCard1, CCCbyNumberCard2}}
}

la famille de méthodes *CCCbyTypeAndCardNumber* qui vérifie la validité d'une carte bancaire par son numéro et son type de carte est définie comme suit :

*CCCbyTypeAndNumberCard_ID* **Family** {
*FamilyName* "CCCbyTypeAndCardNumber",
*CategoryName* "CheckCreditCard",
*Signature* {
*Precondition* Inputs[0].CardType not NULL and Inputs[1].CardNumber not NULL
*PostCondition* True,
*Inputs*{
[0]{*CardType* String},
[1]{*CardNumber* String},
},
*Output* {*Valid* Boolean}},
*ServiceMethods* {CCCbyTypeAndNumberCard1, CCCbyTypeAndNumberCard2}}
}

Pour définir que la famille "*CCCbyNumberCard*" est équivalente à la famille "*CCCbyTypeAndCardNumber*", une instance *CCC* de la classe *Equivalence* est définie par

*CCC* **Equivalence** {
*FamilySource* CCCbyNumberCard_ID,
*EquivalentTo* CCCbyTypeAndCardNumber_ID,

*InputMapping* "CCCbyNumberCard_ID.CardNumber = CCCbyTypeAndCard-Number_ID.CardNumber",
*OutputMapping* "CCCbyTypeAndCardNumber_ID.output= CCCbyNumber-Card_ID.output"
}

L'expression décrite dans l'attribut *InputMapping* spécifie que le paramètre d'entrée *CardNumber* de la famille *CCCbyNumberCard* prend la valeur du paramètre *CardNumber* de la famille *CCCbyTypeAndCardNumber*, tandis que l'expression *OutputMapping* spécifie le mapping de la sortie de la famille *CCCbyTypeAndCardNumber* à la sortie de la famille *CCCbyCardNumber*.

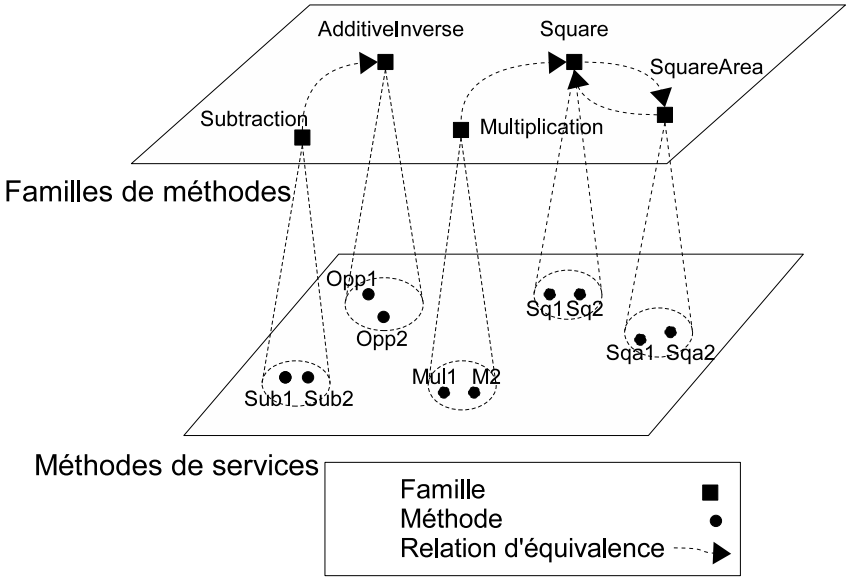


FIG. 3.7 – Equivalences entre les familles

**Propriétés de l'association *Equivalence* :**

Etant donné trois familles de méthodes F1, F2 et F3, l'association *Equivalence* dénotée $\mathscr{E}$ est

– Réflexive : $F1 \mathscr{E} F1$.

– Asymétrique : $(F1 \mathscr{E} F2) \wedge (F2 \mathscr{E} F1) \Rightarrow (F2 = F1)$.

– Transitive : $(F1 \mathscr{E} F2) \wedge (F2 \mathscr{E} F3) \Rightarrow (F1 \mathscr{E} F3)$.

Par exemple, pour mieux illustrer les propriétés de la relation *Equivalence*, nous utilisons les familles de méthodes de mathématique, la figure 3.7 montre les équivalences entre les familles par les flèches. On a :

– les familles *Subtraction* et *AdditiveInverse* : *Subtraction* $\mathscr{E}$ *AdditiveInverse*.

  La famille de méthodes *Subtraction* est équivalente à la famille *AdditiveInverse* dans le sens qu'il est possible d'utiliser la soustraction pour effectuer l'opposition d'un nombre mais pas au sens inverse.

– les familles *Multiplication*, *Square* et *SquareArea* : *Multiplication* $\mathscr{E}$ *Square*, *Square* $\mathscr{E}$ *SquareArea* et *SquareArea* $\mathscr{E}$ *Square*

  Le fait que la famille *Multiplication* soit équivalente à la famille *Square*, et *Square* est équivalente à la famille *SquareArea*, implique que la famille *Multiplication* est équivalente à *SquareArea* qu'il est possible d'utiliser la multiplication pour calculer le carré d'un nombre et également la surface d'un carré (*SquareArea*).

#### 3.2.4.1 Méthodes équivalentes

Etant données deux familles *F1* et *F2*, deux méthodes *m1* et *m2* exportées par deux services *S1* et *S2*, $m1 \in F1.MethodSet$ et $m2 \in F2.MethodSet$, *m1* est équivalente à *m2* si et seulement si

– $F1 \equiv F2$ ou

– ($F1$ $\mathscr{E}$ $F2$).

Note : *m2* n'est pas sûre d'être équivalente à *m1* car la relation $\mathscr{E}$ est asymétrique.

Par exemple, dans la figure 3.7 la méthode *Sub1* est équivalente à *Sub2* car ces méthodes appartiennent à la famille *Subtraction*. La méthode *Sub1* est équivalente à *Opp1* car *Sub1* appartient à la famille *Subtraction* qui est équivalente à la famille *AdditiveInverse*.

## 3.3 Opérations d'adaptation

Pour fournir l'adaptabilité à l'exécution d'une coordination de services nous proposons les trois types d'opérations d'adaptation qui peuvent intervenir lorsqu'une exception se produit lors d'un appel à une méthode de service :

1. la réexécution d'appel à une méthode un nombre de fois limité,

2. la substitution de méthodes de service en remplaçant la méthode appelée dans une activité par une autre méthode équivalente.

3. la substitution d'une activité par une autre activité ou par une sous-composition d'activités équivalente.

Rappelons que les types d'opérations sont proposés pour réagir aux types de fautes ponctuelles, intermittentes ou permanentes. La réexécution permet de s'adapter aux fautes ponctuelles comme l'indisponibilité temporaire d'un service.

Les deux derniers types d'opérations sont responsables de s'adapter aux fautes intermittentes ou permanentes comme l'erreur de la signature d'une méthode de service appelée. Grâce à la classification de méthodes de service en famille et à l'équivalence entre les familles, l'opération de substitution de méthodes peut être réalisée pour remplacer automatiquement une méthode équivalente à une méthode ayant échouée. La substitution d'activités est basée également sur la classification de méthodes et l'équivalence entre les familles pour créer une activité ou une sous-composition d'activités remplaçante.

Dans la suite de cette section nous concentrons sur les deux derniers types d'opérations qui s'adaptent vraiment à l'exécution d'une coordination de services. Le type d'opération réexécution sera présenté dans le chapitre suivant.

### 3.3.1 Substitution de méthodes de service

Comme une activité représente un appel à une méthode, la substitution de méthodes de service consiste à changer la méthode d'appel dans l'activité par une méthode équivalente quand l'appel à cette méthode appelée échoue. D'ailleurs, cette opération doit également s'adapter aux incompatibilités des paramètres d'entrée (sortie) de ces méthodes (s'il y a lieu).

L'opération *ReplaceMethod(M1 : Method, M2 : Method)* réalise la substitution de la méthode *M1* par la méthode *M2*. La substitution de la méthode M1 ne peut pas être réalisée par n'importe quelle méthode. La méthode remplacée M2 doit réaliser une fonctionnalité équivalente à celle de la méthode M1 pour que l'exécution de l'activité soit correcte par rapport à la logique applicative de la coordination de services.

On dit que la substitution de la méthode M1 par M2 réalisée par l'opération *ReplaceMethod(M1 : Method, M2 : Method)* est correcte si la méthode *M2* est équivalente à la méthode *M1*.

Etant donné, deux méthodes de service *M1* et *M2*, deux familles de méthodes *F1* et *F2*, $M1 \in F1.MethodSet$ et $M2 \in F2.MethodSet$,

la substitution de la méthode *M1* par *M2* est correcte si

- $F1 \equiv F2$ (*F1* et *F2* sont identiques) ou
- $F2 \mathscr{E} F1$ (la famille *F2* est équivalente à *F1*).

Pour une substitution de méthodes, les deux cas suivants peuvent être considérés.

1. **Substitution de deux méthodes équivalentes sans adaptation de paramètres :** quand les deux méthodes (remplacée et remplaçante) appartiennent à la même famille de méthodes et que la compatibilité de leurs signatures est garantie, la substitution consiste simplement à remplacer l'appel à la méthode remplacée par l'appel à la méthode remplaçante.

   Par exemple, les méthodes *GetWeatherByZipCode* et *GetWeatherInfo* présentées dans l'exemple de familles de méthodes de la section 3.2.3 qui appartiennent à la famille *USAMeteoConsultationbyZipCodeID* peuvent être remplacées correctement l'une par l'autre.

2. **Substitution de deux méthodes équivalentes avec adaptation de leurs signatures :**
   deux méthodes équivalentes (à un besoin) ne garantissent pas que leurs signatures sont compatibles. Leurs signatures peuvent être incompatibles aux nombres de paramètres d'entrée ou aux types de données de leurs paramètres. Cela nécessite de s'adapter à l'inconformité de leurs signatures.

L'incompatibilité des signatures de deux méthodes *M1* et *M2* appartenant aux deux familles *F1*, *F2* où *F1* est équivalente à *F2*, est résolue grâce aux mappings prédéfinis dans les attributs *InputMapping* et *OutputMapping* de l'objet de la classe *Equivalence* qui représente l'équivalence entre *F1* à *F2*.

Dans ce cas, la substitution de deux méthodes équivalentes avec adaptation de leurs signatures consiste d'abord à traiter à l'incompatibilité entre leurs signatures et après à remplacer l'appel à la méthode remplacée par l'appel à la méthode remplaçante.

Par exemple, les méthodes *CCCbyTypeAndNumberCard1* et *CCCbyNumberCard1* ont la même fonctionnalité de vérification de la validité d'une carte bancaire. Cependant les signatures de ces deux méthodes ne sont pas compatibles car la signature de la méthode *CCCbyTypeAndNumberCard1* a deux paramètres d'entrées tandis que la signature de la méthode *CCCbyNumberCard1* a seulement un paramètre d'entrée. La substitution de la méthode *CCCbyTypeAndNumberCard1* par la méthode *CCCbyNumberCard1* est faite en utilisant l'instance *CCC* de la classe *Equivalence* pour s'adapter à cette incompatibilité (voir l'exemple présenté dans la section 3.2.4).

**Discussion :** La substitution de méthodes de service équivalentes a été étudiée dans les travaux [BSB+05, CDK+06, SBDM02, YL05, TBFM06]. Pour la classification de services, ces travaux ont adopté la notion de type (classe) de services [CDK+06, YL05] ou de communauté de services [SBDM02, TBFM06]. Dans notre approche, nous avons défini les familles de méthodes qui ont la même sémantique de types/classes de services

ou communautés de services. D'ailleurs nous avons défini la notion d'équivalence entre les familles de méthodes pour enrichir la possibilité d'avoir des méthodes équivalentes.

### 3.3.2 Substitution d'activités

La substitution d'activités d'une coordination de services consiste à remplacer une activité ayant échouée par une autre activité ou une sous-composition d'activités. Cette opération modifie également la définition de la coordination de services. Ce changement doit prendre en compte également les modifications de flots de contrôle dans la coordination. Les règles de modification de flot de contrôle sont basées sur les travaux [BVSC05b, BVSC05a].

La substitution d'activités implique le problème de cohérence entre les instances créées (en cours d'exécution) et le schéma du workflow modifié. Plusieurs stratégies sont proposées pour résoudre ce problème [vdABV+00] :

- Les instances en cours d'exécution vont être retournées à l'état origine (rollback) et redémarrer leurs exécutions.

- De multiples versions d'un workflow sont permises. Les instances de l'ancienne version continueront leurs exécutions, tandis que les nouvelles instances seront créées par la nouvelle version.

- Les instances en cours d'exécution doivent changer leurs plans d'exécution pour correspondre au nouveau workflow.

Notre solution pour ce problème sera abordée dans la section 5.3.3.4. La suite de cette section présente les opérations de substitution d'activités.

Par exemple, une coordination de services *Co* est définie par

*Co* **ServiceCoordination** {
    *ServiceCoordinationName* "Co",
    *StartActivity* A1,
    *EndActivity* A2,
    *ActivitySet* {A1, A2}
    *OperatorSet* {"Sequence"},
    *ControlFlowList* {
        [0] {*Operator* {*OperatorName* "Sequence", *Condition* True},
            *Input* {A1},
            *Ouput* {A2}}
    }}

La coordination *Co1* qui remplace l'activité *A2* de la coordination *Co* par une sous-composition d'activités *InAdaptor*, *A22*, *OutAdaptor* est définie par

*Co1* **ServiceCoordination** {
*ServiceCoordinationName* "Co1",
*StartActivity* A1,
*EndActivity* OutAdaptor,
*ActivitySet* {A1, A22, InAdaptor, OutAdaptor}
*OperatorSet* {"Sequence"},
*ControlFlowList* {
[0] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {A1},
*Ouput* {InAdaptor}},
[1] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {InAdaptor},
*Ouput* {A22}},
[2] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {A22},
*Ouput* {OutAdaptor}}
}}

La figure 3.8 illustre la replanification de la coordination *Co* à la coordination *Co1*. Ce changement ne consiste pas simplement de remplacer l'activité *A2* par *A22* mais par une sous-composition d'activités dans laquelle les activités *InAdaptor* et *OutAdaptor* jouent le rôle d'adaptation des incompatibilités des paramètres d'entrée/sortie entre les activités *A2* et *A22*. Pour une substitution d'activités, le problème de la compatibilité des activités doit donc être examiné.

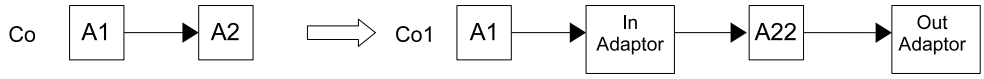


FIG. 3.8 – Replannification du plan d'exécution de Co à Co1

**Compatibilité de deux activités**

Etant données deux activités *A* et *B* ayant respectivement les préconditions *PreCondA* et *PreCondB*, les postconditions *PostCondA* et *PostCondB*, et les méthodes à appeler *MethodA* et *MethodB*.

On dit que *A* est compatible à *B* si et seulement si

– la méthode *MethodA* est équivalente à la méthode *methodB* et

– *PreCondA* équivalente à *PreCondB* et *PostCondA* équivalente à *PostCondB*.

Par exemple, examinons les deux activités qui vérifient la validité d'une carte bancaire suivantes :

*CheckCCbyNumber1* **Activity** {
*ActivityName* "CheckCCbyNumber1",
*PreCondition* Inputs[0].CardNumber not Null,
*PostCondition* True,
*Inputs* {[0] {CardNumber String}},
*Output* Boolean,
*InvocationMethod* {CheckCCbyNumber1}
}}

*CheckCCbyNumber2* **Activity** {
*ActivityName* "CheckCCbyNumber2",
*PreCondition* Inputs[0].CardNumber not Null,
*PostCondition* True,
*Inputs* {[0] {CardNumber String}},
*Output* Boolean,
*InvocationMethod* {CheckCCbyNumber2}
}}

Ces deux activités sont compatibles car

– les méthodes *CheckCCbyNumber1* et *CheckCCbyNumber2* sont équivalentes car elles appartiennent à la même famille "*CheckCCbyCardNumber*" (voir l'exemple dans la section 3.2.4) et

– leurs pré-conditions et post-condition sont équivalentes.

Une substitution d'une activité par une autre activité ou une sous-composition d'activités dépend également du type de flot de contrôle dans lequel l'activité remplacée se situe. Chaque type de flot de contrôle nécessite des opérations de substitution d'activité correspondantes.

#### 3.3.2.1 Substitution d'activités dans un flot de contrôle *Sequence*

Etant donnée une coordination de service *SeqCo* et une activité *B* qui précède une activité *A* et succède par une activité *C*. La coordination *SeqCo* est définie par

*SeqCo* **ServiceCoordination** {
*ServiceCoordinationName* "SeqCo",
*StartActivity* A,
*EndActivity* C,
*ActivitySet* {A, B, C},
*OperatorSet* {"Sequence"},
*ControlFlowList* {
[0] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {A},

*Ouput* {B}}
[1] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {B},
*Ouput* {C}}
}}

Une substitution d'une activité dans une séquence d'activités est réalisée par l'opération *ReplaceSeqActivity*.

```
ReplaceSeqActivity(Co : ServiceCoordination, B : activity,
R : Activity)
```

Etant données une coordination de services *Co* et les activités *A*, *B*, *C* connectées succesivement par les deux opérateurs *Sequence* , la substitution de *B* par *R* est effectuée successivement l'opération *ReplaceSeqActivity*(Co : ServiceCoordination, B : activity, R : Activity).

Par exemple *ReplaceSeqActivity* (*SeqCo*, *B*, *R*) est utilisé pour remplacer l'activité *B* liée entre les activités *A* et *C* par l'activité *R* (voir la figure 3.9).

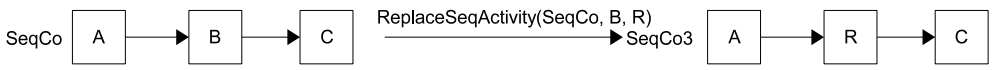


FIG. 3.9 – Opération ReplaceSeqActivity

La nouvelle définition de la coordination *SeqCo* sera *SeqCo2* qui est définie par

*SeqCo2* **ServiceCoordination** {
*ServiceCoordinationName* "SeqCo2",
*StartActivity* A,
*EndActivity* C,
*ActivitySet* {A, R, C},
*OperatorSet* {"Sequence"},
*ControlFlowList* {
[0] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {A},
*Ouput* {R}}
[1] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {R},
*Ouput* {C}}
}}

#### 3.3.2.2 Substitution d'activités dans des flots de contrôle *ANDSplit* et *AndJoin*

Etant donnée une coordination de service *AndSplitCo1* composée par les flots de contrôle *AndSplit*/*AndJoin* définie de la manière suivante :

*AndSplitCo1* **ServiceCoordination** {
*ServiceCoordinationName* "AndSplitCo1",
*StartActivity* A,
*EndActivity* C,
*ActivitySet* {A, B, C, E},
*OperatorSet* {"AndSplit", "AndJoin"},
*ControlFlowList* {
[0] {*Operator* {*OperatorName* "AndSplit", *Condition* True},
*Input* {A},
*Ouput* {B, E}},
[1] {*Operator* {*OperatorName* "AndJoin", *Condition* True},
*Input* {B, E},
*Ouput* {C}}
}}

Une substitution d'une activité dans des flots de contrôle *AndSplit* et *AndJoin* est effectuée par l'opération *ReplaceAndActivity*.

```
ReplaceAndActivity (Co : ServiceCoordination, A : Activity,
index1 : Integer, index2 : Integer, R : Activity)
```

L'opération *ReplaceAndActivity* remplace une activité *A* dans les flots de contrôle *AndSplit* et *AndJoin* par une autre activité *R* dans la coordination de services *Co*.

La substitution de *A* par *R* est effectuée en remplaçant l'activité *A* par l'activité *R* dans les flots de contrôle *AndSplit* et *AndJoin*.

Par exemple, l'opération *ReplaceAndActivity*(*AndSplitCo1*, *B*, *R*) est utilisée pour remplacer l'activité *B* dans les flots de contrôle indexés 0 et 1 par l'activité *R* (voir la figure 3.10). La définition de la coordination *AndSplitCo1* sera transformée en *AndSplitReplaceCo* qui est définie par

*AndSplitReplaceCo* **ServiceCoordination** {
*ServiceCoordinationName* "AndSplitReplaceCo",
*StartActivity* A,
*EndActivity* C,
*ActivitySet* {A, R, C, E},
*OperatorSet* {"AndSplit", "AndJoin"},
*ControlFlowList* {
[0] {*Operator* {*OperatorName* "AndSplit", *Condition* True},
*Input* {A},

*Ouput* {R, E}},
[1] {*Operator* {*OperatorName* "AndJoin", *Condition* True},
*Input* {R, E},
*Ouput* {C}}
}}

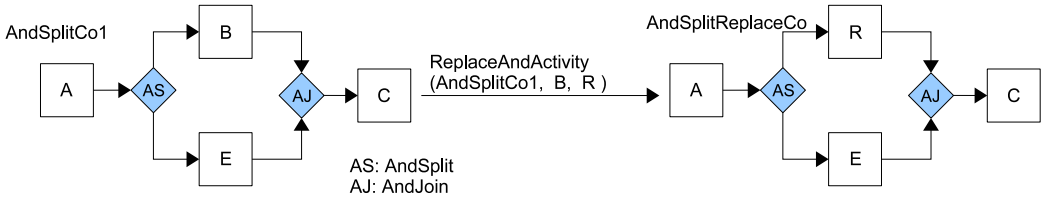


FIG. 3.10 – Opération ReplaceAndActivity

#### 3.3.2.3 Substitution d'activités dans des flots de contrôle *ORSplit* et *ORJoin*

Etant donnée une coordination de services *OrSplitCo1* composée par les flots de contrôle *OrSplit*/*OrJoin*, elle est définie de la manière suivante :

*OrSplitCo1* **ServiceCoordination** {
*ServiceCoordinationName* "OrSplitCo1",
*StartActivity* A,
*Endactivity* C,
*ActivitySet* {A, B, C, E},
*OperatorSet* {"OrSplit", "OrJoin"},
*ControlFlowList* {
[0] {*Operator* {*OperatorName* "OrSplit", *Condition* Condition1},
*Input* {A},
*Ouput* {B, E}},
[1] {*Operator* {*OperatorName* "OrJoin", *Condition* Condition2},
*Input* {B, E},
*Ouput* {C}}
}}

Une substitution d'une activité dans des flots de contrôle *ORSplit* et *ORJoin* est effectuée par l'opération *ReplaceOrActivity*.

```
ReplaceOrActivity (Co : ServiceCoordination, A : Activity,
R : Activity, index1 : Integer, CF2 : Integer,
OrSplitCondition : Predicate, OrJoinCondition : Predicate)
```

L'opération *ReplaceOrActivity*(*Co*, *A*, *R*, index1, index2, OrSplitCondition, OrJoinCondition) remplace l'activité *A* connectée par deux opérateurs *OrSplit* et *OrJoin* à une autre activité *R* dans la coordination de services *Co*,

La substitution de *A* par *R* est effectuée en remplaçant l'activité *A* par l'activité *R* dans les flots de contrôle indexés par *[0]* et *[1]* et les conditions de ces flots de contrôle par les conditions *OrSplitCondition* et *OrJoinCondition*.

Par exemple, l'opération *ReplaceOrActivity*(*OrSplitCo1*, *B*, *R*, 0, 1, *OrSplitCondition*, *OrJoinCondition*) est utilisée pour remplacer l'activité *B* dans les flots de contrôle indexés [0] et [1] par l'activité *R* et les conditions associées à *OrSplitABE* et *OrJoinBEC* par les conditions *OrSplitcondition* et *OrJoincondition* afin de prendre en compte la nouvelle activité *R* remplacée (voir la figure 3.11). La définition de la coordination *OrSplitCo1* sera transformée en *OrSplitReplaceCo1* qui est définie par

*OrSplitReplaceCo* **ServiceCoordination** :-tuple(
*ServiceCoordinationName* "OrSplitReplaceCo",
*StartActivity* A,
*EndActivity* C,
*ActivitySet* {A, R, C, E},
*OperatorSet* {"OrSplit", "OrJoin"},
*ControlFlowList* {
[0] {*Operator* {*OperatorName* "OrSplit", *Condition* OrSplitCondition},
*Input* {A},
*Ouput* {R, E}},
[1] {*Operator* {*OperatorName* "OrJoin", *Condition* OrJoinCondition},
*Input* {R, E},
*Ouput* {C}}
}}

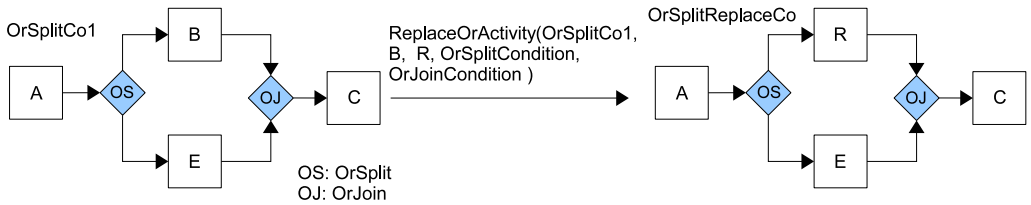


FIG. 3.11 – Opération ReplaceOrActivity

#### 3.3.2.4 Adaptation pour une substitution d'activités incompatibles

L'objectif de l'adaptation pour une substitution d'activés est de s'adapter l'incompatibilité entre les signatures des méthodes équivalentes des activités remplacées et remplaçantes.

- Adaptation de compatibilité des types de donnée des entrées et sorties entre les deux méthodes équivalentes qui ne sont pas pareilles mais pouvant convertir l'un à l'autre.

Par exemple, étant donnée une activité *A1* qui appelle une méthode *M1* ayant une entrée et une sortie de type *Integer*, une activité *A2* qui appele une méthode *M2* ayant

une entrée et une sortie de type *Float*, et *M2* est équivalente à *M1*. Pour remplacer *A1* par *A2*, l'adaptation de compatibilité de type doit convertir le type d'entrée de *M1* du type *Integer* au type *Float* pour correspondre au type d'entrée de *M2* et ensuite convertir le type de sortie de *M2* du type *Float* au type *Integer* pour correspondre au type de sortie de *M1*.

- Adaptation de compatibilité entre les paramètres d'entrées et sorties des deux méthodes : Cette adaptation permet de traiter les incompatibilités entre les signatures des deux méthodes équivalentes grâce aux mappings prédéfinis dans l'instance de la classe *Equivalence* correspondante.

Par exemple, étant donnée une activité *A1* et une activité *A2* qui appellent respectivement les méthodes de service *CCCbyTypeAndNumberCard1* et *CCCbyNumberCard1* pour vérifier la validation d'une carte bancaire. Du point de vue fonctionnelle, l'activité *A1* peut être remplacée par l'activité *A2* car la méthode *CCCbyNumberCard1* est équivalente à la méthode *CCCbyTypeAndNumberCard1*. Cependant, du point de vue opérationnelle if faut s'adapter à l'incompatibilité entre leurs paramètres d'entrée car la méthode *CCCbyTypeAndNumberCard1* a deux paramètres d'entrée tandis que la méthode *CCCbyNumberCard1* a seulement un paramètre d'entrée. L'incompatibilité est résolue grâce à l'instance *CCC* de la classe *équivalence* qui représente l'équivalence entre les familles de ces deux méthodes.

L'adaptation d'une substitution d'activités consiste à remplacer une activité *A* par une sous-composition *S* qui peut être composée de trois activités *InAdapter*, *B* et *OutAdapter* connectées séquentiellement. Dans la sous-composition, *B* invoque une méthode équivalente à la méthode invoquée par *A*, les deux activités *InAdapter* et *OutAdapter* appelent le service d'adaptabilité *Adaptor*.

**Activité *InAdapter* :** Etant donnés *In1*, *In2* des ensembles des entrées de l'activité *A1* et *A2*, un adaptateur des entrées qui est compatible *IN1* à *IN2* est une activité d'adaptation *InAdapter* qui mappe les entrées *IN1* aux sorties *IN2* en appelant le service *Adaptor*.

**Activité *OutAdapter* :** Etant données *Out1*, *Out2* l'ensemble des sorties de l'activité *A1* et *A2*, un adaptateur des sorties qui est compatible *Out1* à *Out2* est une activité *OutAdapter* qui mappent les entrées *Out2* aux sorties *Out1* en appelant le service *Adaptor*.

Le service *Adaptor* qui est un service interne défini dans notre système d'adaptation SEBAS (voir la section 5.1) traite l'adaptation de l'incompatibilité des signatures des méthodes équivalentes en se basant sur les mappings prédéfinis dans l'instance de la classe *Equivalence* correspondante.

**Compatibilité d'activités :** Etant données les activités *A1*, *A2* et les adaptateurs *InAdapter*, *OutAdapter* qui adaptent les incompatibilitées entre les entrées/sorties de *A1* et *A2* (voir la figure 3.8).

```
A1 est compatible à la sous-composition
```

*SC* **ServiceCoordination** {
*ServiceCoordinationName* "SC",
*StartActivity* InAdapter,
*EndActivity* OutAdapter,
*ActivitySet* {A2, InAdapter, OutAdapter},
*OperatorSet* {"Sequence"},
*ControlFlowList* {
[0] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {InAdapter},
*Ouput* {A2}},
[1] {*Operator* {*OperatorName* "Sequence", *Condition* True},
*Input* {A2},
*Ouput*{OutAdapter}}
}}

si et seulement si

– la pré-condition/postcondition de *A1* est équivalente à la pré-condition/postcondition de *A2* et

– la méthode d'appel de *A2* est équivalente à la méthode d'appel de *A1*.

**Exemple 3.1.** *La figure 3.12 illustre la substitution de l'activité A qui appelle la méthode de service CCCbyTypeAndNumberCard1 par la sous-composition S composée par une séquence de trois activités InAdapter, B appelant la méthode de service CCCbyNumber-Card1, OutAdapter. L'activité InAdapter appelle le service Adaptor qui est responsable de résoudre l'incompatibilité entre les entrées de deux méthodes en utilisant l'expression décrite par l'attribut InputMapping de l'instance CCC de la classe Equivalence (voir l'exemple dans la section 3.2.4). L'activité OutAdapter appelle également le service Adaptor pour s'adapter aux sorties de ces deux méthodes.*

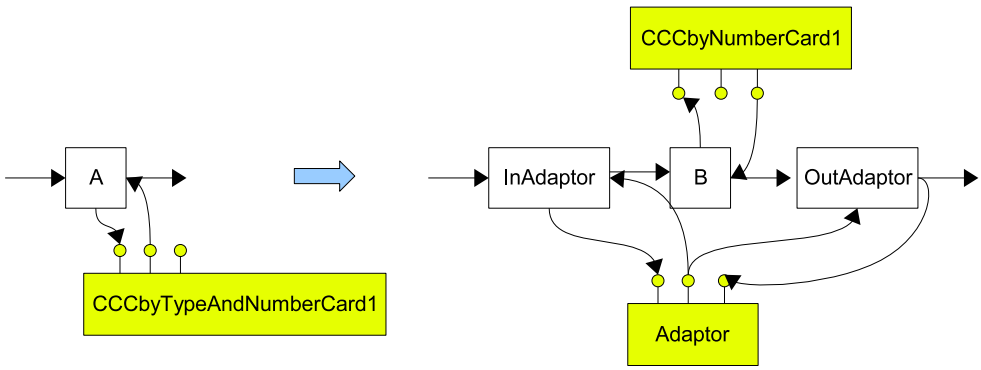


FIG. 3.12 – Substitution de l'activité A par une sous-composition

**Discussion** L'adaptation d'une coordination de services par la replanification de sa définition a été étudiée dans le domaine des workflows [JYH+06, CLK01, SH03]. Dans ces travaux, une replanification d'un workflow est réalisée soit par une intervention humaine [CLK01], soit par des règles [JYH+06, SH03], remplaçant une activité par une autre activité ou une sous-composition prédéfinie. Dans notre approche, nous définissons aussi l'adaptation par la replanification en substituant une activité par une autre équivalente. Nous nous différencions de ces travaux par le fait que l'activité remplaçante n'est pas pré-définie, elle sera créée dynamiquement en se basant sur les familles de méthodes équivalentes pour trouver les méthodes de service équivalentes à la méthode de l'activité remplacée. Parmi les méthodes trouvées, la méthode choisie est la méthode la plus appropriée par rapport aux critères de QoS demandés.

## 3.4 Journal d'exécution

Lors de l'exécution d'une coordination de services adaptative, un journal d'exécution stocke les événements (traces) de son processus d'exécution.

Un événement d'exécution qui se produit au moment de l'exécution d'une coordination de service est caractérisé par un identifiant (*EventID*), une estampille (*TimeStamp*) et les attributs qui fournissent les informations du contexte de cet événement.

Tout événement est un objet de la classe *Event* définie par

**class Event** {
    *EventID* Integer,
    *TimeStamp* Time,
    *CoordinationName* String
    *ActivityName* String
    *ServiceURI* URI
    *MethodName* String
    *InParameter* String
    *Phase* String
    *State* String
    *ExceptionMessage* String
    }

– *CoordinationName* spécifie le nom de la coordination de services,

– *ActivityName* spécifie le nom de l'activité de la coordination de services,

– *ServiceURI* spécifie l'adresse URI du service qui exporte la méthode appelée,

– *MethodName* spécifie le nom de la méthode appelée,

– *InParameter* spécifie les paramètres d'entrée (ParameterName$_i$) de la méthode de

service et leurs valeurs données ($Value_i$). La valeur de ce paramètre est écrite sous forme BNF suivante :
*InParameter* ::= "$ParameterName_1$ = $Value_1$ ; [$ParameterName_i$ = $Value_i$]* "

– *Phase* spécifie l'état d'exécution de la coordination ou/et de l'activité qui est détaillé dans la section suivante

– *State* spécifie l'état réussi (True) ou échoué (False) d'un état d'exécution,

– *ExceptionMessage* spécifie le message d'une exception reçu lorsque l'appel à la méthode échoue.

Pour observer l'exécution d'une coordination de service adaptative, les types d'événements sont définis. Un type d'événement est une expression qui représente un ensemble d'événements et les conditions dans lesquelles ces événements se produisent. Au travers de son type, l'événement identifie la nature d'un changement qui survient dans l'exécution de la coordination de services adaptative.

**Types d'événement de l'exécution d'une activité :** l'exécution d'une activité (*PrepareActivity*, *InvokeActivity*, *EndActivity*) est représentée par les types d'événements suivants :

– le type d'événement *PrepareActivity* représente la préparation d'un appel à une méthode de service : un événement de ce type d'événement est identifié quand son attribut *Phase* a la valeur "PrepareActivity",

– le type d'événement InvokeActivity représente l'appel à une méthode de service : un événement de ce type d'événement est identifié quand son attribut *Phase* a la valeur "InvokeActivity",

– le type d'événement EndActivity représente la réception du résultat de la méthode : un événement de ce type d'événement est identifié quand son attribut *Phase* a la valeur "EndActivity". Ce type d'événement renseigne également l'état réussi ou échoué de l'appel à la méthode de service par la valeur de l'attribut *State* (True ou False).

Par exemple, un événement du type "EndActivity" produit dans le contexte de l'exécution réussie (State True) d'un appel à la méthode "SearchFlight" de l'activité "GetFlight1" de la coordination de recherche de vols "SFCoordination" est définie comme suit :

*E1* **Event** {
*EventID* 110,
*TimeStamp* "2008-07-16T19 :20 :31,4+01 :00",
*CoordinationID* "SFCoordination",

*ActivityID* "GetFlight1",
*ServiceURI* www.AirFrance.fr},
*MethodName* "SearchFlight"},
*InParameter* "From = 'Paris' ; To = 'London' ; DepartureDate = 20/12/2008 ;
ReturnDate = 26/12/2008 ; Adults = 1 ; Children = 0",
*Phase* "EndActivity",
*State* True ,
*ExceptionMessage* ""
}

Le type d'événement *EndActivity* représente également l'état d'une exécution d'une activité réussie (*true*) ou échouée (*false*) et un message qui contient le résultat ou l'exception rendu par la méthode appelée.

Par exemple, au cas où l'appel à la méthode *SearchFlight* dans l'exécution de l'activité *GetFlight1* reçoit une exception d'indisponibilité du service, l'événement du type *EndActivity* est définie par :

*E11* **Event** {
*EventID* 110,
*TimeStamp* "2008-07-16T19 :20 :31,4+01 :00",
*CoordinationID* "SFCoordination",
*ActivityID* "GetFlight1",
*ServiceURI* www.AirFrance.fr},
*MethodName* "SearchFlight"},
*InParameter* "From = 'Paris' ; To = 'London' ; DepartureDate = 20/12/2008 ;
ReturnDate = 26/12/2008 ; Adults = 1 ; Children = 0",
*Phase* "EndActivity",
*State* False,
*ExceptionMessage* "Unavailable service"
}

**Types d'événements de l'exécution d'une coordination de services :** l'exécution d'une coordination est observée par des événements produits lors d'exécution de ses activités et par les deux types d'événements suivants :

- le type d'événement *BeginProcess* représente le début d'un processus d'une coordination de services : un événement de ce type est identifié quand son attribut *Phase* a la valeur "BeginProcess"

- le type d'événement *EndProcess* représente la fin d'un processus d'une coordination de services : l'événement de ce type est identifié quand l'attribut *Phase* de l'instance d'événement a la valeur "EndProcess". Dans ce cas, son attribut *State* renseigne l'état réussi ou échoué de l'exécution de la coordination

Pour ces deux types d'événement, les attributs *ActivityID*, *ServiceURI*, *MethodName InParameter* sont vides.

Par exemple, l'événement qui représente le début de la coordination de services de recherche de vol est représenté par :

*E21* **Event** {
*EventID* 310,
*TimeStamp* "2008-07-16T19 :20 :28,5+01 :00",
*CoordinationID* "SFCoordination",
*ActivityID* "",
*ServiceURI* "",
*MethodName* "",
*InParameter* "",
*Phase* ""BeginProcess"",
*State* True,
*ExceptionMessage* ""
}

**Journal d'exécution** Le journal d'exécution est caractérisé par un nom (*LogName*) et une liste (*EventLog*) d'instances d'événement. Les instances d'événement sont collectées sur les processus d'exécution de coordinations et d'activités. Nous trouverons notamment les informations sur les types d'événements de l'exécution de coordinations de services.

Tout journal d'exécution est un objet de la classe *ExecutionLog* défini par :

**class ExecutionLog** {
*LogName* String,
*EventLog* List(Event)}

## 3.5 Conclusions

Dans ce chapitre nous avons présenté les classes de notre modèle pour la description d'une coordination de services adaptative : service, méthode de service, coordination de service et activités. Nous avons introduit également les classes familles de méthodes et équivalences qui permettent de définir les méthodes de service équivalentes et les mappings entre leurs signatures pour s'adapter à leurs incompatibilités.

Pour fournir l'adaptabilité à l'exécution d'une coordination de services, les opérations d'adaptabilités sont proposées : substitution de méthodes et d'activités. La substitution de méthodes de service consiste à remplacer un appel à une méthode par un appel à une autre méthode équivalente. Cela permet de s'adapter aux fautes produites par l'exécution des méthodes de service extérieures sans arrêter l'exécution de la coordination de services.

La substitution d'activités replanifie la définition de la coordination en remplaçant l'activité échouée par une autre activité ou sous-composition d'activités équivalente. En dépit de la complexité de son processus d'exécution, cette opération permet de résoudre les fautes permanentes.

Le journal d'exécution stocke les événements permettant d'observer l'exécution d'une coordination de services. Ces événements permettent également de calculer les mesures de QoS qui sont présentées dans la section 5.4.

Le chapitre suivant présente le modèle de contrat d'adaptabilité qui fournit des concepts qui aident à spécifier explicitement les comportements d'adaptation et les mesures de QoS qui garantissent les performances de ces comportements.

# Chapitre 4

# Contrat d'adaptabilité

Dans le chapitre précédent, nous avons introduit notre modèle de coordination adaptative de services qui fournit les concepts et les opérations autorisant une adaptation pour l'exécution d'une coordination de services. Ce chapitre complète notre modèle en introduisant le concept de contrat d'adaptabilité (cf 4.1)) permettant de spécifier les opérations d'adaptation réagissant à un type donné d'exceptions et définissant les critères de QoS pour les méthodes de service.

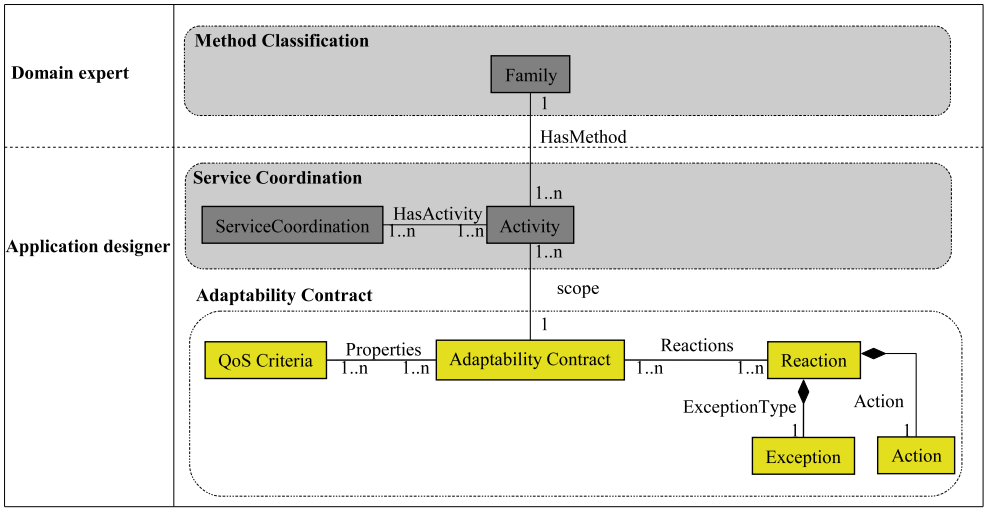


FIG. 4.1 – Modèle du contrat d'adaptabilité

Ce chapitre donne les réponses à ces questions au travers du concept de contrat d'adaptabilité. La figure 4.1 donne la vision complète de notre approche en intégrant ce concept de contrat. Les contrats prédéfinissent les comportements d'adaptation face aux situations spécifiées. Une situation est spécifiée par

- une exception et
- la QoS de la méthode de service qui sera appelée.

Un contrat d'adaptabilité est associé à une ou plusieurs activités d'une coordination

de services. Chaque activité d'une coordination de services est associée à un seul contrat d'adaptabilité. Lorsque l'exécution d'une activité produit une exception, son contrat d'adaptabilité associé sera évalué pour traiter cette exception.

Un contrat d'adaptabilité représente un "mode" d'adaptation lorsqu'une exception se produit lors de l'exécution d'une activité associée à un contrat. Dans un contrat d'adaptabilité, l'utilisateur (concepteur d'application) définit explicitement

- les critères de QoS désirés pour les méthodes de service appelées par les opérations d'adaptations et

- les réactions pour réagir aux exceptions spécifiées en réalisant une action d'adaptation prédéfinie. Une réaction se compose donc d'une exception et d'une action.

Pour illustrer l'utilisation de contrats d'adaptabilité, on reprend l'exemple de la coordination de recherche de vols (cf Section 3.1.1). La figure 4.2 illustre la coordination de recherche de vols qui est associée aux quatre contrats d'adaptabilité :

- "C1", "C2" et "C4" associés respectivement aux activités "GetTravelInformation", "SearchFligh" et "PresentFlightOptions", et

- "C3" associé aux activités "GetFlight1" et "GetFlight2".

Cette figure détaille le contrat "C3" qui sera évalué quand l'exécution de ces deux activités ("GetFlight1" et "GetFlight2") produit une exception spécifiée.

Dans le contrat "C3", l'exécution des méthodes de service appelées par les opérations d'adaptation doit respecter la QoS suivante :

- le temps d'exécution doit être inférieur ou égal à 1000 millisecondes,

- la fiabilité doit être supérieure à 0.5 et

- la disponibilité des méthodes de service doit être supérieure à 0.3.

Le concepteur d'application définit également de manière explicite dans ce contrat les réactions suivantes :

1. Si le délai d'une connexion *Timeout* expire alors l'action de ré-exécuter l'appel à la méthode spécifiée dans cette activité sera effectuée au maximum trois fois.

2. Si une exception produite par un appel à une méthode est détectée, alors l'action de substitution de méthodes de service sera exécutée.

3. Si l'action de réexécution échoue, alors l'action de substitution d'activités sera exé-

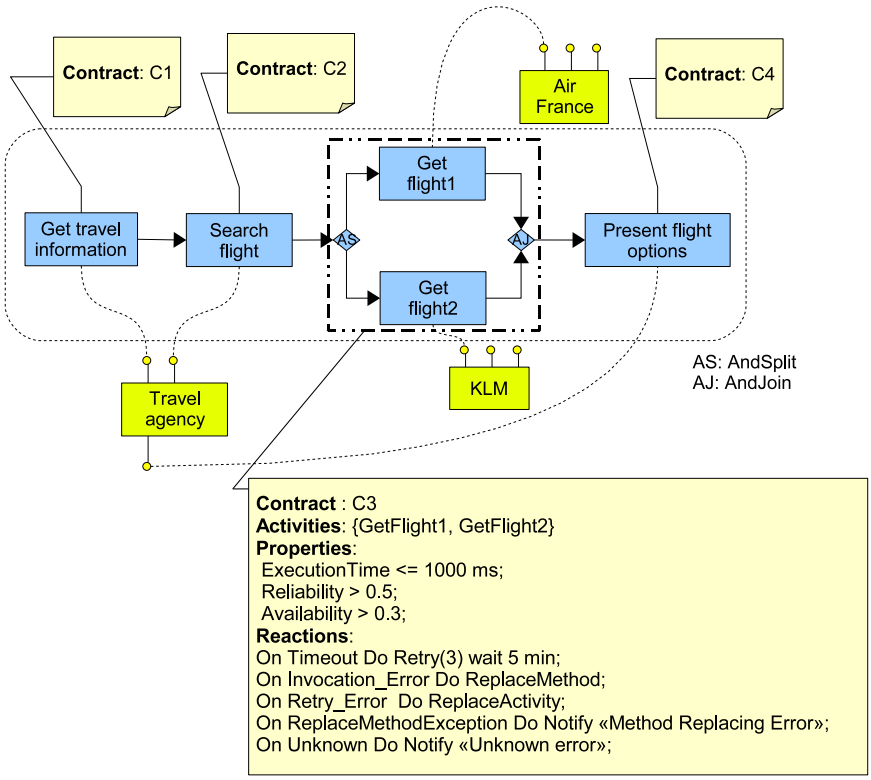


FIG. 4.2 – Le contrat d'adaptabilité "C3" associé aux activités "GetFlight1" et "GetFlight2"

cutée.

4. Si l'action substitution de méthodes de service échoue (*ReplaceMethodException*) se produit alors l'action de notification du message "Service Replacing Error" sera exécutée.

5. Si une exception est inconnue, alors l'action de notification du message "Unknown Error" sera exécutée.

La suite de ce chapitre présente la notion du contrat d'adaptabilité que nous proposons pour décrire les comportements d'adaptations pour une coordination de services. La section 4.1 introduit la classe *AdaptabilityContract* (contrat d'adaptabilité) qui représente un contrat d'adaptabilité. La section 4.2 présente les critères de QoS définis dans un contrat d'adaptabilité. La section 4.3 discute les actions d'adaptation proposées en réaction. La section 4.4 introduit un exemple d'utilisation de contrats pour la coordination de recherche de vols. Finalement, la section 4.5 conclut ce chapitre.

## 4.1 Classe AdaptabilityContract

Soit une instance *sc* de la classe *ServiceCoordination* composant un ensemble d'activités *A*. Un contrat d'adaptabilité d'une (plusieurs) activité(s) d'une coordination *sc* est une instance de la classe *AdaptabilityContract* définie par

**class AdaptabilityContract** {
*ContractName* String,
*Scope* Set(Activity),
*Properties* List(QoSCriterion),
*Reactions* List(Reaction)}

- *ContractName* est le nom du contrat,
- *scope* spécifie l'ensemble d'activités de la coordination associées à ce contrat. Pour *sc*, $scope \subseteq A$
- *Properties* désignent la liste de critères de QoS étant des instances de la classe *QoS-Criterion* que doivent respecter les action d'adaptation prédéfinies dans les réactions et
- (*Reactions*) désigne la liste de réactions représentées chacune par une règle de la forme `On Exception Do Action` (cf Section 4.3).

Dans l'exemple de recherche de vols présenté dans la section 3.2.2, la coordination de service *SF_Coordination_ID* se compose d'un ensemble d'activités A = {*GetTravelInformation_ID*, *SearchFlight_ID*, *GetFlight1_ID*, *GetFlight2_ID*, *PresentFlightOption_ID* }. Le contrat "C3" qui est associé aux activités *GetFlight1* et *GetFlight2* est défini par l'instance *C3Id* de la classe *AdaptabilityContract* comme suit :

*C3Id* **AdaptabilityContract** {
*ContractName* "C3",
*Scope* {GetFlight1, GetFlight2},
*Properties* {
[0] {C31QoS},
[1] {C32QoS},
[2] {C33QoS}
},
*Reactions* {
[0] {C31Retry},
[1] {C32ReplaceMethod},
[2] {C33ReplaceActivity},
[3] {C34Notification},
[4] {C35Notification}
}}

La figure 4.3 illustre l'instanciation du contrat "C3".

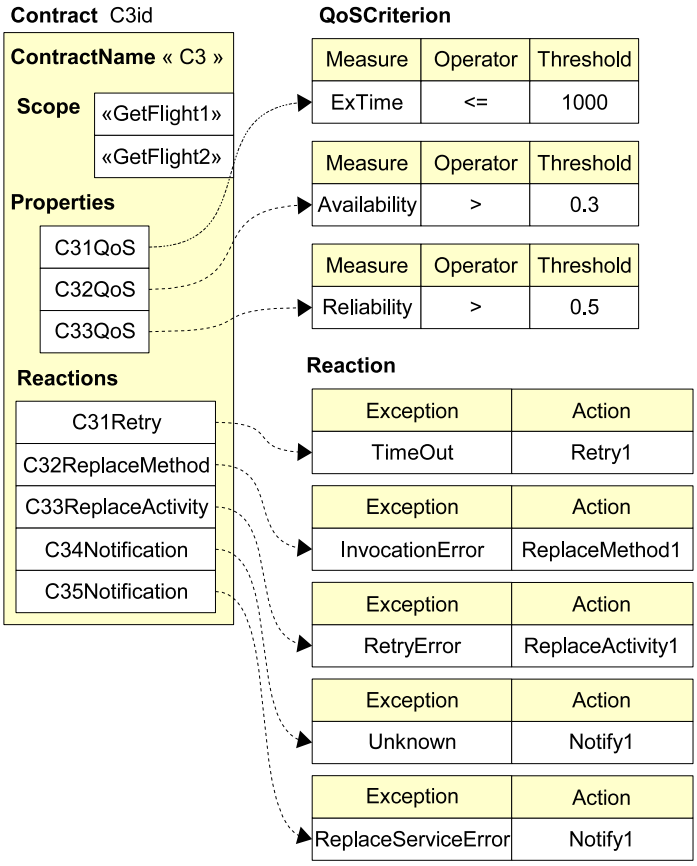


FIG. 4.3 – L'instanciation des instances du contrat "C3"

Les critères de QoS du contrat "C3" sont définis par les instances suivantes :

– *C31QoS* spécifie que le temps d'exécution des méthodes de service appelées doit être inférieur ou égal à 1000 millisecondes,

– *C32QoS* spécifie que la fiabilité des méthodes de service appelées doit être supérieure à 0,5 et

– *C33QoS* spécifie que la disponibilité des méthodes de service appelées doit être supérieure à 0,3.

Le contrat "C3" prédéfinit également les réactions qui sont définies par les instances suivantes :

1. *C31Retry* spécifie que si une exception *Timeout* est détectée alors l'action de réexécution d'appel à la méthode spécifiée dans cette activité sera effectuée au maximum

trois fois,

2. *C32ReplaceMethod* spécifie que si une exception *InvocatioError* est détectée alors l'action de substitution de méthodes de service sera exécutée,

3. *C33ReplaceActivity* spécifie que si l'action de ré-exécuter échoue, une exception *Retry_Exception* se produit alors l'action de substitution de méthodes de service sera exécutée,

4. *C34Notification* spécifie que si l'action substitution de méthodes de service échoue, une exception *ReplaceServiceException* se produit alors l'action de notification du message "Service Replacing Error" sera exécutée,

5. *C35Notification* spécifie que si une exception *Unknown* est détectée, alors l'action de notification du message "Unknown Error" sera exécutée.

## 4.2 Critère de QoS

Tout critère de QoS est caractérisé par un type de mesure de QoS, un opérateur de comparaison, un seuil désiré et l'unité de mesure de QoS (pour le temps d'exécution). C'est une instance de la classe *QoSCriterion* définie par

**class QoSCriterion** {
*MeasureType* QoSMeasure,
*Operator* ComparisonOperator,
*Threshold* Float,
*Unit* TimeUnit)

– *MeasureType* spécifie un type de mesure de QoS qui est une valeur du type *QoS-Measure* défini comme suit :

  *QoSMeasure* :- set("ExecutionTime", "Reliability", "Availability")

– *Operator* spécifie un opérateur de comparaison qui est une valeur du type *ComparisonOperator* défini comme suit :

  *ComparisonOperator* :- set("=", " !=", "<", "<=", ">", ">=")

– *Threshold* de type Float spécifie le seuil accepté de la mesure de QoS.

– *Unit* spécifie une unité du temps d'exécution qui est une valeur du type *TimeUnit* défini comme suit :

*TimeUnit* :- set("msec", "sec", "min", "hr", "").

Pour les mesures de la fiabilité et de la disponibilité, l'attribut *Unit* est vide.

Si l'on considère le contrat "C3" (voir la figure 4.3), les critères de QoS (*C31QoS*, *C32QoS* et *C33QoS*) de ce contrat sont définis respectivement par les instances de la classe *QoSCriterion* comme suit :

**C31QoS** *QoSCriterion* {
*QoSType* "ExecutionTime",
*ComparisonOperator* "<=",
*Threshold* 1000,
*Unit* "msec"}

**C32QoS** *QoSCriterion* {
*QoSType* "Reliability",
*ComparisonOperator* ">",
*Threshold* 0.5,
*Unit* ""}

**C33QoS** *QoSCriterion* {
*QoSType* "Availability",
*ComparisonOperator* ">",
*Threshold* 0.3,
*Unit* ""}

**Expression d'évaluation des critères de QoS d'un contrat :** L'expression d'évaluation des critères QoS d'un contrat est la conjonction des expressions de comparaison des critères de QoS.

Par exemple, l'expression d'évaluation de QoS du contrat "C3" est la conjonction des critères *C31QoS* et *C32QoS* et *C33QoS* :

(ExecutionTime <= 1000) And (Reliability > 0.5) And (Availability > 0.3)

Dans cette expression, le temps d'exécution est mesuré en milliseconde.

## 4.3 Réaction

Une réaction décrit le fait qu'une action d'adaptation prédéfinie est exécutée lorsqu'une exception spécifiée est détectée et lorsque les critères de QoS du contrat d'adaptabilité sont satisfaits. C'est une instance de la classe *Reaction* définie par

**class Reaction** {
*Exception* choice(PhysicsException,SemanticsException,Unknown, Timeout, Unavailable, InvocationError, ActionError),
*Action* choice(RetryAction, ReplaceMethodAction, NotifyAction, ReplannifyAction) }

- *Exception* spécifie qu'une exception d'une classe *PhysicsException*, *SemanticsException*, *Unknown*, *Timeout*, *Unavailable*, *InvocationError* ou *ActionError* déclenchera l'action prédéfinie.

- *Action* est une action d'adaptation d'une des quatre classes *RetryAction*, *ReplaceMethodAction*, *NotifyAction* ou *ReplannifyAction*.

Par exemple,dans le contrat "C3" (voir la figure 4.3), ses réactions sont définies par les instances de la classe *Reaction* suivantes :

1. *C31Retry* **Reaction** {
   *Exception* TimeOut,
   *Action* Retry1
   }

   définit l'instance *C31Retry* de la classe *Reaction* qui réagit aux exceptions de la classe *TimeOut* en exécutant l'action "réexécution" *Retry1* qui est une instance de la classe *RetryAction*.

2. *C32ReplaceMethod* **Reaction** {
   *Exception* InvocationError,
   *Action* ReplaceMethod1
   }

   définit l'instance *C32ReplaceMethod* de la classe *Reaction* qui réagit aux exceptions de la classe *RetryError* en exécutant une action de substitution de méthodes *ReplaceMethod1* qui est une instance de la classe *ReplaceMethodAction*.

3. *C33ReplaceActivity* **Reaction** {
   *ExceptionTypeName* "Retry Error",
   *Action* ReplaceActivity1

}

définit l'instance *C33ReplaceActivity* de la classe *Reaction* qui réagit aux exceptions de la classe *InvocationError* en ré-utilisant l'action de substitution d'activités *ReplaceActivity1* qui est une instance de la classe *ReplannifyAction*.

4. *C33Notification* **Reaction** {
   *ExceptionTypeName* "Replace service error",
   *Action* Notify1
   }

   définit l'instance *C32ReplaceMethod* de la classe *Reaction* qui réagit aux exceptions de la classe *ReplaceMethodError* en exécutant l'action de notification *Notify1* qui est une instance de la classe *NotifyAction*.

5. *C34Notification* **Reaction** {
   *ExceptionTypeName* "Unknown error",
   *Action* Notify2
   }

   définit l'instance *C32ReplaceMethod* de la classe *Reaction* qui réagit aux exceptions de la classe *UnknownError* en exécutant l'action de notification (*Notify2*) qui est une instance de la classe *NotifyAction*.

### 4.3.1 Exceptions

Notre modèle de contrat considère les trois sortes d'exceptions suivants (voir la figure 4.4) :

1. les exceptions physiques qui représentent les exceptions rapportées par les fautes du système, par exemple l'indisponibilité d'un serveur connectée (unvailable) ou l'expiration du délai d'attente (timeout) d'une connexion à un service.

2. les exceptions sémantiques qui représentent
   – les exceptions rapportées par les erreurs d'un appel à une méthode de service (invocationError) comme l'erreur du nom de la méthode ou l'erreur de nombre de paramètres de la méthode, ou par les fautes lors de l'exécution d'une action d'adaptation.

   – les exceptions produites lors de l'exécution des actions d'adaptation (ActionError) comme la réexécution (RetryError), la substitution de méthodes (ReplaceMethodError) et la substitution d'activités (ReplaceActivityError).

3. les exceptions inconnues qui représentent des exceptions non-spécifiées.

Cette taxonomie d'exceptions a pour but de déterminer une faute ponctuelle, intermittente ou permanente. Les exceptions sémantiques sont causées par les fautes permanentes tandis que les exceptions physiques sont souvent causées par les fautes ponctuelles ou intermittentes. Cette détermination est utile pour choisir une action d'adaptation convenable.

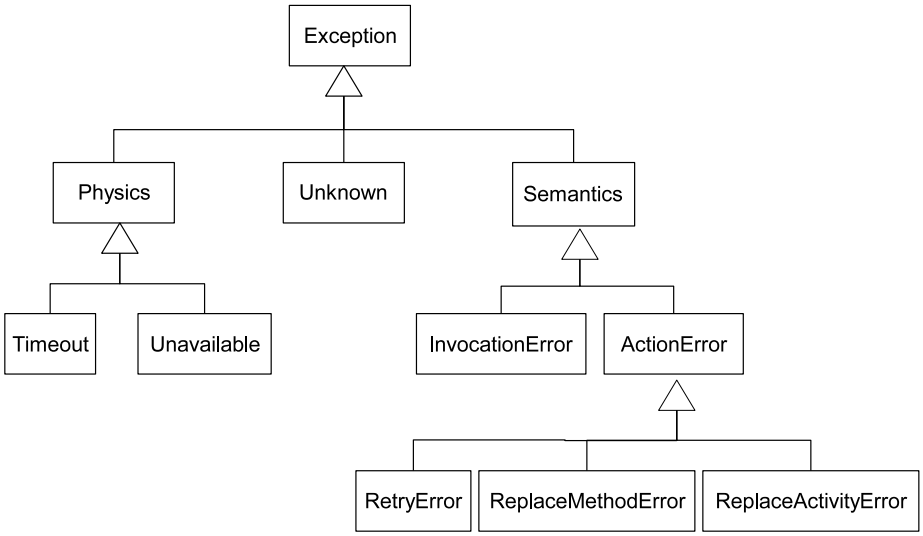


FIG. 4.4 – Hiérarchie de types d'exceptions

Cette classification d'exceptions est basée sur les causes qui les produisent lors d'un appel à une méthode de service ou lors de l'exécution d'une action d'adaptation. La classification d'exceptions est faite par le concepteur d'application. Le tableau 4.1 montre les classes d'exceptions que nous proposons, sa classe parente, les causes qui produisent ses exceptions, et les messages notifiés.

Toute exception est une instance de la classe *Exception* qui est caractérisée par son message de notification. La classe *Exception* est définie par

**class Exception** {
*Message* String }

Par exemple, une exception qui est une instance de la classe *InvocationError* est définie comme suit :

*MethodNameError_ID* **InvocationError** {
Message "Operation Error"
}

| Classe | classe parent | Causes | Messages notifiés |
|---|---|---|---|
| PhysicsException | Exception | | |
| Timeout | PhysicsException | l'expiration du délai d'une connexion à un serveur ou à un service | "Server too busy", "Unavailable Service", "Server timeout" |
| Unavailable | PhysicsException | L'indisponibilité du serveur ou du service | "Unavailable server" "Unavailable Service" |
| SemanticsException | Exception | | |
| InvocationError | SemanticsException | Erreur du nom de la méthode, ou des types des paramètres ou du nombre de paramètres | "Operation error", "Parameter type error", "Parameter number error" |
| ActionError | SemanticsException | Erreur par une action | |
| RetryError | ActionError | Erreur par RetryAction | "Retry exception" |
| ReplaceMethodError | ActionError | Erreur par ReplaceMethodAction | "Replacing method exception" |
| ReplaceActivityError | ActionError | Erreur par ReplannifyAction | "Replacing activity exception" |
| UnknownException | Exception | Exception non spécifiée | "Unknown error" |

TAB. 4.1 – Classes d'exceptions

### 4.3.2 Action d'adaptation

Les types d'actions d'adaptation illustrés par la figure 4.5 sont la réexécution (*RetryAction*), la notification (*NotifyAction*), la substitution de méthodes (*ReplaceMethodAction*) et la replanification (*ReplanifyAction*). La réexécution, la notification sont les types d'action d'adaptation qui ne modifient pas la définition de l'activité tandis que la substitution de méthodes et la replanification sont les types d'action qui modifient respectivement l'appel à une méthode de service et la définition de la coordination. La substitution de méthodes utilise l'opération de substitution présentée dans la section 3.3.1 du chapitre précédent. La replanification de la définition de la coordination remplace l'activité échouée par une autre activité ou une sous composition d'activités. Ce type d'action utilise les opérations présentées dans la section 3.3.2.

La classe *Action* est une classe abstraite.

```
abstract class Action {
        }
```

Ces quatre sous-classes *RetryAction*, *NotifyAction*, *ReplaceMethodAction* et *ReplannifyAction* implantent les quatre types d'actions d'adaptation.

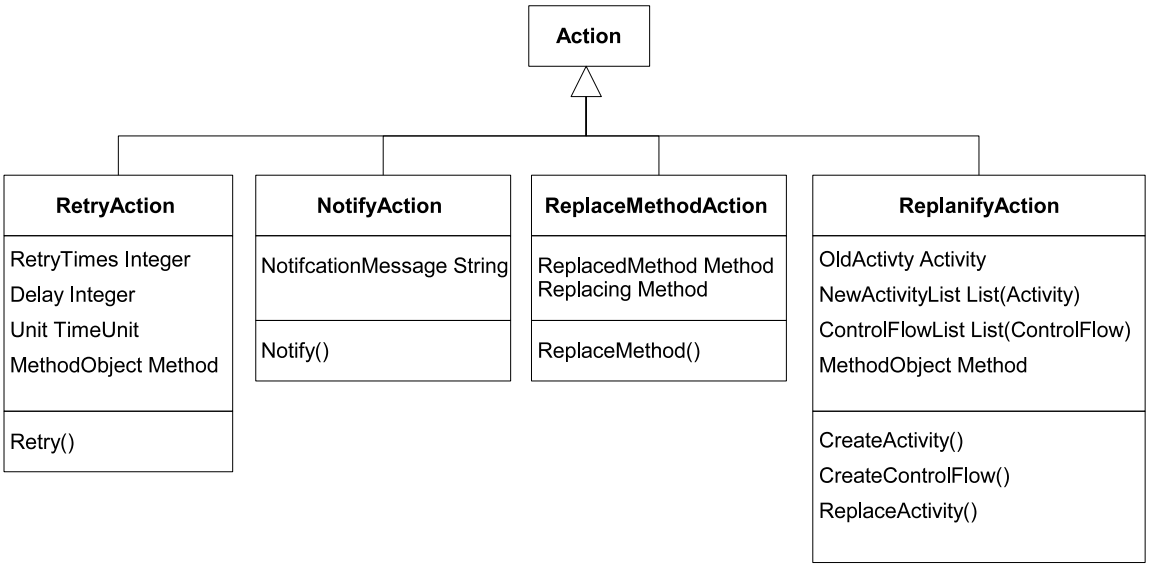


FIG. 4.5 – Hiérarchie de types d'action d'adaptation

#### 4.3.2.1 Classe RetryAction

L'action *RetryAction* réalise la réexécution de l'appel à la méthode de l'activité échouée avec un nombre limité de fois. Entre chaque appel, un délai d'attente peut être spécifié.

Toute action de réexécution est une instance de la classe *RetryAction* qui est une sous-classe de la classe *Action* et qui est définie par

```
class RetryAction : Action {
        RetryTimes Integer,
        Delay Integer,
        Unit TimeUnit,
        MethodObject Method,
        Retry()(String)
         }
```

– *RetryTimes* spécifie le nombre maximum de la réexécution,

– *Delay* spécifie le délai d'attente entre chaque réexécution,

– *TimeUnit* spécifie l'unité de temps,

– *MethodObject* spécifie l'instance de la méthode de service qui sera exécutée,

– *Retry*()(String) est l'opération de réexécution de l'appel de la méthode de l'activité échouée. Elle retourne un message notifiant son état d'exécution.

Les attributs d'une instance de la classe *RetryAction* sont spécifiés par l'utilisateur. L'attribut *MethodObject* peut être spécifié automatiquement en réutilisant la méthode de service de l'activité échouée.

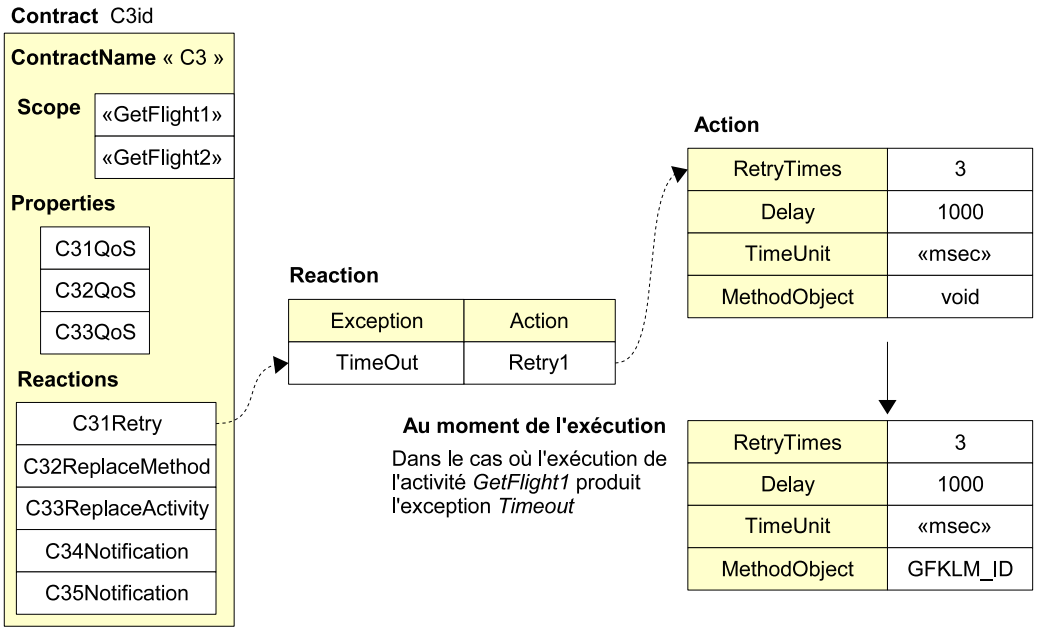


FIG. 4.6 – L'instanciation de l'action de réexécution *Retry1*

Par exemple, dans le contrat "C3", la réaction *C31Retry* spécifie que l'action *Retry1* est exécutée lorsque une exception de type *Timeout* est détectée (voir la figure 4.6). L'action *Retry1* est définie par l'utilisateur comme suit :

*Retry1* **RetryAction** {
*RetryTimes* 3,
*Delay* 1000,
*TimeUnit* "msec"
*MethodObject* Void
}

L'action de réexécution *Retry1* qui spécifie le nombre maximum de réexécution (égal à trois) avec un délai de 1000 millisecondes. Au moment de l'exécution, l'attribut *MethodObject* aura comme valeur la méthode *GFKLM* ou *GFAirFrance* spécifiée dans l'activité *GetFlight1* ou *GetFlight2* (voir la figure 4.6). La réexécution consiste à appeler cette méthode.

#### 4.3.2.2 Classe NotifyAction

Toute action de notification est une instance de la classe *NotifyAction* qui est une sous-classe de la classe *Action* et est définie par

**class NotifyAction** : Action {
*NotificationMessage* String,
*Notify*()(String)
}

– *NotificationMessage* est le message de notification spécifié,
– *Notify*()(String) est l'opération de notifier le message spécifié par l'attribut *NotificationMessage*.

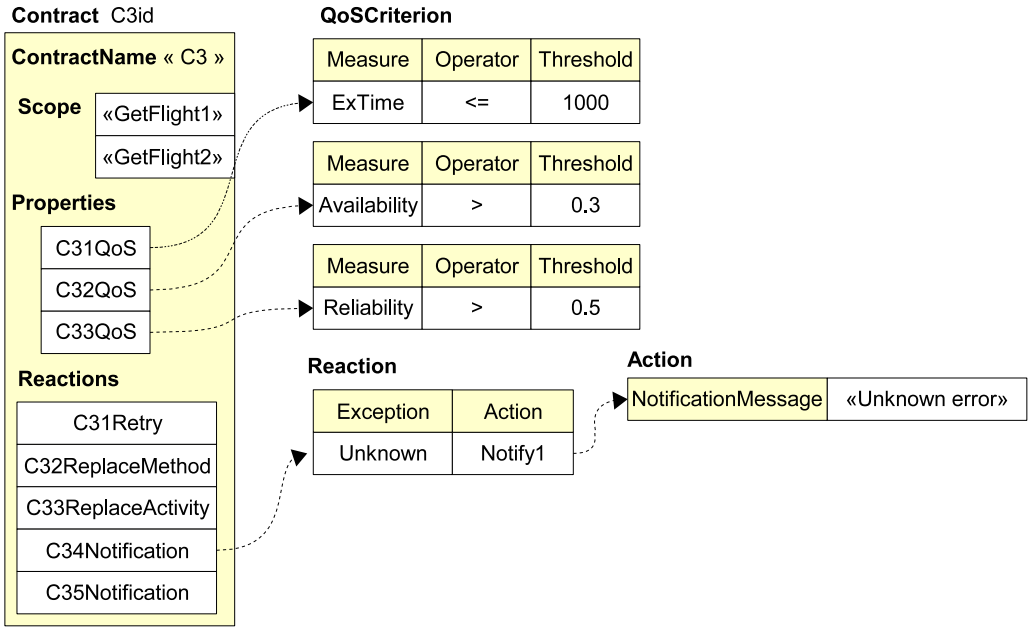


FIG. 4.7 – L'instanciation de l'action de notification Notify1

Par exemple, la figure 4.7 illustre l'instanciation de l'action *Notify1* de la réaction *C34Notification* dans le contrat "C3". L'instance *Notify1* de la classe *NotifyAction* qui notifie le message "Unknown error" est défini de la manière suivante :

*Notify1* **NotifyAction** {
*NotificationMessage* "Unknown error"
}

#### 4.3.2.3 Classe ReplaceMethodAction

Toute action de substitution de méthodes de service est une instance de la classe *ReplaceMethodAction* qui est une sous-classe de la classe *Action* et est définie par

```
class ReplaceMethodAction : Action {
        ReplacedMethod Method,
        ReplacingMethod choice(Method, FamilyComposition),
         }
```

– *ReplacedMethod* spécifie l'instance de la méthode remplacée qui est normalement la méthode de l'activité échouée et associée au contrat,

– *replacingMethod* spécifie l'instance de la méthode remplaçante qui doit être équivalente à la méthode remplacée ou l'instance d'une composition de familles. Cette instance peut être déclarée explicitement par l'utilisateur ou cherchée automatiquement en se basant sur les méthodes équivalentes à la méthode *ReplacedMethod*,

Par exemple, dans la réaction *C32ReplaceMethod* du contrat "C3" (voir la figure 4.3), l'action *ReplaceMethod1* qui est une instance de la classe *ReplaceMethodAction* peut être définie statiquement comme suit :

```
ReplaceMethod1 ReplaceMethodAction {
    ReplacedMethod GFKLM_ID,
    ReplacingMethod GFOpodo_ID
     }
```

L'action *ReplaceMethod1* prédéfinit la substitution de la méthode *GFKLM* par la méthode *GFOpodo*. La vérification de l'équivalence des deux méthodes est réalisée à la phase de conception.

D'une autre manière, l'utilisateur peut également définir l'instance *ReplaceMethod1* sans spécifier explicitement les méthodes remplacée et remplaçante comme suit :

```
ReplaceMethod1 ReplaceMethodAction {
    ReplacedMethod Void,
    ReplacingMethod Void
     }
```

Au moment de l'exécution, les attributs de l'instance *ReplaceMethod1* sont affectés dynamiquement (voir la figure 4.8). Dans ce cas, la valeur de l'attribut *ReplacedMethod* est le nom d'étiquette de l'instance de la méthode de l'activité associée au contrat d'adaptabilité activé et la valeur de l'attribut *ReplacingMethod* est le nom d'étiquette d'une méthode équivalente et la plus appropriée par rapport aux critères de QoS trouvés en cherchant dans les familles de méthodes équivalentes à l'instance de la famille de la méthode donnée à l'attribut *ReplacedMethod*.

Les valeurs des attributs de l'instance *ReplaceMethod1* peuvent être initialisées comme suit :

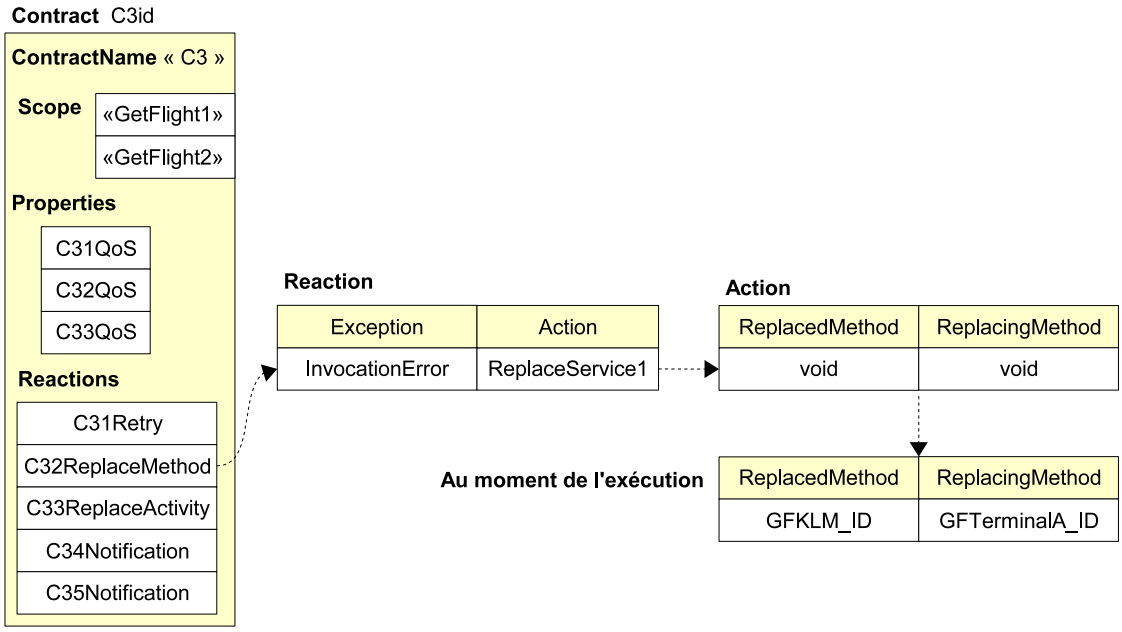


FIG. 4.8 – L'initialisation de l'instance *ReplaceMethod1*

*ReplaceMethod1* **ReplaceMethodAction** {
*ReplacedMethod* GFKLM_ID,
*ReplacingMethod* GFTerminalA_ID
}

Quand l'initialisation de l'instance *ReplaceMethod1* est finie, la substitution de la méthode *GFKLM* par *GFTerminalA* sera réalisée par l'opération *ReplaceMethod*().

#### 4.3.2.4 Classe ReplanifyAction

Toute action de replanification qui substitue une activité par une autre ou par une sous-composition est une instance (objet) de la classe *ReplanifyAction* (une sous-classe de la classe abstraite *Action*) définie par

**class ReplanifyAction** : Action {
*OldActivity* Activity,
*NewActivityList* List(Activity),
*ControlFlowList* List(ControlFlow),
*CreateActivity*() (Activity),
*CreateControlFlow*()(ControlFlow)
*ReplaceActivity*()(String)}

– *OldActivity* spécifie l'instance de la classe *Activity* de l'activité associé au contrat,

– *NewActivityList* spécifie une liste d'instances de la classe *Activity* qui participe au processus de substitution de l'activité *OldActivity*,

– *ControlFlowList* spécifie une liste d'instances de la classe *ControlFlow* qui forme un flôt de contrôle utilisé pour remplacer l'activité *OldActivity*,

– *CreateActivity*()(Activity) est l'opération qui crée les nouvelles instances *Activity*,

– *CreateControlFlow*()(ControlFlow) est l'opération qui crée un flot de contrôle,

– *ReplaceActivity*()(String) spécifie l'opération de substitution d'activités. Elle substitue l'activité *OldActivity* par l'activité unique dans **NewActivity** ou par les flots de contrôle définies dans *ControlFlowList*. Elle retourne un message notifiant son état d'exécution.

Les valeurs des attributs d'une instance de la classe *ReplanifyAction* peuvent être données automatiquement sans intervention de l'utilisateur. La valeur de l'attribut *OldActivity* est trouvée facilement car elle est l'instance de l'activité échouée et associée au contrat d'adaptabilité traité. Les nouvelles instances d'activité sont définies en se basant sur la procédure de substitution d'activités présentée dans la section 3.3.2. Le flot de contrôle est normalement une séquence des activités dans *NewActivityList* présentée aussi dans la section 3.3.2.

Par exemple, dans la réaction *C33ReplaceActivity* du contrat *C3* (voir la figure 4.3), l'action *ReplaceActivity1* qui est une instance de la classe *ReplannifyAction* peut être définie statiquement comme suit :

*ReplaceActivity1* **ReplanifyAction** {
    *OldActivity* Void,
    *NewActivityList* {}
    *ControlFlowList* {}
        }

Au moment de la création de l'action *ReplaceActivity1*, l'attribut *OldActivity* ne référence aucune activité (Void), les listes *NewActivityList* et *ControlFlowList* sont vides. Ces attributs recevront une valeur automatiquement au moment de l'exécution.

Au moment de l'exécution, les valeurs des attributs de l'instance *ReplaceActivity1* sont données dynamiquement (voir la figure 4.9). La valeur de l'attribut *OldActivity* est le nom d'étiquette de l'activité associée au contrat d'adaptabilité (*GetFlight1_ID)*et la liste *NewActivityList* contient le nom d'étiquette de l'activité remplaçante *GetFlight11_ID*. La liste *ControlFlowList* est vide. Concrètement, les valeurs des attributs de l'instance *ReplaceActivity1* peuvent être initialisées comme suit :

*ReplaceMethod1* **ReplaceMethodAction** {
    *OldActivity* GetFlight1_ID,

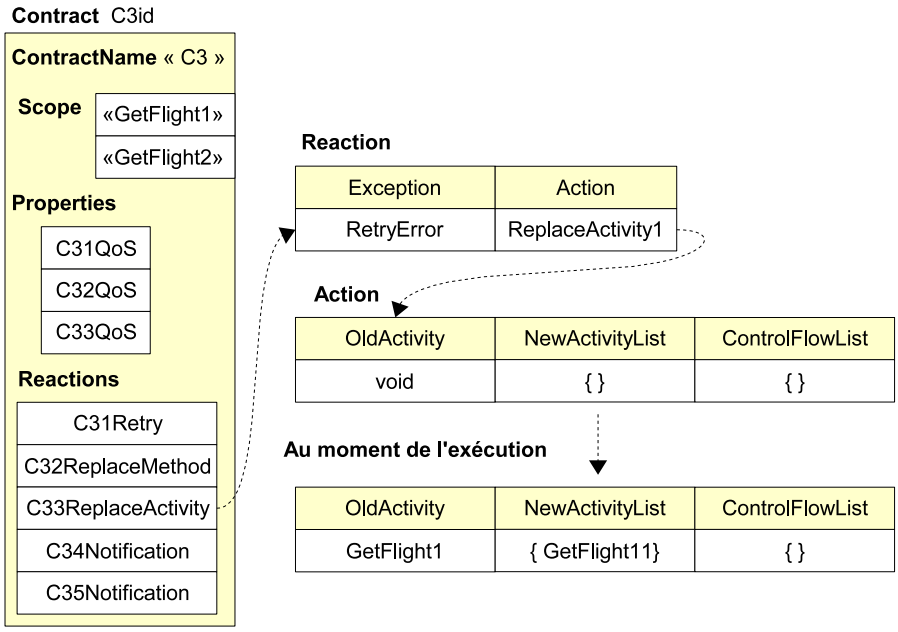


FIG. 4.9 – L'initialisation de l'objet *ReplaceActivity1* au moment de l'exécution

*NewActivityList* {[0] GetFlight11_ID}
*ControlFlowList* { }
}

L'activité remplaçante *GetFlight11* est créée automatiquement en se basant sur la méthode équivalente, par exemple *GFTerminalA* qui remplace la méthode *GFKLM* utilisée dans l'activité *GetFlight1*.

Quand l'initialisation de l'instance *ReplaceActivity1* est finie, la substitution de l'activité *GetFlight1* par *GetFlight11* sera réalisée par l'opération *ReplaceActivity*().

Notons que la substitution d'une activité peut être réalisée par une composition d'activités.

Pour illustrer ce cas, reprenons l'exemple 3.1 dans la section 3.3.2 présentant la substitution d'une activité par une sous-composition de trois activités ordonnées successivement. Dans cet exemple, l'activité A est remplacée par par une sous-composition des trois activités *InAdapter*, *B*, *OutAdapter*. L'activité B appelle une méthode de service ayant la même fonctionnalité de celle appelée par l'activité A. Les activités *InAdapter* et *OutAdapter* sont utilisées pour adapter aux incompabilités entre les entrées et sorties des deux méthodes.

L'instance *ReplaceFDEbySDE* de la classe *ReplaceActivity* qui réalise cette substitution est définie par l'utilisateur comme suit :

*ReplaceAbyB* **ReplanifyAction** {
*OldActivity* Void,
*NewActivityList* {}

*ControlFlowList* {}
}}

Au moment de l'exécution, les activités *InAdaptator*, *B*, *OutAdaptator* seront créées dynamiquement par l'opération CreateActivity()() et ces identificateurs sont insérées dans l'attribut *NewActivityList*, les deux flots de contrôle *Sequence* sont créés par l'opération *CreateControlFlow*() et insérés dans l'attribut *ControlFlowList*. Les valeurs des attributs de l'instance *ReplaceAbyB* seront initialisées comme suit :

*ReplaceAbyB* **ReplanifyAction** {
*OldActivity* A,
*NewActivityList* {
[0] {InAdaptor},
[1] {B},
[2] {OutAdaptor)},
*ControlFlowList* {
[0] {*Operator* {OperatorName "Sequence", Condition True},
*InputActivity* {InAdapter},
*OuputActivity* {B}},
[1] {*Operator* {OperatorName "Sequence", Condition True},
*InputActivity* {A2},
*OuputActivity*{OutAdapter}} }}

Lorsque l'initialisation de l'instance *ReplaceAbyB* est finie, l'opération *ReplaceActivity*() sera réalisée pour remplacer l'activité *A* par cette sous-composition dans la définition de la coordination correspondante.

## 4.4 Exemple de l'instanciation d'un contrat d'adaptabilité

Cette section montre l'instanciation du contrat "C1" de la coordination de recherche de vols. Le détail de ce contrat est illustré par la figure 4.10. Il est associé à l'activité *GettravelInformation*. Ce contrat précise que les critères de QoS associée à la méthode appelée par l'activité *GettravelInformation* associée ou ses méthodes équivalentes doivent respecter sont un temps d'exécution inférieur à 3 secondes, une fiabilité et une disponibilité supérieures à 0.7. Il spécifie également deux réactions :

1. Si une exception *Timeout* est détectée alors l'action de réexécution de l'appel à la méthode spécifiée par l'activité "*GettravelInformation*" sera exécutée au maximum trois fois avec un délai de 30 secondes entre chaque fois d'essai .
2. Si l'action de réexécution échoue, l'action de notification sera exécutée en notifiant le message "Retry Error".
3. Si une exception *Unknown* est détectée, alors l'action de notification sera exécutée en notifiant le message "Unknown exception".

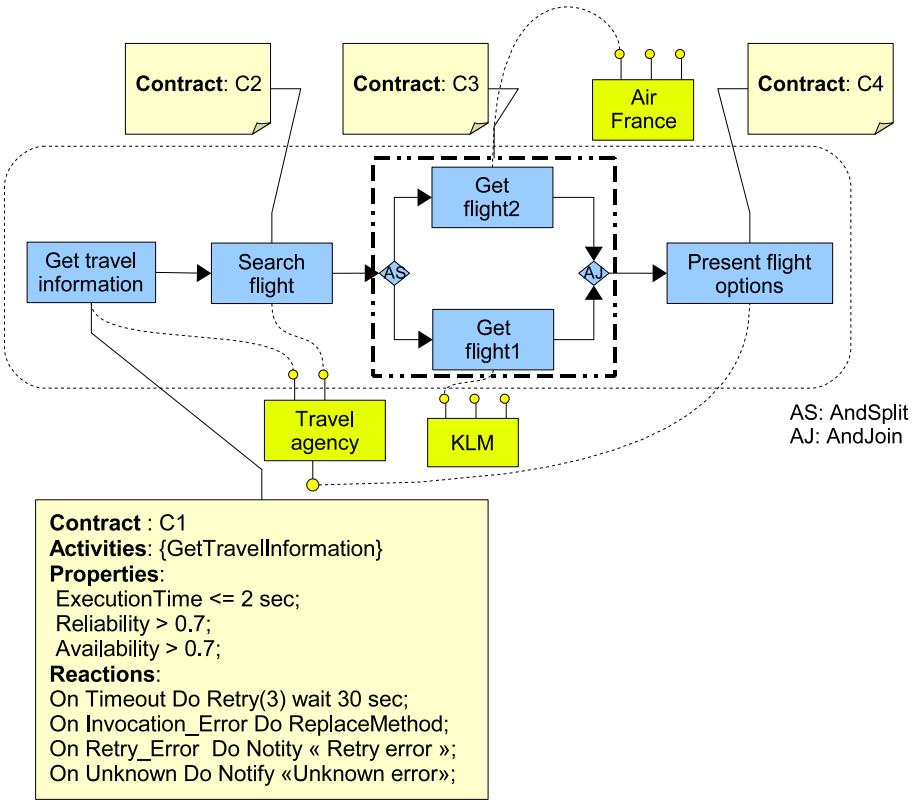


FIG. 4.10 – Contrat C1 de la coordination de recherche de vols

Le contrat "C1" est défini comme suit :

**C1Id** *AdaptabilityContract* {
*ContractName* "C1",
*Scope* {GetTravelInformation},
*Properties* {
[0] {C11QoS}
[1] {C12QoS}
[2] {C13QoS}
},
*Reactions* {
[0] {C11Reaction}
[1] {C11Reaction}
[1] {C13Reaction}
}}

La figure 4.11 illustre une vue globale des instances correpondantes au contrat "C1".

**Instanciation des propriétés de QoS du contrat "C1" :**

le contrat "C1" spécifie une liste de trois propriétés de QoS (*C11QoS*, *C12QoS et C13QoS*) qui sont définies comme suit :

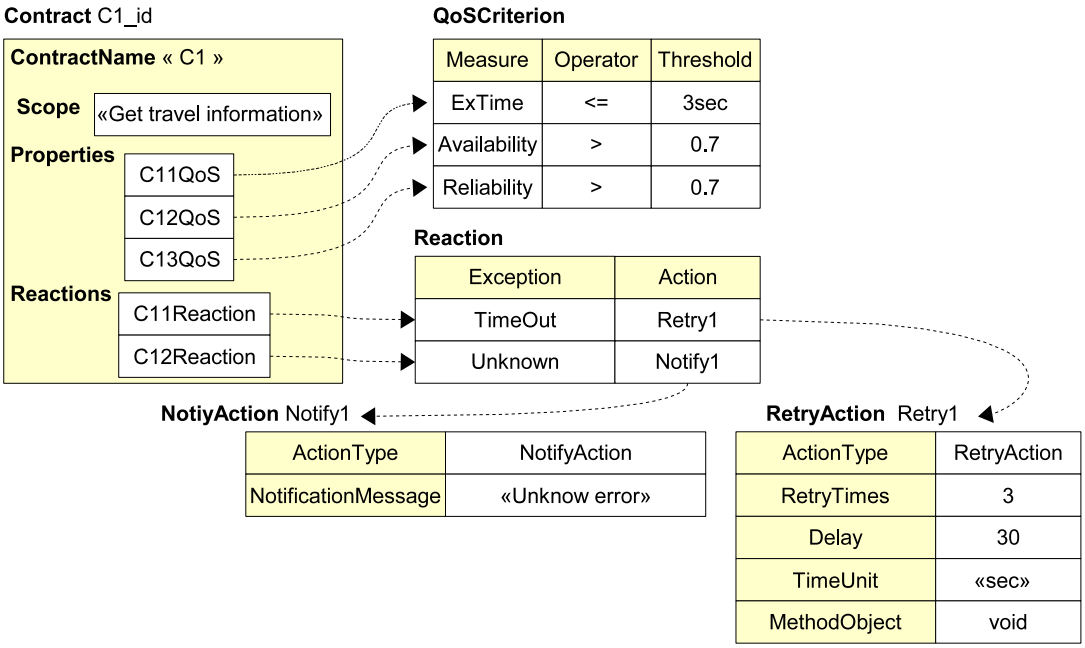


FIG. 4.11 – L'instanciation des instances du contrat "C1"

*C11QoS* **QoSCriterion** {
*QoSType* "ExecutionTime",
*ComparisonOperator* "<=",
*Threshold* 3,
*Unit* "sec"}
}

*C12QoS* **QoSCriterion** {
*QoSType* "Reliability",
*ComparisonOperator* ">",
*Threshold* 0.7,
*Unit* ""}
}

*C13QoS* **QoSCriterion** {
*QoSType* "Availability",
*ComparisonOperator* ">",
*Threshold* 0.7,
*Unit* ""} }

**Instanciation des réactions du contrat "C1" :**

Les instances de la classe *Reaction* définies dans le contrat "C1" sont définies comme suit :

L'instance *C11Reaction* de la classe *Reaction* est définie par

*C11Reaction* **Reaction** {
*Exception* TimeOut,
*Action* Retry1
}.

La réaction *C11Reaction* spécifie que l'action "réexécution" *Retry1* est exécutée lorsque l'exception *TimeOut* se produit.

L'instance *Retry1* de la classe *RetryAction* est définie par

*Retry1* **RetryAction** {
*ActionType* "RetryAction",
*RetryTimes* 3,
*Delay* 30,
*TimeUnit* "sec",
*MethodObject* Void,
}

L'instance *C12Reaction* de la classe *Reaction* est définie par

*C12Reaction* **Reaction** {
*Exception* RetryError,
*Action* Notify1
}.

La réaction *C12Reaction* spécifie que l'action *Notify1* est exécutée lorsque l'exception *RetryError* se produit.

L'instance *Notify1* de la classe *NotifyAction* est définie par

*Notify1* **NotifyAction** {
*NotificationMessage* "Retry Error"
}

L'instance *C13Reaction* de la classe *Reaction* est définie par

*C13Reaction* **Reaction** {
*Exception* Unknown,
*ActionType* Notify2
}.

La réaction *C13Reaction* spécifie que l'action "notification" *Notify1* est exécutée lorsque l'exception *Unknown* se produit.

L'instance *Notify2* de la classe *NotifyAction* est définie par

*Notify2* **NotifyAction** {
*ActionType* "NotifyAction",
*NotificationMessage* "Unknown error"
}

## 4.5 Conclusion

L'approche de contrats dans la programmation orientée objets a été introduite en 1990 par Helm [HHG90]. Les contrats visaient alors à compenser le manque d'outils pour exprimer des relations entre les objets. Ils ont été employés pour spécifier les compositions comportementales. La conception par contrat est une méthode de développement de logiciel [Mey92]. L'idée principale derrière cette méthode est que la liaison entre une classe d'objets fournie par le serveur et des clients est définie par un contrat.

Inspiré par cette approche de contrat, nous avons défini le modèle de contrat d'adaptabilité pour une coordination de services en proposant les concepts : contrat d'adaptabilité, propriété de QoS, réaction. Une classification d'exceptions et des types d'actions d'adaptation sont également définis pour spécifier des réactions sous forme événement-action.

Un contrat d'adaptabilité est utile pour définir un mode d'adaptation lorsque l'exécution d'une coordination de services rencontre une faute. Les intérêts des contrats d'adaptabilité pour l'exécution d'une coordination de services adaptative sont

- la flexibilité, car un contrat est indépendant de la définition de la coordination de services et il peut être modifié lors de l'exécution de la coordination et
- la dynamicité, car il permet de s'adapter dynamiquement aux exceptions par des réactions prédéfinies.

# Chapitre 5

# Evaluation de contrats d'adaptabilité

Ce chapitre décrit l'évaluation des contrats d'adaptabilité d'une coordination de services. Afin d'évaluer un contrat, on doit répondre aux questions suivantes :

- A quel moment d'évaluer un contrat d'adaptabilité ?
- Comment synchroniser l'exécution d'une action d'adaptation avec celle de la coordination ?
- Comment évaluer les critères de QoS de méthodes de service lorsqu'une exception provenant d'un service se produit ?

Ce chapitre propose le processus d'évaluation de contrats et présente notre moteur d'évaluation de contrats d'adaptabilité nommé SEBAS. Il introduit d'abord l'architecture de SEBAS 5.1. Ensuite, la section 5.2 présente le processus d'évaluation d'un contrat d'adaptabilité qui est la phase de l'adaptation d'une activité. L'exécution d'une réaction définie dans un contrat est présentée dans la section 5.3. Par ailleurs, les calculs des mesures de QoS de méthodes de service sont décrits dans la section 5.4. La section 5.5 présente l'implantation de notre prototype SEBAS. Enfin, la section 5.6 conclut ce chapitre.

## 5.1 Architecture générale

La figure 5.1 illustre l'architecture du moteur d'évaluation de contrats d'adaptabilité (appelé SEBAS). SEBAS se compose de deux principaux composants : le gestionnaire d'événements (*EventService*) et l'exécuteur de contrats (*ContractService*). L'exécuteur de contrats est associé aux composants qui soutiennent son fonctionnement : le gestionnaire de contrats (*ContractManager*), l'analyseur de QoS (*QoSAnalyser*), le détecteur de types d'exceptions (*ExceptionTypeDetector*), la recherche de méthodes de service équivalentes (*EquivalentMethodLookup*), et l'exécution d'adaptation (*AdaptationExecutor*). *SEBAS* dispose un journal d'événement (*EventLog*) qui stocke des événements d'exécution de coordinations, une base de QoS de méthodes de service, une base de contrats d'adaptabilité et des ontologies d'adaptation de services et d'exceptions.

Le moteur de workflow a pour entrée une instance de workflow qui est une instance d'une coordination de services. Chaque activité dans la coordination est associée à un contrat d'adaptabilité qui est prédéfini et stocké dans une base de contrats. SEBAS fournit l'adaptation pour l'exécution d'une coordination de services en se basant sur les contrats stockés dans cette base.

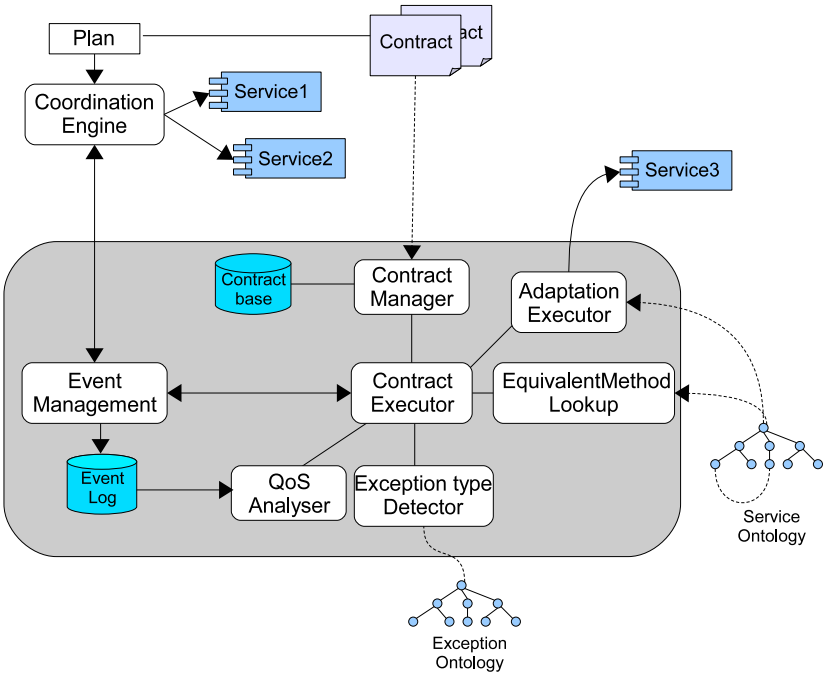


FIG. 5.1 – Architecture de SEBAS

### 5.1.1 Gestionnaire d'événements

Le gestionnaire d'événements détecte des événements produits par le moteur d'exécution de coordination et par l'exécuteur de contrats (*ContractExecutor*) et les notifie à l'exécuteur de contrats. Il interagit avec le moteur de coordination pour intervenir à l'exécution d'une coordination.

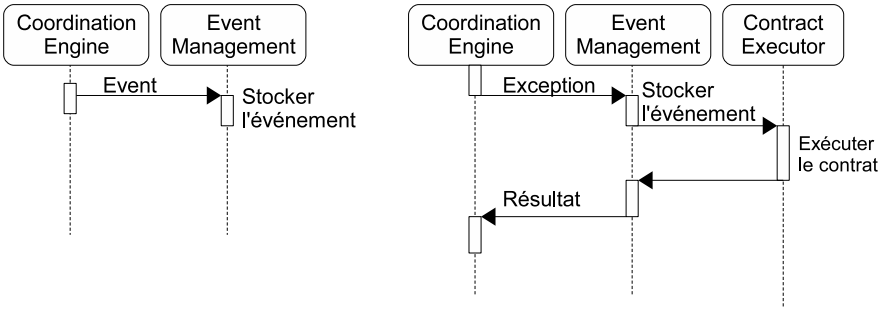


FIG. 5.2 – Interactions entre le moteur de coordination et le gestionnaire d'événements

La figure 5.2 illustre l'interaction entre le moteur de coordination et le gestionnaire d'événements. Quand le moteur envoie un événement au gestionnaire d'événements en

mode asynchrone, le gestionnaire stocke cet événement dans son journal d'exécution. Dans le cas où l'événement est une exception, le gestionnaire stocke cette exception et ensuite l'envoie à l'exécuteur de contrats *ContractExecutor*. Le *contractExecutor* traite cette exception en évaluant le contrat correspondant. En fin, le résultat de l'exécution du contrat sera retourné par le *ContractExecutor* au gestionnaire d'événements qui le renvoie au moteur de coordination.

### 5.1.2 Exécuteur de contrats

Le composant *ContractExecutor* exécute des contrats d'adaptabilité en collaborant avec ses composants partenaires. le processus d'évaluation d'un contrat d'adaptabilité qui se compose de deux phases : déclenchement et exécution. L'évaluation d'un contrat est déclenchée dès qu'il reçoit une notification d'exception envoyée par le gestionnaire d'événements.

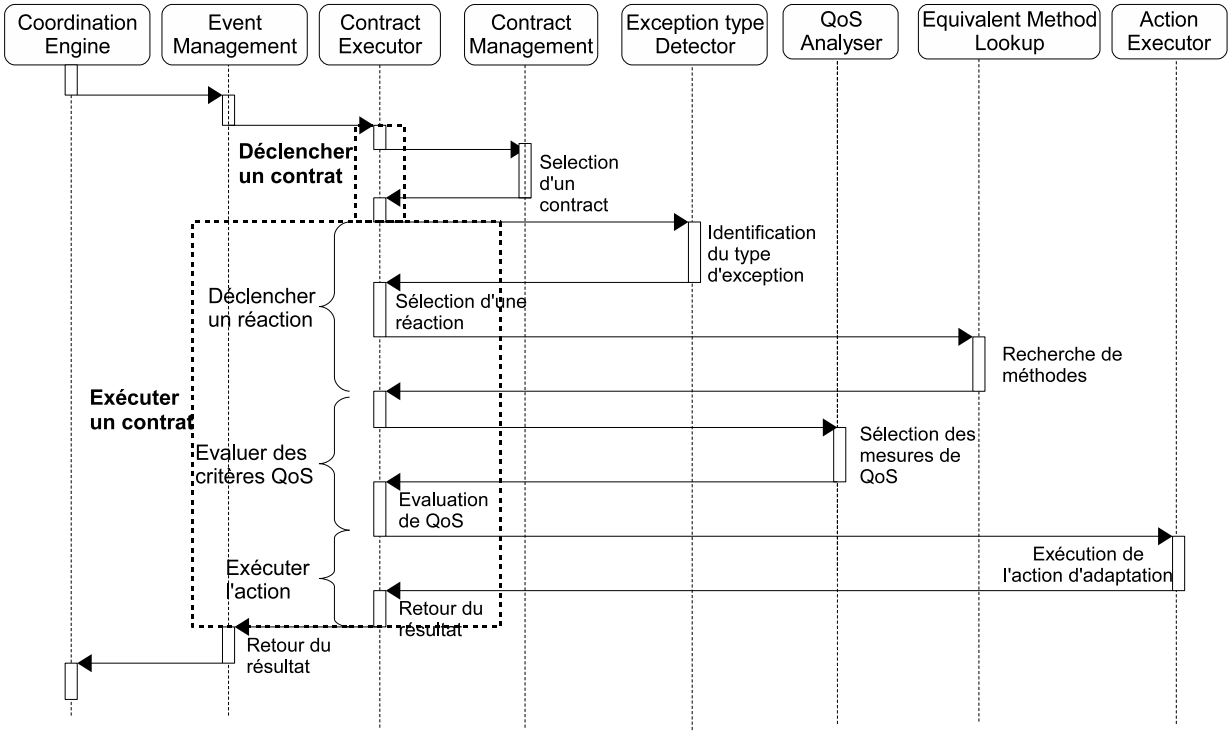


FIG. 5.3 – Interactions entre le ContractExecutor et ses partenaires

La figure 5.3 illustre les interactions entre le *ContractExecutor* et ses composants partenaires pour réaliser le processus d'évaluation d'un contrat d'adaptabilité.

A la réception d'une notification d'exception du gestionnaire d'événements, le *ContractExecutor* déclenche le contrat obtenu en interrogeant le gestionnaire de contrats. Ensuite il déclenche la phase d'exécution de ce contrat. La phase d'exécution d'un contrat consiste à exécuter une ou plusieurs réactions. Le processus d'exécution d'une réaction se compose de trois phases :

- ***Déclenchement*** : dans cette phase, le *ContractExecutor* interroge le *EquivalentMethodLookup* pour chercher les méthodes équivalentes.

- ***Evaluation*** : dans cette phase, le *ContractExecutor* évalue les mesures de QoS des méthodes de service selon les critères de QoS du contrat. Pour avoir les mesures de QoS d'une méthode de service, le *ContractExecutor* interrogent le *QoSAnalyser*.

- ***Exécution*** : dans cette phase, le *ContractExecutor* appelle l'*AdaptationExecutor* pour réaliser l'action d'adaptation. Lorsqu'il reçoit la réponse envoyée par l'*AdaptationExecutor*, il répond au moteur de coordination de services.

**Gestionnaire de contrats d'adaptabilité** est responsable de la gestion des contrats d'adaptabilité de coordinations. Il stocke des contrats de coordinations dans une base de contrats. Il répond à la requête de sélection d'un contrat demandé par le composant *ContractExecutor* en retournant l'instance du contrat d'adaptabilité.

**Analyseur de QoS** est responsable de :

- Sélectionner les mesures de QoS d'une méthode de service : comme les mesures de QoS sont stockées dans une base de données, l'analyseur effectue une requête sur la base de mesures de QoS pour sélectionner les mesures de QoS d'une méthode de service.
- Calculer les mesures de QoS : les calculs des mesures de QoS de méthodes de service sont présentés dans la section 5.4.
- Mettre à jour des mesures de QoS : la procédure de mis à jour des mesures de QoS est effectuée selon la stratégie de mis à jour choisie comme périodique ou instant d'événement présentée dans la section 5.4.

**Détecteur d'exceptions** est responsable d'identifier si une exception est une instance d'une classe d'exceptions spécifiée. Nous avons défini une ontologie d'exceptions présentée dans l'annexe C

**Recherche de méthodes équivalentes** Le composant *EquivalentMethodLookup* est responsable de chercher des méthodes équivalentes à une méthode demandée. La recherche de méthodes est basée sur l'ontologie de familles de méthodes présentée dans l'annexe C. Le composant *EquivalentMethodLookup* retourne une liste de méthodes et de coordinations équivalentes en terme de fonctionnalité à la méthode demandée.

**AdaptationExecutor** Le composant *AdaptationExecutor* est responsable d'exécuter des actions d'adaptations comme la réexécution, la notification, la substitution de méthodes de service et la substitution d'activités. Ce composant fournit également le service *Adaptor* qui permet l'adaptation aux incompatibilités des méthodes de service. Ce service est

appelé par les activités *InAdaptor* et *OutAdaptor* d'une sous-composition d'activité remplaçante. (Cf Section 3.3.2).

## 5.2 Evaluation d'un contrat d'adaptabilité

La figure 5.4 illustre la structure générale d'un contrat d'adaptabilité.

```
Contract : <ContractName>;
Scope: {<ActivityID>};
Properties:
   ExecutionTime <ComparisonOperator> <constant>;
   Reliability <ComparisonOperator> <constant>;
   Availability > <ComparisonOperator> <constant>;
Reactions:
  On <Exception1> Do <Action1>;
  On <Exception2> Do <Action2>;
   ...
  On <ExceptionN> Do <ActionN>;
```

FIG. 5.4 – Structure générale d'un contrat d'adaptabilité

Un contrat est une instance de la classe *AdaptabilityContrat* (cf Section 4.1). Son attribut *Scope* spécifie ses activités associées. Les critères de QoS sont spécifiés dans l'attribut *Properties*. L'attribut Reactions spécifie les réactions qui prédéfinissent les actions réagissant aux exceptions spécifiées.

L'évaluation d'un contrat d'adaptabilité consiste à déterminer si le contrat associé à l'activitée peut être exécutée ou non. Si le contrat contient une réaction réagissant à l'exception notifiée par le gestionnaire d'événement, il sera exécuté.

La figure 5.5 illustre le processus de l'évaluation d'un contrat. L'évaluation d'un contrat est déclenchée par la notification d'un événement (exception).

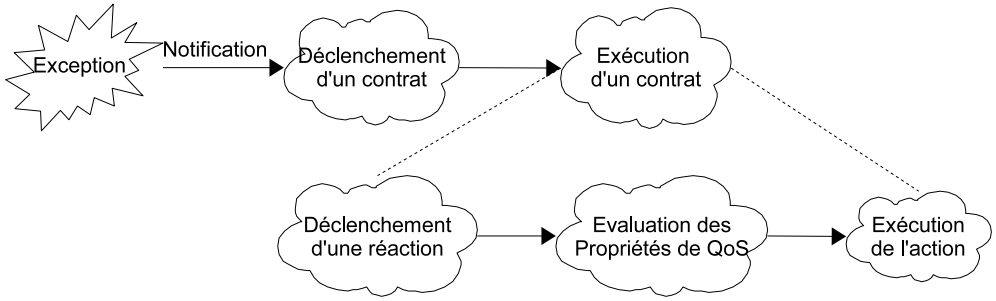


FIG. 5.5 – Processus d'évaluation d'un contrat d'adaptabilité

Le déclenchement de l'évaluation d'un contrat consiste à sélectionner une réaction qui correspond au message d'exception notifié par le gestionnaire d'événements.

La sélection d'une réaction dans la liste de réactions d'un contrat est illustrée par l'algorithme 1.

Cet algorithme a pour entrée le message d'exception notifié par le gestionnaire de contrats et l'instance du contrat. Il cherche d'abord l'exception (instance) correspondant à ce message en interrogeant la base d'exceptions. Ensuite il cherche dans la liste de réactions du contrat associé, une réaction réagissant à cette exception. Si une réaction est trouvée, alors l'algorithme retourne l'index de la réaction, sinon il retourne la valeur -1 pour spécifier qu'aucune réaction n'est trouvée et le contrat ne peut pas être exécuté.

**Algorithme 1** Sélection de réaction : ReactionSelection(msg String, contract Contract)

**ENTRÉES :** e : Exception, contract : Contract
**SORTIES :** Result : integer

```
Exception e = (select e from Exception e where e.message = msg)
pour (i=0 to unbound(contract.Reactions)) faire
   si (e in contract.Reactions[i].Exception) alors
      Retourner i
   finsi
fin pour
Retourner -1
```

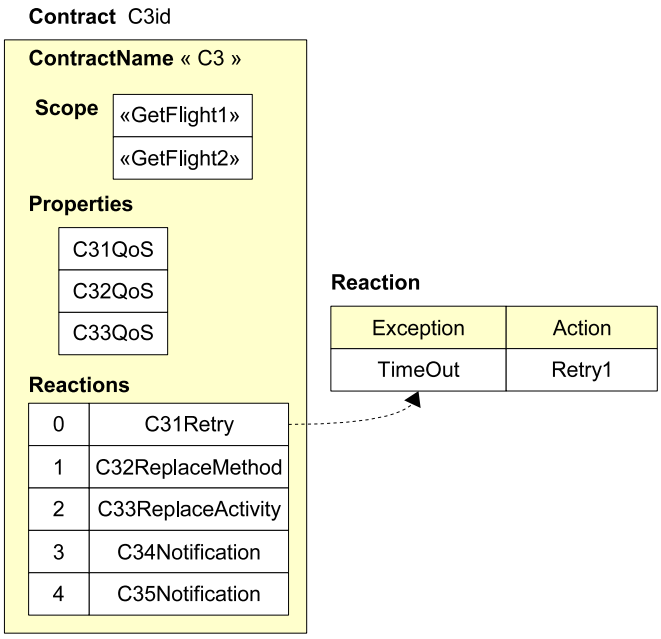


FIG. 5.6 – Sélection d'une réaction de la liste Reactions

Reprenons l'exemple du contrat "C3" présenté dans la section 4.3.2.1. Si l'exception e reçu est une instance de la Timeout (voir la figure 5.6), la réaction ayant l'index 0 dans la liste *Reactions* de ce contrat est trouvée, alors l'index 0 sera retourné. L'exécution d'un contrat consiste à déclencher et à exécuter la réaction. Rappelons que dans un contrat d'adaptabilité, une exception spécifiée ne déclenche qu'une seule réaction. Cependant, quand l'exécution de son action échoue, elle peut déclencher une autre réaction.

Notons que l'interaction entre l'exécution d'une activité et l'exécution de son contrat associé est synchrone, c'est à dire que l'exécution de l'activité sera suspendue et reexé-

cutera après avoir reçu le résultat de l'exécution du contrat renvoyé par le moteur d'évaluation de contrats. La validation d'un contrat implique donc la validation de son activité associée.

## 5.3 Exécution d'une réaction d'un contrat

L'exécution d'une réaction d'un contrat d'adaptabilité consiste à évaluer les critères de QoS et à exécuter son action prédéfinie.

Le processus d'exécution d'une réaction se réalise comme suit :

- Recherche de méthodes équivalentes (si besoin) ;
- Evaluation des mesures de QoS des méthodes de service selon des critères de QoS spécifiés dans le contrat ;
- Exécution de l'action prédéfinie dans la réaction si les critères de QoS satisfont.
- Si l'action réussit, le résultat sera retourné et l'évaluation se terminera. Sinon l'action de notification sera exécutée.

### 5.3.1 Recherche de méthodes équivalentes

La recherche de méthodes équivalentes qui se base sur la classification de méthodes en familles a pour but de trouver les méthodes équivalentes à la méthode de l'activité associée au contrat au moment d'évaluation. Les méthodes trouvées seront appelées lors de l'exécution d'une action d'adaptation.

Cependant il faut déterminer si on a besoin d'exécuter la recherche de méthodes équivalentes ou non car les actions d'adaptations comme la notification, la réexécution ne la demandent pas. La recherche de méthodes équivalentes sera exécutée si

- l'action prédéfinie de la réaction est une substitution de méthodes ou une subtitution d'activités et
- la méthode ou l'activité remplaçante dans cette action n'est pas spécifiée.

Reprenons l'exemple de l'action *ReplaceMethod1* du contrat "C3" présenté de la section 5.3.3.3. Dans cette action (instance) la méthode remplaçante *ReplacingMethod* (cf Section 5.3.3.3) n'est pas spécifiée (voir la figure 5.7). Dans ce cas, la recherche de méthodes équivalentes sera exécutée pour trouver les méthodes équivalentes à la méthode

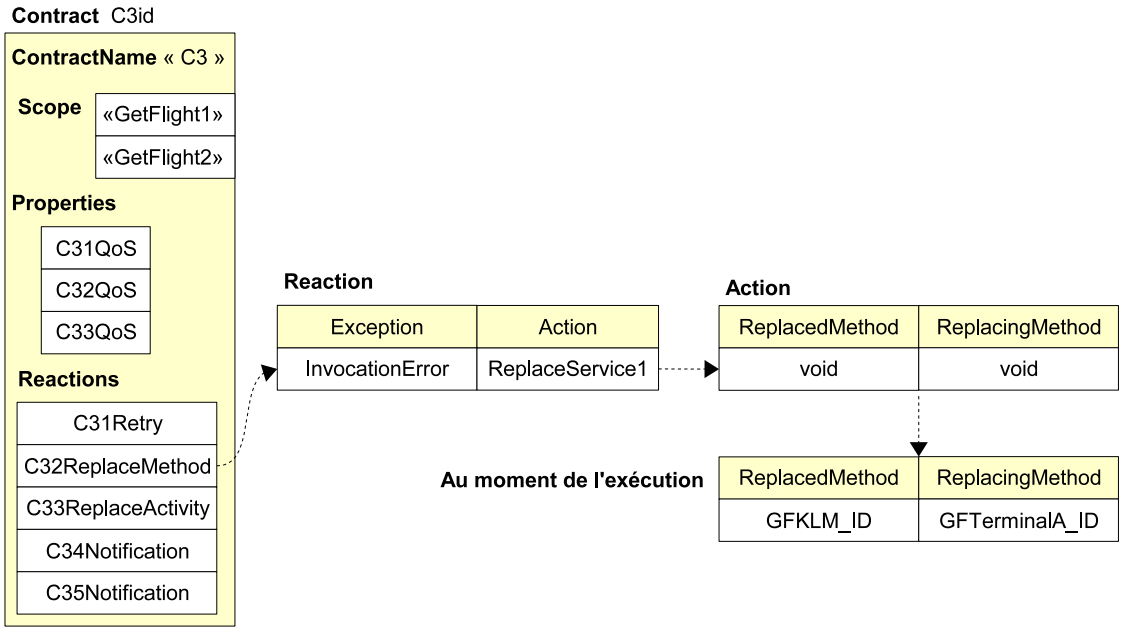


FIG. 5.7 – L'instance *ReplaceMethod1* de l'action *ReplaceMethodAction*

remplacée *ReplacedMethod*.

**Processus de recherche de méthodes de service équivalente** Etant donnée une méthode de service M, la processus de recherche de méthodes de service équivalentes est réalisée par l'algorithme 2. Cet algorithme a pour entrée une méthode de service M et pour sortie une liste de méthodes équivalentes ML. Il récupère d'abord les méthodes de la famille $F_M$ de M et ensuite les méthodes des familles équivalentes à $F_M$. Les méthodes trouvées sont stockées dans la liste ML.

**Algorithme 2** Recherche de méthodes équivalente : MethodLookup(Method M)

**ENTRÉES :** e : Exception, contract : Contract
**SORTIES :** ML : List(Method)
  Recherche la famille $F_M$ de M
  i = 0
  **pour** $M_i \in F_M$.ServiceMethods **faire**
    ML[i] = $M_i$
    i = i +1
  **fin pour**
  **tantque** $F_i$ équivalente à $F_M$ **faire**
    **pour** $M_i \in F_i$.ServiceMethods **faire**
      ML[i] = $M_i$
      i = i +1
    **fin pour**
  **fintantque**
  return ML

Reprenons l'exemple de la recherche de vols (Cf Section 3.1.1), en appliquant l'algorithme 2 pour la recherche de méthodes équivalentes à la méthode *GFKLM* prédéfinie dans l'activité GetFlight2, on obtient les méthodes de la famille "SearchFlight" (Cf Section 3.2.3) : *GFOpodo*, *GFTerminalA*, *GFAirFrance*.

### 5.3.2 Evaluation des critères de QoS

L'évaluation des critères de QoS a pour but de déterminer le déclenchement d'une action d'adaptation. Comme l'exécution d'une action d'adaptation : la réexécution, la substitution de méthodes de service et la substitution d'activités, se base sur une méthode de service, cette évaluation consiste à évaluer les mesures de QoS d'une méthode de service selon la formule de conjonction des critères de QoS du contrat. L'action sera exécutée si les mesures de QoS de la méthode satisfont les critères de QoS.

**L'évaluation des critères de QoS pour une action *réexécution*** consiste à évaluer les mesures de QoS de la méthode de service prédéfinie dans l'activité associée au contrat en cours d'évaluation.

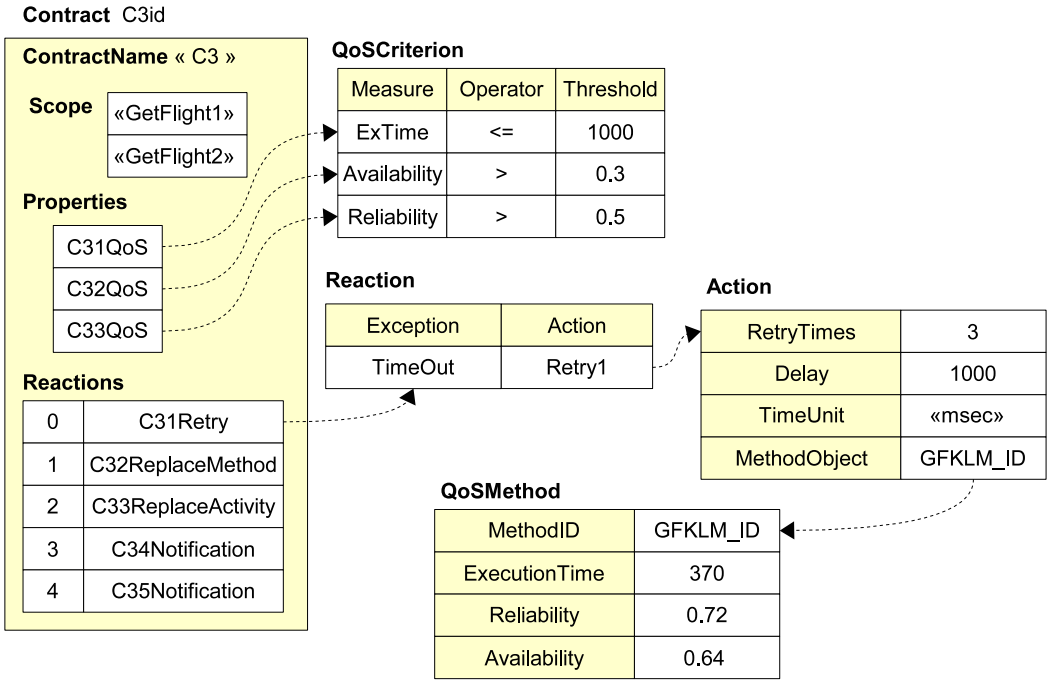


FIG. 5.8 – Les mesures de QoS de la méthode *GFKLM* appelé par l'action *Retry1*

Par exemple, dans l'action Retry1 de la réaction C31Retry du contrat "C3", la méthode de service GFKLM spécifiée dans l'activité GetFlight2 a des mesures de QoS qui satisfont les critères de QoS du contrat (voir la figure 5.8).

Pour l'évaluation des critères de QoS d'une action réexécution , deux stratégies d'évaluation peuvent être considérées :

1. La réexécution ne prend plus en compte des critères de QoS à chaque réexécution.

2. Avant chaque réexécution, l'évaluation des critères de QoS sera évaluée.

**L'évaluation des critères de QoS pour une action substitution de méthodes** consiste à évaluer les mesures de QoS :

1. de la méthode spécifiée dans cette action ou

2. des méthodes de service équivalentes dans la liste de méthodes ML trouvées dans l'étape précédente.

Le deuxième cas consiste à trier les méthodes de service satisfaites aux critères de QoS. Si la liste de méthodes après de trier est vide alors l'action substitution de méthodes ne peut par être exécutée et le contractExecutor notifie un message "Adaptation fails" au gestionnaire d'événements, sinon le processus de l'évaluation passe à l'étape de l'exécution de l'action d'adaptation.

### 5.3.3 Exécution d'une action d'adaptation

L'exécution d'action d'adaptation consiste à exécuter l'action prédéfinie dans la réaction. Comme présenté dans le modèle de contrat d'adaptabilité du chapitre précédent, les quatre types d'actions d'adaptation proposés sont la réexécution, la notification, la substitution de méthodes de service et la substitution d'activités (cf Section 3.3).

L'action d'adaptation sera exécutée si l'évaluation des mesures de QoS selon les critères de QoS est satisfaite. Si l'action réussit, le résultat sera retourné au moteur de coordination et l'exécution du contrat se termine. Dans le cas contraire, l'action notifie un message d'exception qui peut déclencher une autre réaction. Si aucune réaction spécifiée ne réussit, une exception sera notifiée au moteur de coordination de services par le gestionnaire d'événements.

#### 5.3.3.1 réexécution d'un appel à une méthode

Rappelons que l'action de réexécution consiste à rappeler un nombre spécifié de fois la méthode de service définie dans l'activité associée au contrat d'adaptabilité. Cette action est réalisée par l'opération retry() (Cf Section 4.3.2.1).

Reprenons l'exemple de la section 4.3.2, dans le contrat "C3" si l'exception de type "Timeout" est détectée et les mesures de QoS de la méthode appelée satisfont les critères de QoS du contrat associé, alors l'action de réexécution `Retry1` sera effectuée.

#### 5.3.3.2 Notification

La notification est l'action triviale qui ne résout aucune adaptation. C'est le seul type d'actions qui n'a pas besoin d'évaluer les critères de QoS. Elle est réalisée par l'opération Notify() (Cf Section 4.3.2.2).

Par exemple, dans le contrat "C3" de l'exemple présenté dans la section 4.3.2, si une exception de type *Unknown* se produit, alors l'action *Notification1* ci-dessous notifie le message "Unknown error" au moteur de coordination.

**Notify1** *NotificationAction* {
*NotificationMessage* "Unknown error"
}

#### 5.3.3.3 Substitution de méthodes de service

L'action *ReplaceMethodAction* consiste à exécuter une méthode de service remplacée qui peut être

1. la méthode remplacée qui est prédéfinie statiquement,

2. une composition de familles qui est spécifiée statiquement ou

3. une méthode choisie de la liste de méthodes équivalentes.

Pour le premier cas, l'action *ReplaceMethodAction* réalise l'appel à la méthode prédéfinie en exécutant l'opération *call*() de cette méthode.

Pour le deuxième cas, il s'agit d'exécuter l'opération *execute*() de l'instance de la classe *FamilyComposition* prédéfinie.

Pour le troisième cas, il s'agit de choisir la méthode la plus appropriée dans la liste des méthodes équivalentes créée lors du déclenchement de la réaction et triée lors de l'évaluation des critères de QoS. Dans le cas où l'appel à cette méthode échoue, la méthode suivante dans la liste sera appelée. L'algorithme *MethodExecution* illustre l'exécution des méthodes dans la liste selon son ordre jusqu'à ce qu'une méthode réussisse. Le résultat sera envoyé par le *ContractExecutor* au gestionnaire d'événements. Dans le pire des cas où aucune méthode ne réussit ou le délai d'exécution expire, un message d'erreur "Method replacing Error" sera retouné.

**Algorithme 3** Exécution de méthodes : MethodExecution(MList List)

```
ENTRÉES : MList : List
SORTIES : Result : String
  Result = NULL ;
  pour i=0 to Mlist.cardinality() faire
    {
    result = MList[i].call() ;
    si (not exception) alors
      return result ;
    finsi
    si (Timeout()) alors
      break ;
    finsi
    }
  fin pour
  return "Method replacing error !"
```

#### 5.3.3.4 Action de substitution d'activités

Pour les instances d'une coordination en cours d'exécution, nous ne remplaçons pas l'activité échouée mais remplace seulement sa méthode de service échouée par une autre méthode équivalente.

La substitution d'activités sera réalisée aprés un délai spécifié par l'administrateur. Après ce délai, les instances en cours d'exécution seront annulées, l'instanciation de cette coordination sera suspendue pour éviter la création des nouvelles instances lors de la substitution des activités. Quand la substitution d'activités est finie, SEBAS déploie la nouvelle coordination de sevices.

Une substitution d'activités est réalisée en trois phases comme suit :

– **Phase de préparation** :

  La phase de préparation se réalise par la recherche de méthodes de service. Cette tâche est responsable de trouver une méthode équivalente et la plus appropriée (par rapport aux critères de QoS) à la méthode de l'activité remplacée.

– **Phase de construction** :

  L'objectif de cette phase est de construire une activité ou une sous-composition qui puisse remplacer l'activité échouée de la coordination. Deux cas sont possibles :

  - Construction d'une activité : l'activité est créée en se basant sur la méthode de service équivalente à la méthode de l'activité remplacée. Ce cas est choisi quand les signatures de ces deux méthodes sont compatibles.

- Construction d'une sous-composition : ce cas est choisi quand les signatures des deux méthodes remplaçante et remplacée ne sont pas compatibles. Une adaptation de substitution d'activités présentée dans la section 3.3.2 sera réalisée pour remplacer l'activité remplacée par une sous-composition qui se compose de trois activités *InAdapter*, l'activité qui appelle la méthode remplaçante et *OutAdapter*. Les activités *InAdapter* et *OutAdapter* réalisent respectivement l'adaptation des incompatibilités des paramètres d'entrée et de sortie des deux méthodes équivalentes.

– **Phase de substitution** :

La substitution d'activités est réalisée par les opérations de substitution présentées dans la section 3.3.2. Trois cas sont possibles :

1. Si l'activité remplacée se place dans un flot de contrôle Sequence, l'opération *ReplaceSeqActivity* sera utilisée pour réaliser la substitution.

2. Si l'activité remplacée se place dans les flots de contrôle AndSplit et AndJoin, l'opération *ReplaceAndActivity* sera utilisée pour réaliser la substitution.

3. Si l'activité remplacée se place dans les flots de contrôle OrSplit et OrJoin, l'opération *ReplaceOrActivity* sera utilisée pour réaliser la substitution.

### 5.3.4 Exécution de réactions déclenchées en cascade

L'exécution d'une action d'adaptation peut produire une exception qui, à son tour, déclenche une autre réaction. On parle alors de réactions en cascade. Dans notre modèle de contrat, l'exécution d'une réaction dans un contrat ne peut générer qu'une seule exception qui peut déclencher qu'une seule autre réaction prédéfinie également dans le même contrat. Cela évite le risque des déclenchements de réactions sans fin. L'exécution des réactions se termine dans les cas suivants :

1. si une réaction réussit ou

2. toutes les réactions en cascade échouent ou

3. le temps de l'adaptation expire.

Pour le premier cas, le résultat de l'action de la réaction réussie sera retournée. Pour les deux et troisième cas, le *ContractExecutor* notifiera une exception "Adaptation failed".

L'ordonnancement des réactions déclenchées en cascade est séquentiel. Une exécution séquentielle est déterministe car l'ordre d'exécution des réactions dans un contrat est

défini. Une réaction R1 peut déclencher une réaction R2 lorsque l'action d'adaptation de R1 échoue et notifie une exception déclenchant R2.

## 5.4 Calculs des mesures de QoS d'une méthode de service

Les mesures de QoS d'une méthode de service considérées sont le temps d'exécution, la disponibilité, et la fiabilité.

**Caculs des mesures de QoS :** Les mesures de QoS de méthodes de service sont calculées en se basant sur des événements des types d'événement d'activité stockés dans un journal d'exécution.

Le tableau 5.1 montre les calculs des mesures de QoS d'une méthode de service (M1).

| Mesure de QoS | Calcul |
|---|---|
| Temps moyen d'exécution | $T = AVG(t_{InvokeActivity} - t_{EndActivity})$<br>Where Method = "M1" |
| Disponibilité | $Disp = N_2 / N$ |
| Fiabilité | $Fiab = N_1 / N$ |

TAB. 5.1 – Caculs des mesures de QoS d'une méthode de service

où

$N_1$ = select count(EndActivity) From ExecutionLog Where Method = "M1" and state = True ;

$N_2$ = select count(EndActivity) From ExecutionLog Where Method = "M1" and state = False ;

N = select count(EndActivity) From ExecutionLog Where Method = "M1" ; Les mesures de QoS d'une famille de méthodes sont les mesures moyennes de ses méthodes de service. Et les mesures de QoS d'une famille composite sont calculées en se basant sur les mesures de QoS de ses familles. Ces calculs sont présentés dans l'annexe D.

**Instant de calcul :** L'instant de calcul détermine le moment de déclenchement de la procédure de calcul des mesures QoS. Il est déterminé par la périodicité ou par la détection d'un événement.

La périodicité est la fréquence de calcul des mesures QoS qui peut être spécifiée par heure, jour, semaine ou mois.

Chaque fois qu'un événement du type d'événement spécifié est détecté, la procedure de calcul de mesures QoS se déclenche. L'instant peut être spécifié par un événement de

type *BeginProcess*/*EndProcess* ou *PrepareActivity*/*EndActivity*.

## 5.5 Implantation

### 5.5.1 SEBAS

Nous avons implanté un évaluateur de contrats d'adaptabilité appelé SEBAS pour valider notre approche. Les fonctionnalités implantées dans SEBAS sont :

- la gestion de contrats d'adaptabilité,
- la gestion d'événements,
- le calcul des mesures de QoS des méthodes de service,
- l'évaluation de contrats,
- l'exécution d'action d'adaptation : la réexécution, la notification et la substitution de méthodes de service.

Le tableau 5.2 montre les outils utilisés dans notre prototype SEBAS pour implanter ces fonctionnalités.

| | Processus | Service | Journal d'événements | Contrat | Structure de données |
|---|---|---|---|---|---|
| Fonction | Définition d'une coordination | Service Web | Journal d'événement | Contrat d'adaptabilité | Ontologies |
| Standard | XML | SOAP-XML | Base de données | Objet | OWL |
| Outils | XFLOW | Tomcat- Axis, Tomcat | Log4J, MySQL | NeoDatis ODB | Protégé, Jena, Racer |

TAB. 5.2 – Les outils utilisés dans SEBAS

Le moteur de coordination de services utilisé est XFLOW. L'intervention au moteur de coordination est réalisée grâce aux APIs fournies par XFLOW (Cf Annexe E).

Les services d'une coordination sont les services web simulés. Nous utilisons Tomcat et Axis pour installer et gérer les services simulés.

Log4J est utilisé comme le gestionnaire d'événements. Des événements produits lors de l'exécution d'une coordination sont stockés dans une base de données MySQL.

Les classes de notre modèle comme *Coordination*, *Service*, *Method*, *MethodQoS*, *ControlFlow* , *AdaptabilityContract* et leurs instances sont implantés en utilisant une base de données objet NeoDatis ODB.

Les classifications de méthodes en familles et d'exceptions (Cf Annexe C) sont implantées sous forme des ontologies en utilisant Protégé pour créer manuellement les ontologies et Racer pour les interroger.

## 5.5.2 Actions d'adaptation

### 5.5.2.1 La réexécution

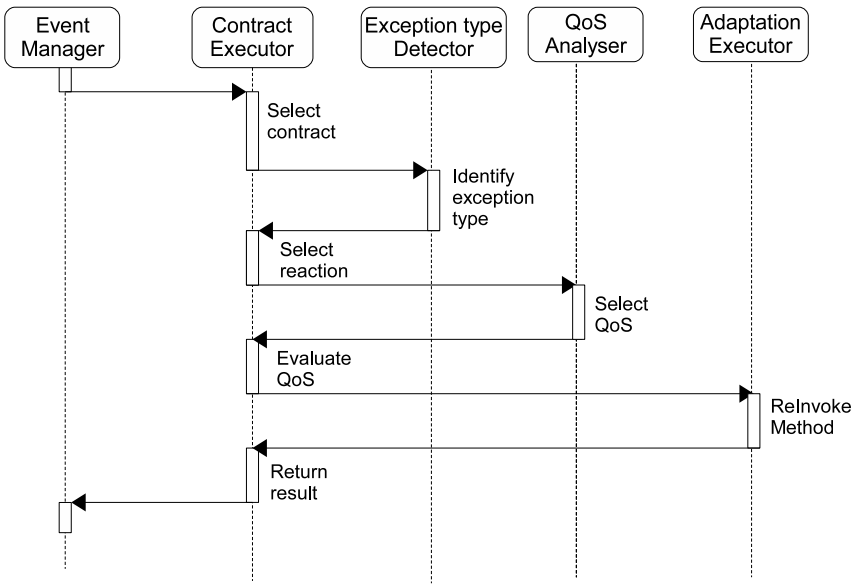


FIG. 5.9 – Interactions entre le ContractExecutor et ses partenaires pour exécuter l'action de réexécution

Le diagramme de séquence 5.9 illustre les interactions entre le *ContractExecutor* et ses partenaires : *ContractManagement*, *ExceptionTypeDetector*, *QoS Analyser* et *AdaptationExecutor*, pour réaliser le processus de l'action de réexécution.

Quand le *ContractExecutor* reçoit un message d'exception, il déclenche le processus d'évaluation d'un contrat. D'abord, il interroge le *ContractManager* pour trouver le contrat d'adaptabilité associé à l'activité dans laquelle s'est produite cette exception.

Ensuite, ayant le contrat, le *ContractExecutor* déclenche une réaction réagissant à l'exception. Il interroge le *ExceptionTypeDetector* pour déterminer le type de l'exception et détermine la réaction correspondante à cette exception dans la liste *Reactions* du contrat.

Comme l'action de la réaction trouvée est la réexécution, le *ContractExecutor* évalue les mesures de QoS de la méthode spécifiée dans l'activité en cours d'adaptation associée à ce contrat selon les critères de QoS. Si l'évaluation des critères de QoS de cette méthode de service satisfait, l'action sera exécutée par l'*AdaptationExecutor*. Le résultat obtenu sera renvoyé par le *ContractExecutor* au gestionnaire d'événements qui l'envoie au moteur de coordination.

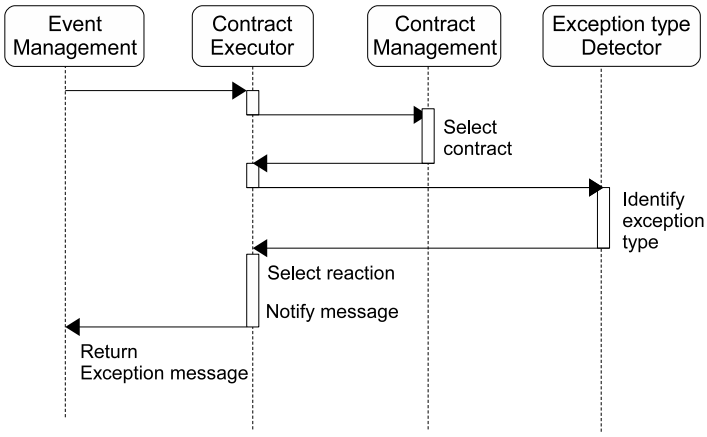


FIG. 5.10 – Interactions entre le ContractExecutor et ses partenaires pour exécuter l'action de notification

#### 5.5.2.2 La notification

Le diagramme de séquence 5.10 illustre les interactions entre le *ContractExecutor* et ses partenaires : *ContractManagement* et *Exception Type Detector* pour réaliser la notification.

Le début du processus de notification se réalise comme le processus de réexécution. Comme l'action de la réaction obtenue est la notification, le *ContractExecutor* exécute cette action en notifiant le message spécifié au gestionnaire d'événements.

### 5.5.3 La substitution de méthodes

Le diagramme de séquence 5.11 illustre les interactions entre le *ContractExecutor* et ses partenaires : *ContractManagement*, *Exception Type Detector*, *QoS Analyser*, *EquivalentMethodLookup* et *AdaptationExecutor*, pour réaliser le processus de substitution de méthodes de service présenté dans la section 5.3.3.3.

La phase déclenchement d'un contrat se réalise de la même manière que l'action de réexécution.

Dans la phase de déclenchement d'une réaction, comme l'action de cette réaction est la substitution de méthodes, si le méthode remplaçante n'est pas spécifiée, le *ContractExecutor* lance la recherche des méthodes de service équivalentes à la méthode remplacée en interrogeant le composant *EquivalentMethodLookup*.

Dans la phase d'évaluation, les méthodes de service équivalentes trouvées seront triées selon les critères de QoS en se basant sur leurs mesures de QoS reçues en interrogeant le *QoS Analyser*. Si aucune méthode de service ne satisfait les critères de QoS, le *ContractExecutor* notifie une exception de type *Replace_Service_error* au gestionnaire d'événements. Sinon, la méthode la plus appropriée par rapport aux mesures de QoS sera exécutée

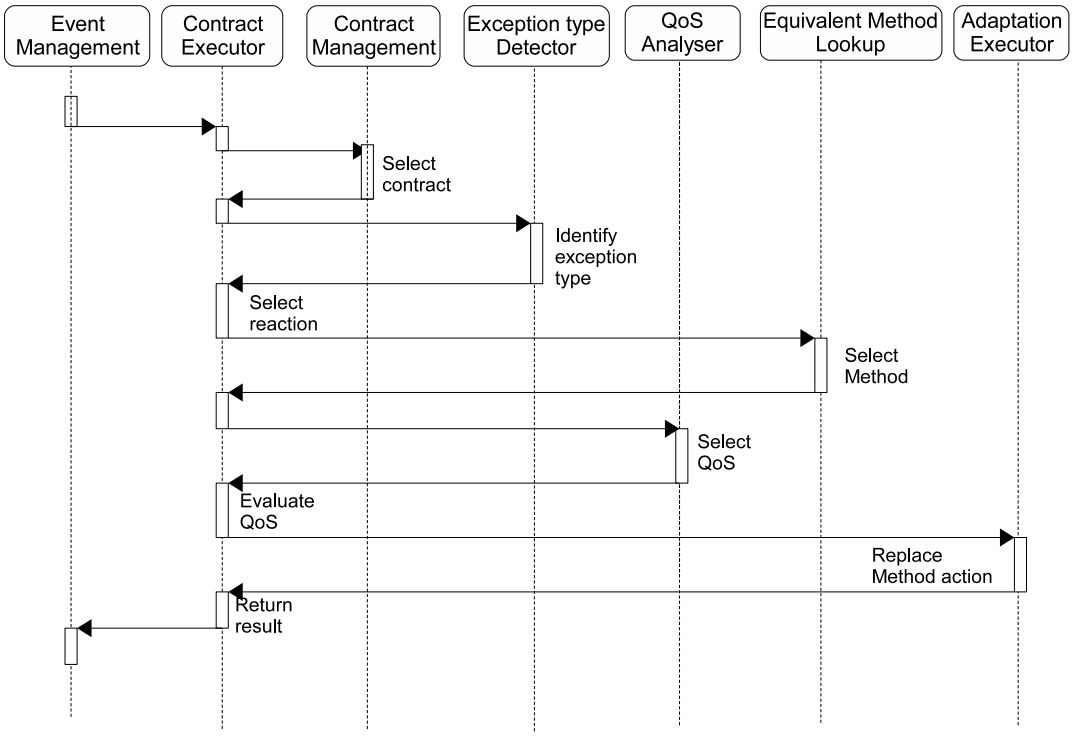


FIG. 5.11 – Interactions entre le ContractExecutor et ses partenaires pour exécuter l'action de substitution de méthodes

par l'*AdaptationExecutor*. Le résultat obtenu sera envoyé par le *ContractExecutor* au gestionnaire d'événements qui le renvoie au moteur de coordination.

## 5.6 Conclusion

Ce chapitre a présenté le processus d'évaluation de contrats d'adaptabilité qui réalise l'adaptation d'une coordination de services. L'évaluation de contrats est réalisée par le moteur d'évaluation de contrats d'adaptabilité qui associe à un moteur de coordination. Le processus d'évaluation d'un contrat d'adaptabilité consiste à exécuter une réaction (ou plusieurs réactions) qui déclenche l'exécution d'une action d'adaptation réagissant à une exception spécifiée. L'action d'adaptation sera exécutée si les critères de QoS du contrat sont respectés.

Les calculs des mesures de QoS de méthodes de service sont brièvement présentés. Finalement, ce chapitre présente les fonctionnalités implantées du prototype SEBAS et les outils utilisés.

# Chapitre 6

# Conclusion et perspective

## 6.1 Résumé du travail effectué

L'objectif de la thèse était de fournir l'adaptabilité à une coordination de services. Nous avons constaté que la composition statique de services manque de moyens d'adaptation dynamique aux exceptions et que la composition dynamique de services est coûteuse. Cela ne permet pas d'appliquer les techniques de composition dynamique pour réaliser l'adaptation.

Au long de notre étude sur l'état de l'art de l'adaptation dynamique d'une coordination de services, nous avons remarqué que l'adaptation a besoin de :

1. la classification de méthodes,

2. la résolution d'incompatibilités des signatures (paramètres d'entrée et sortie) entre les méthodes de service équivalentes,

3. la QoS de méthode de service.

La classification de méthodes permet de trouver les méthodes équivalentes pour répondre à un besoin afin de remplacer l'une par l'autre. Cependant, la classification ne peut pas être construite automatiquement car elle demande d'un grand travail de la part d'un expert du domaine d'application.

Dans certains cas, deux méthodes équivalentes ne peuvent pas être échangées directement et il est nécessaire de résoudre des incompatibilités de leurs signatures afin de pouvoir remplacer correctement l'une par l'autre.

La QoS de méthode de service permet de choisir la méthode la plus appropriée par rapport à un critère demandé. Cela garantit la performance (temps d'exécution) du processus

d'adaptation.

En nous focalisant à ces problèmes, nous avons proposé un modèle de coordination adaptative de services qui fournit :

**Les concepts de coordination de services** permettent de décrire une coordination de services sous forme d'un workflow et de spécifier les mesures de QoS d'une méthode de services. Nous avons également défini trois types d'opérations d'adaptation : la réexécution, la substitution de méthodes de service et la substitution d'activités.

**La classification de méthodes de service en familles** permet de définir l'équivalence entre les méthodes de service. Cette classification est représentée par une base de connaissances grâce à laquelle les méthodes équivalentes peuvent être trouvées.

**La notion de contrat d'adaptabilité** permet de :

- déclarer explicitement les situations et les comportements d'adaptabilité. Un contrat d'adaptabilité définit un mode d'adaptation sur la qualité d'exécution de la coordination de services comme le temps d'exécution, la fiabilité et la disponibilité et sur les réactions pour des exceptions ;

- séparer des propriétés fonctionnelles et non-fonctionnelles. Ceci rend notre approche indépendante et orthogonale aux moteurs d'exécution de la coordination de services ;

- fournir la flexibilité à l'adaptation car un contrat peut être modifié pour répondre aux nouvelles exigences en cours d'exécution sans influencer à la coordination.

Nous avons implémenté un prototype SEBAS qui valide nos propositions. Une première version de SEBAS a été présentée à la conférence BDA2006 [HVSC06]. Cette version a mis en œuvre :

1. l'exécuteur d'adaptation à base d'ontologies de familles de méthodes de service qui remplace une méthode de services par une autre équivalente.

2. l'adaptateur d'interfaces des méthodes qui s'adapte aux incompatibilités des signatures des méthodes équivalentes.

3. le gestionnaire d'événements qui stocke dans le journal d'exécution les événements produits par l'exécution des coordinations et de leurs activités. Ces événements archivés sont des éléments pour calculer les mesures de QoS de méthodes de service et de compositions de services.

4. l'analyseur des mesures de QoS (temps d'exécution, fiabilité, disponibilité) qui calcule les mesures de QoS de méthodes.

5. l'ontologie de familles de méthodes de service en OWL : Cette ontologie nous offre non seulement les descriptions sémantiques des familles de méthodes mais aussi le moyen pour naviguer et raisonner afin de trouver automatiquement des classes (concepts) de familles de méthode équivalentes, des mappings entre les signatures des méthodes équivalentes.

Ensuite, nous avons intégré dans SEBAS la notion de contrat d'adaptabilité qui permet au concepteur d'applications (programmeur) de spécifier explicitement les comportements d'adaptation. Par ailleurs, nous avons développé d'autres actions d'adaptation comme la réexécution et la substitution d'activités.

Dans la nouvelle version de SEBAS qui est un évaluateur de contrats, nous avons ajouté :

1. l'exécuteur de contrats qui est responsable de l'exécution de contrats d'adaptabilité en coordonnant les composants comme le gestionnaire de contrats, la détection de types d'exceptions, l'analyse des mesures de QoS, le service d'adaptation et l'adaptateur d'interface de méthodes ;

2. le gestionnaire de contrats qui gère et permet de manipuler les contrats stockés dans une base de données de contrats.

Nous avons également validé un contrat d'adaptabilité se combinant avec un contrat de transaction dans le contexte mobile. Ce travail a été réalisé dans le cadre du projet d'ORCHESTRA financé par le programme d'ECOS-ANUIES entre les gouvernements mexicains et français, et a été publié dans le cadre des 4èmes journées Francophones Mobilité et Ubiquité (UbiMob2008) [PHEO+08b] et dans le workshop Data and Services Management in Mobile Environments (DS2ME2008) associé à la conférence ICDE 2008 [PHEO08a].

## 6.2 Perspectives

Nous envisageons les perspectives suivantes pour notre travail :

– Combinaison de la propriété d'adaptabilité avec les autres propriétés non fonctionnelles comme la transaction et la sécurité : Ces propriétés sont au coeur des travaux de recherches des collègues de notre équipe dans le projet d'ORCHESTRA. Grâce

à la notion de contrat, nous pouvons composer des contrats d'adaptabilité, de sécurité et de transaction associés à une coordination de services.

– Validation de notre approche dans différents contextes comme l'informatique ubiquitaire : avec l'émergence de l'informatique ubiquitaire apparaît le besoin de construire des applications fiables et sécurisées qui fournissent un accès continu aux informations. L'accès aux ressources et aux applications doit être fait d'une façon flexible et robuste à travers des services qui se représentent comme un nouveau paradigme pour programmer et organiser des opérations. L'approche basée sur l'utilisation de contrats pour construire des applications mobiles fiables orientées services permet d'associer un comportement personnalisé à un flot décrivant la logique d'une application à base de services mobiles. Les contrats garantissent l'adaptabilité en présence d'exceptions et rendent les applications sensibles à leur contexte d'exécution (QoS).

– Validation des performances de notre approche : nous n'avons pas eu l'occasion d'évaluer les performances de notre approche dans un système réel. Les expérimentations que nous avons menées n'ont été considérées uniquement des services locaux.

– Validation de notre prototype SEBAS avec différents moteurs de coordination de services : des expérimentations permettant de mettre en place complètement la coopération entre l'évaluateur de contrats et le moteur de coordination doivent mettre mis en place .

# Bibliographie


[50902] *Workflow management : models, methods, and systems*. MIT Press, Cambridge, MA, USA, 2002.

[ABH+01] Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terence Payne, Katia Sycara, and Honglei Zeng. Daml-s : Semantic markup for web services. In *Proceedings of the International Semantic Web Workshop*, 2001.

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language, version 1.1. ws-bpel.pdf, may 2003.

[ACKM04a] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer Verlag, first edition, 2004.

[ACKM04b] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services : Concepts, Architecture and Applications*. Springer Verlag, 2004.

[Ant02] Antonio. *Quality of Service and Semantic Composition of Workflows*. PhD thesis, University of Georgia, Athens, GA, 2002.

[Ark02] Assaf Arkin. Business process modeling language. Copyright © 2002, BPMI.org, November 2002.

[AVMM04] Rohit Aggarwal, Kunal Verma, John A. Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *IEEE SCC*, pages 23–30, 2004.

[BBB+02] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, and Et. Web services conversation language (wscl) 1.0. Technical report, March 2002.

[BEJ+00] Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The object data standard : ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[BG06] Antonio Bucchiarone and Stefania Gnesi. A Survey on Service Composition Languages and Models. In *Proceedings of the 1st International Workshop on Web Services Modeling and Testing (WsMaTe'06), Palermo, Italy*, 2006.

[BHL+04] Mark Burstein, Jerry Hobbs, Ora Lassila, Drew Mcdermott, Sheila Mcilraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. Owl-s : Semantic markup for web services. Website, November 2004.

[BSB+05] Lucas Bordeaux, Gwen Salaün, Daniela Berardi, Daniela Berardi, and Massimo Mecella. When are Two Web Services Compatible ? In Springer-Verlag Berlin Heidelberg 2005, editor, *Technologies for E-Services(TES 2004,LNCS 3324)*, pages 15–28, 2005.

[BSD03] Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1) :40–48, 2003.

[BvHH+04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference. OWL Web Ontology Language Reference.html, february 2004.

[BVSC05a] K. Belhajjame, G. Vargas-Solar, and C. Collet. Pyros - an environment for building and orchestrating open services. *Services Computing, 2005 IEEE International Conference on*, 1 :155–162 vol.1, July 2005.

[BVSC05b] KhaIid Belhajjame, Genoveva Vargas-Solar, and Christine Collet. Building information systems by orchestrating open services. *Database Engineering and Applications Symposium, International*, 0 :27–36, 2005.

[CDK+06] Girish Chafle, Koustuv Dasgupta, Arun Kumar, Sumit Mittal, and Biplav Srivastava. Adaptation in web service composition and execution. In *ICWS '06 : Proceedings of the IEEE International Conference on Web Services*, pages 549–557, Washington, DC, USA, 2006. IEEE Computer Society.

[CIJ+00] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. eflow : a platform for developing and managing composite e-services, 2000.

[CLK01] Dickson K. W. Chiu, Qing Li, and Kamalakar Karlapalem. Adome-wfms : Towards cooperative handling of workflow exceptions. In *Advances in Exception Handling Techniques (the book grow out of a ECOOP 2000 workshop)*, pages 271–288, London, UK, 2001. Springer-Verlag.

[CMSA02] J. Cardoso, J. Miller, A. Sheth, and J. Arnold. Modeling quality of service for workflows and web service processes, 2002.

[DvdAtH05] Marlon Duma, WIl M. P. van der Aalst, and Arthur H. M. ter Hofstede. *Process-Aware Information Systems*, pages 169–172. John Wiley & Sons, Inc., copyright © 2005 edition, 2005.

[ET07] Exo and Bonita Teams. exo bonita user guide. http ://docs.exoplatform.org/exo-documents/exo-ecm.services.workflow.impl.bonita/, 2007.

[GHB+06] Zhijie Guan, Francisco Hernandez, Purushotham Bangalore, Jeff Gray, Anthony Skjellum, Vijay Velusamy, and Yin Liu. Grid-flow : a grid-enabled scientific workflow system with a petri-net-based interface : Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10) :1115–1140, 2006.

[GNA+98] Malik Ghallab, Ecole Nationale, Constructions Aeronautiques, Craig Knoblock Isi, Keith Golden, Scott Penberthy, David E Smith, Ying Sun, Daniel Weld, and Contact Drew Mcdermott. Pddl - the planning domain definition language, version 1.2. Technical report, 1998.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts : specifying behavioral compositions in object-oriented systems. *SIGPLAN Not.*, 25(10) :169–180, 1990.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4) :293–333, 1996.

[Hoh06] Andreas Hoheisel. User tools and languages for graph-based grid workflows : Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10) :1101–1113, 2006.

[HVSC06] Tan Hanh, Genoveva Vagar-Solar, and Christine Collet. Sebas : a semantic-based system for service adaptation. In *Actes des 22e Journées Bases de Données Avancées (BDA'2006)*, Lille, France, October 2006.

[JYH+06] Meng Jie, Su Stanley Y.W., Lam Herman, Helal Abdelsalam, Xian Jingqi, Liu Xiaoli, and Yang Seokwon. Dynaflow : a dynamic inter-organisational workflow management system. *International Journal of Business Process Integration and Management*, 1 :101–115, 2006.

[KBR+05] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto. Web services choreography description language version 1.0. Web Services Choreography Description Language Version 1.0.html, november 2005.

[KG05] Matthias Klusch and Andreas Gerber. Semantic web service composition planning with owls-xplan. In *In Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, pages 55–62, 2005.

[MBE03] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing web services on the semantic web. *The VLDB Journal*, 12(4) :333–351, 2003.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10) :40–51, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1) :1–40, September 1992.

[MT01] M. Matskin and E. Tyugu. Strategies of structural synthesis of programs and its extensions, 2001.

[MW08] A. Muscholl and I. Walukiewicz. A lower bound on web services composition. *ArXiv e-prints*, April 2008.

[PHEO08a] Alberto Portilla, Tan Hanh, and Javier-Alfonso Espinosa-Oviedo. Building reliable mobile services based applications. In *ICDE Workshops*, pages 121–128, 2008.

[PHEO+08b] Alberto Portilla, Tan Hanh, Javier-Alfonso Espinosa-Oviedo, Christine Collet, and Genoveva Vargas-Solar. Construire des applications fiables &agrave ; base de services mobiles. In *UbiMob*, pages 57–64, 2008.

[RHNT08] Ramy Ragab Hassen, Lhouari Nourine, and Farouk Toumani. Protocol-based web service composition. In *ICSOC '08 : Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 38–53, Berlin, Heidelberg, 2008. Springer-Verlag.

[RS05] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *LNCS*, volume 3387/2005, pages 43–54. Springer, 2005.

[Sam08] Yacine Sam. *Personnalisation de Services Web*. PhD thesis, Université Paul Cézanne (Aix-Marseille III), 04 Décembre 2008. Directeur : Pr. Omar Boucelma.

[SB06] Yacine Sam and Omar Boucelma. Personnalisation de services web : Approche fondée sur la composition. In *COnférence en Recherche d'Informations et Applications, CORIA'06*, pages 237–248, Lyon, France, 15–17 Mars 2006.

[SBDM02] Q. Sheng, B. Benatallah, M. Dumas, and E. Mak. Self-serv : A platform for rapid composition of web services in a peer-to-peer environment, 2002.

[SH03] Yoonki Song and Dongsoo Han. Exception specification and handling in workflow systems, 2003.

[tBBG06] Maurice ter Beek, Antonio Bucchiarone, and Stefania Gnesi. A survey on service composition approaches : From industrial standards to formal methods. Technical Report 2006-TR-15, ACM, D.2.4 Software/Program Verification . Formal Methods, 2006.

[TBFM06] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and ZAKARIA Maamar. Towards an Approach for Web services Substitution. In IEEE, editor, *10th IEEE International Database Engineering and Applications Symposium (IEEE IDEAS 2006)*, dec 2006.

[TMPE04] V. Tosic, W. Ma, B. Pagurek, and B. Esfandiari. Web service offerings infrastructure (wsoi) - a management infrastructure for xml web services. *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, 1 :817–830 Vol.1, 19-23 April 2004.

[TPE+02] V. Tosic, B. Pagurek, B. Esfandiari, K. Patel, and W. Ma. Web service offerings language (wsol) and web service composition management (wscm, 2002.

[vdABV+99] Wil M. P. van der Aalst, Twan Basten, H. M. W. Verbeek, Peter A. C. Verkoulen, and Marc Voorhoeve. Adaptive workflow-on the interplay between flexibility and support. In *International Conference on Enterprise Information Systems*, pages 353–360, 1999.

[vdABV+00] W. M. P. van der Aalst, T. Basten, H. M. W. Verbeek, P. A. C. Verkoulen, and M. Voorhoeve. Adaptive workflow. pages 63–70, 2000.

[vdATKB03] W. M. P. van der Aalst, Ter, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1) :5–51, July 2003.

[Wal01] Richard J. Waldinger. Web agents cooperating deductively. In *FAABS '00 : Proceedings of the First International Workshop on Formal Approaches to*

*Agent-Based Systems-Revised Papers*, pages 250–262, London, UK, 2001. Springer-Verlag.

[WSH+03] D. Wu, E. Sirin, J. Hendler, D. Nau, and B. Parsia. Automatic web services composition using shop, 2003.

[YL05] Tao Yu and K.-J. Lin. A broker-based framework for qos-aware web service composition. *e-Technology, e-Commerce and e-Service, 2005. EEE '05. Proceedings. The 2005 IEEE International Conference on*, pages 22–29, 29 March-1 April 2005.

[ZBD+03] Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. Quality driven web services composition. In *WWW '03 : Proceedings of the 12th international conference on World Wide Web*, pages 411–421, New York, NY, USA, 2003. ACM.

[ZBN01] Liangzhao Zeng, Boualem Benatallah, and Anne H. H. Ngu. On demand business-to-business integration. In *CooplS '01 : Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 403–417, London, UK, 2001. Springer-Verlag.

[ZBN+04] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5) :311–327, 2004.

# Annexe A

# Pré-requis

Les notions suivantes sont requises pour la définition des concepts du modèle de coordination de services.

### A.0.1 Domaine

Un domaine Db est un ensemble de valeurs distinctes. Il contient aussi les valeurs "*NULL*" :
- $\phi \in Db$ (non-information),
- $? \in Db$ (information inconnue).

`Db :-v` dénote la valeur *v* de *Db*

Un domaine qui se compose de valeurs non décomposables est un domaine atomique. Les domaines atomiques considérés sont : *Boolean*, *Char*, *String*, *Integer*, *Unsigned*, *Float*, *Time*, *Date*, *URI*, *Identifier*[1] d'un type et void.

Par exemple, *Integer* :-*62* est une valeur du domaine atomique *Integer*.

$\mathscr{D}$ dénote l'union des domaines. Il inclut la valeur "*NULL*", dénotée $\delta$, qui est une information non existante et n'appartient à aucun domaine.

### A.0.2 Type $\bigtriangledown$

Le domaine $\bigtriangledown$ des types est défini par les règles suivantes :

Un domaine atomique Db est un type de base

[1] Un identificateur (Identifier) est une valeur unique qui identifie un objet (ou une instance)

$$\texttt{Db :-Db} \in \bigtriangledown$$

Un type construit T est défini de manière récursive :

$$\begin{array}{c}
\texttt{T :-tuple(}a_1\ T_1\texttt{,}a_2\ T_2\texttt{,}\ldots\texttt{,}a_n\ T_n\texttt{)} \in \bigtriangledown\ \forall \texttt{i,j} \in \texttt{[1..n],i} \neq \texttt{j,}a_i \neq a_j \\
\texttt{T :-list(}T_1\texttt{)} \in \bigtriangledown \\
\texttt{T :-set(}T_1\texttt{)} \in \bigtriangledown \\
\texttt{T :-ref(}T_1\texttt{)} \in \bigtriangledown \\
\texttt{T :-choice(}T_1\texttt{)} \in \bigtriangledown
\end{array}$$

où $T_1, T_2, \ldots, T_n \in \bigtriangledown$ sont des types et $a_1, a_2, \ldots, a_n \in$ *String* sont des noms distincts.

**Nom et domaine de types**

Les fonctions *name* : $\bigtriangledown \rightarrow$ *String* et *dom* : $\bigtriangledown \rightarrow D$ permettent de retrouver le nom et le domaine d'un type $T$ de $\bigtriangledown$ :

Si $T$ est un type de base de la forme Db :-Db $\in \bigtriangledown$ alors :
- `name(Db :-Db)=Db`
- `dom(Db)=Db`

Db dénote à la fois le domaine (Db $\subset D$) et le type (Db $\in \bigtriangledown$).

Si `T :-tuple(`$a_1\ T_1$`,`$a_2\ T_2$`,...,`$a_n\ T_n$`)`$\in \bigtriangledown$ alors :

- `name(T :-tuple(`$a_1\ T_1$`,`$a_2\ T_2$`,...,`$a_n\ T_n$`))= T;`
- `dom(T) = {T :-tuple(`$a_1\ v_1$`,`$a_2\ v_2$`,...,`$a_n\ v_n$`)|n` $\geq$ `1,` $\forall$`i`$\in$`[1..n],`$v_i$ $\in$ `dom(`$T_i$`)}` est un domaine de valeurs n-uplets, où chaque valeur `T :-tuple(`$a_1\ v_1$`,`$a_2\ v_2$`,...,`$a_n\ v_n$`)` est une aggrégation des valeurs $v_1, v_2, \ldots,$ $v_n$ respectivement de types $T_1, T_2, \ldots, T_n$

Pour simplifier, nous notons `dom(T)= {tuple(`$a_1\ v_1$`,`$a_2\ v_2$`,...,`$a_n\ v_n$`)|n` $\geq$ `1,`
$\forall$`i`$\in$`[1..n],`$v_i$ $\in$ `dom(`$T_i$`)}`.

Si `T :-list(`$T_1$`)` $\in \bigtriangledown$ alors :

- `name(T :-list(`$T_1$`))= T;`
- `dom(T) = {T :-list(`$v_1$`, `$v_2$`, ..., `$v_n$`)| n`$\geq$`1,` $\forall$`i`$\in$ `dom(`$T_i$`)}` est un domaine de listes où chaque valeur `T :-list(`$v_1$`, `$v_2$`, ..., `$v_n$`)` est une liste des valeurs $v_1, v_2, \ldots, v_n$ de type $T_1$.

Pour simplifier, nous notons `dom(T)= {list(`$v_1$`, `$v_2$`, ..., `$v_n$`)| n`$\geq$`1,` $\forall$`i`$\in$ `dom(`$T_i$`)}`.

Si `T :-set(`$T_1$`)` $\in \bigtriangledown$ alors :

- `name(T :-set(`$T_1$`))= T;`

– `dom(T) = {T :-set(`$v_1$`, `$v_2$`, ..., `$v_n$`)| n≥1, ∀i∈ dom(`$T_i$`)}` est un domaine d'ensembles où chaque valeur `T :-set(`$v_1$`, `$v_2$`, ..., `$v_n$`)` est un ensemble de valeurs distintes $v_1, v_2, \ldots, v_n$ de type $T_1$.

Pour simplifier, nous notons `dom(T) = {set(`$v_1$`, `$v_2$`, ..., `$v_n$`)| n≥1, ∀i∈ dom(`$T_i$`)}`.

Si `T :-ref(`$T_1$`)` $\in \bigtriangledown$ alors :

– `name(T :-ref(`$T_1$`))= T;`
– `dom(T) = {T :-ref(`$v_1$`)|`$v_1$ ∈ `dom(`$T_1$`)}` est un domaine d'identificateurs, où chaque valeur T :- ref($v_1$) est une référence à un identificateur de la valeur $v_1$ de type $T_1$.

Si `T :- choice(`$T_1$`; `$T_2$`; ...; `$T_n$`)` $\in \bigtriangledown$ alors :
– `name(T :-choice(`$T_1$`; `$T_2$`; ...; `$T_n$`))= T;`
– `dom(T)= {T :-`$v$ | $v$ ∈ `dom(`$T_i$`), i ∈ [1..n]}` est un domaine de valeurs alternatives, où chaque valeur T :-$v$ est choisie parmi des valeurs typées $T_1$ :-$v_1$ , $T_2$ :-$v_2$, ..., $T_n$ :-$v_n$.

### A.0.3 Classe

**Opération :** Une opération a une signature qui spécifie le nom de l'opération, les paramètres d'entrée et de sortie. Elle est une instance du (méta) type défini par

**Operation** :- tuple(
operationName String ;
input list(tuple(paramName String, paramType $T_i$)),
output T)

où $T_i$, T $\in \bigtriangledown$, ∀i∈[1..n] .

Le type T du paramètre de sortie d'une opération est le type auquel est associé cette opération. L'opération est alors appélé du type T (ou l'opération définie sur le domaine dom(T)). Au cas où le paramétre de sortie est NULL, le type associé à l'opération est le type void.

Pour simplifier, une instance du (méta) type Opération qui a une liste de n paramètres d'entrées ayant comme nom paramName$_i$ typé par $T_i \in \bigtriangledown$ (∀i∈[1..n]), est décrite comme suit :

OperationName(paramName$_1$ $T_1$, ..., paramName$_n$ $T_n$)(T)

– paramName$_i$ sont des paramètres d'entrée typé par un type $T_i \in \bigtriangledown$ (∀i∈[1..n])
– le paramètre de sortie est typé par un type T $\in \bigtriangledown$).

Nous supposons l'existence de :

– opérations de comparaison $>, <, =, \neq, \geq, \leq$, définies sur les domaines de base et les domaines de référence ;
– opérations $\in, \notin, \subset, \subseteq, \cup, \cap$, définies sur les domaines n-uplet, ensemble et liste.

**Classe :** Une classe caractérise des objets de même nature. Ils ont le même type et possède les mêmes opérations. La (meta) classe est définie par

**class** :-tuple(
ClassName String,
attributSet set(tuple(AttributName String, attributType $T_i$),
OperationSet set(Operation)
) (où $T_i \in \bigtriangledown, \forall i \in [1..n]$)

Une instance de cette (meta) classe est la définition d'une classe.

Par exemple

**class** :-tuple(
*ClassName* "Toto",
. . . )

Pour simplier, on peut décrire cette définition sous la forme :

**class ClassName** {
$AttributName_i$ $T_1, \ldots,$
$AttributName_n$ $T_n$,
$OperationName_1$ ($paramName_{11}$ $T_{11}, \ldots, paramName_{1m}$ $T_{1n}$)(T), . . . ,
$OperationName_n$ ($paramName_{n1}$ $T_{n1}, \ldots, paramName_{nm}$ $T_{nm}$)(T)
}

où $T_i, T_{kj} \in \bigtriangledown, \forall i,k \in [1..n]$ et $\forall j \in [1..m]$

**Type et classe**

Une classe est un type *class*,
$\forall$I, instance de **classe** définit un type *Classname* $\in \bigtriangledown$ et dom(*Classname*) = {instance de class} $\subseteq \mathscr{D}$.

**Opérations de base d'une classe**

Toute classe possède les opérations de base suivantes :

1. new()() : créer un objet de la classe.
2. delete()() : supprimer un objet de la classe.

3. setValue(AttributName String, paramValue T)() : pour donner une valeur à un attribut d'un type T.
4. getValue(AttributName String) (T) : permet d'obtenir la valeur d'un attribut d'un type T.

Par exemple, la classe *Person* caractérisant des personnes se définit de la manière suivante

**class Person** {
    *Name* String,
    *PersonAddress* tuple(
        *Street* String,
        *City* String)
    print()() }

L'opération print()() affiche le nom et l'adresse de l'objet instancié par la classe Person.

**Object** est une instance d'une classe. Il possède une valeur typée et un comportement défini par l'ensemble d'opérations exécutables sur l'objet. Chaque objet a un identificateur unique.

Nous utilisons la notation OIF (Oject Interchange Format) pour définir un objet d'une classe. OIF est un langage de spécification utilisé pour définir l'état courant des objets persistants de ODMG (Object Database Management Group) [BEJ+00].

**Création d'un objet**

La création d'un objet est faite par l'opération new()() de sa classe. Pour définir un objet, nous dénotons en OIF

*ObjetTagName* **ClassName** {}

Par exemple, SarahP1 Person{}

Avec cette définition, un objet de la classe Person est créé. Les valeurs de ses attributs ne sont pas initialisées.

**Nomage d'un objet**

L'identificateur d'un objet est assigné à un nom d'étiquette unique dans la base d'objets. Un nom d'étiquette est évident dans l'ensemble entier d'objets.

Par exemple Sarah1 est le nom d'étiquette qui est utilisé pour mettre en référence l'objet défini dans l'ensemble entier

**Initialisation de valeur d'Attribut**

Un sous-ensemble des attributs d'un objet peut être initialiser explicitement.

Par exemple,

*SarahP1* **Person** {
  *Name* "Sarah",
  *PersonAddress* {
    *Street* "Willow Road",
    *City* "Palo Alto"}
    }

définit un objet de la classe Person et initialise l'attribut Name par la valeur "Sarah" et l'attribut PersonAddress par les valeurs "Willow Road" et "Palo Alto" pour les attributs Street et City respectivement.

**Classes List et Set**

La classe *List* instancie une collection d'objets ordonnés. Cette classe définit des opérations pour manipuler une liste d'objets. La définition de la classe List est présentée dans l'annexe.

La classe *Set* instancie un ensemble d'objets uniques et non ordonnés. Elle définit des opérations pour manipuler un ensemble d'objets. La définition de la classe Set est présentée dans l'annexe.

**Hiérarchie de classes**

Une hiérarchie de classes est un triple (G, $\leq$, Types) où

– G est un ensemble de classes,

– $\leq$ est un ordre partiel sur G,

– Types est une correspondance qui associe un type à chaque $g \in G$ tel que la condition suivante est satisfaite :
  $(\forall\ g_1, g_2 \in G)\ g_1 \leq g_2 \rightarrow \text{types}(g_1)$ subtype $\text{types}(g_2)$.

La relation binaire subtype est définie comme suit :

– tuple($a_1\ T_1, a_2\ T_2, \ldots, a_{n+k}\ T_{n+k}$) est un subtype de tuple($a_1\ T_1, a_2\ T_2, \ldots, a_n\ T_n$).

– Si $T_1$ est un subtype de $T_2$, alors set($T_1$) est un subtype de set($T_2$).

– Si $T_1$ est un subtype de $T_2$, alors list($T_1$) est un subtype de list($T_2$).

La première condition ci-dessus indique fondamentalement que le résultat d'ajouter de nouveaux attributs $a_{n+1}$ $T_{n+1}, \ldots, a_{n+k}$ $T_{n+k}$ à un type existant tuple($a_1$ $T_1$, $a_2$ $T_2$, ..., $a_n$ $T_n$) a pour résultat un subtype de tuple($a_1$ $T_1$, $a_2$ $T_2$, ..., $a_n$ $T_n$). Ce nouveau subtype a plus d'attributs que tuple($a_1$ $T_1$, $a_2$ $T_2$, ..., $a_n$ $T_n$) mais reporte toutes les propriétés originales de tuple($a_1$ $T_1$, $a_2$ $T_2$, ..., $a_n$ $T_n$).

Hypothèse : Dans notre approche, nous supposons qu'une classe ne peut héritier qu'une seule classe.

Bien qu'une méthode puisse être définie dans une classe g, elle s'applique pas simplement à cette classe, mais aussi bien à toutes les sous-classes de la classe g .

Supposons généralement que g est une classe. Nous employons la notation ↑g pour dénoter l'ensemble g' $\in$ G | g *leq* g'. La classe g hérite potentiellement la méthode m de la classe g* si et seulemnt si

1. g* $\in$ ↑g et la méthode m est défini dans g* et

2. il existe aucune classe $g^0$ tels que g $\leq$ $g^0$ $<$ g* et la méthode m est définie dans $g^0$.

On dit que la classe g est la spécialisation de la classe g* et la classe g* est la généralisation de la classe g.

Pour décrire la spécialisation/généralisation de deux classes g et g*, nous dénotons : class g : g* { }

Hypothèse : Dans notre approche, nous supposons qu'une classe ne peut hériter directement qu'une seule classe.

Par exemple, la classe Timeout qui représente des exceptions de type TimeOut est une spécialisation de la classe Exception. La définition de ces deux classes est décrite comme suit.

**class Exception** {
*ExceptionType* String,
}

**class Timeout** : Exception {
*Message* String,
*Notify()(String)*
}

### A.0.4 Predicat

Un terme est défini de manière récursive par les règles suivantes :

- une constante c (qui est l'élément d'un domaine $\mathscr{D}$) est un terme ;
- une variable v (qui est définie sur un domaine $\mathscr{D}$) est un terme ;
- une opération o (qui est associée à un domaine $\mathscr{D}$ est un terme ;
- une class c est un terme.

Un prédicat est une formule à base de termes. Le domaine des prédicats $\mathscr{P}$ ($\mathscr{P} \subset \mathscr{D}$) est défini par les règles suivantes :

- une expression de booléenne formée par des termes et des opérations $>, <, =, \neq, \geq, \leq, \in, \notin, \subset, \subseteq, \cup, \cap$, *not* est un prédicat.
  Par exemple : étant donné un variable Age de type Integer, un variable ServiceName de type String :
  - l'expression Age $\geq 0$ est un prédicat,
  - l'expression ServiceName not NULL exprimant que la valeur du nom d'un service doit ne pas être vide est un prédicat.

- une opération qui retourne une valeur de type Boolean est un prédicat.
  Par exemple, étant donné l est un objet de la classe Set, l'opération l.is_empty() qui vérifie si l est vide ou nom est un prédicat.

**Formule bien formée**

Une formule bien formée est décrite à l'aide d'un alphabet de symboles qui comporte :
- l'ensemble de constantes, éléments des domaines $\mathscr{D}$ ;
- l'ensemble de variables définies sur les domaines $\mathscr{D}$ ;
- l'ensemble d'opérations associées aux domaines $\mathscr{D}$ ;
- l'ensemble de prédicats avec arité $\mathscr{P}$ ;
- Les connecteurs logiques $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, les quantificateurs $\exists, \forall$, les symboles de ponctuation ")" et "(".

Le domaine Formula (Formula $\subset \mathscr{D}$) des formules bien formées est défini de manière récursive par les règles suivantes :

1. les termes VRAI et FAUX sont des formules bien formées ;
2. une formule atomique (c'est à dire un prédicat) est une formule bien formée ;
3. si f est une formule bien formée alors $\neg$f est une formule bien formée ;
4. si $f_1$ et $f_2$ sont des formules bien formées alors $f_1 \wedge f_2$, $f_1 \vee f_2$ sont des formules bien formées ;
5. si f est une formule bien formée alors (f) est une formule bien formée.

Toute formule bien formée est une instance du type Formula qui est défini par le domaine Formula $\subset \mathscr{D}$ : Formula :-Formula $\in \bigtriangledown$

Par exemple, la formule

(From not NULL) $\wedge$ (To not NULL)

qui désigne une précondition dans laquelle From, To sont des variables (paramètres typés) est une formule bien formée ($\in$ Formula).

# Annexe B

# Contraintes de flots de contrôle

1. Contraintes de cardinalité d'opérateur :

   – *Sequence* : la cardinalité de *InputActivity* et de *OutputActivity* doit être égal à 1 : $|InputActivity| = 1$ and $|OutputActivity| = 1$

   – *AndSplit* et *OrSplit* : la cardinalité de *InputActivity* doit être égal à 1 et la cardinalité de *OutputActivity* supérier à 1 : $|InputActivity| = 1$ and $|OutputActivity| > 1$

   – *AndJoin* et *OrJoin* : la cardinalité de *InputActivity* doit être supérieur à 1 et la cardinalité de *OutputActivity* égal à 1 : $|InputActivity| > 1$ and $|OutputActivity| = 1$

2. Contraintes de structure de flots de contrôles :

   – Un flot de contrôle ayant un opérateur *AndSplit* doit correspondre à un flot de contrôle ayant un opérateur *AndJoin*, et les activités de sortie du *AndSplit* représentent les activités d'entrée de *AndJoin*.

   – Un flot de contrôle ayant un opérateur *OrSplit* doit correspondre à un flot de contrôle ayant un opérateur *OrJoin*, et les activités de sortie du *OrSplit* représentent les activités d'entrée de *OrJoin*.

# Annexe C

# Ontologies

Les structures de données pour les classifications d'exceptions et de familles de méthodes sont construites par des ontologies en langage OWL.

Nous avons construit deux ontologies principales : l'ontologie d'exceptions et l'ontologie de familles de méthodes.

## C.1 Règles de transformation

Nous définissons les règles de transformation des classes/objet définies dans notre approche en concept/instance d'ontologie. Les concepts et les instances (individus) de concept d'une ontologie sont transformés par les règles suivantes :

1. Une classe est un concept.

2. Les attributs d'une classe sont les attributs d'un concept.

3. L'association entre deux classes est une propriété entre les deux concepts.

4. Une classe-association est un concept.

5. La relation de généralisation/spécialisation est une relation subsomption.

6. Chaque objet d'une classe est une instance (individu) d'un concept.

## C.2 Ontologie de familles de méthodes de service

L'ontologie de familles de méthodes a pour but de définir une taxonomie de méthodes services en famille répondant aux besoins de fonctionnalité et la relation d'équivalence entre les familles de méthodes afin de pouvoir réaliser correctement une substitution d'une méthode de service par une autre équivalente ou par une composition de méthodes.

La figure C.1 présente une vue globale des concepts de l'ontologie de familles de méthodes. Le concept (classe) central est la famille de méthodes Family. Une famille de méthodes correspond à un besoin qui est défini par des concepts Category et Domain spécifiés. Elle a des relations d'équivalence avec d'autres familles. Une famille possède un ensemble de méthodes de service où chaque méthode réalise une fonctionnalité. Une méthode a des propriétés fonctionnelles comme un nom, une adresse, des entrées, une sortie, des pré-conditions et post-conditions. Un besoin peut être également réalisé par une composition de familles.

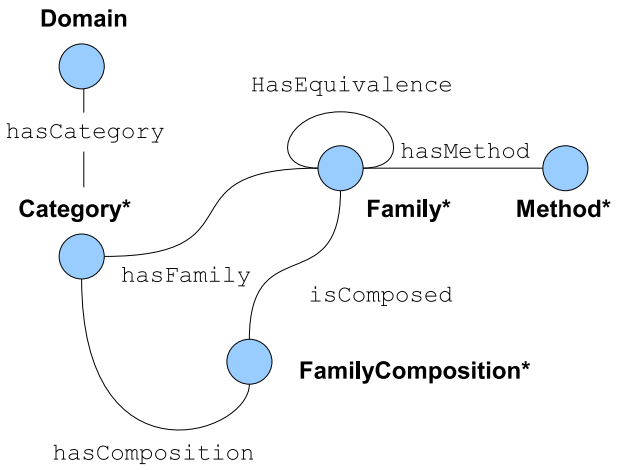


FIG. C.1 – Ontologie de familles de méthodes

**Domaine de familles de service** est défini par les expressions :

Domain := $\forall$ hasCategory.Category $\sqcap \geq 1$ hasCategory
Category := $\forall$ hasFamily.Family $\sqcap \geq 1$ hasFamily $\sqcap \forall$ isDomain.Domain

**Family :** Le concept Family est défini par une expression :

Family := $\forall$ isCategoryOf.Category $\sqcap$
$\exists$ hasMethod.Method $\sqcap$=1 hasMethod $\sqcap$
$\exists$ hasEquivalent.Family $\sqcap$
$\exists$ HasInputType.Type $\sqcap$ $\exists$ HasOutputType.Type $\sqcap$
$\exists$ HasInput.Input $\sqcap$ $\exists$ HasOutput.Output $\sqcap$

$\exists$ HasPreCondition.Expression $\sqcap$ $\exists$ HasPostCondition.Expression

La définition du concept Family décrit trois éléments principaux :

- Classification : une famille appartient à une catégorie d'un domaine (isCategoryOf.Family)
- Relation : une famille peut avoir des relations d'équivalence avec des autres familles (hasEquivalence) et il est exporté au moins par une méthode (hasMethod.Method et $\geq$ 1 hasMethod)
- Propriétés fonctionnelles : une famille a des propriétés fonctionnelles comme types d'entrée (HasInputType.Type), noms d'entrée (HasInput.Input), types de sortie (HasOutputType.Type), noms de sortie (HasOutput.Output), pré-condition et postcondition (HasPreCondition.Expression, HasPostCondition.Expression).

la propriété hasEquivalence qui représente l'équivalence entre les familles est réflexible, asymétrique et transitive.

**Méthode de service :** Le concept Method est défini par une expression :

Method := $\forall$ isMethodOf.Family $\sqcap$
$\forall$ HasMethodName.String $\sqcap$
$\forall$ HasInput.String $\sqcap$
$\forall$ HasOutput.String $\sqcap$
$\forall$ HasAddress.String $\sqcap$

La classe Method caractérise un ensemble de méthodes ayant la propriété isMethod qui exprime un rapport avec une famille et les autres propriétés qui expriment ses propriétés fonctionnelles.

Concrètement, une méthode représente une interface de service. C'est un aspect structural d'un service. Il décrit les propriétés fonctionnelles comme l'adresse de méthode, le nom de méthode, les entrées et les sorties.

Pour exprimer les entrées et sorties d'une méthode, nous décrivons une expression suivante :

Parameter : := "$type_1$ [ : $type_2$ [ : ... [ : $type_n$]]]"

$type_i$ représente un type de donnée du paramètre d'entrée ou sortie qui correspond à la méthode L'ordonnancement des types dans un paramètre doit aussi correspond à l'ordre des paramètres d'entrée/sortie de la signature de la méthode.

**Composition de familles :** Le concept FamilyComposition est défini par une expression suivante :

FamilyComposition := $\forall$ IsComposed.Family $\sqcap$ $\geq$2 IsComposed $\sqcap$
$\forall$ HasOrder.order $\sqcap$
$\forall$ HasInputMapping.Mapping $\sqcap$ $\geq$1 HasInputMapping $\sqcap$
$\forall$ HasOutputMapping.Mapping $\sqcap$ $\geq$1 HasOutputMaping

Le concept FamilyComposition spécifie

- les familles composées ($\geq$ 1IsComposed.Family),
- l'ordre d'exécution de familles qui est décrite par le concept (Order),
- les mappings entre les entrées et sorties d'une famille et de la composition de familles ("HasInputMapping.Mapping, HasOutputMapping.Mapping) par rapport aux individus correspondants dans la classe Mapping.

**Flot de contrôles :**

Un flot de contrôle est spécifié par la classe Order qui est définie suivant :

Order := $\exists$HasFamily.Family $\sqcup$ $\exists$HasSeq.Seq $\sqcup$ $\exists$HasPar.Par $\sqcup$ $\exists$HasIF.If $\sqcup$ Bottom

Seq := $\exists$HasFirstSeq.Family $\sqcap$ $\exists$HasOrder.Order

Par := ($\forall$HasOrder.Order $\sqcap$ $\geq$2 HasOrder) $\sqcap$ $\exists$HasJoined.Order

If := $\exists$HasCondition.Expression $\sqcap$ $\exists$Then.Order $\sqcap$ $\exists$Else.Order

Le concept Order exprime qu'un flot de contrôle peut avoir trois structures principales : séquence, parallèle et condition. Un flot Order peut-être une famille ou une de ces trois structures.

La structure de séquence de familles est décrite par le concept Seq. Elle a deux propriétés HasFirstSeq et HasOrder. La première propriété décrit la famille qui démarre la séquence de familles. La seconde propriété spécifie un flot de contrôle qui la suit.

La structure parallèle est décrite par le concept Par. Elle est basée sur la propriété HasOrder qui spécifie qu'un individu a au moins deux individus du concept Order.

La structure Condition est décrite par le concept IF. La propriété HasCondition décrit une expression conditionnelle pour déterminer quelle branche sera exécutée. Si l'expression conditionnelle a la valeur True, l'individu Order qui rapporte à la propriété Then sera choisi pour s'exécuter, sinon l'individu Order de la propriété Else sera choisie.

**Mappings :** Un mapping entre deux familles est défini par le concept Mapping suivant :

FamilyMapping : := $\exists$hasSourceFamily.Family $\sqcap$ $\exists$hasDestinationFamily.Family $\sqcap$ $\forall$hasInterfaceMapping.Mapping $\sqcap \geq$ 2hasInterfaceMapping

Un FamilyMapping est une relation binaire entre une famille de source (hasSourceService) et une famille de destination (hasDestinationService). Cette relation a deux mappings d'interface (hasInterfaceMapping). Un mapping d'interfaces décrit les mappings des paramètres d'entrée ou de sortie des familles équivalentes.

Le concept Mapping est défini par les expressions suivantes :

Mapping := $\exists$IsAssigned.Operand $\sqcup$ $\exists$HasSource.Expression

Operand := Input $\sqcup$ Output $\sqcup$ Literal

Literal := Number $\sqcup$ Character $\sqcup$ String $\sqcup$ BooleanLiteral
Number := Integer $\sqcup$ Real
BooleanLiteral := True $\sqcup$ False

Le concept Mapping décrit un mapping entre une opérande d'une famille de source et une opérande d'une famille de destination. Une opérande peut être un individu Input, output ou Literal.

la propriété IsAssigned est employée pour assigner un individu du concept Mapping avec un individu de concept Input ou output d'une famille.

la propriété HasSources décrit la relation entre un individu du concept Mapping avec un individu du concept Expression.

**Equivalence entre les familles de méthodes :** La définition d'équivalence de deux familles Family est définie par la propriété HasEquivalence suivante :

Etant données trois classes Family F1, F2, F3 et une base de connaissance $\delta$, F1 équivalent F2 ssi F1 a une relation binaire (propriété) HasEquivalence avec F2 : HasEquivalence(F1,F2) $\in \delta$

la propriété HasEquivalence est réflexive, asymétrique et transitive :

- Réflexive : Les instances de F sont équivalentes.

- Asymétrique : HasEquivalence(F1, F2) n'implique pas HasEquivalence(F2, F1)$\in \delta$.

- Transitive : Si HasEquivalence(F1, F2) $\in \delta$ et HasEquivalence(F2, F3) $\in \delta$, on a HasEquivalence(F1, F3) $\in \delta$

**Equivalence de méthodes :** La définition d'équivalence de deux classes Méthodes est définie par les propriétés : HasMethod et HasEquivalence suivant :

Etant données deux classes Method M1, M2 et deux classes Family F1 et F2, M1 équivalente M2 ssi :

(1) HasMethod(F1,M1) $\in \delta$ and HasMethod(F1,M2) = $\in \delta$.

(2) HasMethod(F1,M1) $\in \delta$ and HasMethod(F2,M2) $\in \delta$ and HasEquivalence(F1,F2) $\in \delta$.

Dans le cas (1), M1 et M2 sont équivalentes car elles sont les classes Method qui ont la relation HasMethod avec la même classe Family F1.
Dans le cas (2), M1 est équivalente à M2 car elles sont les méthodes des deux familles équivalentes F1, F2 respectivement.

## C.3 Ontologie d'exceptions

La Figure C.2 illustre notre ontologie d'exceptions.

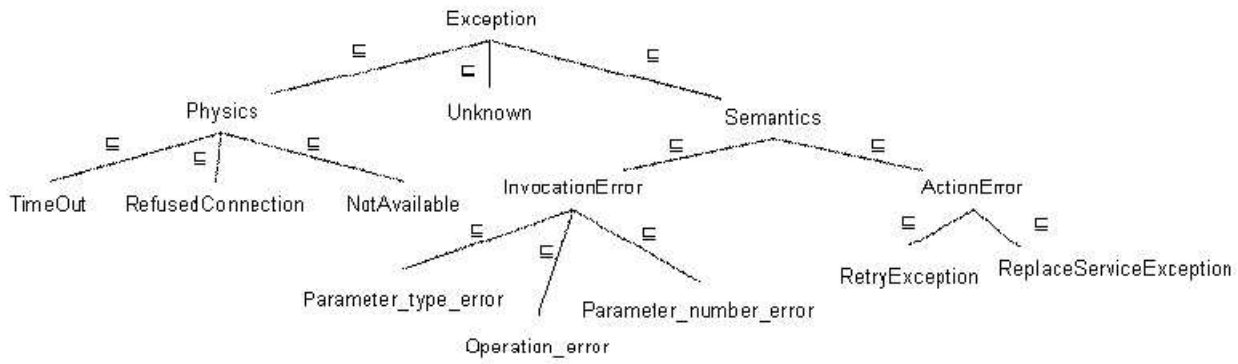


FIG. C.2 – Ontologie d'exception

**Classe Exception** est la classe générale qui inclut trois sous-classes d'exceptions : physique (*Physics*), sémantique (*Semantics*) et inconnu (*Unknown*). La classe d'exceptions *Unknown* inclut des exceptions non déterminées.

Physics $\sqsubseteq$ Exception
Semantics $\sqsubseteq$ Exception
Unknown $\sqsubseteq$ Exception

**Classe Physics :** est une classe qui classifie les fautes de système. Elle a trois sous-classes : *timeout*, *RefusedConnection*, *NotAvailable*. La classe *timeout* représente des exceptions causées par l'expiration du délai d'une connexion à un serveur ou à un service. La classe *RefusedConnection* représente l'indisponibilité du serveur tandis que la classe *NotAvailable* représente l'indisponibilité du service appelé.

Timeout $\sqsubseteq$ Physics
RefusedConnection $\sqsubseteq$ Physics
NotAvailable $\sqsubseteq$ Physics

Par exemple les messages d'exceptions qui expriment l'expiration du délai d'une connexion comme "*Server too busy*", "*Service temporary unavailable*" , "*server timeout*" sont les instances de la classe *Timeout*. Les exceptions "*server is unavailable*"' rapporté par le protocole d'invocation de service RPC, "*Remote Server unavailable*" rapporté par SOAP ou "*internal server error*" rapporté par HTTP sont les instances de la classe *RefusedConnection*. Les exceptions de type "*service_not_available*" comme "*Service unavailable*"' sont rapporté par le serveur web comme IIS ou par le protocole HTTP.

**Classe Semantics** est une classe qui classifie des erreurs rapportées par un service ou une application. Elle a deux sous-classes : *InvocationError*, *ActionError*. La classe *ActionError* présente les types d'erreur venant d'une action effectuée dans un contrat d'adaptabilité.

InvocationError $\sqsubseteq$ Semantics
ActionError $\sqsubseteq$ Semantics

La classe *InvocationError* représente les exceptions rapportées lors d'un appel à une méthode de service. Elle se compose de trois sous-classes *Parameter_number_error*, *parameter_type_error*) et (*Operation_error*).

Parameter_number_error $\sqsubseteq$ InvocationError
Parameter_type_error $\sqsubseteq$ InvocationError
Operation_error $\sqsubseteq$ InvocationError

Par exemple les messages d'erreur d'invocation d'une méthode de service comme "*unable to determine operation*", "*Parameter count mismatch*","type mismatch" rapportés par SOAP sont des instances de la classe *Operation_error*, *Parameter_number_error* et "*Parameter_type_error*" respectivement.

La classe *ActionError* se compose de deux sous-classes *RetryException* et *ReplaceServiceException*. La classe *RetryException* représente les exceptions produites lorsque l'action *RetryAction* échoue et la classe *ReplaceMethodException* représente les exceptiont l'error lors de l'exécution de l'action *ReplaceAction* échoue.

$\text{RetryException} \sqsubseteq \text{ActionError}$
$\text{ReplaceMethodException} \sqsubseteq \text{ActionError}$

Les classes d'exceptions sont séparées. Grâce à cette propriété, nous pouvons indentifier facilement l'instance d'une classe d'exceptions.

$\text{Physics} \sqcap \text{Semantics} = \emptyset$
$\text{Unknown} \sqcap \text{Semantics} = \emptyset$
$\text{Physics} \sqcap \text{Physics} = \emptyset$
$\text{Timeout} \sqcap \text{NotAvailable} = \emptyset$
$\text{RefusedConnection} \sqcap \text{NotAvailable} = \emptyset$
$\text{Timeout} \sqcap \text{RefusedConnection} = \emptyset$
$\text{InvocationError} \sqcap \text{ActionError} = \emptyset$
$\text{Parameter_number_error} \sqcap \text{Parameter_type_error} = \emptyset$
$\text{Parameter_type_error} \sqcap \text{Operation_error} = \emptyset$
$\text{Parameter_number_error} \sqcap \text{Operation_error} = \emptyset$
$\text{RetryException} \sqcap \text{ReplaceServiceException} = \emptyset$

# Annexe D

# Calcul des mesures de QoS

Les mesures de QoS considérées sont le temps d'exécution, la fiabilité et la disponobilité.

Le calcul d'une mesure de QoS d'une méthode de service se base sur des événements stockés dans le journal d'exécution ExecutionLog.

La valeur d'une mesure de QoS d'une famille de méthodes est la valeur moyenne de cette mesure de ses méthodes de service.

Les calculs des mesures de QoS d'une famille composite que nous proposons sont basés sur les travaux de Cardoso [Ant02, CMSA02].

## D.1 Temps d'exécution d'une famille composite

Le temps d'exécution d'une famille composite est basé sur les temps d'exécution de ses familles de méthodes et sur ses structures de flot de contrôle *Sequence*, *AndSplit* et *ORSplit*.

### D.1.1 Temps d'exécution du flot de contrôle *Sequence*

Etant donné n familles de méthode *F1*, *F2*, ..., *Fn* composées séquentiellement et ayant les temps d'exécution *T1*, *T2*, ..., *Tn* respectivement, le temps d'exécution *T* du flot de contrôle *Sequence* est calculé par :

$$T = \sum_{i=1}^{n} T_i$$

Par exemple, une famille composite ayant un flot de contrôle Sequence *C* de trois

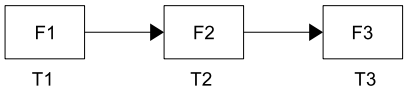


FIG. D.1 – Une séquence de familles de méthodes

familles *F1*, *F2*, *F3* (voir la figure D.1). Les mesures de QoS de ces familles sont données dans le tableau D.1 .

| Mesure QoS | F1 | F2 | F3 |
|---|---|---|---|
| Temps d'exécution | 430 ms | 1200 ms | 578 ms |
| Fiabilité | 0,7 | 0,83 | 0,91 |

TAB. D.1 – Tableau 1 des mesures de QoS

Le temps d'exécution de la famille composite $C$ est calculé comme suit :

Temps d'exécution :

```
T = T1 + T2 + T3
= 430 + 1200 + 578
= 2208 ms
```

### D.1.2 Temps d'exécution du flot de contrôle *AndSplit/AndJoin*

Etant donnée une famille composite définie par un flot de contrôle *AndSplit/AndJoin* de n familles de méthodes *F1*, *F2*, ..., *Fn* ayant les temps d'exécution *T1*, *T2*, ..., *Tn* correspondants respectivement, le temps d'exécution $T$ de la famille composite est calculé par

$$T = max(T1, T2, \quad \ldots \quad , Tn)$$

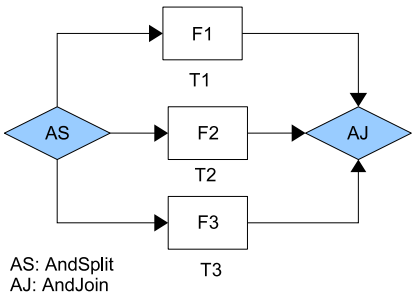


FIG. D.2 – And/Split-join de Familles

Par exemple, une famille composite *AndSplit/AndJoin* (*AJ*) de trois familles *F1*, *F2*, *F3* ayant les mesures de QoS données dans le tableau D.1 est illustrée par la figure D.2).

| Mesure de QoS | F1 | F2 | F3 |
|---|---|---|---|
| Temps d'exécution | 430 ms | 1200 ms | 578 ms |
| Fiabilité | 0,7 | 0,83 | 0,91 |
| proportion | 0.6 | 0.3 | 0.1 |

TAB. D.2 – Tableau 2 des mesures de QoS des familles

Le temps d'exécution de la famille composite*AJ* est calculé comme suit :

Temps d'exécution :

```
T = max (T1, T2, T3)
= 1200 ms
```

***OrSplit/OrJoin*** :

Etant donnée une famille composite*OrSplit/OrJoin* définie par un flot de contrôle *OrSplit/orJoin* de n familles de méthodes *F1*, *F2*, ..., *Fn* ayant les temps d'exécution *T1*, *T2*, ..., *Tn* correspondants respectivement, le temps d'exécution *T* de la famille composite est calculé par

$T = p1.T1 + p2.T2 + ... + +pn.Tn$
$avec(p1 + p2 + ... + pn = = 1)$ où pi (i = 1 ... n) est la proportion du Famille Fi choisi.

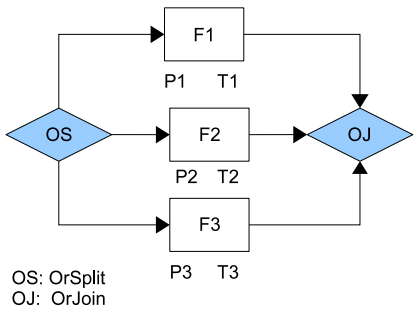


FIG. D.3 – Or/split-join de Familles

Par exemple, une famille composite*OrSplit/OrJoin* (OJ) composée de trois familles *F1*, *F2*, *F3* ayant les mesures de QoS données dans le tableau D.2 est illustrée par la figure D.2).

Le temps d'exécution de la famille composite *OJ* est calculé comme suit :

```
T = p1.T1 + p2.T2 + p3.T3
= 0,6.430 + 0,3.1200 + 0,1.578
= 675,8 ms
```

### D.1.3 Temps d'exécution d'une méthode de service

Le temps d'exécution d'une méthode d'un service est le temps moyen entre l'appel de la méthode et la fin de son exécution.

T = AVG(selection timestamps From ExecutionLog Where ExecutionLog.phase = "InvokeActivity" and ExecutionLog.Method = "M1") - AVG(selection timestamps From ExecutionLog Where ExecutionLog.phase = "EndActivity" and ExecutionLog.Method = "M1")

## D.2 Fiabilité d'une famille composite

La fiabilité d'une famille composite est basée sur les mesures de fiabilité de ses familles de méthodes et ses structures de flots de contrôle *Sequence*, *AndSplit* et *ORSplit*.

### D.2.1 Fiabilité du flot de contrôle textitSequence

Etant données une famille composite de familles *F1*, *F2*, ..., *Fn* composées séquentiellement et ayant les mesures de fiabilité *R1*, *R2*, ..., *Rn* correspondantes respectivement, la fiabilité $R$ de la famille composite est calculée par

$R = \prod_{i=1}^{n} R_i$

Par exemple, reprenons la famille composite séquentielle $C$ de trois familles *F1*, *F2*, *F3* (voir la figure D.1) ayant les mesures de QoS données dans le tableau D.1.

La fiabilité de C est calculée comme suit :

```
R = R1 x R2 x R3
= 0,7 x 0,83 x 0,91
= 0,52871
```

### D.2.2 Fiabilité du flot de contrôle *AndSplit/AndJoin*

Etant donnée une famille composite définie par un flot de contrôle *AndSplit/AndJoin* de n familles *F1*, *F2*, ..., *Fn* ayant les mesures de fiabilité *R1*, *R2*, ..., *Rn* correspondantes respectivement, la fiabilité $R$ de la famille composite *AndSplit/AndJoin* est calculée comme suit :

$$R = \prod_{i=1}^{n} R_i$$

Par exemple, reprenons la famille composite ayant la structure de flot de contrôle *AndSplit/AndJoin* (AJ) de trois familles *F1*, *F2*, *F3* illustré par la figure D.2. Les mesures de fiabilité de ces familles sont données dans le tableau D.1.

La fiabilité $R$ de la famille composite *AJ* est calculée comme suit :

```
R = R1 x R2 x R3
= 0,7 x 0,83 x 0,91
= 0,52871
```

### D.2.3 Fiabilité du flot de contrôle *OrSplit/OrJoin*

Etant donnée une famille composite définie par un flot de contrôle *OrSplit/OrJoin* de n familles *F1*, *F2*, ..., *Fn* ayant les mesures de fiabilité *R1*, *R2*, ..., *Rn* correspondantes respectivement, la fiabilité $R$ de la famille composite*OrSplit/OrJoin* est calculée comme suit :

$$R = p1.R1 + p2.R2 + \ldots + +pn.Rn$$
$$avec(p1 + p2 + \ldots + pn = 1)$$

*pi* (i = 1 ... n) est la proportion de choix de la famille *Si*.

Par exemple, reprenons la famille composite définie par le flot de contrôle *OrSplit/OrJoin* (*OJ*) de trois familles *F1*, *F2*, *F3* illustrée par la figure D.3. Les mesures de QoS de fiabilité de ces familles données dans le tableau D.2.

La fiabilité $R$ de la famille composite*OJ* est calculée comme suit :

```
R = p1.R1 + p2.R2 + p3.R3
= 0,6.0,7 + 0,3.0,83 + 0,1.0,91
= 0,76
```

### D.2.4 Fiabilité d'une méthode de service

La fiabilité d'une méthode de service est la proportion entre le nombre d'appels réussis (résultat pertinent) et le nombre total d'appels.

La fiabilité de la méthode "M1" est calculée commme suit :

```
T = count(selection * From ExecutionLog Where ExecutionLog.phase = "EndActivity" and ExecutionLog.State = true) / count(selection * From ExecutionLog Where ExecutionLog.phase = "EndActivity" and ExecutionLog.Method = "M1")
```

## D.3 Disponibilité d'une famille composite

La disponibilité d'une famille composite est basée sur des mesures de disponibilité des familles de ces familles composants et sa structure de composition. Elle est calculée de la même manière de la fiabilité.

### D.3.1 Disponibilité d'une méthode de service

La disponibilité d'une méthode d'un service est la proportion entre le nombre d'appels disponibles et le nombre total d'appels.

La disponibilité de la méthode "M1" est calculée par

```
T = count(selection * From ExecutionLog Where ExecutionLog.phase = "EndActivity" and ExecutionLog.State = false) / count(selection * From ExecutionLog Where ExecutionLog.phase = "EndActivity" and ExecutionLog.Method = "M1")
```

## D.4 Mise à jour de QoS de méthodes

L'instant de calcul détermine le moment de déclenchement de la procédure de calcul des mesures QoS. Cette dimension est caractérisée la périodicité ou par un type d'événements.

$QoSComputeInstant ::= Periodic|Event$

**Périodicité** La périodicité statique caractérise la fréquence de calcul des mesures QoS spécifié au temps de configuration. Cette fréquence peut être spécifiée par heure, jour,

semaine ou mois. La spécification d'instant de calcul est décrit sous forme BNF suivant :

$PerioidicSpecification ::=< beginning, periodic, ending >$
$beginning := datetime$
$periodic ::= hour|day|week|month$
$hour ::= HOUR(number)$
$day ::= DAY(number)$
$week ::= WEEK(1|2|\ldots|7 : Time)$
$month ::= MONTH(1|2|\ldots|28 : Time)$
$ending ::= date : time|nul$

- beginning spécifie la date de commencement (date et heure) où la périodicité commence.

- ending spécifie la date de terminaison de la périodicité de calcul. Elle peut avoir la valeur NULL pour déterminer que la périodicité dure infiniment.

- La périodicité (periodic) peut être choisie entre heure, jour, semaine et mois avec la spécification des options en BNF suivante : periodic ::= "Hour("h")" | "Day("d")" | "Week("w")" | "Month("m")"

  - l'option *hour* spécifie la périodicité de calcul par le nombre d'heures h (h $\in$ [1..24]) à partir du repère beginning.

  - l'option *day* spécifie la périodicité de calcul par le nombre de jours d (d $\in$ [1..31] à partir du repère beginning.

  - l'option *week* spécifie que la périodicité de calcul sera par semaine et dans un jour et une heure précis de la semaine à partir du repère beginning. Le 1 est équivalent au dimanche ainsi de suite.

  - l'option *month* spécifie que la périodicité de calcul sera mensuelle et dans un jour et une heure précis du mois à partir du repère beginning.

Par exemple une périodicité `P = < 12/05/2008 6 :00am; Day(1); NUL` $>$ détermine que la périodicité de calcul sera chaque jour à partir du 13 mai 2008 à 6 heures du matin.

**Evénement :** Chaque fois qu'une occurrence du type d'événement spécifié se produit, la procedure de calcul de mesures QoS se déclenche. L'instant peut être spécifié par un événement de type *BeginProcess*/*EndProcess* ou *PrepareActivity*/*EndActivity*.

La spécification d'instant d'événement est décrite comme suit :

$Instant ::= processEvent|activityEvent$
$processEvent ::= BeginProcess|EndProcess$
$activityEvent ::== PrepareActivity|EndActivity$

Un instant peut être spécifié par un événement de type *process* ou de type *activity*.

Par exemple, pour l'instant spécifié par un type *processEvent*, si le type d'événement *BeginProcess* est choisi, le calcul de QoS sera exécutée chaque fois qu'une exception du type *BeginProcess* se produit.

Pour l'instant spécifié par un type *activityEvent*, si le type d'événement *PrepareActivity* est choisi, le calcul de QoS sera exécuté chaque fois qu'une exception du type *PrepareActivity*.

# Annexe E

# API du moteur de worflow (*XFLow*)

Les moteurs de workflow fournissent les APIs afin de manipuler les modèles de workflow, les instances de workflow, les activités et leurs attributs. Grâce aux APIs, on peut intervenir dans l'exécution d'une instance de workflow ainsi à l'exécution d'une activité.

Nous citons les méthodes publiques des APIs de *XFlow* permettant de manipuler :

1. *un (des) modèle(s) de workflow :*

– deployModel() : déployer un modèle de workflow, c'est à dire une définition d'un workflow.
– getWorkflowModels() : récupérer les modèles de workflow.

2. *une (des) instances d'un workflow :*

– getWorkflowState() : récupérer l'état du workflow.
– getActiveWorkflowInstances() : récupérer les instances actives d'un workflow.
– resumeWorkflow() : reprendre l'exécution d'une instance d'un workflow.
– suspendWorkflow() : suspendre l'exécution d'une instance d'un workflow.
– abortWorkflow() : annuler l'exécution d'une instance d'un workflow.
– startWorkflow() : démarrer l'exécution d'une instance d'un workflow.

3. *une activité :*

– getProcessNodes() : récupérer toutes les activités d'un workflow.
– getProcessNodeByNodeName() : récupérer une activité par son nom.
– getProcessName() : récupérer le nom d'une activité.
– setProcessName() : nommer un nom pour une activité.
– setTimeStart() : mettre l'heure de démarrage d'une activité.

4. *un attribut d'une activités :*

– setWorkitemId() : donner une valeur à un attribut d'une activité.
– getWorkitemId() : récupérer la valeur d'un attribut d'une activité.