Une approche agile, fiable et minimale pour le maintien de la qualité de service lors de l'évolution d'applications à base de processus métiers

Alexandre Feugas

Université Lille 1 Sciences et Technologies

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Une Approche Agile, Fiable et Minimale pour le Maintien de la Qualité de Service lors de l'Évolution d'Applications à Base de Processus Métiers

THÈSE

présentée et soutenue publiquement le 8 octobre 2014

pour l'obtention du

Doctorat de l'Université Lille 1 Sciences et Technologies

Spécialité Informatique

par

Alexandre Feugas

Composition du jury

Rapporteurs :	Marie-Pierre Gervais, Université Paris Ouest Nanterre La Défense Jean-Michel Bruel, Université de Toulouse
Directeur de thèse :	Laurence Duchien, Université Lille 1
Examinateurs :	Jean-Luc Dekeyser, Université Lille 1 Sébastien Mosser, Université Nice - Sophia Antipolis

Laboratoire d'Informatique Fondamentale de Lille — UMR CNRS 8022
INRIA Lille - Nord Europe

Une Approche Agile, Fiable et Minimale pour le Maintien de la Qualité de Service lors de l'Évolution d'Applications à Base de Processus Métiers

Les logiciels actuels adoptent une méthodologie de développement dite "agile" pour mieux prendre en compte la nécessité de s'adapter constamment aux nouveaux besoins des utilisateurs. Les concepteurs et développeurs se rapprochent alors des futurs utilisateurs du logiciel en proposant des cycles courts d'itération, où le futur utilisateur fait un retour rapide sur l'incrément apporté au logiciel, et fait part de nouveaux besoins à prendre en compte dans les incréments à venir. Ces itérations peuvent être vues comme des évolutions, faisant suite à la définition d'un nouveau besoin de l'utilisateur, à un changement de l'environnement d'exécution, ou encore à une remise en question de l'architecture du logiciel. Dans l'écosystème des architectures orientées services, la conception d'applications passe par la chorégraphie ou l'orchestration de services par des processus métiers. Concevoir ces applications consiste alors à mettre en relation les flots de contrôle et de données de ces services. La phase d'évolution devient une phase complexe, où une simple modification localisée à une sous-partie d'un processus métier peut avoir des conséquences sur l'ensemble du système logiciel, causant par exemple son ralentissement lors de l'exécution. Du point de vue de la qualité de service (QoS), la maîtrise de la fiabilité du processus d'évolution pour maintenir la qualité de service d'un logiciel est alors critique.

Il est donc nécessaire de pouvoir proposer des mécanismes d'évolution agiles et fiables permettant le maintien de la QoS lors de l'évolution d'applications à base de processus métiers. En d'autres termes, il s'agit de s'assurer qu'une évolution ne viole pas les contrats de QoS définis initialement. Cette garantie doit être établie en fonction du contrat soit lors de la conception soit lors de l'exécution. Dans ce dernier cas, le processus de vérification doit être minimal et localisé, afin de ne pas dégrader les performances du système logiciel.

Pour cela, nous proposons de mettre en œuvre un cycle de développement agile, centré sur le maintien de la QoS lors de l'évolution. Il s'agit de prendre l'aspect évolutif du système, ceci dès l'étape de conception initiale, en identifiant les informations requises pour déterminer si la QoS est correcte et si elle est non violée par une évolution. Ces informations étant détenues par plusieurs intervenants, il est également nécessaire d'établir les points d'interaction au cours du cycle de développement, au cours desquels les informations seront partagées de façon à ce que le logiciel qui en est issu reste syntaxiquement et sémantiquement cohérent et que les contrats de QoS soient (re)vérifiés a minima. Les contributions de cette thèse sont donc mises en œuvre dans Blink, un cycle de développement pour l'évolution, et Smile, un canevas de développement pour le maintien de la qualité de service lors de l'évolution d'applications orientées service définies à base de processus métiers. Tandis que le cycle de développement Blink vise à identifier les différents rôles présents dans l'équipe de développement et à expliciter leurs interactions, le canevas Smile propose la réalisation d'une boucle d'évolution. Cette boucle permet de concevoir, d'analyser et d'appliquer une évolution, en détectant les potentielles violations de contrat de QoS. Pour cela, l'analyse de l'évolution détermine son effet sur la QoS du logiciel, en établissant des relations de causalité entre les différentes variables, opérations, services et autres parties du système. Ainsi, en identifiant les éléments causalement affectés par l'évolution et en écartant ceux qui ne le sont pas, notre approche permet de limiter le nombre d'éléments à (re)vérifier, garantissant ainsi une étape d'évolution fiable, avec une étape de (re)vérification minimale.

Nous montrons sur un cas concret de système de gestion de crises, constitué de onze processus métiers et de dix scénarios, que l'utilisation conjointe de Blink et de Smile permet d'identifier, pour chaque évolution, quel sera son effet sur le reste du système, et si la qualité de service sera maintenue ou non.

An Agile, Reliable and Minimalist Approach to Preserve the QoS of Business-Processes Based Applications during their Evolutions

Current softwares are built using "agile" development methods, to better consider the need to adapt to new user requirements. Developers and designers are getting closer to future software users by making short iterative cycles, where the future user gives a fast feedback on the increment made to the software and emits new user requirement to be fulfilled in future increments. These iterations can be seen as evolutions, as an answer to the definition of a new user requirement, or due to a change in the execution environment or in the architecture of the software. In the Service-Oriented Architecture (SOA) world, the design of software is composed of service choreography, or service orchestration using business processes. The design of these applications is equivalent to connecting the services control flow and data flow. As a result, the evolution step becomes a complex step, where a simple modification on a sub-part of a business process can have consequences on the entire system, causing for example its slowing down at runtime. From the Quality of Service (QoS) point of view, ensuring the fiability of the evolution process to maintain software QoS is critical.

As a result, it is necessary to propose agile, reliable evolution mecanisms ensuring QoS preservation during the evolution of software made of business processes. In other words, we need to control that an evolution does not violate any QoS contract initially set up. Depending of the contract, this garanty must be established either at design time or at runtime. In the latter case, the verification process must be minimal and local, in order to not degrade the software performance.

To achieve this goal, we propose to realise an agile development cycle, centered on the QoS preservation during the evolution. It is necessary to care about the evolutive concern of a system from the initial design step, by identifying required information to determine if the QoS continues to be correct and not violated by an evolution. Considering that this information is known by many stakeholders, it is also necessary to set up interaction points during the development cycle, during which information is shared in order to keep building a syntactically and semantically coherent software and to minimally (re)check QoS contracts. The contributions of this thesis are applied in Blink, a development cycle for evolution, and Smile, a framework to maintain QoS during the evolution of a service-oriented software made of business processes. While Blink is intended to identify the different actors and to make their interactions explicit, Smile proposes the realisation of an evolution loop. This loop enables the design, analysis and application of an evolution, by detecting the potential QoS contract violation. The evolution analysis determines its effect on the software QoS, by defining causal relations among variables, operations, services and other parts of the system. In this way, by identifying elements that are causally affected by the evolution, and by avoiding the elements that are not, our approach enables the limitation of the number of elements to (re)check in order to assure a reliable evolution step, with a minimal (re)check step.

We show on the concrete case of a crisis management system, composed of eleven business processes and ten scenarios, that the combined use of Blink and Smile enables for each evolution the identification of its effect on the system, and the QoS preservation of the system.

Remerciements

La thèse est une aventure, une expérience humaine qui vous marque profondément tant elle change vos habitudes,votre façon de penser. J'ai vécu au cours de ces quatre dernières années quelque chose de beau, d'intense, de douloureux et de décourageant aussi. Mais c'est avant tout l'ensemble des rencontres faites durant ce laps de temps qui ont fait de cette thèse un moment fort à mes yeux. Dans cette page, je n'aurai pas la prétention d'être exhaustif, j'oublierai sûrement quelques personnes qui, je l'espère, ne m'en voudront pas. Mais sachez juste que, qui que vous soyez, que je vous ai croisé de près ou de loin, merci à vous d'avoir joué un rôle dans mon aventure doctorante.

Je voudrais remercier en premier lieu mes encadrants, Laurence et Seb, qui ont dû avoir quelques cheveux blancs à cause de moi. Merci de vous être impliqué dans ces travaux, d'avoir su garder patience avec moi. J'espère que vous ne m'en voulez pas trop de vous avoir mené la vie dure. A titre postume, j'ai également une pensée profonde pour Anne-Françoise, qui a su croire en moi ; qui m'a encouragé et qui a été derrière moi à mes débuts. Merci à toi. J'aurais aimé que tu sois présente, auprès de nous.

Plus généralement, je voudrais remercier l'équipe Adam/Spirals, qui a été ma deuxième famille. Merci aux permanents, avec leurs remarques, leurs discussions passionnantes, ou leurs reculs. Merci enfin aux doctorants et ingénieurs de l'équipe, qui ont partagé avec moi le stress des différentes présentations et autres deadlines d'articles. On ne le dit jamais assez, une thèse, c'est pas une sinécure... Et puis merci à tous les collègues embarqués dans la même galère : bravo, merci et courage au 4 autres compagnons ayant démarré cette folle aventure avec moi : Adel, Nico, Rémi et Rémi. Bonne continuation à vous, en espérant que vous garderez le même souvenir d'Inria que moi.

Un clin d'œil pour Clément, frère de thèse, qui a su m'accueillir à Lille malgré son accent sudiste, qui m'a fait découvrir de nombreux endroits sympathiques. J'ai beaucoup apprécié nos longues conversations, autant scientifiques que footballistiques. J'espère que tu mèneras une longue et brillante carrière académique, en démarrant par Milan. Obrigado également à Antonio, mon frère de bière, compagnon de nuit et de braderie, qui m'a fait découvrir sa culture, son pays, et bien plus encore. J'ai été heureux de faire ta connaissance, en espérant te re-croiser un jour, ou dans une vie future. Je ne pourrai pas écrire ces remerciements sans penser à mes camarades de l'association des doctorants, Tilda. Tout particulièrement, je voudrais saluer Fabien et sa bonne humeur, avec qui j'ai eu le plaisir de travailler. Enfin, un immense merci à Julie, voisine de thèse, compagnon de pause café et motivatrice intarissable. Merci de m'avoir épaulé autant dans ma thèse que dans ma vie perso.

Enfin, merci à ma famille, qui m'a soutenu tout au long de ces quatre dernières années, dans les bons et les mauvais moments. Merci d'avoir été auprès de moi, d'avoir cru en moi, lorsque moi-même je n'y croyais plus. Encore une fois, merci à vous. Merci également à tous mes amis sudistes, qui ont sû rester en contact avec moi, malgré la grande distance séparant mon sud natal de cette terre d'accueil qu'a été le Nord. Je ne citerai personne de peur d'oublier quelqu'un, exception faite bien sûr de Laetitia, qui a été mon accroche immuable tout au long du doctorat. Tu as su rester auprès de moi malgré la distance, l'eau qui coulait sur les ponts, et nos chemins qui tendaient à bifurquer. Tu as été présente au bout du téléphone lorsque j'allais bien, lorsque j'allais mal, quand j'avais envie de parler politique, ou simplement de la pluie et du beau temps pour oublier les difficultés de la journée passée. Plus encore, c'est sans nul doute grâce à toi si aujourd'hui, je suis docteur... (le 13ème ?) C'est toi qui, au moment où j'avais envie de baisser les bras, là où je ne voyais plus aucun espoir ni réconfort, a réussi non seulement à me donner la force de terminer ma thèse, mais qui m'a en plus redonné l'envie de sourire, et qui m'a donné un but en m'attendant patiemment dans notre nouvelle contrée toulousaine. Merci, merci infiniment.

Table des matières

1 Introduction **1**
1.1 Problématique . . . 2
1.2 Défis pour maintenir la qualité de service lors de l'évolution . . . 2
1.3 Proposition . . . 3
1.4 Organisation du document . . . 4
1.5 Liste des publications liées à cette thèse . . . 5

I Contexte et Motivations **7**

2 Contexte **9**
2.1 Les architectures orientées services . . . 9
2.1.1 Définitions . . . 9
2.1.2 Organisation des architectures orientées services . . . 10
2.2 La Qualité de Service . . . 12
2.2.1 Qualité du logiciel . . . 12
2.2.2 Méthodes de détermination de la qualité de service . . . 14
2.2.3 Contrats de Qualité de Service . . . 17
2.3 Évolution des systèmes logiciels . . . 17
2.3.1 Définition de l'évolution . . . 17
2.3.2 Caractéristiques d'une évolution . . . 18
2.4 Conclusion . . . 22

3 État de l'art **25**
3.1 Processus de développement . . . 25
3.1.1 Les processus de développement généralistes . . . 26
3.1.2 Les processus de développement spécialisés SOA . . . 27
3.1.3 Prise en compte de la Qualité de Service . . . 28
3.1.4 Prise en compte de l'évolution . . . 28
3.1.5 Comparatif et limitations . . . 29
3.2 Analyse d'impact . . . 29
3.2.1 Fondements de la causalité . . . 29
3.2.2 Application de la causalité : les analyses d'impact . . . 30
3.2.3 Analyse d'impact pour la qualité de service . . . 32
3.2.4 Comparatif et limitations . . . 32
3.3 Conclusion de l'état de l'art . . . 33

4 Présentation du cas d'étude **37**
4.1 Séduite, un système de diffusion d'informations à base de processus métiers 37
4.2 PicWeb, un processus métier de recherche d'images . . . 39
4.2.1 Description de PicWeb . . . 39
4.2.2 Évolution de PicWeb . . . 40

II Contributions 41

5 Un processus de développement pour le maintien de la qualité de service 43
- 5.1 Motivations . . . 43
- 5.2 Définition des acteurs . . . 44
- 5.3 Description du processus de développement . . . 44
- 5.4 Coopération entre acteurs . . . 45
- 5.5 Conclusion du chapitre . . . 46

6 Modélisation d'un système à base de processus métiers 51
- 6.1 Défis . . . 52
- 6.2 Modélisation du système . . . 52
  - 6.2.1 Univers . . . 52
  - 6.2.2 Variables et types . . . 52
  - 6.2.3 Messages et Opérations . . . 53
  - 6.2.4 Services . . . 54
  - 6.2.5 Activités . . . 55
  - 6.2.6 Relations d'ordre . . . 56
  - 6.2.7 Processus métier . . . 57
  - 6.2.8 Système . . . 57
- 6.3 Causalité de l'exécution d'un système . . . 58
  - 6.3.1 Mise en œuvre de la causalité . . . 59
  - 6.3.2 Relations causales fonctionnelles . . . 59
  - 6.3.3 Exemple . . . 60
- 6.4 Déduction des relations causales d'un système . . . 61
  - 6.4.1 Méthodologie . . . 61
  - 6.4.2 Expression des règles causales . . . 62
- 6.5 Conclusion du chapitre . . . 63

7 Modélisation de la qualité de service pour l'évolution 65
- 7.1 Motivations . . . 65
- 7.2 Modélisation de la qualité de service d'un système . . . 66
  - 7.2.1 Propriété, unité et critère de comparaison . . . 67
  - 7.2.2 Domaine d'application . . . 67
  - 7.2.3 Valeur de propriété . . . 68
  - 7.2.4 Influence et ressource . . . 68
  - 7.2.5 Discussion . . . 69
- 7.3 Définition des relations causales de propriété . . . 69
  - 7.3.1 Relations d'environnement . . . 71
  - 7.3.2 Relations de dérivation de propriétés . . . 71
  - 7.3.3 Relations d'agrégation de valeurs . . . 73
  - 7.3.4 Discussion . . . 74
- 7.4 QoS4Evol, un langage de description de la qualité de service . . . 74
  - 7.4.1 Présentation de QoS4Evol . . . 75
  - 7.4.2 Définition du langage . . . 75
  - 7.4.3 Caractéristiques du langage . . . 76
- 7.5 Déduction des règles causales d'une propriété . . . 77
  - 7.5.1 Principe . . . 77
  - 7.5.2 Obtention des relations causales d'environnement . . . 78
  - 7.5.3 Obtention des relations causales de dérivation . . . 79

7.5.4 Obtention des relations causales d'agrégation . . . 79
7.5.5 Discussion . . . 80
7.6 Conclusion du chapitre . . . 81

8 Analyse de l'évolution du logiciel orientée QoS 83
8.1 Présentation générale du processus d'évolution . . . 84
8.1.1 Enjeux . . . 84
8.1.2 Spécification du processus d'évolution . . . 84
8.1.3 Hypothèses . . . 85
8.2 Un langage pour l'évolution des processus métier . . . 86
8.3 Spécification de l'analyse de l'évolution . . . 87
8.3.1 Aperçu des différentes étapes . . . 88
8.3.2 Étape 1 : mise à jour du modèle causal . . . 89
8.3.3 Étape 2 : analyse causale . . . 92
8.3.4 Étape 3 : re-vérification . . . 93
8.3.5 Discussion . . . 96
8.4 Résultat de l'analyse et prise de décision . . . 96
8.4.1 Enjeux . . . 97
8.4.2 Vérification des contrats . . . 97
8.4.3 Détermination des causes racines . . . 98
8.4.4 Déploiement et retour en arrière (roll-back) . . . 99
8.4.5 Discussion . . . 99
8.5 Conclusion du chapitre . . . 101

III Validation 103

9 Mise en œuvre et utilisation de Smile 105
9.1 Réalisation de Smile . . . 105
9.1.1 Présentation du canevas . . . 106
9.1.2 Besoins des utilisateurs de Smile . . . 106
9.1.3 Architecture de Smile . . . 106
9.2 Smile pour la description du système . . . 109
9.2.1 Import de système . . . 109
9.2.2 Description des règles causales . . . 111
9.3 Smile pour la qualité de service . . . 113
9.3.1 Principe . . . 113
9.3.2 Description d'une propriété . . . 113
9.3.3 Gestion des règles causales de propriété . . . 114
9.3.4 Collecte des données . . . 115
9.3.5 Description d'un contrat . . . 116
9.4 Smile pour l'évolution . . . 117
9.4.1 Description d'une évolution . . . 117
9.4.2 Analyse causale de l'évolution . . . 118
9.4.3 Contrôle à l'exécution . . . 119
9.5 Limites . . . 119
9.6 Conclusion du chapitre . . . 120

10 Evaluation **121**
10.1 Défis . . . 122
10.2 Cas d'étude : le Système de gestion de crises . . . 122
10.2.1 Contexte du cas d'étude . . . 122
10.2.2 Description du scénario . . . 123
10.2.3 Cas d'utilisation . . . 124
10.3 Implémentation . . . 127
10.4 Évolutions du scénario . . . 128
10.4.1 Évolutions pour la construction du système . . . 128
10.4.2 Évolutions du processus *Gestion de la Mission* . . . 129
10.5 Évaluation quantitative des contributions . . . 132
10.5.1 Comparaison des éléments à re-vérifier . . . 132
10.6 Discussion . . . 135
10.7 Conclusion du chapitre . . . 135

IV Conclusion **137**

11 Conclusion et Travaux Futurs **139**
11.1 Résumé des contributions . . . 139
11.2 Perspectives à court terme . . . 140
11.3 Perspectives à long terme . . . 141
Bibliographie . . . 145

Table des figures

2.1 Exemple de détermination par dérivation. 16
2.2 Exemple de détermination par agrégation. 16

4.1 Utilisation de Séduite au cours de la nuit de l'info. 38
4.2 Architecture de *Séduite*. 38
4.3 Évolutions de PicWeb. 39

5.1 Schéma du processus de développement Blink. 46

6.1 Extrait du méta-modèle de Smile : variables et types. 53
6.2 modélisation du service Helper. 54
6.3 Extrait du méta-modèle de Smile : services, opérations et messages. 55
6.4 Extrait du méta-modèle de Smile : activités. 55
6.5 Extrait du modèle de PicWeb : activité receive. 55
6.6 Extrait du modèle de PicWeb : relations d'ordre. 57
6.7 Extrait du méta-modèle : relations d'ordre. 57
6.8 Extrait du méta-modèle : processus métier. 58
6.9 Modèle causal fonctionnel de PicWeb. 60
6.10 Procédé de déduction des relations causales. 62

7.1 Modèle et méta-modèle d'une propriété. 67
7.2 Modèle et méta-modèle du domaine d'application. 68
7.3 Méta-Modèle de propriété : Valeurs de propriété. 69
7.4 Modèle du temps de réponse : Valeurs de propriété. 70
7.5 Modèle et méta-modèle des facteurs d'influence. 71
7.6 Modèle causal enrichi avec les relations d'environnement. 72
7.7 Modèle causal enrichi avec les relations de dérivation. 72
7.8 Modèle causal enrichi avec les relations d'agrégation. 73
7.9 Définition du Temps de Réponse. 75
7.10 Chaîne de déduction des relations causales de propriété. 78

8.1 Processus d'évolution. 85
8.2 Évolution de PicWeb. 88
8.3 Étapes de l'analyse de l'évolution. 88
8.4 Traduction de l'évolution en actions sur le modèle causal. 90
8.5 Application à PicWeb de l'analyse causale. 93
8.6 Application à PicWeb de l'analyse causale. 94
8.7 Obtention des données brutes. 95
8.8 Contrat de QoS qualifiant le temps de réponse de PicWeb. 98
8.9 Détermination des causes racines de la violation du contrat de PicWeb. . . 100

9.1 Diagramme des cas d'utilisation de Smile. 107
9.2 Architecture de Smile. 108
9.3 Procédé dirigé par les modèles de l'approche Smile. 109
9.4 Import de PicWeb. 110
9.5 Extrait de fichier composite. 111

9.6 Méta-modèle du modèle causal. 112
9.7 Extrait d'une règle causale écrite en ATL. 112
9.8 Processus de la partie QoS de SMILE. 114
9.9 Chaîne d'outillage pour la description d'une propriété. 115
9.10 Méta-Modèle de traces de SMILE. 115
9.11 Modélisation des contrat de qualité de service. 116
9.12 Méta-Modèle d'évolution de SMILE. 117
9.13 Utilisation de l'éditeur d'évolutions de SMILE. 118

10.1 Description des différents rôles opérationnels. 123
10.2 Description des différents rôles d'observation. 124
10.3 Description des différents rôles stratégiques. 124
10.4 Architecture du système de gestion de crises. 127
10.5 Graphe de dépendance des processus métiers du système de gestion de crises. 127
10.6 Évolution de la taille du modèle causal au cours de la construction du système. 128
10.7 Processus métier du cas d'utilisation "Exécution de la mission". 129
10.8 Évolution du processus métier *Gestion de la mission* : UnavailableIntRes. . 130
10.9 Évolution du processus métier *Gestion de la mission* : UnavailableExtRes. . 131
10.10 Évolution du processus métier *Gestion de la mission* : RehandleOnChange. 132
10.11 Extrait du modèle causal du processus métier *Gestion de la mission* après l'évolution *RehandleOnChange*. 133

Liste des tableaux

2.1 Taxonomie d'une évolution : la dimension "Où". 19
2.2 Taxonomie d'une évolution : la dimension "Quoi". 20
2.3 Taxonomie d'une évolution : la dimension "Quand". 21
2.4 Taxonomie d'une évolution : la dimension "Comment". 22
2.5 Récapitulatif des hypothèses de travail. 23

3.1 Comparatif des différents processus de développement. 29
3.2 Comparatif des différentes analyses d'impact. 35

5.1 Rôles intervenants dans la réalisation d'un système. 45
5.2 Description des étapes 0 à 2 du processus de développement (Blink). . . . 47
5.3 Description des étapes 3 à 5 du processus de développement (Blink). . . . 48
5.4 Description de l'étape 6 du processus de développement (Blink). 49

6.1 Règle causale pour les relations causales de paramètre d'entrée. 62
6.2 Règle causale pour les relations causales de paramètre de sortie. 63

7.1 Formules d'agrégation du temps de réponse. 76
7.2 Règle de production des relations causales d'environnement. 78
7.3 Règle causale d'environnement générée. 78
7.4 Règle de production des relations causales de dérivation. 79
7.5 Règle causale de dérivation générée. 79
7.6 Règle de production des relations causales d'agrégation. 80
7.7 Règle causale d'agrégation générée. 80

8.1 Liste des opérations d'évolution. 86
8.2 Correspondance entre opérations d'évolution et actions du modèle causal. . 90
8.3 Génération de la mise à jour du modèle causal de PicWeb. 91
8.4 Description de l'opération *collecte*. 92

9.1 Description des fonctionnalités par module. 108

10.1 Nombre de valeurs de propriété à re-vérifier pour chaque méthode d'analyse. 134

Chapitre 1

Introduction

Sommaire

1.1 Problématique 2
1.2 Défis pour maintenir la qualité de service lors de l'évolution . . 2
1.3 Proposition 3
1.4 Organisation du document 4
1.5 Liste des publications liées à cette thèse 5

L'informatique prend aujourd'hui une place de plus en plus grande dans le quotidien des personnes, et dans le quotidien des entreprises. L'outil numérique développé au cours du dernier siècle est aujourd'hui omniprésent dans la société. Cela se traduit par l'existence de nombreux systèmes logiciels, allant du simple jeu sur un téléphone portable au moteur de recherche utilisé massivement. Ces systèmes, qui autrefois étaient monolithiques et centralisés, sont devenus avec l'apogée de l'Internet et l'augmentation de la taille des organisations des entreprises des systèmes distribués, de taille conséquente, et à la complexité croissante.

Les architectures orientées services (SOA) constituent une solution permettant de maîtriser cette complexité. La conception d'applications se matérialise par le biais de la chorégraphie ou de l'orchestration de services à l'aide de processus métiers, offrant ainsi un mécanisme de composition permettant de gagner en abstraction et de réduire la complexité. En ce sens, concevoir une application reposant sur les principes des architectures orientées services consiste à mettre en relation les flots de contrôle et de données de ces services.

Au delà de la nécessité de produire des systèmes correspondant aux besoins des utilisateurs, la qualité de service (QoS), définie par Crnkovic *et al.* comme *"une caractérisation de tout ou d'une partie du système, selon une préoccupation particulière"* [Crnkovic 2005], constitue une préoccupation à gérer au cours du développement de logiciels. L'analyse et le maintien de la QoS d'un système sont des opérations critiques, car la QoS sert de base à l'établissement de contrats de service entre fournisseurs et utilisateurs, et un non-maintien de la QoS entrainerait une rupture de contrat. De par la composition des services, la détermination de la QoS d'une orchestration s'effectue par calcul, dépendant ainsi de la QoS des services la constituant. C'est ce que l'on appelle *l'agrégation* [Cardoso 2004].

Les logiciels que nous considérons ici, dont la taille et la complexité échappent à la compréhension d'une seule personne, sont amenés à être modifiés tout au long de leur cycle de vie. Les modifications opérées, que l'on appelle dans ce document *évolutions*, sont la conséquence/réaction d'un changement de besoins de la part des utilisateurs, d'un changement survenu dans l'environnement d'exécution ou encore dans l'architecture du système. Ces évolutions, de par la modification du comportement du système qu'elles entraînent, impliquent de s'assurer que la QoS n'a pas été détériorée. Ainsi, lors de chaque évolution, les changements effectués sur le logiciel impliquent de re-vérifier la QoS de l'ensemble du système.

1.1 Problématique

Dans un contexte où les attentes en termes de fonctionnalités et de qualités du logiciel sont fortes, faire évoluer ce dernier est un processus critique, tant par l'effet que l'évolution peut avoir que par la nécessité d'un processus court permettant d'être réactif à l'égard des utilisateurs. Il est parfois même nécessaire d'appliquer l'évolution au cours même de leur exécution. Dans ce cas, il est vital d'établir un processus d'évolution court et minimal, pour ne pas parasiter ou ralentir son fonctionnement. Il s'agit ici d'un problème complexe, où la nécessité d'effectuer une évolution maintenant les propriétés de QoS implique une collaboration proche entre les différentes expertises de l'équipe de développement, notamment entre l'expertise de la QoS et celle de l'évolution. Le processus d'évolution nécessite d'être court dans sa réalisation, tout en s'assurant du maintien de la QoS du logiciel. L'objectif est donc ici de minimiser la re-vérification des propriétés de QoS, en déterminant de manière précise quelles parties du logiciel ont été réellement affectées par l'évolution.

Dans le cadre de cette thèse, nous nous intéressons tout particulièrement aux systèmes reposant sur une architecture orientée services, pour lesquels le comportement est représenté à l'aide de *processus métiers*. Les mécanismes d'évolution à définir doivent permettre d'établir l'effet d'une évolution sur le reste du système, afin d'éviter un ensemble de dépendances cachées entre les éléments. De plus, le processus d'évolution devra garantir le maintien des contrats de qualité de service. Il s'agit donc non seulement de comprendre l'effet d'une évolution sur le reste du système, mais également sur les différentes propriétés de QoS ayant un intérêt pour l'équipe de développement ou pour la communauté d'utilisateurs. Enfin, ces mécanismes d'évolution devront prendre en compte la répartition des compétences et des expertises de l'équipe de développement, en définissant les différents rôles nécessaires à un processus d'évolution fiable en termes de QoS, et en identifiant les points d'interaction entre ces différents rôles pour partager leurs expertises propres.

1.2 Défis pour maintenir la qualité de service lors de l'évolution

L'objectif de cette thèse est de fournir à l'équipe de développement et de maintenance les outils permettant de s'assurer que la qualité de service est maintenue au cours des différentes évolutions du logiciel, tout au long de son cycle de vie. Le maintien de la QoS lors d'une évolution est une tâche complexe, nécessitant la collaboration de différentes expertises telles que l'expertise du métier, de l'environnement d'exécution, ou encore de la propriété de QoS considérée.

De plus, le contexte des architectures orientées services positionne notre travail dans un environnement où l'agilité du développement est pré-dominante. En effet, avec des changements fréquents des besoins des utilisateurs, mais également des services externes utilisés, la fiabilité de la QoS et son maintien dans de tels systèmes sont des valeurs importantes.

Nous énumérons dans cette section les différents défis liés à l'évolution d'un logiciel en terme de qualité de service :

1. **Collaboration entre acteurs :** pour gérer une évolution et assurer le maintien de la qualité de service d'un système, la connaissance et la coopération des acteurs œuvrant au niveau du système développé, des propriétés de qualité de service étudiées, et des évolutions sont nécessaires. En effet, effectuer une évolution tout en s'assurant du maintien de la qualité de service requiert une connaissance transversale à ces différents domaines, que seule la coopération peut apporter. L'identification des différents rôles et des différentes expertises au sein de l'organisation permettront de concevoir et de

faire évoluer un système où la qualité de service est l'une des préoccupations. Cette identification est nécessaire dans le but de responsabiliser et de définir les différentes collaborations à mettre en place.

2. **Interactions entre les différentes parties d'un logiciel :** le logiciel réalisé par l'équipe de développement est bien souvent constitué de sous-parties, qui interagissent pour proposer des fonctionnalités attendues par les utilisateurs. Ces interactions influent sur la qualité du service du système, et sont bien souvent implicites. L'identification explicite des interactions entre les éléments d'un système permet d'améliorer la compréhension du système dans son ensemble, notamment lors d'une évolution.
3. **Minimisation de la vérification :** afin de s'assurer du maintien de la qualité de service, l'étape de l'évolution est accompagnée d'une étape de vérification, où chaque propriété de QoS est contrôlée pour l'ensemble du système. Toutefois, avec l'augmentation de la taille des systèmes, cette phase peut devenir chronophage. Ce phénomène s'accentue si l'on considère des systèmes développés de manière agile, où les évolutions sont fréquentes. Dans ces conditions, l'étape de vérification doit être la plus rapide possible afin de ne pas interférer avec le bon fonctionnement du système et de ne pas dégrader ses performances. Nous devons rendre l'étape de vérification la plus rapide possible, en minimisant le nombre d'éléments à re-vérifier.
4. **Identification de la cause de la violation d'un contrat :** lorsqu'une évolution viole un contrat de QoS, l'équipe de développement a pour tâche de détecter l'origine de cette violation, dans le but de pouvoir corriger l'évolution. Partant du contrat violé, il s'agit donc d'identifier l'ensemble des interactions du logiciel impliquées dans la dégradation de la QoS, afin de cibler au plus près son origine. Pour cela, nous cherchons à établir les dépendances existant entre l'exécution du logiciel et sa qualité de service.

1.3 Proposition

Pour répondre à ces défis, nous présentons dans cette thèse un ensemble de contributions constituant un mécanisme d'évolution à la fois **agile** dans la réaction aux besoins de l'utilisateur, **fiable** dans le maintien de la qualité de service une fois l'évolution réalisée, et **minimale**, dans le sens où le processus de vérification de la qualité de service doit être le plus court possible dans le cas où l'évolution s'opèrerait à l'exécution.

Ces contributions s'articulent autour d'un cycle de développement agile, nommé Blink, portant une attention particulière sur le maintien de la QoS lors de l'évolution. Notre cycle de développement met au centre des préoccupations la notion d'évolution comme élément à prendre en compte dès les premières étapes de conception du logiciel. En identifiant les différents rôles nécessaires au maintien de la QoS de l'évolution, Blink permet d'identifier l'ensemble des informations devant être fourni, et de délimiter les points d'interaction dans le but d'échanger ces informations.

En complément du processus de développement, nous présentons Smile, un canevas de développement mettant en œuvre Blink. Celui-ci implémente une boucle d'évolution, permettant de concevoir, d'analyser et d'appliquer une évolution, en portant une attention particulière au maintien de la qualité de service. Le but de cette boucle est de déterminer au plus tôt l'effet que peut avoir une évolution sur le reste du système et sur sa QoS. Pour cela, notre analyse de l'évolution consiste à déterminer les relations de causalité qu'une évolution a sur le reste du système. Il s'agit ici d'utiliser ces relations causales, dans le but d'identifier les différents éléments affectés et de re-vérifier par la suite si la qualité de service de ces éléments a été affectée. Cela permet ainsi de pouvoir identifier dans un premier temps la

portée d'une évolution, afin de pouvoir dans un second temps minimiser le processus de re-vérification.

En résumé, les contributions de la thèse sont les suivantes :

- **Blink, un processus de développement collaboratif pour le maintien de la QoS lors de l'évolution** : nous définissons dans ce document un processus de développement pour lequel la problématique du maintien de la QoS lors de l'évolution est au centre des préoccupations. Pour cela, nous définissons un ensemble d'acteurs, devant collaborer tout au long du cycle de développement en partageant les informations propres à leur expertise qui pourraient être utile au maintien.
- **Une méthode de détermination des interactions entre les différentes parties du système** : nous définissons dans un premier temps quels types d'influence nous pouvons retrouver dans un système et au niveau de la qualité de service. Puis, nous présentons une méthode permettant, à partir d'un système et de la description d'une propriété de qualité de service, de déterminer de manière automatique ces influences.
- **Une analyse d'évolution permettant de déterminer l'effet d'une évolution sur la QoS du système** : en réutilisant les modèles représentant les interactions existantes au sein d'un système, nous présentons une analyse permettant, à partir d'une évolution, de savoir l'effet de cette dernière au sein du système, mais également de déterminer si un contrat de QoS a été violé ou non, gage du maintien de la QoS.
- **Smile, un canevas de développement automatisant les différents traitements nécessaires à Blink** : nous introduisons dans ce document notre canevas supportant les différents points abordés ci-dessus, et montrons ce que ce canevas de développement apporte lors de l'évolution.

Afin de valider l'apport de notre approche dans le maintien de la QoS lors de l'évolution, nous appliquons Blink et Smile sur un cas d'étude nommé *Système de gestion de crises*. Lors de cette validation, nous vérifions notamment si notre outil est en mesure de détecter le non-maintien de la QoS lors de l'évolution du cas d'étude.

1.4 Organisation du document

Ce document est organisé en trois parties :

1. La première partie se concentre sur le contexte et les motivations de la thèse. Dans le chapitre 2, nous posons le contexte de travail, d'un point de vue conceptuel et technologique. Nous définissons ici les notions liées aux architectures orientées services, à la qualité de service, et à l'évolution des systèmes logiciels. Dans le chapitre 3, nous dressons l'état de l'art des travaux de recherche sur ce domaine. Pour cela, nous nous focalisons sur deux axes, à savoir les cycles de développement d'un logiciel, et les différentes analyses d'impact pouvant déterminer l'effet d'une évolution, ou d'un changement en général. Enfin, dans le chapitre 4, nous décrivons PicWeb, que nous utilisons comme exemple fil rouge nous permettant d'illustrer les différentes contributions de la thèse. Après avoir présenté l'exemple, nous expliquons comment un problème de qualité de service est survenu lors de son évolution.
2. La deuxième partie regroupe les contributions de la thèse, réparties en quatre chapitres. Dans le chapitre 5, nous présentons Blink, un processus de développement orienté pour le maintien de la qualité de service au cours des différentes évolutions. Nous mettons notamment en évidence les différents acteurs nécessaires au maintien de la QoS. Les chapitres 6 à 8 se focalisent alors successivement sur chaque acteur

impliqué dans Blink. **Le chapitre 6** se centre sur la réalisation d'un système, et sur le rôle de l'architecte du système. Nous définissons un méta-modèle permettant de représenter un système à base de processus métiers. Puis, nous introduisons la notion de causalité dans un système, permettant de représenter les interactions entre ses différentes parties, et dressons une liste de causalités que l'on peut retrouver. **Le chapitre 7** est focalisé sur le rôle de l'expert en qualité de service. Nous présentons ici notre méta-modèle pour représenter une propriété, ainsi que *QoS4Evol*, un langage permettant à l'expert en QoS de la définir de manière textuelle. De manière similaire au chapitre précédent, nous définissons ensuite la notion de causalité pour les propriétés de qualité de service. Nous listons les différents types de relation causale que nous pouvons retrouver pour une propriété de QoS, et les illustrons sur PicWeb. Enfin, nous présentons une méthode permettant d'extraire automatiquement les relations causales de la description d'une propriété. **Le chapitre 8** porte sur l'architecte de l'évolution. Nous introduisons un langage d'évolution des processus métiers, permettant d'ajouter ou de supprimer des fonctionnalités. Puis, nous nous focalisons sur l'étape d'analyse d'une évolution, où nous utilisons les relations causales présentées dans les chapitres précédents pour déterminer l'effet d'une évolution sur la qualité de service d'un système, et ainsi qualifier si une évolution permet le maintien de la QoS ou non.

3. La troisième partie du document s'intéresse à l'évaluation de notre approche. Le chapitre 9 détaille la mise en œuvre des différentes contributions au travers de notre canevas, Smile, tandis que le chapitre 10 est une évaluation de notre approche sur un autre cas d'étude, le *système de gestion de crise*. Ce cas d'étude a l'avantage d'être de taille plus importante que PicWeb, d'être constitué de plusieurs processus métiers, et d'être doté d'une base d'évolutions conséquentes. Enfin, nous terminons avec le chapitre 11, où nous faisons le bilan des différents chapitres de la thèse, et présentons quelques pistes à explorer dans de futurs travaux.

1.5 Liste des publications liées à cette thèse

Les communications mentionnées ci-dessous ont été réalisées dans des manifestations avec comité de sélection.

Communications internationales

– **A Causal Model to predict the Effect of Business Process Evolution on Quality of Service.** Alexandre Feugas, Sébastien Mosser et Laurence Duchien. Dans *International Conference on the Quality of Software Architectures (QoSA'13)*, pages 143-152, Vancouver, Canada, juin 2013. (Rang A selon le classement CORE Austalian)

Communications nationales

– **Déterminer l'impact d'une évolution dans les processus métiers.** Alexandre Feugas, Sébastien Mosser, Anne-Françoise Le Meur et Laurence Duchien. Dans *IDM*, pages 71-76, Lille, France, juin 2011.

Posters

– **Un processus de développement pour contrôler l'évolution des processus métiers en termes de QoS.** Alexandre Feugas, Sébastien Mosser et Laurence Duchien. Poster dans *GDR GPL*, pages 237-238, Rennes, France, Juin 2012.

Première partie

Contexte et Motivations

Chapitre 2

Contexte

Sommaire

2.1 Les architectures orientées services 9
2.1.1 Définitions 9
2.1.2 Organisation des architectures orientées services 10
2.2 La Qualité de Service 12
2.2.1 Qualité du logiciel 12
2.2.2 Méthodes de détermination de la qualité de service 14
2.2.3 Contrats de Qualité de Service 17
2.3 Évolution des systèmes logiciels 17
2.3.1 Définition de l'évolution 17
2.3.2 Caractéristiques d'une évolution 18
2.4 Conclusion 22

Nous avons présenté dans l'introduction l'objectif de cette thèse. Dans ce chapitre, nous présentons ses enjeux d'un point de vue conceptuel et technologique : nous abordons dans un premier temps les architectures orientées services. Puis, nous présentons la notion de qualité de service, ainsi que les approches existantes permettant de l'étudier. Nous introduisons enfin la définition de l'évolution : quelles notions se trouvent derrière ce concept, comment l'équipe de développement s'y prend pour faire évoluer un logiciel.

2.1 Les architectures orientées services

2.1.1 Définitions

Les *architectures orientées services* (SOA) sont un nouveau paradigme de conception apparu dans la fin des années 90. Il existe différentes définitions pour caractériser les architectures orientées service. Parmi elles, nous pouvons retenir la définition du W3C [Haas 2004] :

Définition 1

> ***Architecture Orientée Services (W3C)*** *: "une architecture orientée services regroupe un ensemble de composants qui peuvent être invoqués, et pour lesquels les descriptions de leurs interfaces peuvent être publiées et découvertes".*

Cette définition peut être complétée par une autre définition, d'IBM :

Définition 2

> ***Architecture Orientée Services (IBM)*** *: "une architecture orientée services est un canevas d'information qui considère les applications métiers de tous les jours pour les fragmenter en fonctions et processus métiers individuels, appelés services. Une architecture orientée service permet la construction, le déploiement et l'intégration de ces*

> *services indépendamment des applications et des plate-formes d'exécution".*

De ces deux définitions, nous pouvons comprendre que la notion de *service*, comme entité de calcul invocable, et pour lequel une interface est définie, est principale. Ces services sont indépendants, ils sont déployés pour pouvoir par la suite être publiés, découverts et invoqués.

Il existe différentes technologies dites orientées services, reprenant les principes des architectures orientées services plus ou moins fidèlement. Parmi elle, nous retiendrons notamment les web-services [Kuebler 2001], où l'on considère que les services sont définis et exposés en utilisant les standards du Web.

Propriétés remarquables

Les architectures orientées service ont été définies pour concevoir des systèmes logiciels pour lesquels les entités de première classe sont les services, des entités autonomes, invocables, dont l'interface est clairement explicitée. Plusieurs travaux ont proposé la caractérisation des architectures orientées services [Papazoglou 2003, Huhns 2005, Breivold 2007, Papazoglou 2007]. Dans la suite, nous recensons les propriétés suivantes :

- **Couplage lâche des éléments :** afin de pouvoir faciliter la reconfiguration d'un service, *e.g.*, son remplacement, les éléments d'un système doivent bénéficier d'un couplage lâche, c'est-à-dire qu'il n'est pas nécessaire pour un appelant à un service de connaître la structure ou l'implémentation de ce dernier pour pouvoir l'utiliser. De même, une façon de permettre un couplage lâche au niveau d'un service consiste à séparer l'interface de l'opération et son implémentation. Cela offre la possibilité de pouvoir modifier l'implémentation, ou de pouvoir exposer facilement l'interface du service.
- **Neutralité technologique :** pour permettre à tout système d'interagir avec un service, la dépendance à un contexte technologique donné est à proscrire. Cette propriété s'inscrit dans un contexte d'inter-opérabilité, facilitant ainsi l'établissement de partenariats entre différentes organisations.
- **Transparence de la localisation :** un service doit être explicitement atteignable, de façon à permettre à tout client de pouvoir l'invoquer, dans son périmètre de disponibilité. Dans le cas des web-services, un service doit être disponible à n'importe quel point de l'Internet.
- **Modularité et Composabilité :** les architectures orientées services mettent en avant le côté réutilisable des services. Par le biais d'une interface exposée et de l'inter-opérabilité des services, le concept de service a été conçu pour facilité la modularité, la réutilisabilité et la composabilité d'une application. Ainsi, il est possible de délimiter le cadre de modification d'un service par son aspect modulaire, l'exposition de son interface permet de réutiliser le service dans une autre application, et, partant de différents services, il est possible de composer un nouveau service avec une fonctionnalité différente, en utilisant d'autres services par le biais d'invocations.

Pour atteindre cet ensemble de propriétés, un éco-système des architectures orientées services a été défini par Papazoglou [Papazoglou 2003] comme étant constitué de trois couches. Dans la suite de cette section, nous détaillons cette organisation en trois couches.

2.1.2 Organisation des architectures orientées services

Dans son travail sur la définition des architectures orientées service, Papazoglou a imaginé l'éco-système des SOA comme étant constitué de trois couches, où chaque couche s'appuie sur les définitions des couches précédentes. La première couche concerne les services à

leur niveau le plus simple. Ici, on s'intéressera aux notions d'interface, de publication et de sélection d'un service. La seconde couche s'appuie sur les services basiques pour définir de nouveaux services par composition. On appelle composition de services la réalisation d'un service en réutilisant d'autres services, le plus souvent pour construire une fonctionnalité de complexité supérieure. Ici, on attachera plus d'importance à la sémantique du nouveau service composé, mais également à des caractéristiques non fonctionnelles d'un service telles que les propriétés de qualité de service. Enfin, le dernière couche considère la gestion des services. Ici, il s'agit en fait de considérer l'infrastructure dans son ensemble, en ayant en tête des préoccupations telles que le cycle de vie du service, son contrôle, et la gestion du changement d'un service.

2.1.2.1 Services simples

Définition d'un service

Comme introduit dans les propriétés des SOA, une opération d'un service, ou par abus de langage un service, a une interface explicitement définie. Cette interface définit a minima la signature de l'opération. Concrètement, il peut s'agir d'une structure écrite dans un *Language de Description d'Interface* (IDL) [Bachmann 2002]. Dans le cadre des web-services, le langage adopté est le *Langage de Description des Web Services* (WSDL) [Chinnici 2007]. Il est ainsi possible d'y décrire des types de données, les signatures des différentes opérations, tout comme consigner la localisation du service par le biais d'un *identifiant de ressource* (*Uniform Resource Identifier*, *URI* en abbrégé) [Coates 2001]. Un service est en soi autonome et indépendant. C'est cette caractéristique qui permet aux architectures orientées services d'offrir un couplage lâche entre les éléments.

Interactions d'un service

Au niveau des interactions entre les services, il est important de différencier deux acteurs différents. Ces acteurs auront des préoccupations différentes :

- **Fournisseur de Service :** il est en charge de l'implémentation des différentes opérations du service. Il doit notamment prendre en compte la qualité de service, en adoptant une politique, cette police pouvant aller d'une politique de "meilleur effort" (où aucun niveau minimal de qualité est à atteindre), jusqu'à mettre la qualité de service au rang de préoccupation majeure. Ce dernier scénario se rencontre fréquemment dans les logiciels à base de services. En effet, dans un système où les services sont publiés et disponibles pour tous, le critère de différenciation et de choix entre deux services implémentant les mêmes fonctionnalités portera sur le niveau de qualité de service garanti pour l'opération. Il est important dans ce cas de fournir le service le plus efficient possible. Afin de pouvoir établir ces contraintes et ces attentes en termes de qualité de service, il est nécessaire pour le fournisseur du service et pour l'utilisateur du service d'établir un contrat, nommé *Service Level Agreement* (SLA) [Ludwig 2003], où chacune des parties s'engagent respectivement à fournir un certain niveau de qualité de service et à utiliser le service dans certaines conditions. Nous reviendrons sur ce point dans la section 2.2.3.
- **Utilisateur du service :** il s'agit de l'entité faisant appel à l'opération proposée par le fournisseur. Si l'utilisateur n'a a priori aucun engagement auprès du fournisseur de service, ce dernier doit être en mesure d'établir des limites dans l'utilisation du service invoqué, afin de pouvoir garantir un niveau de qualité de service satisfaisant. En effet, invoquer un service de façon disproportionnée (trop fréquemment par exemple) pourrait notamment détériorer les performances de ce dernier [Menasce 2002]. De

plus, l'utilisateur du service doit être le garant de la bonne intégration du service dans l'éco-système du logiciel qu'il réalise, notamment dans le cadre de transactions.

2.1.2.2 Composition de Services

La notion de service est enrichie avec le concept de *composition*, où l'on va chercher ici à réutiliser des services existants pour, en les combinant, produire un nouveau service ayant une fonctionnalité différente.

Dans les architectures orientées services, il existe principalement deux types de composition : la composition par orchestration et la composition par chorégraphie. Nous appelons orchestration de services un processus métier exécutable, pouvant interagir avec des services internes et externes [Peltz 2003]. Il s'agit en fait de la combinaison d'un flot de contrôle et d'un flot de données pour décrire la séquence des opérations à effectuer pour aboutir à un but donné. L'orchestration d'un service diffère de la chorégraphie dans le sens où cette dernière a un rôle de vision globale des interactions d'un système, là où l'orchestration représente le comportement d'une seule opération, et ses échanges avec d'autres services.

2.1.2.3 Gestion de services

Le dernier niveau organisationnel consiste à gérer les différents services existants. Dans cette partie, il s'agit de gérer le cycle de vie d'un service (marche/arrêt) ainsi que le contrôle de son exécution. Dans certains cas, il pourra également s'agir de gérer le remplacement à chaud de services par un autre, en cas de défaillance par exemple. Ces techniques sont implémentées dans les *Bus de Services pour les Entreprises* (ESB), véritable plate-forme d'exécution gérant entre autres l'exécution des services et leurs communication.

La modularité présente dans les architectures orientées services, associée à la notion d'inter-opérabilité, permet notamment de faciliter la sélection ou le remplacement d'un service par un autre aux fonctionnalités similaires. Dès lors, le concepteur a face à lui un panel de services équivalents parmi lesquels il devra en choisir un. Pour effectuer son choix, il peut désormais prendre en compte un nouveau critère de sélection : la qualité de service.

2.2 La Qualité de Service

Dans cette section, nous présentons la notion de qualité de service. Nous introduisons dans un premier temps le domaine de qualité du logiciel, avant de nous concentrer sur ce qu'est la qualité de service, quelles sont les différentes méthodes de détermination, et comment il est possible de la maintenir dans les SOA.

2.2.1 Qualité du logiciel

La notion de qualité du logiciel regroupe l'ensemble des caractéristiques visant à évaluer la manière dont le logiciel a été réalisé. Derrière ce terme, on retrouve un ensemble de caractéristiques ; certaines sont quantifiables, tandis que d'autres sont totalement à l'appréciation de l'humain. Afin de pouvoir caractériser les caractéristiques d'un logiciel qui ne sont pas propres à son exécution ou à sa logique métier, la notion de qualité du logiciel a été introduite dans le milieu des années 70 pour définir une classification d'un ensemble de propriétés de qualité du logiciel [Boehm 1976]. Cette notion fut normalisée par la norme ISO/CEI9126 [ISO 2001] et est régulièrement ré-actualisée [ISO 2004], pour s'inscrire désormais dans une démarche d'établissement des besoins de qualité et de son évaluation. Cette

démarche se nomme *Square*. On retrouve notamment dans cette norme une classification regroupant l'ensemble des propriétés de qualité autour de 5 catégories :

- **La capacité fonctionnelle** : il s'agit de la manière dont les fonctionnalités développées correspondent au cahier des charges. On s'intéresse notamment ici aux sous-caractéristiques d'*aptitude*, d'*exactitude* et d'*interopérabilité* [Brownsword 2004].
- **La fiabilité** : il s'agit de la faculté du logiciel à continuer d'être digne de confiance. Pour cela, on s'intéresse aux sous-caractéristiques de *maturité*, de *tolérance aux fautes* et de *capacité de récupération* [Ucla 2001, Clements 2003].
- **La facilité d'usage** : cette catégorie qualifie la capacité d'un utilisateur quelconque à utiliser le logiciel développé. On s'intéresse ici aux sous-caractéristiques d'*exploitabilité*, de *facilité d'apprentissage* et de *facilité de compréhension*
- **L'efficacité** : cette caractéristique étudie l'adéquation entre les moyens financiers et humains mis en œuvre pour réaliser le logiciel, et les besoins effectifs. On s'intéresse ici à l'efficacité en parlant des *ressources employées*, et des *temps de réalisation*
- **La maintenabilité** : cette catégorie regroupe toutes les caractéristiques propres à la faculté de modifier le logiciel pour ,par exemple, corriger un bogue. On s'intéresse ici aux sous-caractéristiques de *stabilité*, de *facilité de modification*, de *facilité d'analyse*, et de *facilité à être testé*.
- **La portabilité** : enfin, une dernière caractéristique propre à la qualité se centre sur la capacité à déployer une application sur une machine quelconque. On s'intéresse ici aux sous-caractéristiques de *facilité d'installation*, de *facilité de migration*, d'*adaptabilité* et d'*interchangeabilité*.

De toutes ces sous-caractéristiques, certaines sont mesurables et quantifiables, tandis que d'autres sont subjectives. Dans le cadre de cette thèse, nous considérons que les propriétés de qualité de service sont quantifiables, et dont un changement de valeur démontre un changement dans le comportement du système. À noter qu'il est important pour les différentes personnes gravitant autour du logiciel développé de prendre en compte la qualité du logiciel. Pour le client, il s'agit du gage d'un système performant. Pour les développeurs, développer un système de qualité permet de s'assurer que l'entreprise engrangera du profit au lieu d'essuyer des pertes.

Qualité de Service

Il existe de nombreuses définitions de la qualité de service. Parmi elles, nous retenons dans ce document la définition de Crnkovic et al. [Crnkovic 2005] :

Définition 3

> ***Qualité de Service*** *: On appelle propriété de qualité de service "une caractérisation de tout ou d'une partie du système, selon une préoccupation particulière".*

Il s'agit ici de propriétés portant directement sur le comportement du logiciel. Elles sont le plus souvent quantifiables. Dans le cadre de cette thèse, nous restreignons notre étude des propriétés de QoS aux propriétés quantifiables de manière non équivoque. Un exemple de propriétés de qualité de service correspondant à notre définition est les propriétés de performance telles que la sécurité, le temps de réponse ou la disponibilité [O'Brien 2007, Rosenberg 2006], pour lesquelles il existe bien souvent des outils d'analyse ou de contrôle à l'exécution permettant d'obtenir une valeur numérique. Tout particulièrement, on regardera du côté de [Lelli 2007, Fakhfakh 2012] pour des exemples de propriétés de qualité de service que l'on rencontre dans les SOA.

Il existe de nombreux langages de modélisation de la qualité de service. Les travaux les plus proche de notre thématique sont *CQML* [Rottger 2003], *WQSDL* [Newcomer 2002] et *MARTE* [OMG 2009b]. Dans ces langages, une propriété est définie par son nom. À ce nom est associé a minima une unité de grandeur et un critère de comparaison permettant de savoir si une valeur est meilleure qu'une autre. Par exemple, on cherchera le plus souvent à minimiser des durées (le plus petit est meilleur), mais on cherchera à augmenter un pourcentage dans le cas de la disponibilité (le plus grand est meilleur).

Outre ces informations, les langages de QoS décrivent une, ou plusieurs manières de déterminer la valeur de qualité de service. Nous discutons des différentes méthodes de détermination dans la section suivante.

2.2.2 Méthodes de détermination de la qualité de service

Dans le contexte de la thèse, les propriétés de qualité de service sont des grandeurs quantifiables caractérisant un spécimen donné, à savoir le service. Nous présentons dans cette section les différentes méthodes de détermination de la qualité de service existantes.

Détermination par analyse statique

Une manière de déterminer les valeurs des propriétés de qualité de service consiste à effectuer une analyse statique du logiciel. Cette méthode s'opère le plus souvent au cours de la phase de conception, et peut s'appuyer sur des informations provenant de l'exécution du logiciel. De nombreuses techniques existent pour analyser un logiciel. Par exemple, de nombreux travaux transforment le logiciel/processus métier en réseau de Pétri [Reisig 1998] pour effectuer des analyses formelles. On retiendra notamment les travaux de van der Aalst et de Massuthe qui cherchent à déterminer si un logiciel est correct [Van der Aalst 2008, Massuthe 2005].

Dans le domaine de l'ingénierie des performances, l'établissement de modèles tels que les réseaux de file d'attente (*"queuing network"* en anglais) ou les modèles d'utilisation permettent de simuler l'exécution du système. Il est également possible de traduire un processus métier dans une logique temporelle linéaire (LTL) [Pnueli 1977, Giannakopoulou 2001] ou dans un langage de Pi-Calcul [Milner 1999]. Cela permet par exemple de déterminer des propriétés de sûreté et de correction (justesse). [Havelund 2002, Ferrara 2004].

La détermination par analyse est particulièrement adaptée pour des propriétés portant sur la structure ou l'architecture du système, et qui ne dépendent pas de l'état des éléments du système à l'exécution. Cette technique s'applique hors-ligne, à un moment où on peut se permettre de prendre plus de temps pour exécuter les analyses. Toutefois, dans certains cas, comme lorsqu'on souhaite déterminer la taille d'un message où la taille de certaines variables n'est pas fixe, l'analyse n'est pas déterminable hors ligne. Dans ce cas, l'analyse raffine en déterminant tout ce qui est possible à la conception, pour aller chercher uniquement les informations nécessaires à l'exécution. On parle alors d'analyse dynamique.

Détermination par analyse dynamique

Dans le cas où certaines informations ne sont pas disponibles hors ligne, la détermination d'une valeur de propriété s'effectue à l'exécution. C'est le cas notamment pour des propriétés telles que le temps de transmission d'un message ou le temps de réparation d'un service [OASIS 2010], qui ont une forte dépendance au contexte d'exécution. Pour cela, la détermination s'effectue à l'aide d'un **moniteur**. Un moniteur a pour rôle d'observer un

ou plusieurs événements survenant au cours de l'exécution d'un logiciel, en capturant des informations sur l'état du logiciel avant, après, ou en parallèle de ces événements.

Outre la propriété qu'il observe, il est possible de qualifier un moniteur selon différents critères. Nous retiendrons ici le **type** de moniteur. Wetzstein *et al.* classifient dans leurs travaux trois **types** de moniteurs : moniteur externe au moteur d'exécution, instrumentation du moteur d'exécution et instrumentation du processus métier [Wetzstein 2009]. Selon la solution choisie, l'insertion de code lié à la capture des informations à l'exécution engendrera un sur-coût en terme de performance pouvant influer sur le bon fonctionnement du système et sur sa qualité de service. On dit que le contrôle à l'exécution est **intrusif** [Cheng 2009]. L'intrusion du moniteur ainsi que son niveau d'interopérabilité seront plus ou moins grands. Par exemple, un moniteur instrumentant le moteur d'exécution sera optimisé pour que son intrusion soit moindre, mais il ne sera par compatible avec un autre moteur d'exécution.

Nous retiendrons également la **granularité** de l'observation d'un moniteur : il s'agit du degré de détail auquel le moniteur est sensible [Ghezzi 2007]. À titre d'exemple, une mesure peut être effectuée au niveau du processus pour avoir une vision d'ensemble, au niveau de l'activité, ou encore basé sur des événements bas niveau tels que le changement d'état d'une activité en passe d'être exécutée [De Pauw 2005]. Ici encore, cette finesse dans la granularité a un coût : plus le contrôle sera fin, et plus de données seront collectées et précises. C'est autant de données qu'il faut par la suite analyser et interpréter. En revanche, rester à niveau d'abstraction trop élevé empêchera l'observation de certains phénomènes.

Détermination par dérivation de propriétés

Il est également possible de définir la détermination d'une propriété comme étant une fonction mathématique dépendant de valeurs d'autres propriétés. On parle de propriété **dérivée** de sous-propriétés. La valeur de propriété pour un élément du système (tel qu'un service) est définie en fonction de valeurs d'autres propriétés pour le même élément du système. Par exemple, le temps de réponse d'un service peut être défini comme étant la somme du temps de transmission et du temps d'exécution. Cette définition est opérée à l'aide d'une formule mathématique, *nommée formule de dérivation*. La FIGURE 2.1 est un exemple de dérivation où le temps de réponse est calculé en se basant sur les valeurs des propriétés dont il dérive.

La détermination des valeurs de propriétés par dérivation a pour avantage d'être légère, les formules définies utilisant le plus souvent des opérateurs arithmétiques simples. Toutefois, il est nécessaire de déterminer les opérandes utilisés dans la formule ; ces valeurs étant d'autres valeurs de propriétés, la détermination par dérivation nécessitera au préalable d'effectuer des analyses à la conception et/ou des mesures à l'exécution.

Détermination par agrégation de valeurs

Nous avons restreint dans le cadre de cette thèse le champ des propriétés étudiées aux propriétés composables. Cela veut dire que pour un élément de composition (tels que les processus métiers, les séquences, ou les appels en parallèle), il existe une formule mathématique permettant de déduire la valeur de propriété de la composition, à partir des valeurs de propriétés des activités contenues. Il suffit donc d'observer ces activités (par analyse statique ou par mesure), et de composer une valeur de propriété pour la composition en utilisant une formule mathématique nommée *formule d'agrégation*. Il est important de noter qu'ici, contrairement à la détermination par dérivation de propriétés, la détermination par agrégation de valeurs ne repose pas sur plusieurs propriétés différentes. Ici, il s'agit de

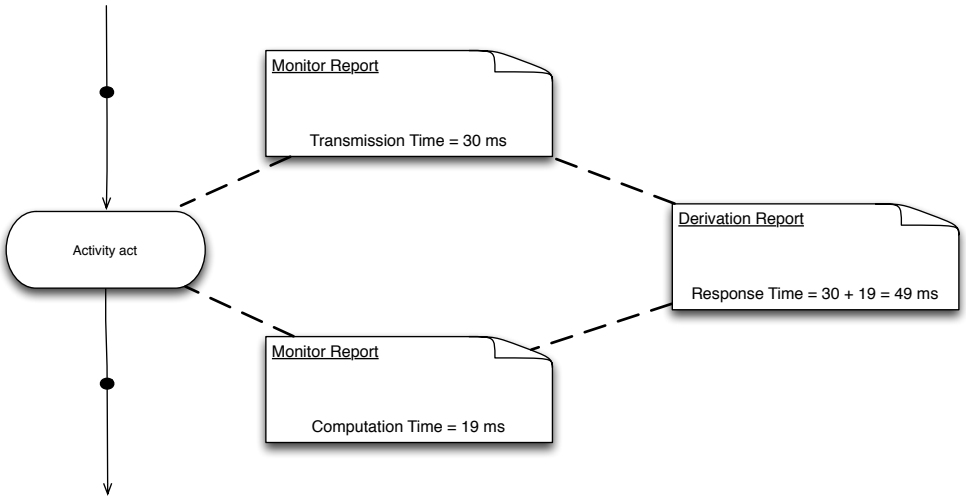


FIGURE 2.1 – Exemple de détermination par dérivation.

déduire une valeur de propriété pour un élément de composition à partir des valeurs de la même propriété, pour les éléments le constituant.

La FIGURE 2.2 illustre la détermination par agrégation sur un processus métier quelconque. Ici, le temps de réponse du processus est la somme des temps de réponse des activités qu'il contient. De nombreux travaux ont porté sur l'établissement de ces formules d'agrégation. Les premiers à les avoir formalisées sont Cardoso [Cardoso 2004] et Jaeger [Jaeger 2004]. Ces travaux furent repris par Mukherjee [Mukherjee 2008], et par Dumas pour permettre leur établissement dans le cadre de processus métiers mal formés [Dumas 2010]. Canfora a même étendu le concept d'agrégation à des propriétés non fonctionnelles, mais spécifiques au domaine du logiciel étudié [Canfora 2006].

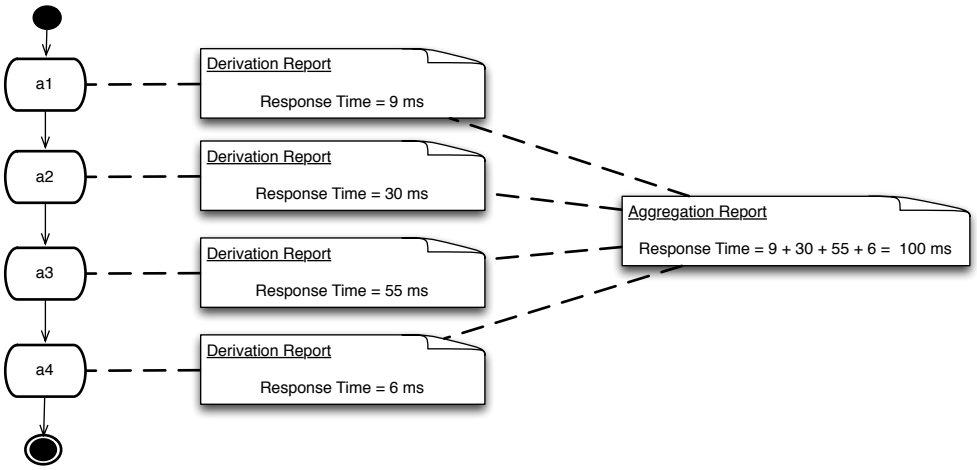


FIGURE 2.2 – Exemple de détermination par agrégation.

De manière similaire à la détermination par dérivation, la méthode par agrégation est peu coûteuse en elle-même en termes de ressources. Toutefois, elle nécessitera également une phase préliminaire de détermination à la conception et/ou à l'exécution.

2.2.3 Contrats de Qualité de Service

Dans l'éco-système d'un service, différentes parties œuvrant pour sa réalisation et son utilisation doivent collaborer. Il s'agit notamment du fournisseur du service, responsable de son implémentation et de sa disponibilité, et du consommateur du service, qui l'utilise. Afin de pouvoir garantir que le service proposé reste stable au niveau de sa qualité de service, les différentes parties impliquées doivent se mettre d'accord.

Pour pouvoir établir cet engagement, il est nécessaire d'établir des **contrats** entre ces différentes parties. Il existe différents supports pour pouvoir établir ces contrats. Le plus utilisé se nomme *Web Service Level Agreement* (WSLA) [Keller 2003]. Un contrat est un accord liant deux ou plusieurs parties sur certaines conditions, telles que les responsabilités de chaque partie ou les engagements à respecter. Dans le cadre des WSLA, un contrat est constitué de trois sections. D'abord, il s'agit d'établir l'identité de chaque partie impliquée. Puis, le contrat se poursuit par la désignation du service pour lequel les parties se mettent d'accord. Enfin, la dernière section du contrat établit les différentes clauses d'engagement. On parle alors d'objectifs de niveau de service (SLO). Un objectif est constitué d'une partie engagée par l'objectif (l'*Obligee* en anglais), d'une période de validité de l'objectif, et d'une condition à respecter. Cette condition prend la forme d'une expression, indiquant un ensemble d'opérateurs sur des métriques. Par exemple, on pourra décrire que la métrique de disponibilité doit être supérieur ou égale à 80%, ce qui signifie que le service en question doit être accessible pour le consommateur au minimum 80% du temps.

Une fois que le contrat est mis en place, il est nécessaire de s'assurer que tous les objectifs sont bien respectés. Une approche possible pour cela consiste à utiliser un contrôleur, qui vérifie bien qu'aucun objectif n'est violé. On retiendra notamment les travaux de Oriol *et al.*, qui ont développé un système nommé SALMon pour l'analyse et le contrôle de SLA [Oriol 2008].

Nous venons de présenter dans cette section les travaux existants autour de la qualité de service dans les architectures orientées services. Nous avons vu différentes manières de déterminer une valeur de propriété de QoS permettent de qualifier un service, avant de s'intéresser à la manière dont la QoS peut être conservée en définissant un contrat entre le fournisseur du service et son consommateur. Si ce dernier point est facilement vérifiable au moment du contrat, il convient de s'intéresser à l'étape de l'évolution, au cours de laquelle les changements opérés peuvent interférer la validité du contrat. Pour cela, nous nous intéressons dans la section suivante aux différentes caractéristiques de l'évolution des systèmes logiciels.

2.3 Évolution des systèmes logiciels

2.3.1 Définition de l'évolution

Le domaine de cette thèse se concentre sur la manière de faire évoluer un logiciel au fil du temps. Cela veut dire par exemple d'ajouter des fonctionnalités, ou de maintenir le code pour effectuer les mêmes fonctionnalités. Traditionnellement, les logiciels ont besoin de changer pour correspondre aux besoins des utilisateurs, pour corriger un bug ou pour supporter l'évolution technologique.

La notion d'évolution est apparue dans la fin des années 60 [Lehman 1969]. Lehman *et al.* ont été les premiers à en donner une définition [Lehman 1985] :

Définition 4

Évolution du logiciel *: "La satisfaction [des besoins] continue demande des change-*

ments continus. Le système aura à être adapté en fonction d'un environnement changeant et des besoins changeants, en développant des concepts et en faisant progresser les technologies. L'application et le système doivent évoluer. "

Ce genre de logiciel est appelé un **logiciel de type-E**.

Pour accompagner cette définition, les auteurs ont érigé une série de lois de l'évolution, établies de manière empirique suite à l'étude de plusieurs systèmes industriels monolithiques [Lehman 1997, Lehman 2006]. Parmi elles, nous retenons les suivantes, car elles s'inscrivent tout particulièrement dans notre contexte :

Définition 5

Lois de l'évolution du logiciel *:*

- ***Continuité du changement :*** *Un système satisfera progressivement de moins en moins les besoins des utilisateurs au fil du temps, à moins qu'il s'adapte de manière continue pour satisfaire les nouveaux besoins.*
- ***Croissance de la complexité :*** *Un système deviendra progressivement de plus en plus complexe, à moins qu'un travail soit effectué pour réduire la complexité.*
- ***Déclin de la qualité :*** *Un système sera perçu comme perdant en qualité au fil du temps, à moins que sa conception soit maintenue avec précaution et adaptée à de nouvelles contraintes opérationnelles.*

En s'appuyant sur des expériences industrielles concrètes, ces trois lois renforcent notre position sur le besoin de faire évoluer un logiciel, et du besoin de s'assurer que la qualité du système continue à être maintenue au fil du temps. Il est important de noter ici que la notion de *changement* est au centre de ces définitions. Ce changement peut s'opérer de plusieurs manières : si l'on considère l'éco-système d'un système logiciel comme étant constitué du code source du système, de l'environnement dans lequel il est exécuté, des besoins utilisateurs desquels le système a émané, le changement peut s'opérer sur chacun de ces éléments.

Pour nous positionner dans cet éco-système, nous nous appuyons sur les travaux de Godfrey *et al.* [Godfrey 2008]. Dans leur papier, les auteurs dressent un état des lieux du vocabulaire utilisé et des différentes catégories d'opérations que l'on peut qualifier de "changement". Ils se positionnent sur l'utilisation des mots *maintenance* et *évolution*. Dans le contexte de la thèse, nous considérons que la maintenance est une phase au cours de laquelle le logiciel est modifié pour corriger des bugs. Nous appelons adaptation la modification du logiciel pendant son exécution pour réagir à un changement de son environnement. Enfin, nous appelons évolution la modification d'un système logiciel suite à l'apparition de nouveaux besoins des utilisateurs.

2.3.2 Caractéristiques d'une évolution

Cette section s'intéresse aux différents facteurs permettant de caractériser une évolution. Nous présentons dans un premier temps les différentes catégories, avant de positionner nos travaux pour chaque catégorie, dessinant ainsi le type d'évolution que nous traitons dans cette thèse.

Afin de pouvoir décrire les caractéristiques d'une évolution, Mens *et al.* ont dressé une taxonomie de l'évolution du logiciel [Mens 2003]. Dans ces travaux, les auteurs ont défini quatre dimensions pour caractériser une évolution : le *Où*, le *Quoi*, le *Quand* et le *Comment* de l'évolution. Sous chaque dimension, un ensemble de caractéristiques caractérise

cette dimension, proposant parfois diverses alternatives. Nous reprenons chacune de ces caractéristiques dans la suite pour positionner les travaux de cette thèse, en définissant quel type d'évolution nous traitons.

Le "Où" de l'évolution :

Cette dimension cherche à caractériser la localisation de l'application d'une évolution. Pour cela, les auteurs se sont intéressés à différents facteurs :

- *le type d'artéfact logiciel modifié* : une évolution va consister en la modification d'un élément du système au sens large. Ici, il peut s'agir bien évidemment du code source de l'application ; mais on peut également considérer l'évolution comme un changement au niveau des besoins de l'utilisateur, de l'architecture, ou encore de la documentation.
- *la granularité* de l'évolution : ce facteur est un ordre de grandeur évaluant à quel niveau de finesse l'évolution opère. Il peut s'agir d'une granularité fine, telle qu'un changement d'instruction ou de paramètre, jusqu'à un changement opéré à l'échelle du fichier, du package, du service, ou même du système dans son ensemble.
- *l'impact* de l'évolution : le facteur d'impact est un critère d'importance pour qualifier une évolution. Il a pour but de caractériser la portée de l'effet du changement sur le reste du système. Il s'agit ici également d'un ordre de grandeur, allant d'un impact local, jusqu'à un impact à l'échelle du système.
- *la propagation du changement* : ce facteur reprend la notion d'impact, en l'appliquant à l'échelle du processus de développement. Là où le facteur d'impact d'une évolution se concentre sur l'effet de l'évolution sur le comportement du système, la propagation du changement s'intéresse aux effets sur d'autres éléments tels que la documentation ou le modèle conceptuel.

Nous résumons l'ensemble des facteurs de la dimension "Où" dans la Table 2.1, où les éléments notés en gras sont ceux que nous traitons dans le cadre de la thèse. Compte-tenu de la définition de l'évolution que nous avons choisie, nous traitons des évolutions portant sur une partie du code source de l'application, à savoir les processus métiers. Ces évolutions ont une granularité fine : nous souhaitons pouvoir caractériser tout changement dans un processus métier, et un impact pouvant être local comme global. Enfin, nous nous intéressons uniquement à son effet sur le comportement du système, nous ne traitons pas la propagation du changement sur d'autres documents de conception.

Table 2.1 – Taxonomie d'une évolution : la dimension "Où".

Où			
Type d'artéfact	*Granularité*	*Impact*	*Propagation du changement*
Documentation	**Paramètre**	Local	Documentation
Conception	Instruction	**Système**	**Conception**
Implémentation	Fichier		**Implémentation**
Tests	Package		Tests

Le "Quoi" de l'évolution :

Cette dimension s'intéresse aux caractéristiques du système sur lequel l'évolution est appliquée.

- *Disponibilité :* la disponibilité du système peut être définie comme la capacité du système à répondre à une requête dans un laps de temps donné. Ici, il est important de

savoir quelles contraintes sont définies sur le système pour pouvoir prévoir l'application d'une évolution. On considère qu'il peut être requis d'un système qu'il soit tout le temps disponible, qu'il peut y avoir des périodes d'arrêt du système sur des périodes courtes, ou simplement que le critère de disponibilité n'a pas d'importance.

- *Dynamisme :* ce facteur détermine si les changements à opérer sur le système sont guidés de l'extérieur (système réactif), ou si les changements proviennent d'une observation faites par des contrôleurs propres à l'application (système proactif). Pour ce dernier cas, on parle de système autonome, et le changement est opéré à l'exécution.
- *Exposition :* les auteurs parlent d'ouverture du système pour désigner sa faculté à autoriser son extension. On parle de système ouvert lorsque l'architecture et l'environnement d'exécution sont constitués de mécanismes d'extension.
- *Sûreté :* la sûreté de l'évolution désigne la propriété du système à s'assurer que l'évolution ne met pas le système dans un état erroné. Il peut s'agir de mécanismes à la compilation, ou de contrôles effectués à l'exécution. On parle de sûreté statique lorsque le contrôle de sûreté est effectué hors ligne. À l'inverse, la sûreté dynamique consiste à vérifier l'évolution au cours de l'exécution du système.

Nous résumons l'ensemble des facteurs de la dimension "Quoi" dans la Table 2.2. Pour cette dimension, nous nous positionnons dans un cadre où les systèmes doivent être réactifs (le client pouvant potentiellement poser des contraintes fortes de disponibilité d'un service), mais pour lesquels un temps d'arrêt pour effectuer de l'évolution est prévisible. Nous traitons des évolutions réactives, dans le sens où elles sont déclenchées par un changement dans les besoins du client, et ne sont pas le fruit d'une adaptation à l'environnement. De part le contexte technologique des SOA, l'exposition du système est ouverte, donnant de par le couplage lâche et l'aspect modulaire des services la possibilité de l'étendre. Enfin, nous nous positionnons dans un contexte d'évolution dont nous assurons la sûreté à la conception, par le biais d'une analyse.

Table 2.2 – Taxonomie d'une évolution : la dimension "Quoi".

Quoi			
Disponibilité	*Dynamisme*	*Exposition*	*Sûreté*
Toujours disponible	**Réactif**	**Ouvert**	**Statique**
Temps d'arrêt acceptable	Proactif	Fermé	Dynamique
Indifférent			

Le "Quand" de l'évolution :

Cette dimension s'intéresse aux caractéristiques temporelles d'une évolution. Plus particulièrement, les auteurs étudient dans cette dimension les propriétés suivantes :

- *le moment* au cours duquel une évolution peut survenir (à la conception, à la compilation, ou à l'exécution)
- *l'historique* des évolutions. Cette caractéristique permet de savoir s'il est possible d'appliquer plusieurs évolutions en concurrence. En effet, avec l'augmentation de la taille des systèmes, la taille des équipes de développement a également augmenté, décentralisant ainsi le développement et de fait les évolutions. On distingue ici trois cas : les évolutions peuvent être prises de manière *séquentielle*, c'est à dire qu'il n'est pas possible d'appliquer une évolution tant qu'une autre évolution est en cours, de manière *parallèle synchrone*, où ici, le développement des évolutions peut être effectué

indépendamment, mais leur application concrète est réalisée à un point précis dans le temps, ou encore de manière parallèle asynchrone, où le développement et l'application des évolutions sont indépendants. Ce dernier cas est complexe à gérer, dans le sens où l'aboutissement à un système incohérent est possible, sans pouvoir effectuer une analyse lors du point de synchronisation, comme cela est possible dans la deuxième méthode.

– *la fréquence* de l'évolution, à savoir si le système étudié évolue fréquemment de sorte que des périodes de temps d'arrêt du système sont à prévoir, ou non. On distingue ici une évolution continue, où le système est en perpétuel changement, une évolution périodique, où les évolutions sont appliquées de manière régulière, ou encore arbitraire, où les évolutions sont effectuées à un rythme irrégulier et peu soutenu.

Nous résumons l'ensemble des facteurs de la dimension "Quand" dans la Table 2.3. Nous traitons dans cette thèse des évolutions subvenant au moment de la conception. De part l'ensemble des problématiques liées à l'historique d'une évolution, nous nous focalisons sur des évolutions entrant dans une logique d'application séquentielle. Enfin, la fréquence n'influençant pas la nécessité du maintien de la qualité de service, nous traitons les évolutions qu'elles soient à fréquence soutenue, ou plus disparates dans le temps.

Table 2.3 – Taxonomie d'une évolution : la dimension "Quand".

Quand		
Moment	*Historique*	*Fréquence*
Conception	**Séquentiel**	**Continue**
Compilation	Parallèle synchrone	**Périodique**
Exécution	Parallèle asynchrone	**Arbitraire**
Chargement		

Le "Comment" de l'évolution :

Cette dimension s'intéresse à la manière dont l'évolution est opérée. Pour cela, les auteurs étudient les facteurs suivants :

– *le degré d'automatisation* de la mise en œuvre de l'évolution. Une évolution peut être *complètement automatisée*, notamment dans le cadre des systèmes auto-adaptatifs, *partiellement automatisée*, où l'équipe de développement décrit les modifications à apporter dans un langage d'évolution, laissant au moteur d'évolution la tâche d'appliquer ces modifications sur les différents documents formant le système, ou *manuelle*, entraînant l'équipe de développement à modifier à la main tous les documents, en prenant à bien garde à garder un système cohérent.

– *le degré de formalité* de la mise en œuvre de l'évolution. Ici, il s'agit de caractériser à quel degré de formalisme est exprimée une évolution. Par exemple, une évolution peut être effectuée de manière ad-hoc (*i.e.*, sans aucun support), ou dans un formalisme mathématique tel que la réécriture de graphe, permettant ainsi de pouvoir s'intéresser à des propriétés de propagation de changement ou de re-factoring.

– *le support* du processus d'évolution. Il s'agit de savoir ici si l'évolution s'effectue de manière manuelle, ou bien si un ou plusieurs outils viennent accompagner le développeur de l'évolution pour automatiser certaines tâches.

– *le type de changement*, à savoir s'il s'agit d'un changement structurel ou comportemental. Dans le premier cas, on modifiera les fonctionnalités par un ajout ou une

TABLE 2.4 – Taxonomie d'une évolution : la dimension "Comment".

Comment			
Degré d'automatisation	*Formalisme*	*Support du processus*	*Type de changement*
Automatisé	Ad-hoc	Aucun	**Structurel**
Partiellement automatisé	**Formalisme mathématique**	Re-factoring	**Sémantique**
Manuel		**Analyse d'impact**	

suppression de paramètres ou de fonctions. Dans le deuxième, il s'agit de la modification de la logique fonctionnelle du programme. À noter ici que ces deux critères ne sont pas mutuellement exclusifs : une évolution est le plus souvent à la fois structurelle et comportementale.

Nous résumons l'ensemble des facteurs de la dimension "Comment" dans la TABLE 2.4. Nous traitons dans cette thèse des évolutions partiellement automatisées, dans le sens où elle ne sont pas le fruit d'une auto-adaptation. Nous cherchons à faire évoluer des processus métiers, en nous appuyant sur un langage pour exprimer ces évolutions. Ce langage nous permettra de raisonner dans une logique des graphes. Enfin, nos évolutions, à la fois structurelles et comportementales, seront analysées dans le but de déterminer son impact sur la qualité de service.

2.4 Conclusion

Dans ce chapitre, nous avons présenté le contexte de cette thèse, en étudiant successivement les architectures orientées services, la qualité de service et la notion d'évolution. Lors de l'étude de ces domaines, nous avons émis un certain nombres d'hypothèses, que nous récapitulons dans la TABLE 2.5. Cette thèse s'inscrit dans un contexte de développement d'applications orientées services. Ici, les logiciels développés sont notamment modulaires et autonomes. Afin de pouvoir différencier plusieurs services aux fonctionnalités similaires, la notion de qualité de service peut être utilisée comme indicateur de comparaison. Enfin, ces systèmes répondants à des besoins changeants de la part de l'utilisateur, il est nécessaire de prévoir des évolutions semi-automatisées, subvenant à la conception, dans le but de pouvoir les analyser pour savoir si, après leur application, la qualité de service est maintenue dans le système.

TABLE 2.5 – Récapitulatif des hypothèses de travail.

Architectures Orientées Services	
Hypothèse 1	*Nous étudions l'évolution dans le contexte des services. Les entités étudiées, nommées services, peuvent être **distribuées**. Nous considérons qu'ils sont **modulaires**, que leur localisation est **transparente** vis-à-vis de l'équipe de développement, et que leur couplage est **lâche**.*
Qualité de Service	
Hypothèse 2	*Pour toutes les propriétés que nous étudions, **il existe au moins une méthode** permettant de déterminer la valeur de propriété d'un service.*
Hypothèse 3	*Dans un système, les besoins en terme de qualité de service sont définis à l'aide d'un **contrat de qualité de service**. Le plus souvent, ils sont représentés sous la forme d'une SLA.*
Évolution	
Hypothèse 4	*Les évolutions que nous considérons dans cette thèse s'opèrent au niveau de l'**implémentation** du système. Elles sont définies au niveau de **granularité** de la réalisation d'un processus métier, c'est-à-dire au niveau des activités qui le constitue.*
Hypothèse 5	*Lorsqu'une évolution est appliquée, nous considérons qu'un **temps d'arrêt** du système est acceptable. Cette évolution est réalisée en **réaction** à un changement dans les besoins de l'utilisateur. L'analyse de la validité de l'évolution est effectuée de manière **statique**.*
Hypothèse 6	*Le processus d'évolution est effectué au moment de la **conception**. Nous considérons dans notre cas que les évolutions sont appliquées de manière **séquentielle**, i.e., qu'à tout moment du cycle de vie, il y a au plus une évolution en train d'être appliquée. La fréquence de ces évolutions, elle, n'importe peu.*
Hypothèse 7	*Une évolution, si elle est appliquée de manière **automatique** sur le système, est déclenchée par un acteur humain. Il peut s'agir de changements **structurels**, ou **sémantiques**. Pour supporter ce processus d'évolution, nous cherchons à effectuer une **analyse d'impact** de son effet sur le reste du système.*

Chapitre 3

État de l'art

Sommaire

3.1 Processus de développement 25
3.1.1 Les processus de développement généralistes 26
3.1.2 Les processus de développement spécialisés SOA 27
3.1.3 Prise en compte de la Qualité de Service 28
3.1.4 Prise en compte de l'évolution 28
3.1.5 Comparatif et limitations 29
3.2 Analyse d'impact 29
3.2.1 Fondements de la causalité 29
3.2.2 Application de la causalité : les analyses d'impact 30
3.2.3 Analyse d'impact pour la qualité de service 32
3.2.4 Comparatif et limitations 32
3.3 Conclusion de l'état de l'art 33

Nous avons vu dans le chapitre 2 que les applications à base de services avaient pour particularité de devoir être réactives face aux changements des besoins des utilisateurs, tout en ayant une préoccupation particulière pour la QoS. Dans ce contexte, l'équipe en charge du système doit mettre en place une stratégie pour faire évoluer le logiciel, tout en prenant en compte le maintien de la qualité de service. Dans ce chapitre, nous dressons dans un premier temps l'état de l'art des processus de développement logiciel. Puis, nous étudions la notion d'analyse d'impact comme possibilité d'établir l'effet d'une évolution sur le reste du système.

3.1 Processus de développement

Nous présentons dans cette section l'état de l'art concernant les processus de développement. Pour étudier les différents travaux, nous nous focalisons sur les critères suivants :

- **Agilité** : le critère d'agilité est le minimum requis pour pouvoir effectuer des évolutions de manière continue. On considèrera ainsi que dans un processus de développement agile, le côté itératif et incrémental peut être vu comme une évolution.
- **Collaboration** : nous étudions ici des systèmes complexes où plusieurs expertises au niveau de l'élaboration de services, de processus métiers, d'évolution ou de qualité de service sont requises. Il est donc important qu'une place soit accordée à ces rôles dans le processus de développement.
- **Prise en compte de la QoS** : le but d'une équipe de développement est de produire un logiciel fonctionnel. La qualité de service n'est pas forcément considérée comme une préoccupation principale ; il s'agira de voir si elle est intégrée au sein du processus de développement.

Dans la suite, nous considérons dans un premier temps les processus de développement généralistes. Puis, nous nous centrons sur les processus orientés SOA, et étudions ceux qui prennent en compte la qualité de service et l'évolution. Enfin, nous effectuons un comparatif des différentes approches.

3.1.1 Les processus de développement généralistes

Les processus de développement ont pour rôle de guider l'équipe de développement dans la réalisation de logiciels. Il existe de nombreux processus de développement, chacun ayant ses avantages et ses inconvénients. Pour la plupart, ils sont construits autour des mêmes grandes étapes, comme référencées dans l'ISO 12207 [IEEE 2008] :

- **Analyse des besoins** : au cours de cette étape, les besoins du client sont identifiés (par exemple à l'aide d'un diagramme de cas d'utilisation UML) et consignés dans un cahier des charges.
- **Conception** : l'architecte logiciel conçoit une solution en se basant sur les besoins collectés précédemment. Ici, il s'agit d'une solution à haut niveau d'abstraction. L'architecte décrit ce qui doit être réalisé, sans donner les détails de comment le réaliser.
- **Implémentation** : l'équipe de développement se base sur la conception de l'architecte pour écrire le code du logiciel.
- **Vérification et validation** : une fois la solution logicielle développée, l'étape suivante consiste à vérifier que l'on a bien implémenté le logiciel, *i.e.*, que le logiciel ne présente pas de bogue. Cette phase se nomme *"vérification"* du logiciel. On s'assure également que l'on a implémenté le bon logiciel, *i.e.*, que le comportement du logiciel correspond aux besoins du client. Dans ce cas, on parle de validation du logiciel.
- **Déploiement** : enfin, l'étape de déploiement consiste à installer le logiciel sur la machine de production, pour pouvoir être utilisé.

Parmi les processus de développement les plus utilisés, nous retenons le modèle en cascade [Royce 1970], comme étant le déroulement séquentiel des étapes listées ci-dessus. Un autre processus est le modèle en V [IABG 1997], développé dans les années 80 par la République Fédérale d'Allemagne, qui est une extension du modèle en V. Ici, au lieu de considérer les étapes de manière séquentielle, une association est faite entre les étapes de réalisation, et les étapes de test, afin de pouvoir les mener en parallèle, et de renforcer leur cohérence. Ces approches sont dites en cascade : le client établit en amont du projet un certain nombre de besoins fonctionnels et non fonctionnels. Le porteur du projet déclenche alors le début du cycle de développement choisi. S'en suit le déroulement des différentes étapes, pour aboutir à la production d'un logiciel correspondant aux besoins établis. Toutefois, ce genre d'approche a été vivement critiqué, principalement pour un manque de réactivité vis-à-vis des besoins du client. Pour pallier ce genre de problème, de nouveaux cycles de développement ont été proposés. C'est le début des méthodes dites *agiles* [Beck 2001] : ici, l'accent est porté sur la livraison fréquente et le retour d'expérience, par le biais de méthodes dites *itératives*, l'équipe en charge de réaliser le logiciel exécute plusieurs fois le processus de développement, et *incrémentales*, où chaque itération augmente les fonctionnalités du logiciel réalisé [Larman 2003]. Le client est désormais au centre du processus, là où il n'était consulté auparavant qu'en début et en fin de projet, permettant ainsi d'éviter de réaliser un logiciel trop éloigné des besoins. Parmi les méthodes agiles, nous retiendrons l'*Extreme Programming* (XP) [Beck 2004], le *Rationale Unified Process* (RUP) [Kruchten 2004] ainsi que la méthode *Scrum* [Schwaber 2004], qui font partie des méthodes de développement les plus utilisées à ce jour.

Pour positionner les processus de développement par rapport à la notion d'évolution, les cycles de développement agiles sont les plus à même à en supporter les différents mécanismes liés. En effet, le caractère itératif et incrémental des méthodes agiles permet une réalisation naturelle de l'évolution, là où les cycles de développement classiques, de par leur long déroulement, nécessite soit d'interrompre le déroulement classique du cycle pour prendre en compte l'évolution, soit d'attendre la fin d'une itération pour prendre en compte l'évolution. Dans le premier cas, cela peut engendrer des problèmes de consistance, par exemple en abandonnant momentanément le développement en cours pour s'occuper de l'évolution. Dans le second cas, cette approche ne permet pas d'être réactif face aux besoins de l'évolution. Les approches agiles sont donc les plus à même de supporter l'évolution.

3.1.2 Les processus de développement spécialisés SOA

Selon les principes énoncés dans le chapitre 2, la construction de logiciels basés sur des architectures orientées services diffère des systèmes traditionnels, notamment dans le sens où la notion de réutilisabilité est prépondérante. Pour cela, de nombreux travaux ont tenté d'adapter des processus de développement généralistes aux architectures orientées services. Ceux-ci sont comparés dans les études de Ramollari *et al.*, et de Shahrbanoo *et al.* [Ramollari 2007, Shahrbanoo 2012]. Nous retiendrons ici les travaux de Papazoglou *et al.*, qui ont développé un certain nombre de recommandations pour l'élaboration d'un processus de développement pour les SOA [Papazoglou 2006]. Leur processus de développement agile est constitué de six étapes :

- **Planification** : l'étape initiale consiste à étudier la faisabilité de la mise en œuvre d'une solution à base de services. Il s'agit dans un premier temps d'établir les besoins de l'utilisateur. Également, c'est dans cette phase que l'équipe de développement prendra garde à s'assurer que la solution à développer s'intègrera dans un environnement existant.
- **Analyse et conception** : Il s'agit de la première étape du cycle de développement. Ici, on établit l'existant en énumérant les différents services, et en revenant potentiellement sur les besoins de l'utilisateur pour identifier quels sont les services en place, et quels sont les services à développer ou à faire évoluer.
- **Construction et test** : au cours de cette étape, on reprend les besoins identifiés dans l'étape précédentes pour concevoir, réaliser et tester de nouveaux services et processus métiers. Il s'agira de déterminer parmi l'ensemble des besoins quels services devront être réalisés, réutilisés, ou composés.
- **Approvisionnement** : cette étape cherche à établir l'équilibre entre les services fournis et leur offre en termes de propriétés. Ici, l'équipe s'intéresse à l'établissement de contrats (de type SLA), en déterminant quel niveau de qualité peut être fourni, comment il peut être monnayé à une potentielle entreprise, et quel usage doit être fait du service pour garantir le niveau de qualité spécifié.
- **Déploiement** : une fois que les nouveaux services ont été validés, ils peuvent être déployés et être promus au niveau des autres organisations potentiellement intéressées.
- **Exécution et Contrôle** : enfin, les nouveaux services peuvent être exécutés. S'il est nécessaire, des contrôles (notamment de la qualité de service) peuvent être enclenchés, afin de pouvoir collecter des données sur la qualité du système, et amorcer ainsi la réflexion d'une autre boucle de développement.

La méthode de développement nommée *"Service-Oriented modeling and Architecture"* (SOMA) est une autre approche développée par IBM [Arsanjani 2008]. Cette approche a

la particularité de mettre en évidence les rôles nécessaires pour la réalisation d'un logiciel à base de services. La méthode est constituée de sept activités assez similaire à l'approche de Papazoglou *et al.*.

La différence principale entre ces approches et les approches généralistes résident dans la notion de réutilisation, mais également la notion de niveau de service fourni. Ainsi, certaines actions, comme par exemple l'implémentation, diffèrent des approches généralistes de par la sélection de services existants.

3.1.3 Prise en compte de la Qualité de Service

Si les cycles de développement se focalisent dans un premier temps sur la livraison d'un logiciel fonctionnel correspondant au cahier des charges établi, tous ne considèrent pas la qualité de service dans leurs préoccupations. Toutefois, certains travaux ont consisté à dériver des processus de développement existants pour y inclure la qualité de service.

Par exemple, Koziolek *et al.* [Koziolek 2006] ont dérivé le cycle de développement RUP pour intégrer la QoS dans les différentes étapes du développement de logiciels à base de composants. Ce processus a notamment pour but d'aider l'architecte à choisir parmi des composants sur l'étagère, en utilisant la QoS comme critère de comparaison. Comme RUP, leur nouveau processus est itératif et incrémental, permettant aux développeurs de proposer de nouveaux composants à chaque itération. Toutefois, chaque itération implique une re-vérification complète de l'architecture. Cette approche identifie quatre rôles de référence : l'expert du domaine, le responsable du déploiement du système, l'architecte du système, et le développeur de composants.

Gatti *et al.* proposent également de prendre compte des propriétés de qualité comme le temps d'exécution pire-cas à travers l'ensemble du cycle de développement [Gatti 2011]. Dans leur approche, chaque étape est guidée par des éléments émanant de l'étape précédente, dans le but d'assurer la traçabilité d'une propriété de l'étape des besoins jusqu'à l'exécution.

3.1.4 Prise en compte de l'évolution

Tous les processus de développement ne prennent pas en compte explicitement l'évolution comme une de leurs étapes. Il est évidemment possible de considérer dans un processus de développement itératif que l'itération est une évolution. Toutefois, il est également possible de considérer l'évolution comme une étape à part entière, en allouant également un temps pour analyser les effets de l'évolution sur le reste du système [Lewis 2010].

Il existe différents processus prenant en compte l'évolution comme une étape à part entière : Kijas *et al.* ont développé un processus d'évolution pour les architectures orientées services [Kijas 2013]. Pour cela, ils ont réalisé un modèle d'évolution de manière empirique, en s'appuyant sur les différents scénarios que l'on peut rencontrer dans la modification d'un système à base de services. Par exemple, la création d'un service entraîne d'autres opérations, telles que la publication de l'interface de ce service. De là, ils analysent l'effet de l'évolution sur le reste du code, en se reposant sur ces scénarios.

Kim *et al.* proposent une approche fondée sur le tissage d'aspect pour établir et faire évoluer un système à base de services [Kim 2010]. Pour cela, ils se basent sur des réseaux de Pétri et sur la séparation des préoccupations par aspects pour établir une évolution, sur laquelle leur outil est en mesure d'effectuer des analyses de performance.

TABLE 3.1 – Comparatif des différents processus de développement.

	Rôles	*Support de la QoS*	*Support de l'évolution*	*Agilité*
[Koziolek 2006]	Oui	Oui	Non	Oui
[Gatti 2011]	Non spécifiés	Oui	Non	Non
[Papazoglou 2006]	Non spécifiés	Oui	Non	Oui
[Kijas 2013]	Non spécifiés	Non	Oui	Non indiqué
[Kim 2010]	Non spécifiés	Oui	Oui	Oui

3.1.5 Comparatif et limitations

Le TABLE 3.1 regroupe les différents processus de développement que nous avons présentés. Nous effectuons la comparaison selon les points suivants :

- **Rôles :** nous pouvons voir ici qu'une minorité de processus intègre la notion de rôles ; mais si un rôle est défini, aucun des travaux ne prévoit dans le processus de développement des points de rencontre explicite où les rôles sont amenés à collaborer.
- **Support de la QoS :** de par l'importance donnée à la qualité de service dans les fondements des architectures orientées services, presque tous les processus s'intéressent à la qualité de service. Toutefois, si la majorité des processus supporte la qualité de service, il est important de s'intéresser à la manière donc la QoS est étudiée. En effet, pour chaque évolution, une complète réévaluation du niveau global de QoS du système est effectuée. Si cette pratique permet d'obtenir le résultat escompté, bon nombre des analyses effectuées sont ré-exécutées de manière arbitraire, sans porter attention au fait que l'élément analysé n'a peut être pas été affecté par l'évolution.
- **Support de l'évolution et agilité :** seuls les travaux les plus récents ont choisi de considérer l'évolution au sein du cycle de développement. Des cinq processus de développement que nous étudions, seul le processus développé par Gatti *et al.* n'est pas un processus agile, et avec un manque d'information sur le processus de Kijas *et al.*. En mettant en perspective ce critère avec le support de l'évolution, nous pouvons voir que tous à l'exception du processus de Gatti *et al.* ont une possibilité plus ou moins évidente de réaliser une évolution du logiciel, que ce soit en considérant une itération du processus, ou en l'exprimant de manière explicite.

Nous venons d'effectuer une comparaison des différents processus de développement. De tous ces processus, aucun ne satisfait l'ensemble des critères nécessaires au maintien de la QoS lors de l'évolution. De manière plus spécifique, chacun de ces processus considère une re-vérification complète de la QoS du système pour chaque évolution, sans essayer de cibler les éléments affectés par l'évolution. Pour pallier ce genre de problème, il s'agit de considérer les techniques d'*analyse d'impact.*

3.2 Analyse d'impact

3.2.1 Fondements de la causalité

La notion de causalité est un principe trans-disciplinaire, touchant à la fois la physique, les mathématiques, la philosophie. Nous retrouvons notamment les origines des concepts de cause et de conséquence dès les dialogues de Platon, dans les récits d'Artistote, ou encore dans le Discours de la méthode de Descartes.

Les travaux de Judea Pearl ont posé les fondements de la causalité, en les formalisant d'un point de vue logique [Pearl 2000]. En accord avec ses travaux, nous parlons dans ce

document de relation de dépendance causale (ou relation causale en abrégé) entre deux éléments A et B, noté A $\longrightarrow$ B, signifiant le fait que A est une cause de B, ou que B est la conséquence de A.

De ces fondements, la notion d'analyse causale est apparue, introduisant des valeurs probabilistes pour représenter la potentialité d'une relation causale. Ces travaux reposent notamment sur l'établissement du modèle causal à l'aide d'un réseau Bayésien et de chaines de Markov [Spohn 2001].

Le besoin d'effectuer une analyse d'impact pour comprendre une évolution a déjà été évoqué dans les travaux de Papazoglou [Papazoglou 2011], où l'auteur explique que ce genre d'analyse serait nécessaire pour prédire et réduire le sous-ensemble des éléments impactés, et comprendre concrètement l'effet de l'évolution.

3.2.2 Application de la causalité : les analyses d'impact

Les théories établies dans la section précédente ont une applicabilité directe dans l'étude de systèmes logiciels. En 2002, Moe *et al.* ont motivé le besoin d'étudier la causalité au sein de systèmes distribués, dans le but de résoudre certains problèmes liés à la compréhension de l'effet du système, d'un sous-système, ou même d'un message [Moe 2002]. Parmi ces problèmes, on retiendra notamment la découverte de goulots d'étranglement. Dans la suite de ce paragraphe, nous nous focalisons sur la causalité lié à un changement opéré dans un système, en présentant différentes techniques permettant de déterminer son impact. Nous les classons en deux catégories, à savoir les méthodes d'analyse de causes racines, et les méthodes d'analyse de l'impact du changement.

Analyse des causes racines

La détermination de la ou les causes à l'origine d'un phénomène est une discipline connue sous le nom d'analyse des causes racines (*root cause analysis* en anglais). Elle a été formalisée dans les années 80 [Busch 1986] dans le contexte du département de l'énergie. L'objectif des analyses de causes racine est, partant d'une situation établie (le plus souvent un problème rencontré), d'obtenir les causes de cette situation. Il s'agit d'une approche réactive, dans le sens où on cherche à remonter à l'origine d'un problème. On opposera ainsi les approches réactives aux approches proactives, où la cause est connue, et où on cherche à en prédire les conséquences.

Les analyses de causes racines fonctionnent en quatre temps [Rooney 2004] :

- **Collecte de données :** afin de pouvoir établir les causes du changement, il est nécessaire de comprendre concrètement ce qu'il se passe dans le système. Pour cela, il s'agit d'établir son état courant, c'est à dire l'état dans lequel se trouve chacun des éléments qui le constituent, mais également l'ensemble des événements l'ayant mené à cet état.
- **Tracé des facteurs de causalité :** en partant des informations collectées, il s'agit de dresser une cartographie des différentes influences entre les éléments. Il s'agit le plus souvent d'un diagramme de séquence, où les différents événements jouent le rôle de garde et où la séquence représente la chaîne de causalité. Ainsi, une fois que le diagramme est tracé, un ensemble de branches apparaît, menant au nœud final correspondant à l'état courant.
- **Identification des causes racines :** une fois que tous les facteurs causaux ont été intégrés au diagramme, un diagramme de décision, appelé carte des causes racine, est établi pour identifier la ou les causes ayant mené à l'état courant. Cette carte permet

d'établir comment, à partir d'une cause éloignée, par un effet de cascade, le système a pu aboutir à son état courant.

– **Recommandations :** enfin, une fois que les causes racines sont établies, l'analyse préconise une ou plusieurs actions à effectuer pour éviter à l'avenir de se retrouver dans cet état.

Une manière de représenter les résultats de l'analyse s'effectue à l'aide des diagrammes de Pareto. Dans ces diagrammes, l'ensemble des causes dont le problème émane est représenté en colonnes, classées par ordre décroissant de probabilité d'implication sur la conséquence. Les diagrammes de Pareto font partie, avec le diagramme en arêtes de poisson, des sept outils à utiliser pour contrôler la qualité [Tague 2004].

Si cette catégorie de méthode permet d'établir un diagnostic face à un problème, tel que la dégradation de propriété de qualité de service par exemple [Ben Halima 2008], elle ne s'inscrit cependant pas complètement dans le cadre de cette thèse. En effet, nous avons pour objectif de maintenir la qualité de service tout au long du cycle de vie du logiciel. Or, ce genre de méthode se positionne davantage dans la situation où la qualité de service n'est plus maintenue.

Analyse de l'impact du changement

L'analyse de l'impact du changement (*change impact analysis* en anglais) s'intéresse à l'identification des conséquences d'un changement effectué sur un système. Ces méthodes ont été définis à l'origine par Bohner et Arnold [Bohner 1996] comme étant *"The determination of potential effects to a subject system resulting from a proposed software change"*. Cette description générale peut s'appliquer à l'ingénierie logicielle en particulier. On retrouve dans la littérature de nombreuses applications de ce principe, agissant sur différentes parties du cycle de vie : l'analyse de l'effet sur les phases de collecte des besoins, sur la conception de la solution, sur son implémentation, et sur la vérification.

Les techniques d'analyse de l'impact du changement n'ont pas pour but unique de se centrer sur l'effet d'une évolution. Dans leurs travaux, Xiao *et al.* utilisent ces techniques pour calculer le coût d'une évolution [Xiao 2007]. En effet, modifier un processus métier s'accompagne souvent d'une modification des interfaces, voir même de la logique d'implémentation de certains services. Les auteurs effectuent une analyse d'impact sur le processus métier dans le but de quantifier le coût de l'évolution sur le reste du système. Nous pensons que cette technique serait une bonne approche si elle pouvait être étendue pour prendre en compte non pas le coût d'une évolution mais son effet sur la qualité de service.

Cette approche contemplative de l'étude de l'évolution est appliquée de manière active dans l'outil nommé Morpheus [Ravichandar 2008]. Ici, les auteurs proposent de s'appuyer sur les relations de dépendance établies pour automatiser l'évolution, en propageant les changements sur les autres éléments constituant le système.

Analyse des effets d'une évolution

Plusieurs travaux se sont intéressés à l'analyse d'impact de l'évolution d'un logiciel. Par exemple, E*lbaum et al.* ont présenté une étude empirique sur l'effet de l'évolution du logiciel sur les tests de couverture de code [Elbaum 2001]. Ils ont montré que même une modification minime dans le logiciel peut modifier les instructions au niveau du code, impliquant de le re-tester. Fondamentalement, nous visons à réaliser le même genre d'approche au niveau des orchestrations de service, et en se préoccupant principalement de la qualité de service.

Analyse de l'évolution dans les SOA

L'analyse d'impact, ou la détermination des éléments affectés par un changement, a été étudiée dans le cadre des processus métiers. Soffer a défini dans ses travaux la notion de *portée du changement*, dont l'identification permet de *"faciliter l'ajustement du système face aux changements des processus métiers"* [Soffer 2005]. Pour cela, l'auteur dresse une taxonomie des différentes modifications pouvant être effectuées sur un processus métier : modification d'un état, modification d'une transition d'état (que ce soit au niveau de la structure du flot de contrôle, ou d'une potentielle condition de garde), modification des variables, ou modification totale, en remplaçant le processus métier par un autre.

Partant de cette taxonomie, l'auteur définit quel est le scope de changement pour chacune des catégories, à savoir le changement contenu (où il n'y a pas de changement indirect), le changement local, ou encore le changement global.

Le canevas proposé par Ravichandar *et al.* vise à contrôler l'évolution de processus métiers [Ravichandar 2008]. Ils établissent les relations entre les différents éléments du système, et opèrent à la fois sur le code source et sur les descriptions de haut niveau d'abstraction pour conserver la cohérence du système. Enfin, l'élaboration d'une évolution entraîne grâce à leur outil une propagation du changement dans l'ensemble de l'architecture. Ces travaux sont similaires à ceux de Dam et al. [Dam 2010] .

3.2.3 Analyse d'impact pour la qualité de service

Enfin, un certain nombre de travaux ont porté sur l'étude de l'effet d'un changement sur la qualité de service. Dans le langage CQML+ [Rottger 2003], il existe la possibilité de représenter les relations de causalité pouvant exister entre les différentes propriétés. Toutefois, aucun outil, ou aucune application, n'est proposée pour pouvoir déterminer pour un système donné la chaine de conséquences entre une évolution et son effet sur la qualité de service. Becker *et al.* proposent une approche nommée Q-Impress visant à prédire les conséquences des évolutions, au moment de la conception du logiciel, sur les différents attributs de qualité. Pour cela, les auteurs se basent sur des modèles de prédiction tels que les chaînes de Markov ou les réseaux de file d'attente [Becker 2008].

Cicchetti *et al.* ont développé une approche pour analyser les effets de l'évolution sur la qualité de service d'applications à base de composants [Cicchetti 2011]. Leur approche considère une évolution effectuée de manière ad-hoc : l'équipe de développement réalise une nouvelle version du système, et l'incrément est calculé en effectuant la différence des modèles architecturaux. En fonction de cette différence, leur outil calcule l'impact de cette différence selon un ensemble de règles. Toutefois, l'inconvénient de leur approche réside dans la nécessité de mesurer a priori l'ensemble des valeurs pour pouvoir ensuite déterminer quel en a été l'impact. Il en résulte une re-vérification complète du système.

3.2.4 Comparatif et limitations

La Table 3.2 regroupe les différents types d'analyse d'impact que nous avons présentés. Nous effectuons la comparaison selon les points suivants :

– **Adaptabilité au contexte :** notre contexte circonscrit les domaines des processus métiers, de la QoS et de l'évolution. Sur l'ensemble des analyses que nous avons étudié, aucune de ces méthodes ne s'inscrit complètement dans notre contexte, prenant en compte au mieux deux des trois critères, comme le font Becker *et al.* ou Cicchetti *et al.*.

- **Identification du sous-ensemble impacté :** de par notre analyse précédente, nous pouvons opposer les méthodes d'analyse des causes racines aux méthodes d'analyse de l'impact du changement. En considérant que pour garantir le maintien de la QoS lors de l'évolution, nous avons besoin de déterminer en premier lieu quel est l'impact de l'évolution, l'analyse des causes racines n'est pas adapté à notre problématique.
- **Quantification de l'impact :** une fois que l'impact a été déterminé, c'est-à-dire que l'analyse a identifié quels sont les éléments du système affecté par une évolution, il est nécessaire de savoir si cet impact est bénéfique ou néfaste du point de vue de la QoS. Toutes les méthodes d'analyse n'effectuent pas cette vérification cependant, essentielle pour pouvoir dire si la QoS est maintenue. C'est le cas notamment de Ravichandar *et al.*, Elbaum *et al.* ou Rottger *et al.*.
- **Coût de l'analyse :** enfin, le dernier critère de comparaison consiste à étudier le coût pour l'utilisateur de l'analyse. Nous cherchons en effet ici à avoir une analyse la plus automatisée possible, et pour laquelle l'utilisateur sera le moins impliqué. Si la plupart des méthodes ont un coût important, en terme d'effort de modélisation ou de temps d'exécution de l'analyse en elle-même, Ben Halima *et al.* ainsi que Ravichandar *et al.* ont un faible coût.

En résumé, aucune approche ne propose une analyse d'impact qui soit à la fois adapté à notre contexte, faible en coût, et fournissant l'ensemble des informations nécessaires à garantir le maintien de la QoS lors de l'évolution d'un logiciel.

3.3 Conclusion de l'état de l'art

Nous venons de dresser l'état de l'art en matière de processus de développement et d'analyses d'impact pour l'évolution. Beaucoup de travaux gravitent autour de ces domaines depuis de nombreuses années. Il existe de nombreux processus de développement spécialisés dans la détermination de la qualité de service, dans l'élaboration de systèmes à base de services, ou incluant l'évolution comme étape du processus.

Nous avons également étudié les travaux portants sur la détermination des effets de l'évolution en termes d'impact. Là encore, de nombreux travaux existent, s'intéressant à la cause d'un changement ou à son effet. Ces techniques ont été appliquées à différents domaines et dans différents contextes.

Dans cette thèse, nous souhaitons maintenir la qualité de service d'un logiciel au cours de ses différentes évolutions, tout en minimisant la phase de re-vérification. Dans cette optique, les défis sont :

- **faire collaborer les acteurs de l'équipe de développement**, en identifiant les connaissances requises pour le maintien de la QoS lors de l'évolution, et en caractérisant les moments au cours desquels les acteurs devront collaborer.
- **déterminer les interactions au sein d'un logiciel**, afin de pouvoir établir le lien existant entre la modification d'un élément du système et la modification d'une propriété de qualité de service.
- **minimiser la vérification de la QoS**, en déterminant les éléments du système impactés par l'évolution de manière à ne vérifier la QoS que sur ces derniers.
- **identifier la cause de la violation d'un contrat de QoS**, en appliquant les techniques d'analyse des causes racines sur le contrat déterminé comme violé.

Dans la suite de ce document, nous présentons nos différentes contributions permettant d'assurer le maintien de la qualité de service lors de l'application à base de processus métiers.

Nous présentons dans le chapitre 5 un processus de développement, nommé Blink, mettant en évidence les différentes collaborations entre acteurs. Il s'agit dans un premier temps d'identifier ces acteurs, et de déterminer à quel moment du processus de développement une coopération entre acteurs est nécessaire.

Puis, nous nous focalisons dans le chapitre 6 sur le rôle de *l'architecte du système*. Nous présentons dans un premier temps comment modéliser un système selon les principes des architectures orientées services. Nous cherchons pour cela à définir ce qu'est un système, et quelles sont les informations que l'équipe de développement doit fournir pour établir son bon fonctionnement. Puis, nous définissons le concept de causalité, à savoir l'influence que peut avoir un élément d'un système sur un autre élément. À travers la notion de modèle causal, nous identifions l'ensemble des interactions dans un système, dans le but de pouvoir déterminer l'effet d'une évolution sur les éléments du système.

Dans le chapitre 7, nous considérons le rôle de *l'expert en qualité de service*. Après avoir fourni une définition de la qualité de service, nous présentons un langage de modélisation d'une propriété de QoS. À partir de la description d'une propriété, nous cherchons alors à enrichir le modèle causal établi dans le chapitre 6 pour déterminer les interactions entre les éléments du système et ses propriétés de QoS, et cibler les propriétés de QoS à re-vérifier lors d'une évolution.

Enfin, dans le chapitre 8, nous nous intéressons au rôle de *l'architecte de l'évolution*. Après avoir donné une définition de ce qu'est une évolution, nous présentons un langage permettant à l'architecte de décrire des évolutions. Puis, nous présentons un outil d'analyse permettant de déterminer si oui ou non la qualité de service est maintenue une fois l'évolution appliquée. Le résultat de cette analyse est une chaine de conséquence, rendant explicite la cause possible du non-maintien de la qualité de service.

L'ensemble des contributions de cette thèse est regroupé au sein de Smile, un canevas de développement pour le maintien de la qualité de service. Notre canevas regroupe la réalisation des contributions, en s'inscrivant de pair avec Blink, notre processus de développement identifiant les interactions entre les expertises de l'équipe de développement, et en implémentant les différentes contributions propres aux expertises de la modélisation du système, de la qualité de service et de l'évolution. Ces contributions sont illustrés à l'aide de PicWeb, notre exemple fil rouge dont la description est réalisée dans le chapitre suivant.

	Critères					
	Adapté aux processus métiers	*Adapté à la QoS*	*Adapté à l'évolution*	*Identification du sous-ensemble impacté*	*Quantification de l'impact*	*Coût de l'analyse*
Analyses des causes racines [Ben Halima 2008]	Non	Oui	Non	Non	Oui	Bas (analyse de trace)
Analyse de l'impact du changement [Ravichandar 2008]	Non	Non	Oui	Oui	Non	Bas (analyse de dépendance)
Analyse de l'évolution [Elbaum 2001]	Non	Non	Oui	Oui	Non	Élevé (étude empirique)
Analyse de Processus Métiers [Soffer 2005]	Oui	Non	Oui	Oui	Oui	Élevé (taxonomie)
Analyse de Processus Métiers [Dam 2010]	Oui	Non	Non	Oui	Oui	Moyen (modélisation du système nécessaire)
Analyse de la QoS [Rottger 2003]	Non	Oui	Non	Non	Non	élevé (pas d'outil disponible)
Analyse de la QoS [Becker 2008]	Non	Oui	Oui	Oui	Oui	Élevé (besoin de modèles de performance supplémentaires)
Analyse de la QoS [Cicchetti 2011]	Non	Oui	Oui	Oui	Oui	Moyen (besoin de re-vérifier l'intégralité du système)

TABLE 3.2 – Comparatif des différentes analyses d'impact.

Chapitre 4

Présentation du cas d'étude

Sommaire

4.1 Séduite, un système de diffusion d'informations à base de processus métiers 37

4.2 PicWeb, un processus métier de recherche d'images 39

4.2.1 Description de PicWeb 39

4.2.2 Évolution de PicWeb 40

Ce chapitre présente PicWeb, un service de recherche d'images provenant de différentes sources qui correspondent à un mot-clé passé en paramètre. PicWeb est représenté par un processus métier dont le développement s'est déroulé de manière agile. Au fil du temps, ce processus métier a évolué pour continuer de correspondre aux besoins des utilisateurs. Il constitue à ce titre un cas d'étude idéal de par sa taille raisonnable et sa compréhension aisée pour illustrer les problèmes rencontrés lors de l'évolution d'application à base de processus métiers. Tout particulièrement, nous montrons ici qu'il est difficile de garantir le maintien de la qualité de service lors de l'évolution. Nous utilisons PicWeb tout au long du document pour illustrer l'apport des différentes contributions de cette thèse. Dans ce chapitre, nous présentons le scénario d'utilisation de PicWeb : nous montrons comment il s'inscrit dans un système plus large nommé *Séduite* [Delerce-Mauris 2009] et en quoi les différentes évolutions de PicWeb ont été source de problèmes pour maintenir la qualité de service du système.

4.1 Séduite, un système de diffusion d'informations à base de processus métiers

Séduite est un système de diffusion d'informations à destination du personnel et des étudiants d'une université. Son origine provient d'un besoin de diffuser sur un campus universitaire des informations diverses, provenant de sources différentes telles que la scolarité, le restaurant universitaire, ou encore les différentes associations étudiantes souhaitant communiquer sur leurs activités. De par l'hétérogénéité du public ciblé, ces informations doivent être affichables sur des périphériques variés tels que des écrans de contrôle positionnés au sein de l'institution, des smartphones, ou encore via un site web. La Figure 4.1 illustre l'interface graphique de Séduite.

Le développement de Séduite a débuté en 2005. Originellement utilisé comme prototype de recherche à des fins d'expérimentation (il a servi notamment de validation au projet FAROS [1]), le système prit de l'ampleur suite à l'implication d'une équipe de huit développeurs. Le système est aujourd'hui constitué de vingt-six services implémentés en Java et de sept processus métiers implémentés dans le *Business Process Execution Language*

1. Projet RNTL FAROS, 2005.

FIGURE 4.1 – Utilisation de Séduite au cours de la nuit de l'info.

BPEL [OASIS 2007]. Ces services ont pour tâches de collecter les différentes informations provenant des différents services administratifs de l'université, mais également de services web externes. La FIGURE 4.2 représente l'architecture globale de Séduite.

En huit ans, Séduite a été déployé dans plusieurs institutions. Son développement s'est effectué de manière agile : en s'appuyant fortement sur les nombreux retours de la part des utilisateurs, l'implémentation de Séduite a subi de nombreuses évolutions. Nous nous focalisons dans le cadre de cette thèse sur l'évolution de l'un de ces processus métiers, nommé PICWEB.

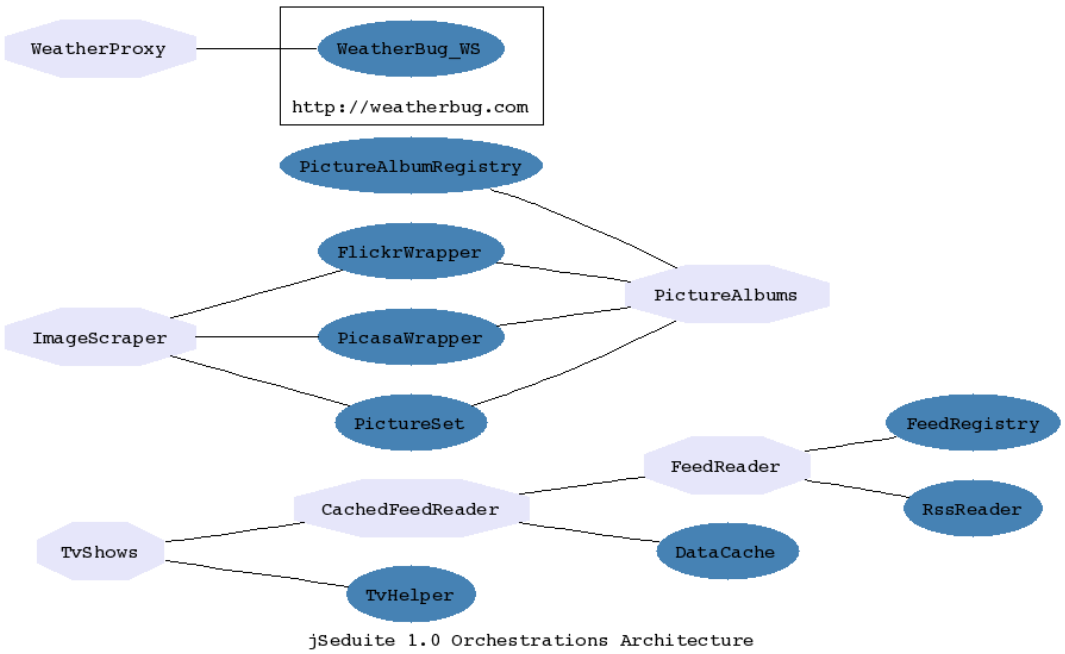
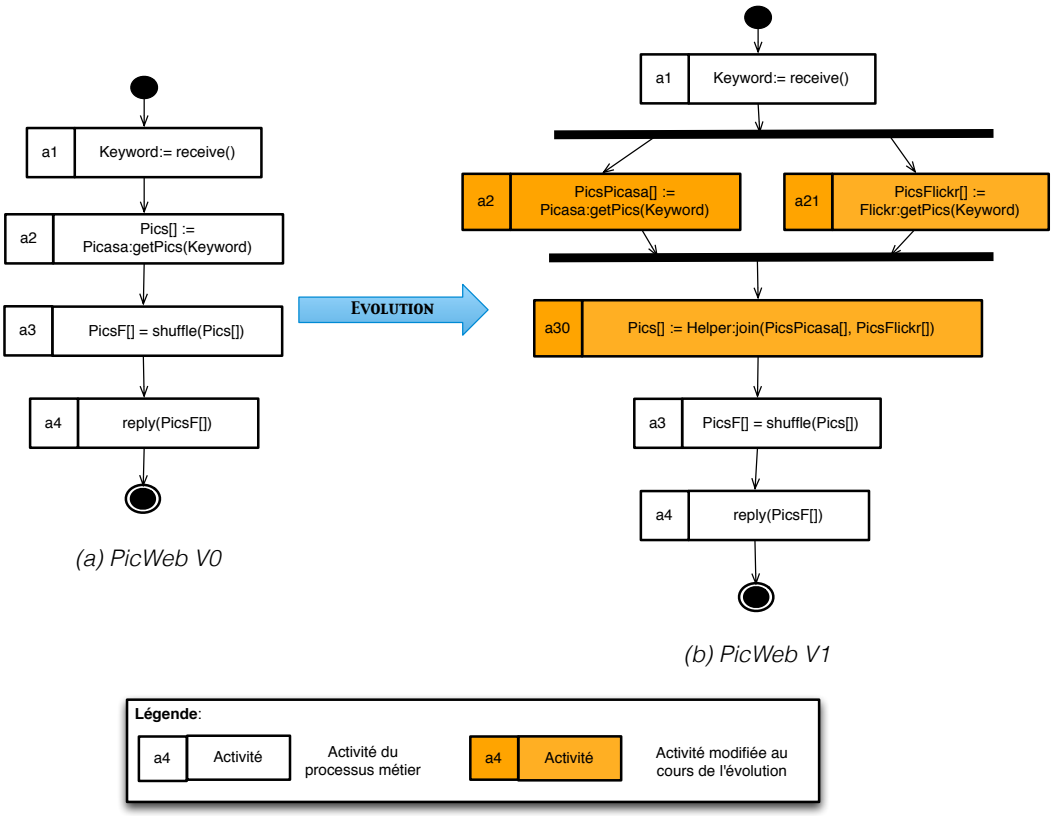


FIGURE 4.2 – Architecture de *Séduite*.



(a) PicWeb V0

(b) PicWeb V1

FIGURE 4.3 – Évolutions de PicWeb.

4.2 PicWeb, un processus métier de recherche d'images

De l'ensemble des évolutions de *Séduite*, le cas des évolutions de PicWeb se dégage particulièrement. En effet, à l'issue de l'une de ces évolutions, le déploiement de la nouvelle version entraîna le ralentissement de l'ensemble du système. Nous nous focalisons dans cette section sur cette évolution, qui nous servira d'exemple tout au long du document.

4.2.1 Description de PicWeb

PicWeb est un processus métier faisant partie de Séduite. Il a pour but d'afficher sur les écrans de diffusion différentes photos de la vie du campus. Pour cela, le processus métier interroge des services web de stockage de photos tels que Flickr de *Yahoo!* ou encore Picasa de *Google*, en récupérant les photos correspondant à un mot-clé donné. Le processus métier récupère ces ensembles de photos et effectue un mélange afin de rendre l'affichage des photos aléatoire. Par exemple, sur le campus de Lille, PicWeb est utilisé pour diffuser les photos des différents événements de la vie étudiante.

La Figure 4.3(a) est une représentation de la version initiale de PicWeb selon le formalisme des diagrammes d'activités d'UML, permettant la description du processus métier. Au départ, le processus métier récupère les photos correspondant à un mot-clé donné (activité *a1*). Le processus reçoit le mot-clé, effectue une invocation au service Picasa (activité *a2*) pour récupérer les correspondances, les mélange pour obtenir un ordre aléatoire (activité *a3*), et les retourne en tant que résultat (activité *a4*).

Dans le cadre du développement de PicWeb, l'équipe de développement s'est intéressé particulièrement à étudier le temps de réponse du service. Concrètement, il s'agissait de caractériser un appel à un service ou à un processus métier, en mesurant le temps passé entre la réception du message d'entrée, et l'envoi du résultat en sortie. Ce temps comprend le temps passé pour l'envoi des messages d'entrée et de sortie (appelé *temps de transmission*), et le temps passé pour calculer la réponse (appelé *temps de calcul*). Pour s'assurer que le temps de réponse de PicWeb se situait dans un intervalle de temps acceptable, des mesures ont été effectuées sur l'ensemble du processus. Puis, lors de chaque évolution, l'équipe de développement considérait uniquement les activités modifiées par l'évolution pour s'assurer du maintien de la qualité de service pour le temps de réponse.

4.2.2 Évolution de PicWeb

Au cours de son cycle de vie, PicWeb évolua à plusieurs reprises pour répondre aux changements des besoins des utilisateurs. Nous nous focaliserons dans ce document sur l'évolution décrite dans la Figure 4.3, car elle montre comment le simple ajout d'un appel de service peut avoir un impact sur la qualité de service de l'ensemble du processus métier, voire même du système tout entier. PicWeb évolue pour prendre en considération un nouveau fournisseur d'images (Flickr), comme montré dans la figure Figure 4.3 (b). Ce fournisseur est désormais appelé en parallèle de Picasa, avant que le résultat soit concaténé dans la réponse.

Lors de cette évolution, si les temps de réponse des activités *a20*, *a21* et *a30* de la Figure 4.3(b) présentaient individuellement des valeurs d'un ordre de grandeur attendu, le déploiement de la nouvelle version de PicWeb révéla une toute autre réalité. En effet, les temps de réponse de PicWeb, mais également de Séduite dans son ensemble, augmentèrent de façon significative. Cette augmentation provoqua le ralentissement du système, obligeant l'équipe de développement à déployer dans l'urgence une version précédente du système.

Il est intéressant de noter ici que la cause du problème ne vient pas directement de l'évolution mais d'un autre service. Ici, l'appel à Flickr ou au service de concaténation n'était pas la cause du ralentissement du système. Toutefois, l'augmentation de la taille des données traitées par l'activité de formatage des images ralentit considérablement son exécution, causant le ralentissement de l'ensemble du système. En effet, l'implémentation de l'opération *shuffle* consiste, pour un ensemble de n éléments donnés en entrée, à effectuer n^2 permutations afin de mélanger l'ensemble. L'évolution appliquée à PicWeb n'a pas modifié l'implémentation de *shuffle* ; toutefois, le contenu de la variable *Pics*, donnée en entrée de *shuffle*, est passé de n éléments à $2n$. Au niveau de *shuffle*, l'évolution a entrainé les n^2 permutations à devenir $\mathbf{4}n^2$ permutations, causant ainsi un ralentissement de l'exécution de l'opération, de PicWeb et du système dans son ensemble.

De ce fait, il n'est pas raisonnable de limiter l'analyse de l'évolution aux seuls éléments modifiés dans l'évolution, car cette modification peut avoir des effets indirects sur le reste du système. Il est donc nécessaire de pouvoir déterminer la relation entre l'évolution et son effet sur le reste du système, afin de pouvoir qualifier l'effet concret de l'évolution sur l'ensemble du système.

Ce chapitre vient de présenter le cas d'étude de ce document. Dans le chapitre suivant, nous présentons les différents problèmes rencontrés au cours de l'évolution de PicWeb. Nous en extrayons les différents défis liés à l'évolution, avant de donner un aperçu des contributions de la thèse pour apporter une solution à cette problématique.

Deuxième partie

Contributions

Chapitre 5

Un processus de développement pour le maintien de la qualité de service

Sommaire

5.1 Motivations 43
5.2 Définition des acteurs 44
5.3 Description du processus de développement 44
5.4 Coopération entre acteurs 45
5.5 Conclusion du chapitre 46

Ce chapitre présente notre approche pour guider l'équipe de développement dans la réalisation d'un système suivant les principes des architectures orientées services, avec un point de vue particulier pour la qualité de service. Nous identifions tout d'abord les différents acteurs opérant sur le logiciel, en décrivant leur rôle, et en quoi leurs expertises propres interviennent dans le maintien de la QoS lors de l'évolution d'un logiciel. Nous présentons ensuite le cycle de développement en lui-même. Enfin, nous mettons l'accent sur le besoin d'interactions entre les différents acteurs pour parvenir à maintenir la QoS tout au long du cycle de vie.

5.1 Motivations

La conception et la réalisation d'un système informatique nécessitent la mobilisation d'un ensemble de compétences propres à un domaine particulier [IEEE 2008, Tarr 1999]. Nous nous intéressons dans le cadre de cette thèse à la réalisation d'une évolution et au maintien de la qualité de service d'un système. Si ces deux compétences peuvent être traitées indépendamment, leur implication dans la réalisation du système entraine des conflits à prendre en compte. Typiquement, l'évolution de PicWeb et les mesures de son temps de réponse peuvent être gérées de manière isolée ; toutefois, en agissant de la sorte, l'application de l'évolution a eu de fait une influence sur la propriété étudiée, ayant ainsi des conséquences inattendues.

Pour pouvoir contrôler ces conséquences, il est nécessaire d'identifier les influences entre les différentes expertises, c'est-à-dire les moments dans le cycle de vie d'un logiciel où les choix techniques et architecturaux réalisés par une expertise donnée auront un impact sur une ou plusieurs autres expertises. Il s'agit donc dans un premier temps d'expliciter les compétences existantes, pour ensuite définir les moments dans le cycle de vie où les choix d'une expertise ont un impact sur une autre expertise, et où une concertation est nécessaire. Nous avons décidé dans un premier temps d'identifier les compétences requises et de les

affecter à des entités nommées *acteurs* [Taskforce 2011]. Un acteur est une, ou plusieurs personnes, responsable d'un ensemble de tâches propres à une préoccupation. Ainsi, une tâche commune à deux acteurs implique un échange de connaissances entre les deux acteurs, que ce soit par le biais d'une rencontre, ou encore par un échange de documents et de données ; c'est ce que l'on appelle un *point d'interaction*.

Le défi ici est de définir quelles sont les interactions nécessaires entre les différents membres de l'équipe de développement. Pour cela, nous proposons BLINK, un processus de développement permettant d'ordonner les tâches en étapes, et de définir le but de chaque tâche. En identifiant les compétences fournies par les acteurs et les compétences requises pour chaque étape, nous mettons en évidence les interactions nécessaires. Dans la suite de ce chapitre, nous présentons quelles sont les acteurs gravitant autour de la réalisation d'un système. Puis, nous introduisons BLINK, et détaillons l'ensemble des étapes nécessaires pour le développement d'un système avec une préoccupation pour la qualité de service. Enfin, nous explicitons les différentes interactions au sein de l'équipe de développement.

Pour illustrer nos propos, nous nous appuyons sur le cas de l'évolution de PICWEB.

5.2 Définition des acteurs

Au cours du développement et de l'évolution de l'application, différents acteurs interviennent pour apporter leur expertise propre. Ces acteurs ont un rôle spécifique, et sont amenés à prendre des décisions modifiant la structure et le comportement du système réalisé. Dans le contexte de la réalisation de systèmes orientés services se préoccupant de la Qualité de Service, nous listons dans la TABLE 5.1 les différents acteurs identifiés, et décrivons leur rôle. Ces acteurs ont été identifiés en nous inspirant des travaux de Bejaoui *et al.* [Bejaoui 2008], de Kajko-Mattson *et al.* [Kajko-Mattsson 2007], ou encore de Zimmermann *et al.* [Zimmermann 2006]. Nous comptons cinq rôles. Parmi eux, un est extérieur à l'équipe de développement (l'utilisateur). Le dernier rôle est un acteur non humain : il s'agit de notre canevas de développement, SMILE, dont nous présentons les différentes parties tout au long du document, et dont l'implémentation est décrite dans le chapitre 9.

Nous cherchons maintenant à déterminer la manière dont les acteurs œuvrent pour réaliser un système. Pour cela, nous répertorions chacune des étapes de BLINK, notre processus de développement. Nous présentons chacune des étapes en décrivant leur rôle dans le processus global, et les acteurs amenés à y participer.

5.3 Description du processus de développement

Afin d'éviter les problèmes d'influence entre les expertises et de pouvoir définir un système dont la qualité de service est maintenue au fil du temps, nous identifions dans cette section les étapes clés du processus de développement, au cours desquelles une mise en commun des informations et des choix effectués par les acteurs concernés est nécessaire. Pour définir notre processus de développement, BLINK, nous nous sommes inspirés de processus de développement existants, tels qu'ils sont présentés dans [Papazoglou 2006] ou encore dans [Kijas 2013]. De ces processus, nous avons conçu BLINK, un processus de développement pour le maintien de la qualité de service à l'évolution. La FIGURE 5.1 représente le diagramme d'activités de ce processus de développement. Chacune des étapes est décrite dans les TABLES 5.2, 5.3 et 5.4, accompagnée d'un diagramme selon la spécification SPEM [1]. Ce processus a été conçu pour réaliser un système de façon itérative et incrémentale, où chaque

1. http://www.omg.org/spec/SPEM/2.0/

TABLE 5.1 – Rôles intervenants dans la réalisation d'un système.

Utilisateur du système	Représente la communauté utilisant le logiciel. Il est responsable de faire part auprès de l'équipe de développement des nouveaux besoins, nécessitant de faire évoluer le logiciel.
Architecte du processus métier	Réalise la logique métier du système au moyen de processus métiers et de services. Il a une connaissance approfondie du comportement du système et de la panoplie de services déjà implémentés qu'il peut utiliser.
Expert en Qualité de Service	Définit les propriétés de qualité de service. Il a une connaissance approfondie des propriétés et des outils pour les observer au sein de n'importe quel système. Il est en mesure de déterminer en cas de dégradation des valeurs de propriétés les facteurs d'influence pouvant en être la cause.
Architecte de l'évolution	Décrit les évolutions à produire pour modifier le comportement de système, en accord avec les nouveaux besoins de l'utilisateur. Il a une connaissance partielle du comportement du système.
SMILE	Joue le rôle d'interlocuteur avec les autres acteurs de l'équipe de développement, en leur communiquant par exemple les informations nécessaires à leurs prises de décisions. Il contient des informations sur le système, son implémentation, et les différentes analyses effectuées. Dans le cadre de BLINK, SMILE automatise certaines tâches comme l'application de l'évolution au système, ou encore son analyse.

incrément est une évolution à considérer. Il implique l'ensemble des acteurs du système et permet, à partir de l'expression de nouveaux besoins de la part de l'utilisateur, de définir une évolution qui sera analysée afin de s'assurer du maintien de la qualité de service dans le système post-évolution.

Au cours des différentes étapes, certains acteurs sont amenés à interagir. Nous définissons dans la section suivante les points d'interaction du processus, où plusieurs expertises sont nécessaires pour accomplir des tâches communes.

5.4 Coopération entre acteurs

Le but de cette section est de mettre en évidence les coopérations entre les différents acteurs. Il s'agit de définir pour un point d'interaction à quel moment celui-ci intervient dans le processus de développement, quels acteurs sont nécessaires, et quelles informations ils vont échanger. Nous distinguons trois points d'interaction, indiqué sur la FIGURE 5.1 par un titre d'étape souligné :

- **Évolution des besoins** : au cours de cette étape, l'utilisateur et l'architecte du système collaborent ensemble pour établir le plus précisément possible les nouveaux besoins. Il en résulte une spécification des besoins.
- **Définition de l'évolution** : pour pouvoir définir une évolution, l'architecte de cette évolution a pour tâche de proposer une modification qui satisfera les besoins spécifiés. Pour cela, la collaboration de l'architecte du système et de l'architecte de l'évolution consiste à la mise en commun des besoins, une spécification des besoins, guidée par la

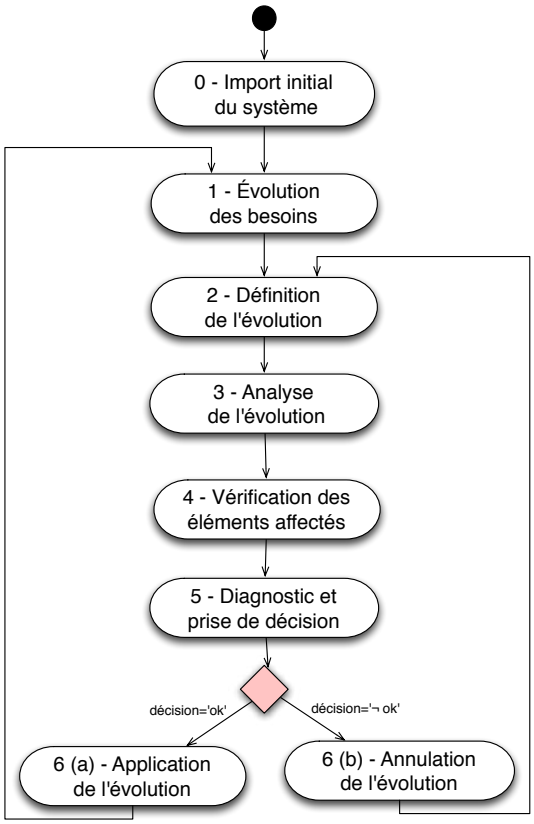


FIGURE 5.1 – Schéma du processus de développement BLINK.

connaissance pointue du système que l'architecte du système a acquis depuis le début du projet.

– **Diagnostic et prise de décision** : une fois que l'architecte de l'évolution a conçu une modification, et que l'expert en qualité de service l'a analysée, il s'agit de mettre en commun ces deux expertises dans le but de dresser un diagnostic. Pour cela, l'expert en qualité de service apporte les différentes mesures effectuées : l'architecte de l'évolution est en mesure de les interpréter, afin de comprendre pourquoi une évolution a violé un contrat de qualité de service, le cas échéant. Cette coopération a pour but de faciliter le diagnostic, dans l'optique de l'écriture d'une nouvelle évolution qui ne violera pas de contrat de qualité de service.

En se basant sur ces différentes coopérations, nous mettons en évidence à quel moment du cycle de vie il est important d'unir plusieurs expertises, évitant ainsi un cloisonnement des acteurs de l'équipe de développement.

5.5 Conclusion du chapitre

Nous venons de présenter BLINK, un processus de développement pour gérer le maintien de la qualité de service à l'évolution. Pour atteindre cet objectif, nous avons établi un ensemble d'acteurs, et défini leur rôle dans la réalisation d'un système logiciel. Notre processus est constitué d'un ensemble d'étapes, pour lesquelles le ou les acteurs impliqués sont désignés. De plus, cette répartition des rôles tout au long du processus de développe-

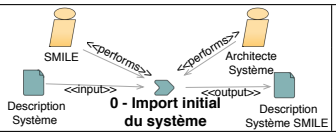
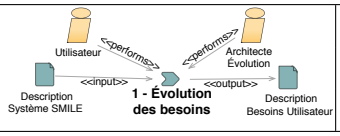
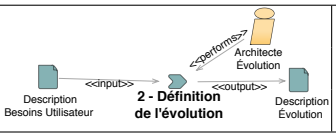
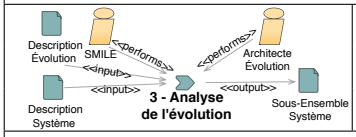
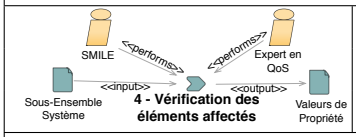
0 - Import initial du système	
	**Entrée :** Description architecturale d'un système **Sortie :** représentation du système dans SMILE **Acteur(s) :** Architecte du système, SMILE
Description : afin de pouvoir unifier l'utilisation du processus de développement BLINK pour des nouveaux systèmes comme pour des systèmes existants, l'import initial du système est une étape optionnelle permettant d'importer un système existant, constitué de un ou de plusieurs processus métiers, dans notre canevas de développement. **Application à PICWEB :** l'architecte du système initie le cycle de développement en important le processus métier dans sa première version (voir FIGURE 4.3). En sortie de l'étape, SMILE produit une structure de données représentant le système, lui permettant par la suite de raisonner dessus. Plus d'informations dans le chapitre 6.	
1 - Évolution des besoins	
	**Entrée :** Représentation du système **Sortie :** Description des besoins de l'utilisateur **Acteur(s) :** Utilisateur, Architecte de l'évolution
Description : cette étape est déclenchée à l'initiative de l'utilisateur, qui souhaite voir le système évoluer. Pour cela, l'utilisateur et l'architecte de l'évolution se réunissent pour capturer les nouveaux besoins conduisant à faire évoluer le système. **Application à PICWEB :** l'utilisateur a émis le souhait de voir davantage de photos différentes sur les écrans du campus. L'architecte de l'évolution a proposé alors d'inclure une nouvelle source de données, *Flickr*.	
2 - Définition de l'évolution	
	**Entrée :** Description des besoins de l'utilisateur **Sortie :** Description de l'évolution **Acteur(s) :** Architecte de l'évolution
Description : l'étape consiste à définir une évolution du système répondant aux nouveaux besoins. L'architecte de l'évolution exprime l'évolution sous formes d'opérations applicables aux processus métiers. Il décrit la séquence des opérations d'évolution (telles que l'ajout d'une variable, la modification d'une activité) permettant de rendre le système satisfaisant vis-à-vis des nouveaux besoins de l'utilisateur. **Application à PICWEB :** l'architecte de l'évolution décrit les opérations nécessaires pour ajouter un appel à *Flickr* en parallèle à Picasa, et pour concaténer les résultats des deux services.	

TABLE 5.2 – Description des étapes 0 à 2 du processus de développement (BLINK).

ment nous a permis de rendre explicite les interactions nécessaires entre acteurs. De cette manière, chaque choix effectué au cours du processus est fondé sur l'ensemble des données nécessaires, permettant de s'assurer à tout moment qu'une décision prise n'altèrerait pas le maintien de la QoS.

Il s'agit donc maintenant de savoir, pour chacun de ces acteurs, quels sont les outils à mettre en place pour leur permettre d'apporter les informations nécessaires lors des points d'interaction. Dans la suite du document, nous prenons tour à tour la place de chaque acteur, en déterminant leur problématique propre pour aider au maintien de la qualité de service, et en proposant une contribution permettant de résoudre ces problématiques.

3 - Analyse de l'évolution	
	**Entrée :** Évolution, Système **Sortie :** Sous-ensemble du système post-évolution **Acteur(s) :** Architecte de l'évolution, SMILE
Description : l'étape d'analyse a pour objectif de déterminer quelles valeurs de propriétés du système sont affectées par l'évolution. Pour cela, SMILE identifie les influences entre les éléments manipulés dans l'évolution et le reste du système, pour déterminer quels éléments ont leur comportement affecté par l'évolution. Nous verrons comment ces influences sont déterminées dans les chapitres 6 et 7, et comment l'analyse est réalisée dans le chapitre 8. **Application à PICWEB :** l'étape d'analyse de l'évolution a pour objectif de déterminer quelles activités sont influencées par l'ajout de *Flickr*. Ici, il s'agit de lever la limitation présentée dans le chapitre 4, où l'équipe de développement a vérifié uniquement le temps de réponse des activités manipulées par l'évolution. En dressant la chaine de conséquences de l'évolution, l'analyse désigne un sous-ensemble du système à re-vérifier.	

4 - Vérification des éléments affectés	
	**Entrée :** Sous-ensemble du système post-évolution **Sortie :** Valeurs de propriétés collectées **Acteur(s) :** Expert en QoS, SMILE
Description : Une fois que l'analyse de l'évolution a déterminé les éléments affectés par l'évolution, il s'agit maintenant de savoir si leur modification améliore ou dégrade la qualité de service. Pour cela, l'expert en qualité de service est en charge de contrôler la qualité de service des éléments affectés, pour déterminer les nouvelles valeurs de propriété à l'aide d'outils de détermination de la qualité de service tels que des outils d'analyse statique du système, ou des contrôleurs présents à l'exécution. **Application à PICWEB :** une version de test est déployée, dans le but de mesurer à l'exécution le temps de réponse des activités désignées lors de l'étape précédente. Ainsi, il est possible de quantifier la différence causée par l'évolution en terme de temps de réponse en la comparant avec les valeurs mesurées sur la version précédente de PICWEB.	

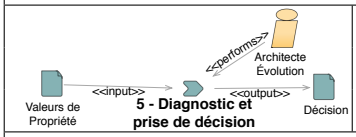
5 - Diagnostic et prise de décision	
	**Entrée :** Valeurs de propriétés collectées **Sortie :** Décision **Acteur(s) :** Architecte de l'évolution
Description : le but de cette étape est de décider de l'application d'une évolution. Pour cela, l'architecte compare les nouvelles valeurs de propriétés du système post-évolution avec les contrats de QoS définis. À partir de cette comparaison, l'architecte de l'évolution peut déterminer si un contrat de QoS a été violé, et décider d'appliquer ou non l'évolution. **Application à PICWEB :** les données collectées révèlent une augmentation importante du temps de réponse, allant au delà du niveau maximum autorisé par le contrat de qualité de service. En observant les valeurs du temps de réponse mesurées à l'exécution, l'architecte de l'évolution est en mesure de détecter que le temps de réponse de l'activité *shuffle (a3)* a augmenté (voir FIGURE 4.3). En conséquence, il prend la décision de ne pas appliquer l'évolution.	

TABLE 5.3 – Description des étapes 3 à 5 du processus de développement (BLINK).

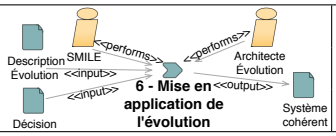
6 - Mise en application de l'évolution	
	**Entrée :** Décision, Évolution **Sortie :** Système dans un état cohérent **Acteur(s) :** Architecte de l'évolution, Smile
Description : la dernière étape consiste à mettre en œuvre la décision prise précédemment. Pour cela, deux choix sont possibles : 1. **6 (a) - Application de l'évolution :** Dans cette alternative, il s'agit d'appliquer de manière effective l'évolution sur le système en cours d'exécution. Dans le cas où l'évolution apportée au système satisfait l'architecte de l'évolution, les processus modifiés sont mis à jour et déployés sur le serveur de production de façon atomique. 2. **6 (b) - Annulation de l'évolution :** Cette alternative annule les modifications faites dans la représentation du système. Dans le cas où l'évolution apportée au système viole les contrats de qualité de service ou ne satisfont pas l'architecte de l'évolution, il est nécessaire d'annuler l'application de l'évolution. Cela se traduit par une réinitialisation des modèles de raisonnement du système à la version pré-évolution et par un effacement des mesures effectuées au cours de l'étape de vérification, afin de conserver la cohérence entre l'état du système exécutée et sa représentation. Le système retrouve un état cohérent ; l'architecte de l'évolution peut reprendre le cycle de développement en écrivant une nouvelle évolution. **Application à PicWeb :** Ici, le modèle du système est rétabli tel qu'il était avant d'appliquer l'évolution, et les valeurs de propriétés mesurées sur la version de test sont effacées. Partant de la cause identifiée lors de l'étape 5, l'équipe de développement modifie l'opération *shuffle* en réduisant le nombre d'échanges effectués et en parallélisant les traitements, permettant ainsi d'obtenir un comportement similaire tout en respectant les contrats de QoS de *Séduite*.	

Table 5.4 – Description de l'étape 6 du processus de développement (Blink).

Chapitre 6

Modélisation d'un système à base de processus métiers

Sommaire

6.1 Défis 52
6.2 Modélisation du système 52
6.2.1 Univers 52
6.2.2 Variables et types 52
6.2.3 Messages et Opérations 53
6.2.4 Services 54
6.2.5 Activités 55
6.2.6 Relations d'ordre 56
6.2.7 Processus métier 57
6.2.8 Système 57
6.3 Causalité de l'exécution d'un système 58
6.3.1 Mise en œuvre de la causalité 59
6.3.2 Relations causales fonctionnelles 59
6.3.3 Exemple 60
6.4 Déduction des relations causales d'un système 61
6.4.1 Méthodologie 61
6.4.2 Expression des règles causales 62
6.5 Conclusion du chapitre 63

Ce chapitre se centre sur le rôle de l'architecte du système. Celui-ci a pour tâche de décrire la structure et le comportement du système ; il doit être en mesure de comprendre les interactions entre ses différentes parties. Pour l'aider dans sa tâche, nous présentons dans un premier temps un méta-modèle permettant de décrire un système. Puis, nous définissons la notion de *causalité* dans le cadre d'un système à base de processus métier, et motivons l'intérêt de connaitre les relations de causalité dans un système pour comprendre les interactions existantes. Ces *relations causales* ont pour but de rendre explicite les interactions présentes au sein d'un système. Après avoir illlustré les différents types de relations causales que l'on peut trouver dans un système, nous montrons comment il est possible de les déterminer de manière automatique. Ces causalités, matérialisées sous la forme de *relations causales*, serviront de base dans le chapitre 8 pour analyser les effets d'une évolution sur la qualité de service d'un système.

6.1 Défis

Dans ce chapitre, nous répondons aux défis suivants :

- **Modélisation du système :** afin de pouvoir répondre aux besoins des utilisateurs, l'architecte a besoin de pouvoir décrire la structure et le comportement du système. Pour cela, nous devons déterminer quels sont les concepts nécessaires à sa modélisation.
- **Comportement du système à l'exécution :** lors de l'exécution du système, processus métiers, variables et appels à des opérations de services interagissent dans le but de pouvoir produire un résultat en sortie. Si l'architecte décrit l'ensemble du comportement du système, la complexité et la taille de ce dernier rendent malgré tout la compréhension des interactions difficile. Comment peut-on rendre explicite les interactions liées à l'exécution d'un système et extraire celles qui nous intéressent ?

Nous proposons dans la suite de ce chapitre de présenter dans un premier temps comment l'architecte modélise un système en introduisant notre méta-modèle. Nous avons fait le choix ici de définir notre propre méta-modèle, reprenant les concepts nécessaires à la modélisation de processus métiers. Ainsi, nous offrons la possibilité d'être inter-opérable avec les différents langages de description de processus métiers, en effectuant une correspondance entre les concepts du langage utilisé et notre méta-modèle. Puis, nous définissons la notion de causalité dans un système. Cette notion nous servira à représenter les interactions entre les éléments du système (comme par exemple l'influence d'un paramètre d'un appel à un service sur le comportement de ce dernier). Après avoir présenté différentes relations causales que l'on peut retrouver dans un processus métier tel que PicWeb, nous définissons des groupes de relations, représentant le même type de causalité. Nous introduisons alors la notion de *règle causale*, qui sont des patrons de relation causale qu'il est possible d'appliquer à un système pour obtenir de manière automatique ses relations causales.

6.2 Modélisation du système

Cette section présente notre méta-modèle permettant à l'architecte du système de modéliser un système. Nous définissons chacun des concepts, avant d'illustrer comment ils sont mis en œuvre pour modéliser PicWeb.

6.2.1 Univers

L'ensemble des concepts présentés dans la thèse, qui servent à définir un système ou sa qualité de service, sont regroupés autour d'un concept appelé *Univers*. Un univers est composé d'un ensemble d'éléments (*UniverseElement*), dont tous les concepts que nous allons présenter héritent.

6.2.2 Variables et types

On appelle *type de donnée*s la caractérisation du sens et de la taille d'une donnée. Nous considérons deux sortes de types :

- **Types primitifs (SimpleType)** : on appelle *type primitif* la structuration des données à un niveau indivisible. Par exemple, les types *Integer*, *String* ou *Float* sont des types primitifs, définis sur la base de normes existantes [Singh 2006].

- **Types complexes (ComplexType)** : il est également possible de définir des types de données comme étant une composition de types simples. Par exemple, une *séquence* est un type complexe constitué d'une suite d'éléments dont la taille n'a pas été définie à l'avance. L'architecte déclarera alors par exemple utiliser un type *Sequence<Integer>*, comme étant une suite d'éléments de type Integer.
- **Types structurés (StructuredType)** : en s'appuyant sur les types complexes et les types primitifs, l'architecte a la possibilité de définir ses propres structures de données. Pour cela, un *type structuré* est constitué d'un ensemble de types, que l'on différencie au sein de la structure de données à l'aide d'un *nom* (*name* en anglais). C'est ce que l'on appelle un *champ (Field en anglais)*. À titre d'exemple, l'architecte pourrait définir un type structuré *Étudiant*, constitué de trois champs *nom* (de type *String*), *prénom* (de type *String*), et *notes* (de type *Sequence<Integer>*).

Une *variable* est une donnée dont la valeur peut évoluer au cours de l'exécution. Elle est caractérisée par un nom et un type de donnée. À titre d'exemple, la variable *keyword* est une variable de PicWeb, de type simple *String*. Une autre variable de PicWeb, *PicsPicasa*, est une variable de type complexe *Sequence<String>*. La Figure 6.1 est un extrait de notre méta-modèle permettant de représenter des types et des variables : une variable a un et un seul type ; *Type* est un méta-élément abstrait, dont héritent *SimpleType*, *ComplexType* et *StructuredType*.

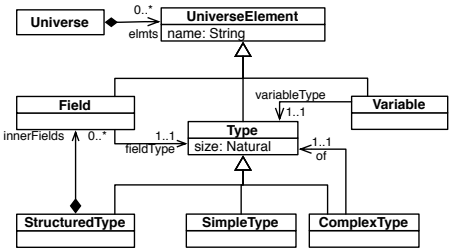


Figure 6.1 – Extrait du méta-modèle de Smile : variables et types.

6.2.3 Messages et Opérations

Nous appelons *Opération* dans le contexte des architectures orientées services la déclaration d'une tâche effectuant un traitement. Une opération est déclarée à l'aide d'une signature. Celle-ci définit le *nom* de l'opération, ainsi que le *type* de donnée en entrée de l'opération et le *type* de données produites en sortie. Ainsi, on définit une opération par le tuple :
Opération(nom : String, entrée : Type, sortie : Type).
Pour interagir avec une opération, il est souvent nécessaire de lui communiquer un ensemble de données à traiter. Cela est réalisé par le biais d'un *message*. On appelle *message* une structure de données encapsulant dans une même unité logique l'ensemble des variables nécessitant d'être transmis. Un *message* est déclaré en lui donnant un *nom* et en indiquant son *type* de données.

L'invocation d'une opération s'effectue la plupart du temps en envoyant un message, dont le type correspond au type d'entrée de la signature de l'opération. La définition du comportement de l'opération est, dans le contexte des architectures orientées services, réalisée à l'aide d'un processus métier, ou par une implémentation dans un langage de programmation donné.

À titre d'illustration, PicWeb est un processus métier implémentant l'opération du même nom. Sa signature est :
Opération(nom="PicWeb",entrée="String",sortie=Sequence<"String">).
Lorsqu'il est invoqué, PicWeb reçoit en entrée un message constitué d'une seule variable, *keyword*, de type String. Le processus décrit l'ensemble des traitements à effectuer, pour retourner au final un autre message, constitué d'une seule variable *PicsF* ; cette variable est de type *<"String">*, un type complexe constitué d'une séquence d'URLs, de type *String*.

6.2.4 Services

Le concept principal du paradigme des architectures orientées services est le *service*. Le consortium OASIS définit un service comme étant *"un mécanisme pour permettre l'accès à une à plusieurs fonctionnalités, où l'accès est fourni par une interface prescrite et respectant les contraintes et les polices spécifiées dans la description du service"*[OASIS 2006]. Un service est une entité logique regroupant un ensemble d'*opérations*. Nous distinguons deux sortes de services :

- **Service intra-domaine :** il s'agit ici d'un service interne à l'organisation, l'entité en charge du système modélisé par l'architecte du système. Ce service peut être vu comme une boîte blanche, sur lequel des ajustements sont possibles. L'implémentation est disponible, modifiable, maintenable et évolutive. Dans le cadre de PicWeb, *Helper* est un service dont l'implémentation dépend de l'équipe de développement du système : il s'agit là d'un service intra-domaine. Ce service comprend plusieurs opérations, dont l'opération *Format*.
- **Service extra-domaine :** en plus des services intra-domaines, il est parfois nécessaire de faire appel à des services proposés par des organisations tierces. Ces services sont vus comme des boîtes noires : l'équipe de développement n'a aucune emprise sur leur implémentation ou leur déploiement. Par exemple, le service *Picasa* est un service extra-domaine fourni par *Google*. L'équipe de développement n'a aucune possibilité de changer l'implémentation des opérations de ce service.

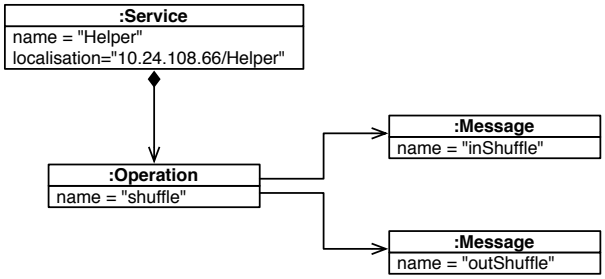


Figure 6.2 – modélisation du service Helper.

Un service est défini comme le tuple :
Service(nom:String, localisation:String, ops:Ensemble<Opération>).
À titre d'illustration, la Figure 6.2 est un extrait du modèle de PicWeb, représentant le service *Helper* et son unique opération. Ce modèle est conforme à l'extrait de notre méta-modèle illustré dans la Figure 6.3, représentant les différents concepts introduits dans ce paragraphe. Un service a pour attribut sa localisation. Il est constitué de une à plusieurs opérations, possédant au plus un message en entrée et au plus un message en sortie.

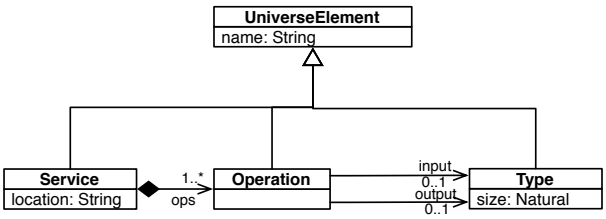


FIGURE 6.3 – Extrait du méta-modèle de SMILE : services, opérations et messages.

6.2.5 Activités

Le point de vue choisi pour modéliser un système est celui d'un ensemble de tâches ordonnées dans le temps. Nous appelons *activité*, en lien avec la définition d'une activité dans le langage *BPEL*, l'instanciation d'une tâche dans le contexte d'un processus métier. Cette tâche a pour but d'effectuer un traitement sur des données. Par exemple, *a2* dans la FIGURE 4.3 est une activité de PICWEB instanciant une invocation au service *Picasa*. L'activité prend en entrée un mot-clé, et produit en sortie un ensemble de photos.

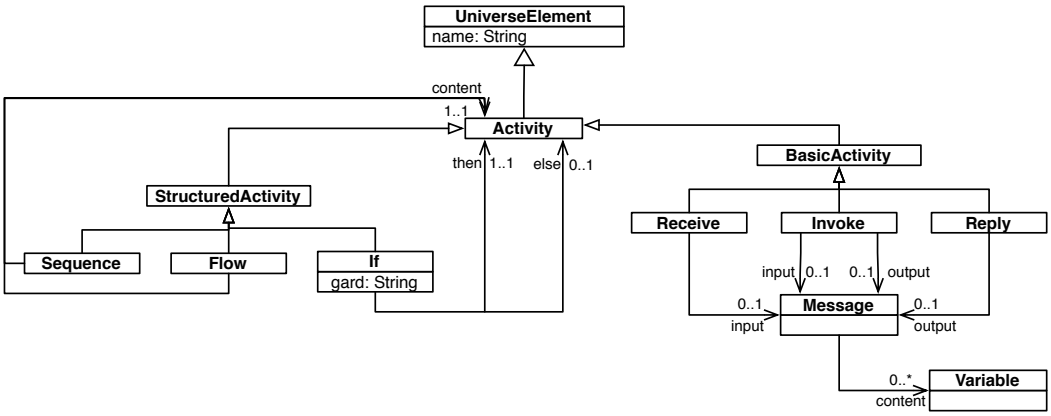


FIGURE 6.4 – Extrait du méta-modèle de SMILE : activités.

Nous modélisons l'ensemble des activités par la FIGURE 6.4. Nous distinguons différents types d'activités :

– **Réception** : l'activité de *réception* consiste en l'attente bloquante d'une information de la part des partenaires. Le plus souvent, nous retrouvons une activité de réception en début du processus métier, pour récupérer les données en entrée. Le partenaire est alors l'invoquant de l'opération. Une activité de réception est définie comme un tuple *Rcp(nom:String, output:Message)*. La FIGURE 6.5 est une partie du modèle de PICWEB représentant l'activité de réception d'un message, constitué d'une seule variable *keyword*.

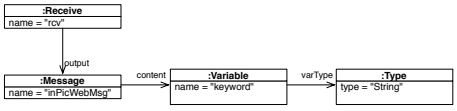


FIGURE 6.5 – Extrait du modèle de PICWEB : activité receive.

- **Envoi** : l'*envoi* est l'équivalent de l'activité de réception, mais pour transmettre des informations aux partenaires. Il sert le plus souvent à retourner le résultat d'un processus métier. Une activité d'envoi est définie comme un tuple *Env(nom:String, input:Message)*.
- **Invocation** : lorsqu'il est nécessaire de faire appel à d'autres services pour effectuer le traitement du processus métier, nous utilisons le type d'activité *invocation*. Les informations nécessaires pour qualifier une invocation sont le nom du service et l'opération à invoquer. Pour une invocation, il est également nécessaire de définir le message en entrée de l'opération invoquée, et de définir le message destiné à recevoir le résultat de l'opération. Une activité d'invocation est définie comme un tuple *Inv(nom:String, op:Operation, input:Message, output:Message)*.

En plus des activités présentés ci-dessus, nous introduisons d'autres activités, dites *structurantes*. Elles ont pour rôle de regrouper, sous une même entité, différentes activités, en ajoutant une sémantique supplémentaire. Ces activités sont les suivantes :

- **Condition** : Afin de pouvoir effectuer différents traitements selon une condition donnée, l'activité *condition* permet de décrire les différentes alternatives. Une activité de type *condition* est constituée d'une *condition de garde*, expression pour laquelle l'évaluation retournera *Vrai* ou *Faux*, et de deux activités nommées *Alors* et *Sinon*. Si l'évaluation de la garde retourne *Vrai*, l'exécution du processus métier se poursuivra par l'activité *Alors*. Si l'évaluation de la garde retourne *Faux*, l'activité exécutée sera l'activité *Sinon*, si elle est spécifiée. Dans le cas contraire, l'activité de condition se termine.
- **Séquence** : une activité de type *séquence* contient un ensemble ordonné d'activités, pour laquelle la sémantique d'exécution donnée consiste à exécuter les différentes activités dans l'ordre donné. Par exemple, les activités de la première version de PicWeb (Figure 4.3) sont toutes regroupées dans une même séquence.
- **Flot** : une activité de type *flot* est associée à une sémantique d'exécution différente de celle de *séquence*. En effet, aucun ordre n'est défini pour l'exécution des activités contenues : elles peuvent être exécutées les unes à la suite des autres et dans un ordre aléatoire, ou bien de manière parallèle. Le flot se termine lorsque l'ensemble des activités contenues a terminé son exécution. Un exemple d'activités de type *flot* consiste à considérer les deux activités *a2* et *a3* dans la Figure 4.3 (b). Ces deux activités sont contenues dans une activité de type *flot* : elles s'exécutent sans aucune garantie de leur ordre relatif.

6.2.6 Relations d'ordre

Afin de pouvoir définir l'ordre d'exécution des activités, il est nécessaire d'introduire un concept permettant de prioriser l'exécution des activités. Nous utilisons pour cela la notion de relation d'ordre partiel. Une *relation d'ordre* Ord($a1$, $a2$), ($a1$,$a2$) $\in$ ($Activity \times Activity$) est un couple d'activités tel que *a1* s'exécute avant *a2*. Grâce à cette notion, il est possible de définir un ensemble de relations d'ordre déterminant l'ordre dans lequel les activités vont s'exécuter. Par exemple, la Figure 6.6 représente la modélisation de l'ordre pour les activités *a1*, *a2* et *a3* de la Figure 4.3 (a). Toutefois, nous parlons d'ordre *partiel* puisque s'il est possible de définir l'ordre d'exécution entre deux activités, l'ordre d'exécution de trois activités n'est pas défini. En effet, si nous prenons les activités *a1* *a2* et *a21* de la Figure 4.3 (b), les activités *a2* et *a21* peuvent s'exécuter en parallèle, ou l'une avant l'autre, sans avoir nécessairement besoin de connaitre cet ordre.

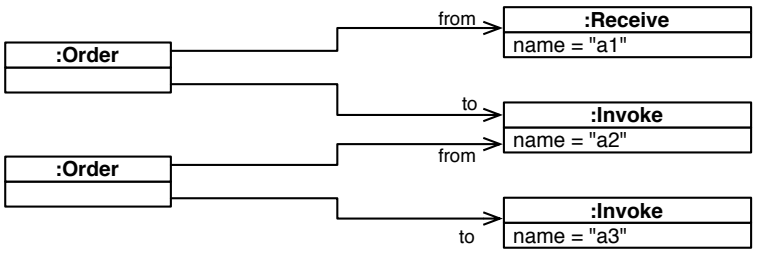


FIGURE 6.6 – Extrait du modèle de PICWEB : relations d'ordre.

Il est important de noter ici que la modélisation de l'ordre d'exécution est redondante avec la présence d'activités structurantes. En effet, puisque les activités structurantes sont associées à une sémantique d'exécution précise, dire que deux activités appartiennent à une séquence et dire qu'il existe une relation d'ordre entre ces deux activités est équivalent. Cette dualité permet à la fois d'être souple dans l'expression d'un flot de contrôle, par le biais des relations d'ordre partielle, tout en donnant un cadre hiérarchique aux différentes activités par le biais des activités structurantes. Ce dernier point permet notamment de pouvoir qualifier directement une activité structurante du point de vue de la QoS.

La FIGURE 6.7 représente la partie de notre méta-modèle responsable de modéliser les relations d'ordre.

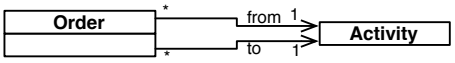


FIGURE 6.7 – Extrait du méta-modèle : relations d'ordre.

6.2.7 Processus métier

Nous avons vu dans la définition d'une opération qu'il était possible d'en définir l'implémentation par le biais d'un processus métier. Nous appelons *processus métier* la réalisation d'une opération par un ensemble d'activités ordonnées. Pour cela, nous réutilisons les concepts introduits précédemment : un processus métier a besoin de variables pour stocker le résultat des différents traitements, d'activités pour instancier les différentes tâches à effectuer, de messages pour communiquer avec les activités, et de relations d'ordre pour prioriser les activités. Enfin, un processus métier a besoin d'un point d'entrée, indiquant par quelle(s) activité(s) démarrer. La FIGURE 6.8 représente la partie de notre méta-modèle correspondant à un processus métier : formellement, nous définissons un processus métier comme un tuple *Proc(vars:Ensemble<Variable>, msgs:Ensemble<Message>, acts:Ensemble<Activity>, ord:Ensemble<Order>, init:EntryPoint)*.

6.2.8 Système

Il existe dans la littérature de nombreuses définitions des systèmes logiciels. Par exemple, le consortium OASIS définit un système comme étant *"une collection de composants organisés pour accomplir une fonction ou un ensemble de fonctions spécifiques"* [OASIS 2006]. Nous appelons *système logiciel* l'ensemble des ressources et infrastructures logicielles qu'une équipe de développement est amenée à gérer dans le cadre d'un projet. Un système peut ainsi être vu comme un agglomérat de services, pouvant être implémentés à l'aide de pro-

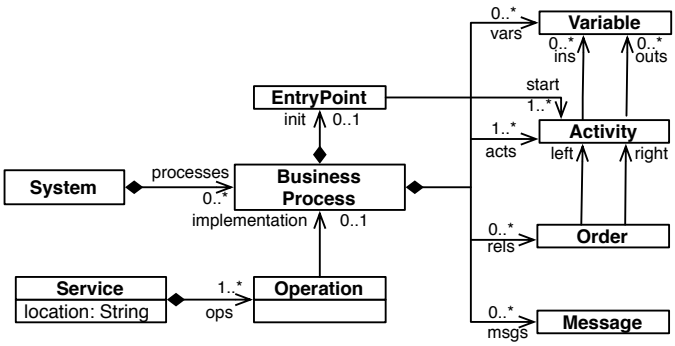


FIGURE 6.8 – Extrait du méta-modèle : processus métier.

cessus métiers, agissant dans un but commun. Formellement, on considère un système *Sys(process:Ensemble<BusinessProcess>, serv:Ensemble<Service>)*.

Nous venons de voir dans cette section l'ensemble des concepts nécessaires à l'architecte du système pour modéliser un système. Notre méta-modèle, représenté dans son intégralité en annexe, reprend les concepts communs aux langages de description d'architecture et aux langages de description de processus métier pour fournir un méta-modèle à la fois centré sur le processus métier, mais permettant de décrire la logique d'un système dans son ensemble, comme assemblage de plusieurs processus métiers permettant à l'architecte du système d'en définir son comportement. Dans la section suivante, nous nous intéressons à la causalité présente au cours de l'exécution du système.

6.3 Causalité de l'exécution d'un système

L'exécution d'un système peut être vue comme les modifications successives de son état [Turing 1936]. Dans le contexte de la thèse, ceci peut être vu comme une série d'interactions entre les différents éléments constituants le système. Par exemple, la fin d'une activité d'un processus métier peut engendrer le début d'une autre activité. Ce sont ces interactions qui modifient l'état du système, tout au long de son exécution. Lorsque l'architecte fait évoluer un système, il ajoute, modifie ou supprime des éléments. Cela a pour conséquence de changer la manière dont les éléments du système vont interagir, et par la même occasion influe sur leur qualité de service. Afin de pouvoir suivre ces interactions et par la suite calculer l'effet d'une évolution sur la qualité de service du système, il est nécessaire d'établir ces interactions de manière explicite. Ainsi, lorsqu'un élément du système évolue, savoir les conséquences de cette évolution consistera à déterminer l'ensemble des interactions que cet élément a avec le reste du système, pour pouvoir identifier quels sont les éléments pouvant être affectés. Notre objectif est de définir les causalités de notre application, afin de les utiliser lors de l'analyse d'une évolution pour prédire quelles seront les conséquences de l'évolution sur l'état du système. C'est grâce à ces causalités que nous parvenons à identifier concrètement l'effet d'une évolution sur le reste du système et sur sa qualité de service.

Dans cette section, nous nous intéressons à la notion de causalité, et définissons dans le contexte de l'exécution d'un système les relations de cause à effet qui sont présentes. Après avoir défini le concept de causalité et de relation causale, nous présentons un moyen d'établir la causalité, en vue de s'en servir pour analyser l'impact d'une évolution.

6.3.1 Mise en œuvre de la causalité

La notion de causalité a des origines provenant de plusieurs disciplines, passant des sciences physiques à la philosophie. En informatique, la causalité a été introduite à de nombreuses reprises également. Nous retiendrons tout particulièrement la définition de la causalité introduite par Judea Pearl [Pearl 2000]. Dans ces travaux, la notion de causalité est définie de manière informelle comme étant *"notre conscience de ce qui cause quoi dans le Monde, et en quoi cela est important"*. Dans notre contexte de système à base de services, nous nous centrons sur cette relation de "ce qui cause quoi". Nous appelons *relation causale* le couple $(X, Y) \in UniverseElement \times UniverseElement$, représentant l'influence de X sur Y, ou encore le fait que X cause Y. Ici, cette influence veut dire que le comportement de X régit celui de Y. De ce fait, dire que "X est une cause de Y" signifie que si l'état de X change, alors l'état de Y change également. Partant de cette notion, Rooney *et al.* introduisirent *l'analyse de cause racine* [Rooney 2004]. Il s'agit ici d'un "*processus conçu pour être utilisé dans l'investigation et la catégorisation des causes racines des événements liés à la sûreté, la santé, l'environnement, la qualité, la fiabilité et la production d'impacts*". En d'autres termes, ce processus permet, en se basant sur la notion de causalité, de remonter la chaîne de conséquences afin de trouver le facteur-clé à l'origine de la réaction en cascade ayant causé le changement de comportement d'un élément. Nous verrons dans le chapitre 8 que le principe d'analyse de cause racine est à la base de notre analyse des évolutions.

Pour établir la chaîne de conséquences constituée lors de l'application d'une évolution, nous avons besoin dans un premier temps de déterminer les changements d'états dans notre système, *i.e.*, les changements opérés par l'évolution, et de déterminer en quoi ces changements vont avoir une influence sur d'autres éléments. Cela nécessite au préalable d'établir les causalités au sein même de notre système. Cette section se concentre sur l'établissement des causalités. Notre objectif ici est de dresser une représentation de notre système d'un point de vue causal. Il s'agit donc d'un **modèle**, dont la préoccupation principale est de représenter en quoi un élément du système va avoir une influence sur d'autres éléments du système. Nous appelons ce modèle un *modèle causal.*

Définition 6.3.1 (*Modèle causal*)

Un modèle causal *est un couple C<A,R>, où A est l'ensemble des artéfacts du système et R l'ensemble des relations causales régissant les artéfacts étudiés.*

Nous verrons dans la suite du document qu'il existe différents types de relations causales au sein d'un système, et relatives aux propriétés de qualité de service étudiées. La question est donc maintenant de savoir quels types de relations causales peuvent opérer dans un système. Dans ce chapitre, nous nous concentrons dans un premier temps sur les relations causales régissant l'exécution d'une système, avant de voir dans le chapitre 7 les relations propres à la qualité de service.

6.3.2 Relations causales fonctionnelles

Le système décrit par l'architecte est interprété par une plate-forme d'exécution. Cette plate-forme est définie selon une sémantique précise, régissant les interactions entre les différents éléments du système. Il est donc important de voir que les éléments du système, par le biais de la plate forme d'exécution, interviennent dans l'état d'autres éléments. Cette forme d'interaction entre les éléments du système est ce que l'on cherche à représenter par des relations causales. Le but ici est de permettre à l'architecte du système de comprendre dans un premier temps quels éléments agissent sur d'autres éléments, pour pouvoir ensuite déterminer, partant du changement d'un élément, comment le reste du système va être

affecté.

Nous cherchons dans un premier temps à représenter les relations causales propres au comportement de la plate-forme d'exécution pour orchestrer les processus métier. Il est intéressant de noter ici que ces relations causales sont une représentation de la sémantique de la plate-forme d'exécution. Cette sémantique conditionne les relations existantes entre les éléments du système. Par exemple, prenons la notion d'appel à un service. Un service prend des données en entrée, qui régissent l'exécution du service, pour au final produire d'autres données en sortie. Nous pouvons dire ici que l'entrée du service influence le calcul du service, qui lui-même influence la donnée en sortie. Il s'agit bien ici de relations de causalité. Nous introduisons par la suite deux types de relations que l'on peut trouver dans un système.

Définition 6.3.2 (*Relation Causale de paramètre d'entrée*)

Soit O<nom,entrée,sortie> une opération du système. Selon la sémantique du moteur d'exécution, le message en entrée de l'opération conditionne l'exécution de l'opération. La relation causale de paramètre d'entrée représente l'influence du paramètre en entrée sur le comportement de l'exécution d'une opération. On note ainsi que ***entrée*** $\xrightarrow{in}$ ***S*** .

De manière similaire, le principe de relation causale de paramètre d'entrée établit ci-dessus s'applique sur le message produit par le service :

Définition 6.3.3 (*Relation Causale de paramètre de sortie*)

Soit O<nom,entrée,sortie> une opération du système. Selon la sémantique du moteur d'exécution, le message en sortie de l'opération est calculé à partir de l'exécution de l'opération. La relation causale de paramètre de sortie représente l'influence du comportement de l'exécution d'une opération sur le paramètre de sortie. On note ainsi que ***S*** $\xrightarrow{out}$ ***sortie*** .

6.3.3 Exemple

Pour illustrer les différents types de relations causales fonctionnelles présentés précédemment, nous nous plaçons dans le contexte du développement de PicWeb. La Figure 6.9 est le modèle causal obtenu en appliquant les définitions de relations causales de paramètre d'entrée et de sortie à PicWeb.

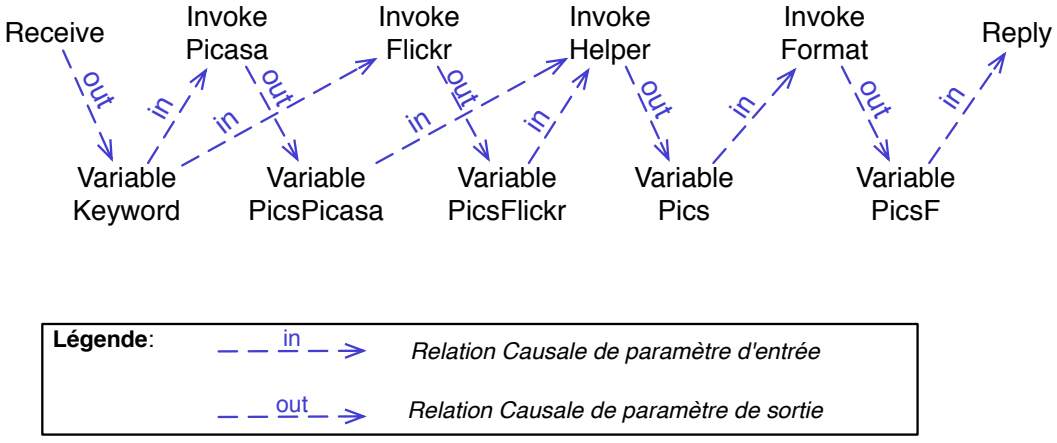


Figure 6.9 – Modèle causal fonctionnel de PicWeb.

L'observation de ce modèle permet de déterminer par exemple l'effet que pourrait avoir une modification de la variable *Pics* : en suivant les relations causales sortants de cette variable sur le modèle causal, l'architecte peut identifier l'activité *Format* comme directement affectée, mais également la variable *PicsF* et l'activité *Reply* par transitivité.

Dans cette section, nous avons présenté les concepts de causalité, de relation causale fonctionnelle et de modèle causal. En faisant le focus sur le moteur d'exécution du système, nous avons exhibé deux types de relations opérant dans un système. À l'aide de ces deux types, nous avons montré à quoi pouvait ressembler un modèle causal. Si la description faite ici ne représente qu'une infime partie de la sémantique du moteur, elle n'a pour but que de poser les bases nécessaires à la compréhension des modèles causaux et à montrer comment faire le lien entre la sémantique de la description du système et celle du moteur d'exécution, ainsi qu'avec la notion de causalité. Toutefois, s'il est possible de construire le modèle causal de PicWeb à la main, la prise en compte d'autres types de relations causales, ainsi que la taille plus conséquentes des systèmes sont des critères qui nous limitent dans cette méthode. Il est nécessaire pour l'architecte du système de pouvoir automatiser la constitution du modèle causal. Dans la suite de ce chapitre, nous introduisons une méthode permettant d'extraire les relations causales d'un système donné, en caractérisant les types de relations sous la forme de règles causales. Cette méthode est implémentée dans Smile, afin de déduire automatiquement le modèle causal d'un système.

6.4 Déduction des relations causales d'un système

Nous venons de définir la notion de modèle causal d'un système, permettant d'identifier les interactions entre ses éléments. Si son utilité n'est plus à montrer, une limitation réside dans son établissement. En effet, la construction manuelle d'un modèle causal est fastidieuse et sujette à erreur, compte tenu du nombre de relations causales présentes dans un système. Dans cette section, nous présentons une méthode permettant de déduire de manière automatisée le modèle causal fonctionnel d'un système. Après avoir présenté les principes de notre méthode, nous définissons la notion de *règle causale* qui nous sert à décrire, indépendamment de tout système, les types de relations causales que l'on peut déduire d'un système. Nous montrons enfin comment, en appliquant les règles causales sur un système donné, il est possible d'en déduire automatiquement ses relations causales.

6.4.1 Méthodologie

Le but de la méthodologie est de décrire une transformation permettant, en partant de la description d'un système, d'obtenir son modèle causal. Nous avons présenté précédemment différents types de relations causales fonctionnelles. Il est important de noter que ces types de relations sont décrits indépendamment du système étudié, mais en fonction de la plateforme d'exécution. Il convient donc de souhaiter "instancier" par la suite ces types de relations causales, pour un système donné. Pour cela, notre méthode est un procédé de déduction : il s'agit d'appliquer au système étudié des *règles* correspondant aux types de relations causales, pour en déduire ses relations causales.

Afin de pouvoir mettre en œuvre un tel procédé, nous introduisons la notion de règle causale.

Définition 6.4.1 (*Définition d'une règle causale*)

Une règle causale est une requête permettant de sélectionner dans un système donné des couples $(a, b) \in (SystemElement \times SystemElement)$ *tels qu'il existe une relation causale entre* a *et* b*.*

TABLE 6.1 – Règle causale pour les relations causales de paramètre d'entrée.

Situation	Action
A : Activity, M : Message, v : Variable **and** (A.type = **Invoke or** A.type = **Receive**) **and** $M = A$.input **and** $v \in M$.content	$v \xrightarrow{in} A$

Notre méthode consiste à définir en une seule fois les règles causales d'un moteur d'exécution, dans le but de les appliquer sur n'importe quel système pour en extraire ses relations causales. Pour cela, nous proposons le procédé représenté dans la FIGURE 6.10.

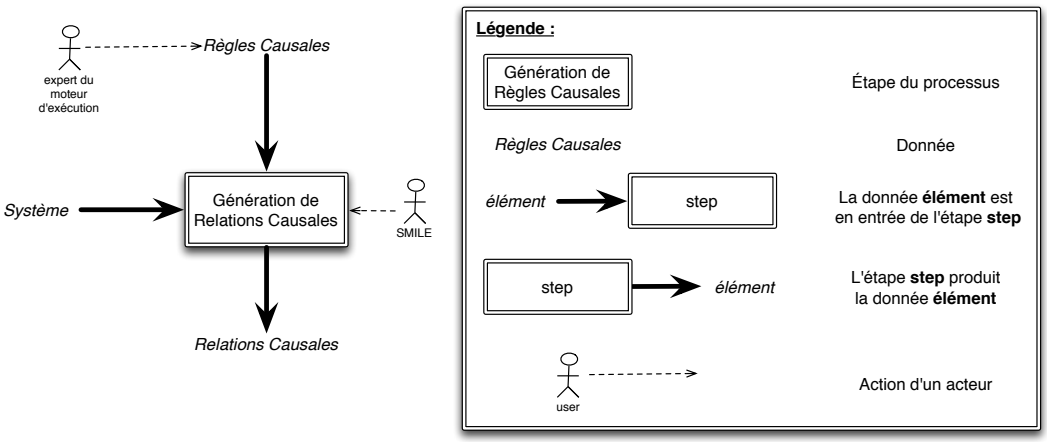


FIGURE 6.10 – Procédé de déduction des relations causales.

6.4.2 Expression des règles causales

Concrètement, nous avons choisi de représenter les règles causales comme un *système de règles de production*. Selon le spécification de l'OMG [OMG 2009a], une règle de production est *"une instruction de logique de programmation qui spécifie l'exécution de une ou plusieurs actions dans le cas où ses conditions sont satisfaites. Les règles de production ont donc une sémantique opérationnelle (formalisant les changements d'état, e.g., sur la base d'un formalisme d'un système de transition d'état)"*. Une règle de production est représentée comme une instruction *Si [condition] alors [liste d'actions]*. À titre d'exemple, la TABLE 6.1 et la TABLE 6.2 sont l'expression des règles causales correspondant respectivement aux relations causales d'entrée et de sortie. On parle de *Situation* pour décrire la condition, et d'*Action* pour l'ensemble des actions à effectuer. Dans le contexte de la règle causale de paramètre d'entrée, la situation décrite consiste à raisonner sur une activité A, un message M et une variable v. Nous posons comme condition que A soit une activité de type *Invoke* ou *Receive*, que M soit le message d'entrée de A, et que v soit contenue dans M. Dans ces conditions, la situation, *i.e.*, la relation causale à déduire, est $v \xrightarrow{in} A$. Le principe est similaire pour la règle causale de paramètre de sortie, en considérant ici les activités de type *Invoke* et *Reply*. En utilisant ces deux règles causales sur PICWEB, notre méthodologie produit le modèle causal de la FIGURE 6.9.

TABLE 6.2 – Règle causale pour les relations causales de paramètre de sortie.

Situation	**Action**
A : Activity, M : Message, v : Variable **and** (A.type = **Invoke or** A.type = **Reply**) **and** $M = A$.output **and** $v \in M$.content	$A \xrightarrow{out} v$

6.5 Conclusion du chapitre

Dans ce chapitre, nous nous sommes centrés sur les besoins de l'architecte du système. Nous avons introduit les différents concepts nécessaires pour modéliser un système. Partant de cette modélisation, nous nous sommes intéressés à la notion de causalité pour retranscrire les interactions existantes au cours de l'exécution d'un système. À travers la notion de modèle causal, nous avons établi un ensemble de relations causales telles que les relations causales de paramètres d'entrée et de sortie, représentant l'effet d'un élément du système sur un autre. Nous avons enfin présenté une méthode permettant, en exprimant une fois pour toute des types de causalité par le biais de règles causales, de déduire de manière automatique le modèle causal d'un système. Notre approche reste cependant limitée par l'intervention d'un expert du moteur d'exécution qui doit transcrire la sémantique du moteur en règles causales. De plus, si le modèle causal obtenu permet de représenter la causalité d'un point de vue fonctionnel, nous pouvons toutefois nous demander ce qu'il en est de la causalité en vue de l'étude de la variation de la qualité de service dans un système. Nous traitons ce point dans le chapitre suivant.

Chapitre 7

Modélisation de la qualité de service pour l'évolution

Sommaire

7.1 Motivations 65
7.2 Modélisation de la qualité de service d'un système 66
7.2.1 Propriété, unité et critère de comparaison 67
7.2.2 Domaine d'application 67
7.2.3 Valeur de propriété 68
7.2.4 Influence et ressource 68
7.2.5 Discussion 69
7.3 Définition des relations causales de propriété 69
7.3.1 Relations d'environnement 71
7.3.2 Relations de dérivation de propriétés 71
7.3.3 Relations d'agrégation de valeurs 73
7.3.4 Discussion 74
7.4 QoS4Evol, un langage de description de la qualité de service . . 74
7.4.1 Présentation de QoS4Evol 75
7.4.2 Définition du langage 75
7.4.3 Caractéristiques du langage 76
7.5 Déduction des règles causales d'une propriété 77
7.5.1 Principe 77
7.5.2 Obtention des relations causales d'environnement 78
7.5.3 Obtention des relations causales de dérivation 79
7.5.4 Obtention des relations causales d'agrégation 79
7.5.5 Discussion 80
7.6 Conclusion du chapitre 81

Dans l'étude d'un système, l'expert en qualité de service a pour rôle de s'assurer du maintien de la QoS du système, à tout moment de son cycle de vie. Il apporte son expertise du domaine pour vérifier le respect des contrats de QoS, et pour identifier les causes d'une potentielle violation. Nous présentons dans ce chapitre un ensemble d'outils nécessaire à l'expert en QoS pour étudier l'effet d'une évolution sur la QoS.

7.1 Motivations

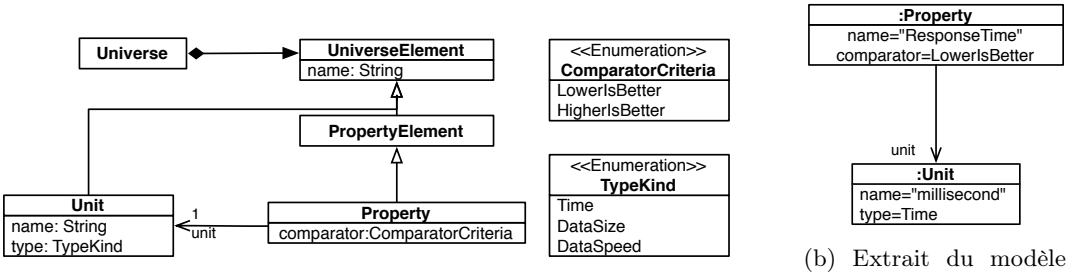
Pour rappel, nous considérons qu'une évolution est une modification des fonctionnalités du système dans le but de satisfaire de nouveaux besoins des différentes parties prenantes. Si une évolution cherche en premier lieu à satisfaire de nouveaux besoins fonctionnels, il

convient de s'interroger si celle-ci a également un effet sur les besoins non fonctionnels, et notamment sur la qualité de service. Selon les principes de notre cycle de développement BLINK, cette préoccupation est à la charge de *l'expert en QoS*. Celui-ci a pour tâche d'établir les contrats de qualité de service, de mettre en œuvre les outils de contrôle de la QoS, de s'assurer à chaque évolution qu'aucun contrat de QoS n'a été violé et, le cas échéant, il doit être en mesure de définir les causes de la violation. Concernant ce dernier point, nous avons défini dans le chapitre 6 le concept de relation causale d'un point de vue fonctionnel. Nous avons établi un modèle causal destiné à représenter ce qu'il se passe dans un système lors de l'exécution, permettant de comprendre quels éléments du système affectent d'autres éléments. Toutefois, la simple considération de ces relations ne permet pas de décrire la causalité de l'exécution du point de vue de la qualité de service. En guise d'illustration, faisons le parallèle à une machinerie mécanique. Dans ce contexte, la détermination du modèle causal fonctionnel établirait que le moteur a une influence sur les pièces mécaniques de la machine, qui ont elles une influence sur des rouages. En supposant un changement du moteur, le modèle causal indiquerait la chaîne de conséquences jusqu'aux rouages. En revanche, cette chaîne de conséquences n'indique pas de quelle manière la modification influence les rouages, c'est-à-dire si le changement de moteur a entraîné une modification de la vitesse de rotation du rouage, de son usure, de son sens de rotation, *etc.*. Dans une telle situation, il est donc prudent de vérifier l'ensemble des propriétés du rouage, ce qui est certes plus sûr, mais qui entraîne bien souvent un ensemble de vérifications inutiles.

Pour lever ces limitations, nous cherchons à représenter quels sont les facteurs d'influence d'une propriété, dans le but de cibler de manière plus précise quelles valeurs de propriété vont être affectées par une évolution. Ces facteurs d'influence ont pour vocation d'enrichir le modèle causal établi en introduisant des relations causales propres aux propriétés de qualité de service, afin de cibler l'identification des propriétés influencées et minimiser le nombre de vérifications à effectuer. La construction de ce nouveau modèle causal nécessite dans un premier temps d'être en mesure de pouvoir représenter les valeurs de propriété d'un système, afin de pouvoir les inclure en tant qu'élément conceptuel. L'expert en QoS a pour cela besoin de modéliser la qualité de service d'un système à l'exécution, comme présenté dans la section 7.2. Dans un deuxième temps, nous nous intéressons à ce qui peut influencer une valeur de propriété. Nous présentons dans la section 7.3 les différents facteurs d'influence, traduits par le biais de relations causales de QoS. Enfin, nous introduisons dans la section 7.4 un langage permettant de décrire des propriétés de qualité de service. C'est à partir de cette description que nous présentons dans la section 7.5 un procédé permettant de déduire de manière automatique les relations causales d'un système pour une propriété donnée.

7.2 Modélisation de la qualité de service d'un système

Nous avons vu dans le chapitre 2 que l'expert en qualité de service a à sa disposition un ensemble d'outils permettant de déterminer la qualité de service d'un élément du système : l'analyse statique de l'élément, le contrôle à l'exécution, ou le calcul à partir d'autres valeurs. Ces outils produisent différents ensembles de données sur lesquels l'expert devra raisonner dans le but de déterminer si oui ou non le système respecte les besoins de l'utilisateur final. Toutefois, la diversité des méthodes de détermination pose un problème d'hétérogénéité pour l'expert. En effet, la diversité des outils de détermination de la qualité de service entraîne une répartition des informations sur différents supports, rendant compliqué tout raisonnement. L'expert a besoin d'un support commun pour représenter ces informations, qui seront ensuite reprises dans le modèle causal.



(a) Extrait du méta-modèle de propriété.

(b) Extrait du modèle du temps de réponse

FIGURE 7.1 – Modèle et méta-modèle d'une propriété.

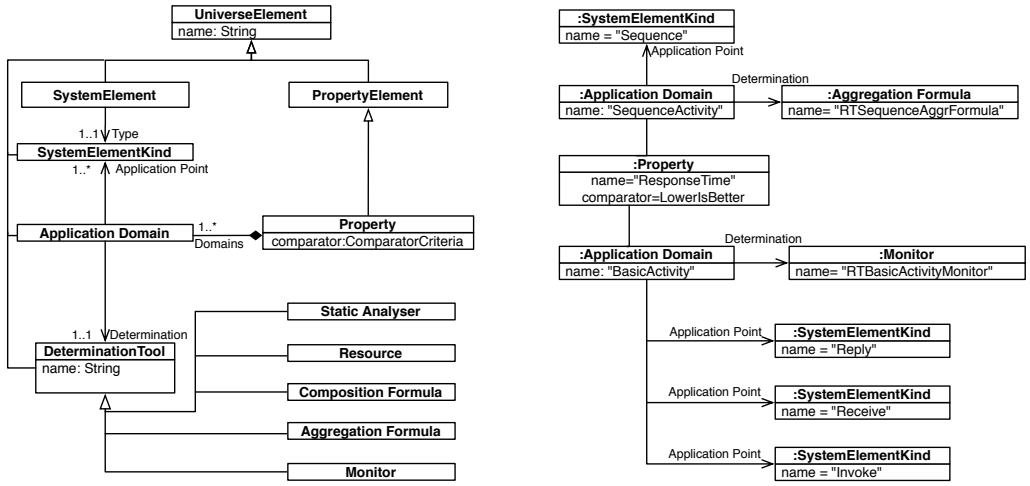
Nous proposons dans cette section de résoudre ce problème en définissant un méta-modèle pour la qualité de service permettant de centraliser les informations collectées. Nous présentons les différents concepts et les illustrons sur PICWEB en étudiant le temps de réponse.

7.2.1 Propriété, unité et critère de comparaison

Une *propriété* est un élément mesurable qui donne une information qualifiant la manière dont le système fonctionne. Il s'agit là du concept central que l'expert en QoS manipule, pour lequel il/elle doit définir un ensemble de caractéristiques. La FIGURE 7.1a est la partie du méta-modèle représentant les concepts présentés dans cette partie. Une propriété a un nom et une unité, représenté dans notre méta-modèle par les méta-éléments **Property** et **Unit**. Par exemple, l'expert en QoS créé une propriété "Temps de réponse", dont l'unité est "millisecondes", comme représenté dans la FIGURE 7.1b. Associée à la propriété, le *critère de comparaison* définit pour deux valeurs de cette propriété laquelle sera considérée comme la meilleure. Par exemple, cela permet de savoir lors d'une évolution si une valeur de propriété s'est améliorée ou s'est dégradée. Dans le cas du temps de réponse, l'expert déclare que plus petite la valeur est, le mieux c'est.

7.2.2 Domaine d'application

Les systèmes étudiés sont constitués de plusieurs éléments, de types différents. Par exemple, il existe dans PICWEB des activités de type Invoke, Receive, Reply. Nous voulons donner la possibilité à l'expert en QoS de définir une manière de calculer une valeur de propriété en fonction de chaque type d'élément. Par exemple, l'expert veut mesurer le temps de réponse pour les activités de type *Invoke*, mais calculer les temps de réponse d'une *séquence* en fonction des temps de réponse des activités qu'il contient. Nous appelons *type d'élément du système (SystemElementKind)* un type d'élément. Pour pouvoir définir comment déterminer la qualité de service pour un type d'élément donné, nous introduisons le concept de domaine d'application. Un *domaine d'application (ApplicationDomain)* associe une méthode de détermination (analyse, contrôle ou calcul) à un type d'élément. Nous représentons cela dans notre méta-modèle par la partie décrite dans la FIGURE 7.2a : un méta-élément *ApplicationDomain*, possède une relation avec une méthode de détermination (*DeterminationTool*) et un type d'élément du système (*SystemElementKind*). En guise d'illustration, la FIGURE 7.2b est la partie du modèle du temps de réponse représentant comment déterminer une valeur de propriété pour une activité simple (via le moniteur *RTBasicActivityMonitor*), et pour une séquence en décrivant le domaine d'application *SequenceActivity*.



(a) Extrait du méta-modèle de propriété. (b) Extrait du modèle du temps de réponse.

FIGURE 7.2 – Modèle et méta-modèle du domaine d'application.

7.2.3 Valeur de propriété

Une valeur de propriété (*PropertyValue*) est une valeur logique caractérisant un élément du système donné. La FIGURE 7.3 représente la partie du méta-modèle responsable de définir les valeurs de propriétés. Pour chaque élément du système pour lequel l'expert en QoS a défini un domaine d'application, une valeur de propriété sera créée. Si le méta-élément a pour but de représenter l'ensemble des mesures effectuées, il ne contient lui-même aucune valeur. En effet, les valeurs numériques concrètes sont représentées par le méta-élément *Measurement*. Ce choix de modélisation a été effectué pour simplifier la recherche de toutes les mesures effectuées pour un élément du système donné, et pour une propriété donnée. Enfin, afin de pouvoir regrouper les mesures effectuées au cours du même contexte d'exécution, nous introduisons le concept de trace d'exécution (*ExecutionTrace*). Ce concept a pour but de regrouper les valeurs de propriétés qui ont été capturées lors de la même instance d'exécution [Comes 2009]. Il est caractérisé par une *estampille* (*"timestamp"* en anglais), et par un *numéro de version* (*"versionNumber"* en anglais). Tandis que le premier a pour rôle de capturer l'heure précise à laquelle la trace a été enregistrée, le dernier sert à caractériser sur quelle version du système la trace a été capturé. Lors de chaque évolution, ce numéro est incrémenté.

La FIGURE 7.4 est un exemple de modèle représentant les données mesurées. Ici, l'expert s'intéresse à deux propriétés, le temps de transmission et le temps de calcul. Par souci de lisibilité, le lien entre une propriété et ses valeurs de propriété est représenté par un code couleur. Chaque activité est associée à deux valeurs de propriété, une pour chaque propriété. À chaque valeur est associée un ensemble de mesures. Ici, quatre traces sont présentes, représentant quatre appels à PICWEB.

7.2.4 Influence et ressource

Une valeur de propriété de qualité de service dépend de nombreux facteurs. Comme énoncé précédemment, il est courant que le contrôle à l'exécution d'une propriété fournisse des données très différentes pour les mêmes paramètres d'exécution. Cela est dû à l'influence des facteurs extérieurs, tels que l'utilisation de certaines ressources physiques, mais

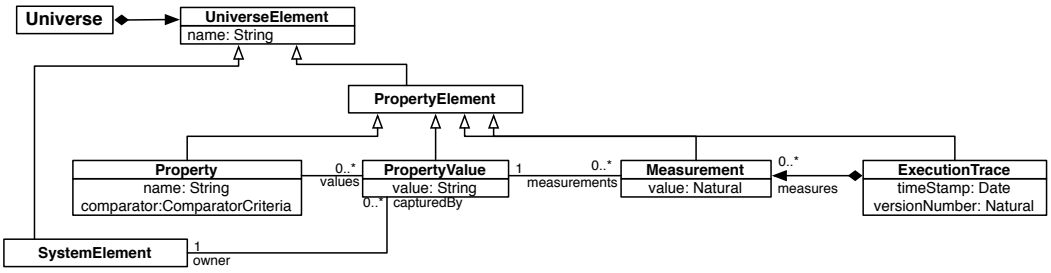


FIGURE 7.3 – Méta-Modèle de propriété : Valeurs de propriété.

également des facteurs internes tels que des éléments du système. Afin de donner la possibilité de représenter quel type de ressource peut influencer une valeur de propriété, nous introduisons le concept de *Resource* dans notre méta-modèle. Par exemple, la propriété de temps de réponse d'une activité dont l'implémentation est située sur une autre machine que celle de l'appelant sera influencée par le réseau. Également, l'expert souhaite représenter l'effet d'un message en paramètre d'entrée d'une activité sur son temps de transmission.

La FIGURE 7.5a représente la partie du méta-modèle responsable de définir les facteurs d'influence. Ces concepts sont illustrés dans la FIGURE 7.5b, représentant l'influence du réseau et l'influence d'un message en entrée sur une valeur de propriété.

7.2.5 Discussion

Le méta-modèle que nous venons de présenter sert de base de représentation des différentes données nécessaires par l'expert en qualité de service. Nous avons fait le choix de regrouper l'ensemble des mesures effectuées pour une propriété et pour une élément du système donnée autour de l'élément *PropertyValue*. Selon les méthodes de détermination des valeurs de propriété, le résultat de tout calcul, toute analyse ou toute mesure effectuée sur un élément du système est représenté dans le modèle de données et associé à l'élément analysé. Par cette méthode, nous gardons trace de la provenance des données. Nous avons ici le moyen de tracer pour une mesure le moment de sa capture (permettant de savoir si la donnée a été déterminée sur la version courante du système, ou sur une version antérieure), ainsi que le type d'outil l'ayant déterminé. Ainsi, nous fournissons un support unifié pour les différents outils de détermination de la qualité de service d'un système.

De plus, les éléments modélisés servent également de base pour enrichir le modèle causal du système : ce sont les nouveaux nœuds à introduire. Il s'agit donc maintenant de déterminer les nouvelles relations du modèle causal à introduire. Pour cela, nous nous intéressons aux différents éléments pouvant influencer une valeur de propriété.

7.3 Définition des relations causales de propriété

Dans cette section, nous nous intéressons à ce qui peut influencer une valeur de propriété. Cette influence peut provenir du comportement du système à l'exécution, de son environnement, ou encore des valeurs de propriétés entre elles. De ces facteurs d'influence, nous en réifions 3 types de relations causales de propriété : *i)* les relations causales d'environnement, lorsqu'un élément de l'éco-système autour de l'application (tel que le réseau) influence une valeur de propriété, *ii)* les relations causales de composition, lorsqu'une valeur de propriété rentre dans le calcul d'une autre propriété, et *iii)* les relations causales d'agrégation, lorsqu'une valeur de propriété d'une activité composite est influencée par une

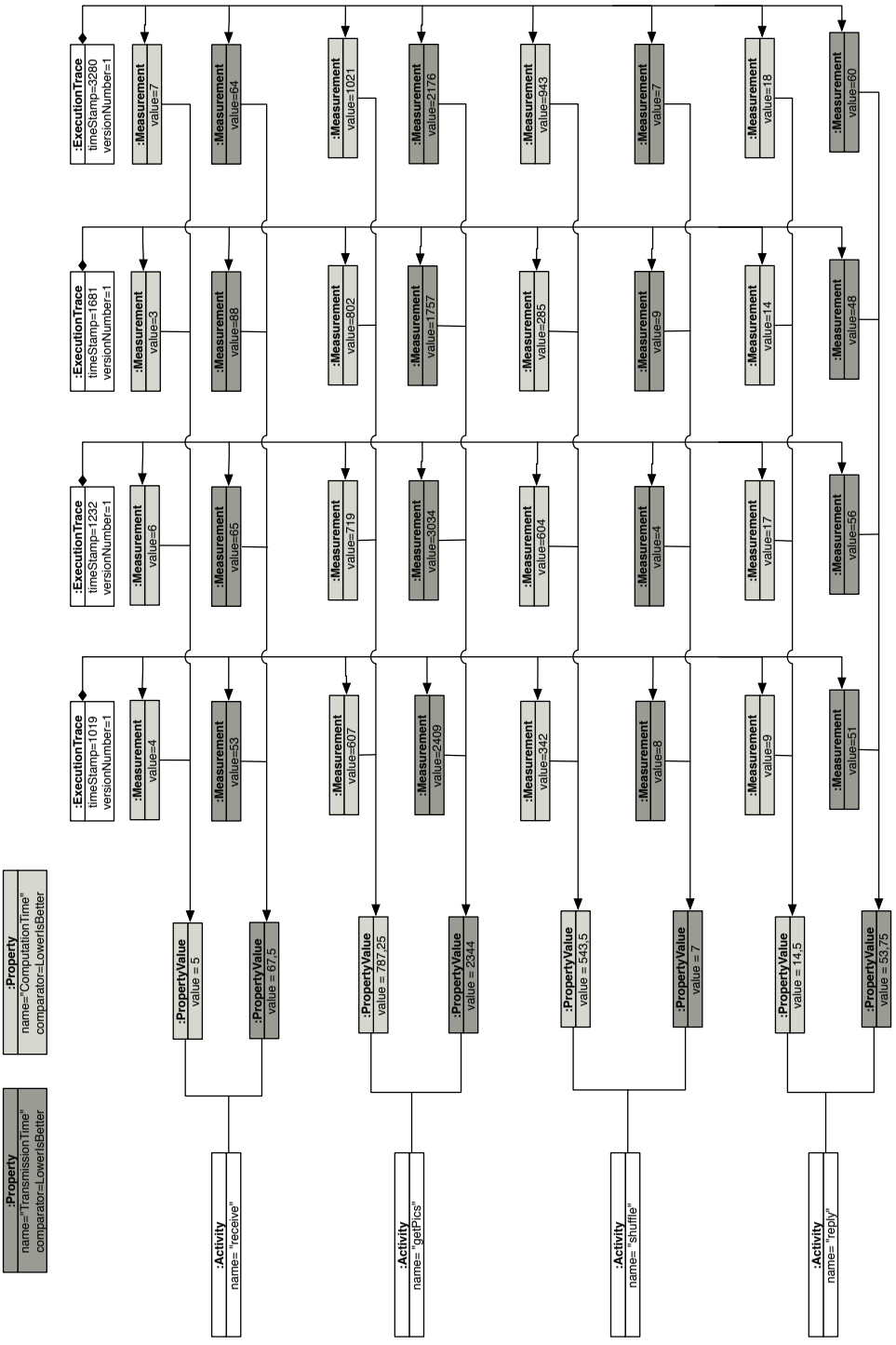
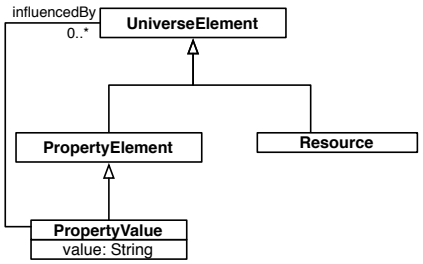
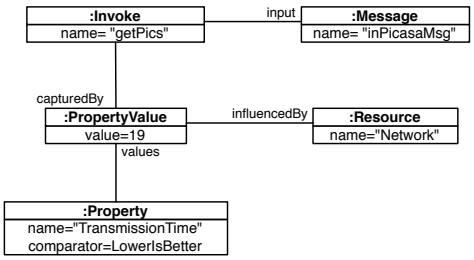


FIGURE 7.4 – Modèle du temps de réponse : Valeurs de propriété.

valeur de propriété d'une de ses activités contenues. Nous détaillons dans la suite chacun de ces types de relation causale en les illustrant sur PICWEB.



(a) Extrait du méta-modèle de propriété.



(b) Extrait du modèle du temps de réponse.

FIGURE 7.5 – Modèle et méta-modèle des facteurs d'influence.

7.3.1 Relations d'environnement

On appelle environnement toutes les ressources physiques et logiques extérieures (tels que les liens réseaux ou la mémoire) à la définition de la logique du système. De fait, les caractéristiques des ressources de l'environnement influent sur la qualité de service du système. Pour cela, nous introduisons une relation causale liée à l'environnement :

Définition 7.3.1 (*Relation d'environnement*)

Soient A un élément du système, R une ressource de l'environnement du système, P une propriété dont R est déclarée comme influence, et $PV_P(A)$ la valeur de propriété de A pour la propriété P. Une relation causale d'environnement représente la causalité établie entre une valeur de propriété et une ressource de l'environnement : on peut dire que $\boldsymbol{R} \xrightarrow{env} PV_P(A)$.

À titre d'exemple, nous voulons représenter que la rupture d'un lien réseau peut affecter le temps de réponse d'un service S. Ici, il est important de pouvoir indiquer qu'un lien rompu affectera le temps de réponse, mais n'affectera pas la logique métier propre au service. En revanche, nous pouvons dire que toute propriété de QoS, de type temporel, dont l'élément caractérisé utilise ce lien, sera affecté.

Nous établissons donc que $\boldsymbol{NetworkLink} \xrightarrow{env} PV_{ResponseTime}(S)$, comme décrit pour PICWEB dans la FIGURE 7.6.

Pour pouvoir utiliser ce type de relation causale, nous faisons ici l'hypothèse que l'information concernant l'état d'un lien réseau est établie à l'aide d'un moniteur, chargé d'observer le comportement des ressources de l'environnement. Ainsi, lorsque le moniteur détecte qu'un lien est rompu, il suffit de parcourir les relations causales d'environnement propre à la ressource *NetworkLink* pour déterminer en quoi cela impacte la QoS du système.

7.3.2 Relations de dérivation de propriétés

Nous avons vu qu'une propriété de QoS peut être définie comme étant dérivée d'autres propriétés, *i.e.*, qu'il est possible de déterminer une valeur de propriété en fonction des valeurs d'autres propriétés. Ces dernières ont une influence sur la valeur dérivée. Par le concept de relation de dérivation, nous voulons matérialiser cette influence.

Définition 7.3.2 (*Relation de dérivation*)

Soit un service S, caractérisé par trois propriétés, P1, P2 et P3, telles que P3 est dérivée de P1 et de P2. La valeur de S pour la propriété P3 est définie en fonction des valeurs de S pour P1 et P2. Par exemple, $PV_{P3}(S) = PV_{P1}(S) + PV_{P2}(S)$. P1 et P2

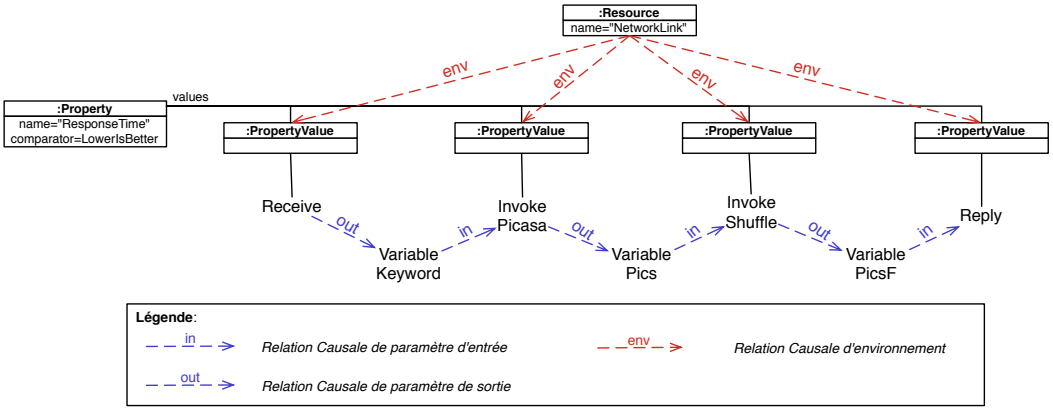


FIGURE 7.6 – Modèle causal enrichi avec les relations d'environnement.

ont une influence sur P3. Nous pouvons donc dire qu'il existe les relations causales de dérivation :

- $PV_{P1}(S) \xrightarrow{der} PV_{P3}(S)$
- $PV_{P2}(S) \xrightarrow{der} PV_{P3}(S)$

À titre d'exemple, nous considérons la définition du temps de réponse (RT) d'un service S comme étant la somme du temps de transmission (TT) et du temps de calcul (CT). Nous avons donc la définition $\mathrm{PV}_{RT}(\mathrm{S}) = \mathrm{PV}_{TT}(\mathrm{S}) + \mathrm{PV}_{CT}(\mathrm{S})$. Cette définition se traduit par deux relations causales de dérivation, $\boldsymbol{PV_{TT}(S)} \xrightarrow{der} \boldsymbol{PV_{RT}(S)}$ et $\boldsymbol{PV_{CT}(S)} \xrightarrow{der} \boldsymbol{PV_{RT}(S)}$. Appliquée à PICWEB, nous obtenons le modèle causal de la FIGURE 7.7. Ce genre de relation se déduit directement de la formule définissant le temps de réponse.

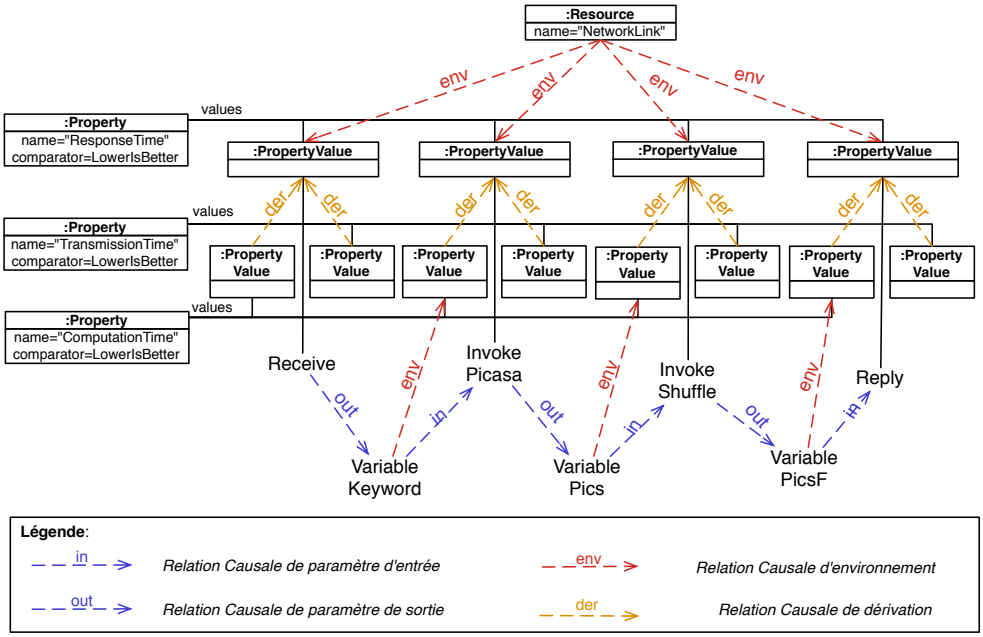


FIGURE 7.7 – Modèle causal enrichi avec les relations de dérivation.

7.3.3 Relations d'agrégation de valeurs

Un processus métier est un ensemble d'activités dont l'exécution est définie par des relations d'ordre partiel. Comme présenté dans le chapitre 6, il est possible de représenter cet ordre partiel par le biais d'activités structurées, constituant ainsi une hiérarchie en forme d'arbre, où chaque activité est imbriquée dans une autre activité. Sous cette forme, il est possible de déterminer la valeur de propriété d'une activité structurée en se basant sur les valeurs de propriété des activités qu'elle contient. Par extension, la valeur de propriété d'une activité structurante pour une propriété donnée, décrit par une formule d'agrégation, est influencé par les valeurs de propriété des activités la constituant. À ce titre, nous introduisons la notion de relation causale d'agrégation.

Définition 7.3.3 (*Relation d'agrégation*)

Soit C une activité structurante d'un processus métier, constituée de 3 activités, A1, A2 et A3. La propriété P pour une activité structurante est définie comme étant fonction des valeurs de propriétés des activités la constituant. Ici, les valeurs de propriété des constituants (A1, A2 et A3) ont une influence sur la valeur de propriété de l'activité structurante (C). Nous avons donc les relations causales d'agrégation
$\boldsymbol{PV_P(A1)} \xrightarrow{agr} \boldsymbol{PV_P(C)}$, $\boldsymbol{PV_P(A2)} \xrightarrow{agr} \boldsymbol{PV_P(C)}$ *et*
$\boldsymbol{PV_P(A3)} \xrightarrow{agr} \boldsymbol{PV_P(C)}$

À titre d'exemple, nous considérons le temps de réponse (RT) d'une activité structurante de type **Séquence** comme étant la somme des activités la composant. Si nous considérons la séquence présente dans PicWeb, son temps de réponse est la somme des temps de réponse des activités Receive, Picasa, Shuffle et Reply. Nous avons ici plusieurs relations causales d'agrégation, comme décrit dans la Figure 7.8. Dans cette figure, nous avons voulu représenter le modèle causal complet de PicWeb, c'est-à-dire constitué de la somme de toutes les relations causales, qu'elles soient fonctionnelles, d'environnement, de dérivation ou d'agrégation.

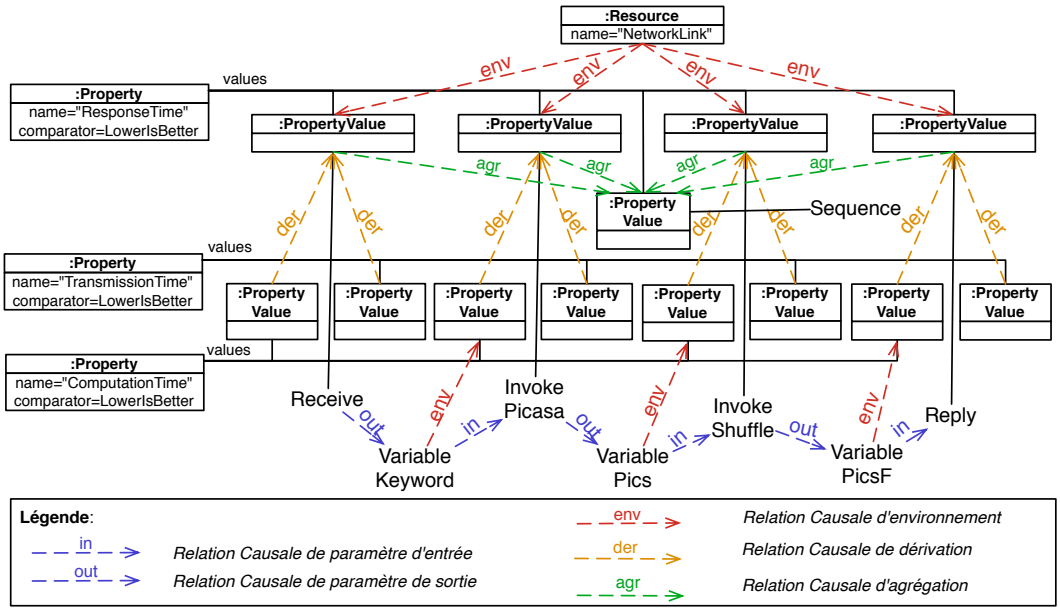


Figure 7.8 – Modèle causal enrichi avec les relations d'agrégation.

7.3.4 Discussion

Nous venons de voir les trois types de relations causales que nous pouvons définir sur une propriété de qualité de service. L'application de l'ensemble de ces relations causales permet de décrire de manière plus précise ce qui influence une valeur de propriété. En effet, là où le modèle causal fonctionnel indique les relations de causalités entre éléments du système, les relations causales introduites montrent directement comment les valeurs de propriétés ont une influence entre elles, et comment elles sont influencées par des éléments extérieurs.

Nous pouvons cependant nous poser la question de l'exhaustivité de ces types de relations. Nous avons ici émis l'hypothèse que les propriétés étudiées étaient influencées par l'environnement, par d'autres valeurs de propriétés, ou pas le comportement du système. À partir de ces relations, nous donnons une indication sur l'interaction entre les éléments du système afin de définir le périmètre des éléments affectés. Nous pourrions aller plus loin dans l'élaboration du modèle causal, comme par exemple en caractérisant plus finement l'influence d'une ressource à l'aide d'une fonction d'utilisation [Crnkovic 2005]). Dans un tout autre domaine, nous pourrions également aller plus loin en caractérisant si l'influence représentée par une relation causale est positive ou négative pour une propriété donnée, par le biais par exemple de règles *helps/hurts* [Chung 2009, Bass 2003, Perez-Palacin 2014]. Ces points seront abordés dans des travaux futurs.

Les différents types de relations causales que nous venons de voir forment avec les relations causales du chapitre 6 un modèle causal dont la construction manuelle serait source d'erreurs et trop couteux en temps. Nous cherchons à présent à définir une méthode pour obtenir le modèle causal de façon automatique. Pour cela, nous avons besoin dans un premier temps d'un support commun regroupant l'ensemble des informations sur une propriété de QoS. Dans la section suivante, nous présentons QoS4Evol, un langage permettant la description de propriétés. À partir de ces descriptions, nous verrons ensuite que notre canevas, Smile, est en mesure de les analyser pour en extraire des règles de transformations permettant de déduire le modèle causal enrichi de n'importe quel système.

7.4 QoS4Evol, un langage de description de la qualité de service

Nous avons vu dans les sections précédentes que la détermination des propriétés de qualité de service pour un processus métier nécessitait un ensemble d'informations en provenance de l'expert en Qualité de Service. Ces informations, qualifiant si une valeur de propriété est déterminée par analyse, par mesure ou par calcul, sont représentables dans le modèle causal. La formulation de toutes ces informations est réalisée à différents moments du cycle de vie, et ne disposent pas à l'heure actuelle d'un support d'expression centralisé. En effet, l'expert doit manuellement déclencher les analyses et déployer les moniteurs d'observation. De plus, nous cherchons à permettre l'expression des relations causales. Pour cela, il nous faut cette centralisation, mais il nous faut également pouvoir définir l'influence que certains éléments du système peuvent avoir sur les autres éléments. Nous proposons dans cette section un langage pour unifier l'expression de ces informations, nommé *QoS4Evol*. Ce langage a pour but d'apporter une manière simple à l'expert, pour décrire une propriété dans sa manière de la déterminer, de la calculer, ainsi que ses facteurs d'influence. Nous présentons ici la grammaire du langage, avant de montrer par la suite comment il est possible d'en déduire des règles causales à appliquer sur le système étudié pour obtenir automatiquement un modèle causal enrichi.

```
1  Property RT{
2    Unit : ms;
3    Range : positive;
4    BasicComputation: CT + TT;
5    ApplicationPoint:
6    Sequence is Sum(children);
7    Flow is Max(children);
8    Loop is k × children;
9  }
10
11 Property CT{
12   Unit : ms;
13   Range : positive;
14   BasicComputation: monitor named CTMonitor;
15   IsInfluencedBy: Input(a);
16 }
17
18 Property TT{
19   Unit : ms;
20   Range : positive;
21   BasicComputation: monitor named TTMonitor;
22   IsInfluencedBy: Network;
23 }
```

FIGURE 7.9 – Définition du Temps de Réponse.

7.4.1 Présentation de QoS4Evol

Afin de pouvoir exprimer une propriété de QoS, nous définissons un langage spécifique au domaine (DSL) décrivant l'ensemble des informations nécessaires à la détermination de valeurs de propriété pour le système [Mernik 2005]. *QoS4Evol* est un DSL textuel, inspiré du langage CQML+ [Rottger 2003] et supporté par notre canevas SMILE. Dans la suite, nous détaillons chaque instruction du langage en utilisant ce dernier pour modéliser une propriété, à savoir le Temps de Réponse. Cette propriété est un propriété dérivée, définie comme étant la somme du temps pour transmettre un message (Temps de Transmission) et le temps pour exécuter le dit service (Temps de Calcul). La FIGURE 7.9 est la description du Temps de réponse écrite par l'expert en qualité de service et conforme à notre langage. Il est important de noter que la description d'une propriété est indépendante du système à étudier. Ces informations ont besoin seulement d'être décrites une fois, pour être ensuite appliquées par SMILE sur n'importe quel système pour calculer les valeurs de propriétés. La description proposée fournit les informations usuelles pour gérer une propriété de QoS d'un système.

7.4.2 Définition du langage

Une propriété est définie par :

Déclaration de l'unité et du domaine de valeur :

Considérant l'hypothèse que les valeurs de propriété sont des valeurs numériques, il est nécessaire de définir une unité, et un domaine de valeurs autorisées. Par exemple, le temps de réponse est défini comme une quantité en millisecondes, qui est toujours positif. Dans notre langage, l'unité de la valeur est décrite à l'aide du mot-clé **Unit**, et le domaine de valeur à l'aide du mot-clé **Range**. Ces instructions sont utilisées par exemple aux lignes 2

TABLE 7.1 – Formules d'agrégation du temps de réponse.

	Temps de réponse
Séquence	$\sum RT(Sequence.children)$
Flot	$\max(RT(Flow.children))$
Boucle	$k \times RT(Loop.activity)$

et 3 de la FIGURE 7.9

Détermination des valeurs pour les éléments basiques du système :

La technique de détermination de la QoS est fondée sur la composition, c'est-à-dire que les valeurs de propriétés sont déterminées au niveau le plus bas (ici pour des activités simples), pour être ensuite composées pour des éléments de plus haut niveau (les activités structurantes, les processus métiers, et le système). Par exemple, la taille d'un message est calculée par composition de la taille des variables le composant. Dans ce cas précis, les éléments de base sont les variables. Nous considérons ici trois moyens pour calculer une valeur de propriété pour les éléments basiques : par analyse statique, par contrôle, ou par calcul. (voir section 2.2.2). Dans notre exemple, l'expert a introduit trois propriétés : le temps de réponse (RT), le temps de calcul (CT) et le temps de transmission (TT). Tandis que le temps de calcul et le temps de transmission sont décrits aux lignes 14 et 21 comme étant déterminés par des moniteurs (respectivement nommés **CTMonitor** et **TTMonitor**), le temps de réponse est déterminé comme étant la somme du temps de calcul et du temps de transmission (voir ligne 4).

La déclaration des analyseurs statiques et des contrôleurs permet à SMILE d'effectuer de manière automatique les analyses, et de positionner les contrôleurs dans le système. Dans notre cas, le Temps de Réponse est défini comme une propriété dérivée. Le Temps de Calcul (CT) et le Temps de Transmission (TT) sont mesurés en utilisant les contrôleurs fournis.

Détermination des valeurs de propriétés pour les éléments composites :

Pour calculer la valeur de propriétés d'éléments composite comme les activités structurantes, l'expert en qualité de service décrit une *formule d'agrégation*, comme on peut en trouver dans la littérature ([Mukherjee 2008]) et rappelée ici dans le tableau 7.1 : il s'agit ici de définir la valeur de propriété d'une activité structurante en fonction de la qualité de service des éléments qu'elle contient. Par exemple, le temps de réponse d'une séquence est la somme des temps de réponse de ses activités filles. De telles formules permettent de calculer une valeur de propriété pour une activité structurante, basée sur les valeurs de propriétés des activités la constituant. Dans notre langage, l'expert en QoS définit dans la section *ApplicationPoint* la formule d'agrégation pour les activités *Sequence*, *Flow* et *Loop* (voir lignes 5 à 8).

Influence de l'environnement :

Les formules définies précédemment donnent une manière de calculer une valeur de propriété pour chaque élément du système. Cependant, cette formule n'explicite pas le fait qu'une valeur de propriété peut être influencée par des facteurs extérieurs. Modéliser le fait qu'une ressource spécifique influence une valeur de propriété permet d'inclure cette influence dans notre analyse de l'évolution. Ici, l'expert en QoS écrit la dépendance de la valeur de propriété à d'autres facteurs. Par exemple, l'expert définit que le temps de transmission d'une activité est influencé par le délai du réseau.

7.4.3 Caractéristiques du langage

Le langage que nous venons de définir fournit les caractéristiques suivantes :

Indépendance au système : nous proposons ici d'abstraire complètement la description d'une propriété du système sur lequel elle sera appliquée. Nous obtenons ainsi des propriétés *réutilisables*, indépendantes du système considéré.

Modularité de l'expression : la définition des *Points d'Applications* permet de diviser les analyses en fonction du type d'élément étudié. Cela permet à l'expert en QoS de se focaliser sur un type d'élément à la fois.

Expressivité de la causalité : chacune des sections définies dans la description permet d'établir une règle de causalité du système donné. Nous donnons également la possibilité de décrire comment l'environnement va pouvoir influencer le comportement du système, et donc la valeur de la propriété.

En nous basant sur cette description, nous cherchons maintenant à appliquer la définition d'une propriété sur un système donné pour en déduire les relations causales propres à cette propriété. Pour cela, nous présentons dans la section suivante une méthode de déduction des règles causales d'un système propre à une propriété de qualité de service. Cette méthode repose sur le même principe que la méthode de déduction du modèle causal fonctionnel présenté dans le chapitre 6, en se basant sur des règles causales appliquées au système étudié. Toutefois, l'approche appliquée dans ce chapitre diffère dans la méthode d'obtention des règles causales : alors qu'un expert de la plate-forme était obligé d'écrire manuellement les règles causales régissant la plate-forme d'exécution, nous allons ici déduire les règles causales de la description de la propriété.

7.5 Déduction des règles causales d'une propriété

L'ensemble des relations causales présentées dans la section 7.3 permet de classifier les différentes dépendances d'un point de vue qualité de service. Si ces dépendances permettent de représenter la causalité d'une propriété, la question de savoir comment obtenir ce genre de modèle reste entière. En effet, la construction manuelle d'un modèle causal, avec un ensemble non négligeable de relations, ne peut pour des raisons évidentes s'envisager : la taille du système, l'aspect fastidieux de la collecte des relations, et le haut risque d'erreur dans la construction nous amène à rejeter l'hypothèse d'une construction à la main. Nous proposons dans la suite une méthode pour enrichir automatiquement le modèle causal d'un système avec les relations causales de propriété. Dans un premier temps, nous présentons comment les informations contenues dans la description d'une propriété de QoS peuvent être transformées en règles de déduction des relations. Puis, nous montrons comment les règles causales sont appliquées à un système donné pour obtenir les différentes relations. Enfin, nous montrons sur notre exemple comment cette méthode peut être appliquée.

7.5.1 Principe

La méthode que nous mettons ici en œuvre se déroule en deux temps : il s'agit d'abord de déduire de la description de la propriété un ensemble de règles causales. Ces règles causales sont, de la même manière que dans le chapitre 6, indépendantes de tout système. Dans un deuxième temps, les règles déduites sont appliquées au système étudié pour obtenir ses relations causales propres à la propriété étudiée.

La FIGURE 7.10 est une vue schématisée de notre approche : le procédé est initié en entrée par la description de propriété. La première étape consiste à appliquer sur la description de la propriété des règles de production que nous détaillons dans la suite de ce paragraphe. Ces règles de production génèrent en sortie d'autres règles de production, que nous appelons règle causale. Dans un deuxième temps, nous appliquons les règles de productions générées

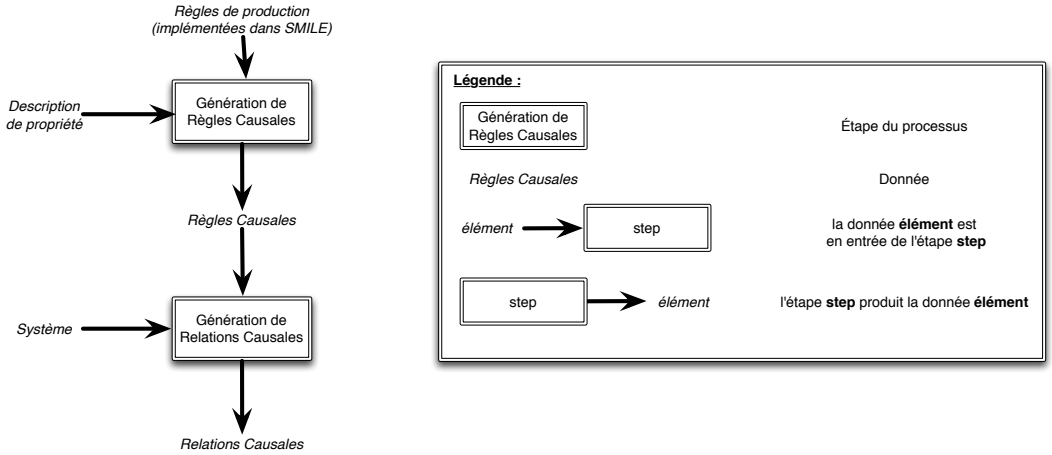


FIGURE 7.10 – Chaîne de déduction des relations causales de propriété.

TABLE 7.2 – Règle de production des relations causales d'environnement.

Situation	**Action**	
	Situation	**Action**
P.influencedBy = **R**	PV.getProperty = **P**	$\boldsymbol{R} \xrightarrow{env} PV$

TABLE 7.3 – Règle causale d'environnement générée.

Situation	**Action**
PV.getProperty = TT	$Network \xrightarrow{env} PV$

sur le système à étudier, pour en déduire les relations causales de propriété.

Nous présentons dans la suite l'application de notre méthode pour chacun des trois types de relations causales que nous avons présenté dans la section 7.3.

7.5.2 Obtention des relations causales d'environnement

Pour rappel, une relation causale d'environnement établie la causalité entre une ressource de l'environnement et une valeur de propriété. Dans le langage de description des propriétés que nous avons défini, ce genre de dépendance peut-être déduit de la section **influencedBy**. Par exemple, l'expert a déclaré dans la FIGURE 7.9 que le temps de transmission est influencé par le réseau. Ainsi, pour toute valeur de propriété PV_{TT} du temps de transmission, il existe une relation causale $\boldsymbol{Network} \xrightarrow{env} PV_{TT}$. Ce comportement est identique à toute ressource déclarée comme source d'influence. De ce fait, nous pouvons écrire une règle de production qui est applicable à n'importe quelle section *influencedBy*. Cette règle est consignée dans le TABLE 7.2.

Une fois ce patron appliqué à la description de propriété (lors de l'étape de génération de règles), nous obtenons un ensemble de règles causales d'environnement. Par exemple, la génération de règles a produit plusieurs règles, dont celle consignée dans le TABLE 7.3.

Il est important de noter que chacun des patrons écrits dans cette section est universel : il est valide pour toute règle causale de dérivation, d'agrégation ou d'environnement.

TABLE 7.4 – Règle de production des relations causales de dérivation.

Situation	Action	
P.*basicComputation*=f(**Props**)	**Situation**	**Action**
	$\forall P_x \in$ **Props**, PV_1.prop=P_x **and** PV_2.prop=**P** **and** PV_1.owner=PV_2.owner	$PV_1 \xrightarrow{der} PV_2$

TABLE 7.5 – Règle causale de dérivation générée.

Situation	Action
$\forall P_x \in$ **{TT,CT}**, PV_1.prop=P_x **and** PV_2.prop=**RT** **and** PV_1.owner=PV_2.owner	$PV_1 \xrightarrow{der} PV_2$

7.5.3 Obtention des relations causales de dérivation

Une relation causale de dérivation intervient lorsque une propriété est définie à l'aide d'une formule de dérivation. Cette information est également présente dans la description de la propriété : dans la section **BasicComputation**, une valeur possible pour cette section est d'indiquer une formule de dérivation. Le cas échéant, chacune des opérandes de la formule est une source d'une relation causale de dérivation. Pour construire les relations causales de dérivation, il s'agit donc de trouver toutes les valeurs de propriétés dérivées, et créer une relation entre chaque opérande de la formule de dérivation et la propriété dérivée. Pour obtenir ces règles causales de dérivation, nous avons écrit une autre règle de production spécifique aux relations causales de dérivation. Celui ci est consigné dans le TABLE 7.4.

Une fois la règle de production appliquée sur la description de propriété, nous obtenons une règle causale de dérivation similaire à celle consignée dans la TABLE 7.5. Celle-ci sélectionne toute propriété dont la méthode de calcul basique est une fonction dont les paramètres sont l'ensemble **Props**. Partant de l'ensemble des paramètres, la règle de production générée recherche les valeurs de propriété correspondant à la propriété donnée (PV_1.prop=P_x), ainsi que les valeurs de propriété de P (PV_2.prop=**P**), et sélectionne l'unique couple PV_1 et PV_2 tel qu'ils caractérisent le même élément du système (PV_1.owner=PV_2.owner). Cette situation résulte en l'action de la création de la relation causale de dérivation $PV_1 \xrightarrow{der} PV_2$.

7.5.4 Obtention des relations causales d'agrégation

Les relations causales d'agrégation interviennent pour calculer la valeur de propriété d'une activité structurante à partir des valeurs de propriété de ses activités contenues. Pour cela, il suffit de collecter l'information à partir de la section **ApplicationPoint** de la description de propriété. Nous décrivons dans le TABLE 7.6 la règle de production correspondant aux règles causales d'agrégation. Une fois appliquée à la description du temps de réponse, notre méthode génère la règle de production décrite dans le TABLE 7.7, correspondant au point d'application des activités de type Séquence. La situation de cette règle est constituée d'un ensemble de conditions, le but étant de sélectionner les valeurs du temps de réponse d'une séquence et de ses activités filles. Pour cela, nous raisonnons sur deux acti-

TABLE 7.6 – Règle de production des relations causales d'agrégation.

Situation	**P**.*ApplicationPoint.isStructuredType*=**True and** **P**.*ApplicationPoint*=*f*(**Children**)	
Action	**Situation**	**Action**
	A_1, A_2 : Activity **and** A_1.type=**P**.applicationDomain. applicationPoint **and** $A_2 \in A_1$.content **and** PV_1.prop = P **and** PV_2.prop = P **and** PV_1.owner = A_1 **and** PV_2.owner = A_2	$\mathbf{PV}_P(\mathbf{A}_1) \xrightarrow{agr} \mathbf{PV}_P(\mathbf{A}_2)$

TABLE 7.7 – Règle causale d'agrégation générée.

Situation	**Action**
A_1, A_2 : Activity **and** A_1.type=**Sequence and** $A_2 \in A_1$.content **and** PV_1.prop = **RT and** PV_2.prop = **RT and** PV_1.owner = A_1 **and** PV_2.owner = A_2	$\mathbf{PV}_{RT}(\mathbf{A}_1) \xrightarrow{agr} \mathbf{PV}_{RT}(\mathbf{A}_2)$

vités A_1, A_2 telles que A_1 correspond aux activités de type *Séquence* (A_1.type=**Sequence**), et A_2 les activités qu'elle contient ($A_2 \in A_1$.content). Nous raisonnons ensuite sur l'ensemble des valeurs de propriété, en sélectionnant les valeurs correspondant au temps de réponse (PV_1.prop = **RT and** PV_2.prop = **RT**), pour enfin sélectionner les deux valeurs de propriétés attachées à A_1 et A_2.

Nous ne montrons ici que l'application de la règle de production pour le point d'activité *séquence*, l'équivalent pour le *flot* (*"flow"* en anglais) et la *boucle* (*"loop"* en anglais) étant similaire.

Une fois que les règles causales ont été déduites, il est possible de les appliquer à n'importe quel système pour en extraire ses relations causales de propriété. Comme décrit dans le chapitre 6, SMILE applique les règles causales de propriété sur le système pour produire un modèle causal correspondant à celui que nous avons décrit dans la section 7.3 (voir FIGURE 7.8).

7.5.5 Discussion

Le procédé que nous venons de décrire permet de déduire de manière automatique le modèle causal de propriété d'un système, en se basant sur la description d'une propriété. Nous avons identifié un ensemble de trois types de relations causales propres aux propriétés de qualité de service. Toutefois, notre approche dépend fortement de la description de la propriété réalisée par l'expert en QoS. Elle est ainsi limitée dans le sens où des informations supplémentaires doivent y être inscrites, nécessitant un effort supplémentaire de la part de l'expert en QoS.

Malgré cette limitation, notre approche améliore le modèle causal du système décrit dans le chapitre 6, en apportant une granularité plus fine dans la détermination des causes et des conséquences au sein d'un système. De plus, les règles de production décrites pour générer les règles causales sont génériques pour toutes les propriétés : il suffit pour l'expert en qualité de service de décrire sa propriété pour obtenir les règles causales correspondantes,

pour pouvoir les appliquer à n'importe quel système.

Cette contribution permet d'apporter une réponse au défi **"interactions entre les différentes parties d'un logiciel"**, en enrichissant le modèle causal par des relations causales propres à la qualité de service. En augmentant notre modèle causal de relations propres à la QoS, nous apportons également des éléments de réponses au **"minimisation de la vérification"** : grâce à un niveau de précision supérieur, nous permettons ici de cibler au plus près les valeurs de propriété à re-vérifier.

7.6 Conclusion du chapitre

Dans ce chapitre, nous avons proposé un modèle de données permettant de modéliser la définition d'une propriété de QoS, ainsi que la représentation unifiée des données de détermination. Pour faciliter la tâche de l'expert, nous avons défini un langage spécifique au domaine permettant de décrire des propriétés de qualité de service. L'intérêt de la définition d'un nouveau langage ici provient du besoin d'exprimer les influences entre les différents éléments du système. En effet, la plupart des langages existants se focalisent sur la manière d'observer la propriété. Il est important de noter toutefois que ce prototype de langage est à fin expérimentale. Dans un futur proche, il serait utile d'étendre un langage existant avec les concepts de ce langage, afin de bénéficier de toute la puissance du langage étendu.

À partir de la description de propriété, nous avons présenté une méthode permettant d'extraire automatiquement des règles causales spécifiques à la propriété, dans le but de les appliquer de manière automatique à n'importe quel système afin d'en extraire un modèle causal enrichi. Cette méthode permet de s'affranchir d'une construction manuelle qui aurait été coûteuse en temps et sujette à erreur.

De manière générale, si notre solution pour l'expert en qualité de service diminue et simplifie la gestion de la qualité de service en unifiant l'ensemble des outils dans une seule description, un effort reste à faire pour automatiser complètement la gestion de la qualité de service. En effet, pour chaque outil envisagé, il sera nécessaire de définir une transformation des données produites par l'outil pour les intégrer dans le modèle unifié. De plus, si notre canevas permet de s'assurer de la cohérence du modèle de données tout au long du cycle de vie du système, le principe de pérennité des données capturées au cours de la détermination reste à la charge de l'expert. Il s'agira à l'avenir de réfléchir à une méthode automatique visant à supprimée les données obsolètes, après un certain nombre d'évolutions par exemple.

Dans le chapitre suivant, nous nous intéressons au rôle de l'architecte de l'évolution. Nous présentons son rôle et expliquons comment ce modèle causal enrichi peut l'aider dans l'analyse d'une évolution.

Chapitre 8

Analyse de l'évolution du logiciel orientée QoS

Sommaire

8.1 Présentation générale du processus d'évolution 84
8.1.1 Enjeux 84
8.1.2 Spécification du processus d'évolution 84
8.1.3 Hypothèses 85
8.2 Un langage pour l'évolution des processus métier 86
8.3 Spécification de l'analyse de l'évolution 87
8.3.1 Aperçu des différentes étapes 88
8.3.2 Étape 1 : mise à jour du modèle causal 89
8.3.3 Étape 2 : analyse causale 92
8.3.4 Étape 3 : re-vérification 93
8.3.5 Discussion 96
8.4 Résultat de l'analyse et prise de décision 96
8.4.1 Enjeux 97
8.4.2 Vérification des contrats 97
8.4.3 Détermination des causes racines 98
8.4.4 Déploiement et retour en arrière (roll-back) 99
8.4.5 Discussion 99
8.5 Conclusion du chapitre 101

Dans ce chapitre, nous présentons notre approche pour effectuer et analyser une évolution. Cette approche se centre sur l'architecte de l'évolution, pour lequel il est nécessaire de savoir si l'évolution qu'il/elle a décrit ne cause pas le viol d'un contrat de qualité de service à l'échelle du système lorsqu'elle est appliquée. Ce chapitre présente la contribution finale de ce document, où nous expliquons comment le modèle causal déduit dans le chapitre précédent est utilisé pour effectuer une analyse des conséquences de l'évolution sur la qualité de service du système. L'objectif de l'analyse est d'identifier le sous-ensemble du système affecté, directement ou indirectement, par l'évolution, afin de déterminer son effet sur la qualité de service. Le résultat de l'analyse permet de renseigner l'architecte de l'évolution sur une potentielle amélioration ou dégradation de la qualité de service. Au final, l'analyse de l'évolution détermine si l'application de l'évolution sur le système maintient les différentes propriétés de qualité de service considérées.

Ce chapitre est constitué comme suit : dans un premier temps, nous présentons la spécification des différentes étapes constituant une évolution, de la définition des nouveaux besoins, à l'expression des modifications à appliquer au système. Puis, nous détaillons chacune des étapes de l'analyse en expliquant leurs spécificités et les choix conceptuels que

nous avons pris. Enfin, la dernière section détaille comment le résultat de l'analyse de l'évolution, à savoir l'ensemble des éléments affectés du système, est utilisé pour re-vérifier le système dans le but de quantifier l'effet de l'évolution et de dire si la qualité de service est maintenue.

8.1 Présentation générale du processus d'évolution

8.1.1 Enjeux

L'objectif de ce chapitre est de présenter une méthode permettant de garantir le maintien de la qualité de service au cours de l'évolution. Pour cela, nos objectifs sont les suivants :

- **Maintien de la qualité de service** : nous cherchons par le biais de notre analyse à savoir si une évolution a un effet sur la qualité de service. Pour cela, nous introduisons le concept de *caractérisation d'une évolution* :

  Définition 8.1.1 (*Caractérisation d'une évolution*)

  > *Une évolution peut être caractérisée comme étant neutre, bénéfique ou néfaste, si elle n'a respectivement aucun effet, un effet d'amélioration ou un effet de dégradation sur la valeur de la propriété de QoS considérée. On considère également qu'une évolution sera jugée satisfaisante en terme d'une propriété de QoS si l'évolution ne cause pas de violation de contrat.*

  Concrètement, la caractérisation d'une évolution s'effectue en comparant la valeur de qualité de service pour le système *avant* l'évolution à la valeur *après* l'évolution. Si la valeur s'améliore (voir chapitre 7), alors l'évolution sera bénéfique.

- **Diagnostic en cas de non-maintien de la qualité de service :** lorsqu'une évolution dégrade une valeur de QoS et qu'elle cause la violation d'un contrat de QoS, l'architecte de l'évolution doit être en mesure de modifier la description de l'évolution pour corriger l'erreur. Dans ce cas, il est nécessaire de savoir quel est l'élément du système responsable de la violation.

Pour répondre à ces défis, nous proposons un processus d'évolution s'assurant du maintien de la qualité de service et de l'identification de la cause d'une violation de contrat. Afin de montrer le fonctionnement du processus, nous utilisons PicWeb comme illustration.

8.1.2 Spécification du processus d'évolution

Le processus d'évolution peut être vu comme une transformation du système d'un état S à un état S'. À titre de guide, la Figure 8.1 en représente les différentes étapes : l'architecte de l'évolution décrit en (1) une évolution à appliquer sur le système. En s'appuyant sur cette description, ainsi que sur les descriptions de propriétés à contrôler, l'évolution est alors analysée en (2), afin de déterminer si elle est *satisfaisante* du point de vue de la QoS. Le cas échéant, l'évolution est en (3) appliquée de manière effective, ou bien l'architecte de l'évolution observe les causes déterminées par l'analyse pour pouvoir corriger son évolution et en proposer une nouvelle.

Nous détaillons chacune des étapes dans la suite de cette section.

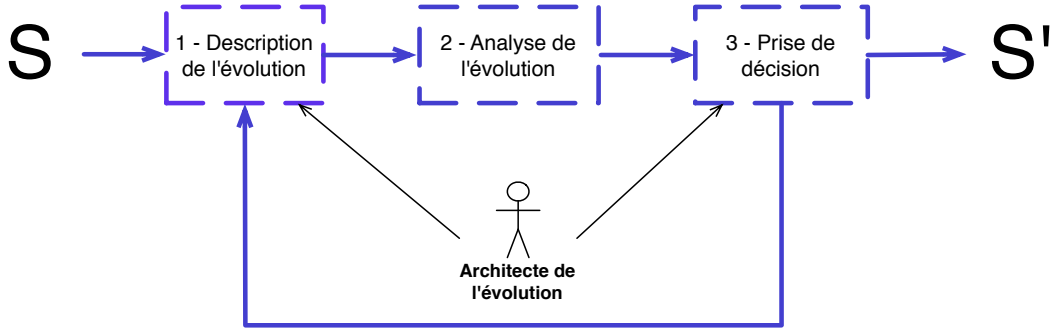


FIGURE 8.1 – Processus d'évolution.

8.1.2.1 Étape 1 : description de l'évolution

WriteEvolution : $\emptyset \longrightarrow \textit{Évolution}$

La première étape consiste à décrire les modifications que l'on veut apporter au processus métier. L'architecte du système décrit dans un premier temps la séquence de modifications à opérer sur le système, telles que les ajouts, suppressions ou mises à jour des différents éléments du système. L'objectif de cette étape est de construire une modification de la structure et du comportement du système, dans le but de satisfaire les changements des besoins de l'utilisateur. Nous montrons dans la section 8.2 comment cette description peut être établie avec notre langage pour l'évolution.

8.1.2.2 Étape 2 : analyse de l'évolution

Analyse : $\textit{Évolution} \longrightarrow \textit{Set<PropertyValue>}, \textit{Set<Contract>}$

La deuxième étape consiste à déterminer les effets de l'évolution sur la qualité de service du système. Pour cela, l'analyse repose sur l'établissement du modèle causal futur, *i.e.*, tel qu'il serait si l'évolution était établie. Partant de ce modèle, l'analyse identifie le sous-ensemble des valeurs de propriétés affectées par l'évolution. Enfin, l'analyse se termine par une re-vérification de ces valeurs, afin de pouvoir déterminer si les contrats de QoS sont toujours respectés, ou si au contraire ils ont été violés.

8.1.2.3 Étape 3 : prise de décision

Decision : $\textit{Set<PropertyValue>}, \textit{Set<Contract>} \longrightarrow \textit{Boolean}$

La dernière étape de l'analyse a pour rôle d'apporter les informations nécessaires à l'architecte de l'évolution afin de savoir si l'évolution proposée respecte bien les contrats de qualité de service définis. Au cours de cette étape, l'architecte choisit d'appliquer l'évolution de manière effective sur le système en production. En cas de problème sur un contrat, il s'agira de rentrer dans une phase de diagnostic afin de comprendre pourquoi le contrat a été violé. Les modifications faites sur le modèle causal sont alors annulées avant de revenir au début du processus, afin de définir une nouvelle évolution.

8.1.3 Hypothèses

Afin de pouvoir garantir la cohésion du système, nous devons poser un certain nombre d'hypothèses :

– Nous considérons ici que pour garantir la cohérence du système, deux évolutions ne peuvent être appliquées en parallèle. Notre processus d'évolution devra être conçu

TABLE 8.1 – Liste des opérations d'évolution.

Opération	Description
addVariable(var)	var est ajoutée au modèle
delVariable(var)	var est retirée du modèle
addActivity(id, nom)	act est ajoutée au modèle
delActivity(id)	act est retirée du modèle
addInParameter(act,var,pos)	var est dans les paramètres de act, en position pos
delInParameter(act,var)	var est retirée des paramètres de act
addOutParameter(act,var,pos)	var est dans le résultat de act, en position pos
delOutParameter(act,var)	var est retirée du résultat de act
addRelation(idFrom, itTo)	une relation d'ordre est créée entre les activités idFrom et idTo
delRelation(idFrom,itTo)	la relation d'ordre entre les activités idFrom et idTo est supprimée

pour effectuer les évolutions de manière séquentielle et atomique, ce dernier point permettant ainsi de revenir à la version précédente en cas de problème.

– Pour pouvoir déterminer si la QoS est maintenue, nous supposons que la description des propriétés de QoS est disponible, et que des contrats sont définis au préalable.

Dans la suite de ce chapitre, nous détaillons chacune des étapes du processus d'évolution. Après avoir introduit notre langage pour décrire une évolution, nous nous focalisons sur les spécificités de chacune des étapes du processus d'analyse. Nous détaillons chacune des étapes, avant de les illustrer à l'aide du cas d'étude. Nous désignons dans le reste du chapitre les parties du système telles que les variables, les services, les activités, ou les paramètres en entrée/sortie d'un service comme étant des *éléments* du système (voir chapitre 6).

8.2 Un langage pour l'évolution des processus métier

L'étape initiale du processus d'évolution consiste à exprimer les changements à effectuer sur le système décrit par l'architecte du système pour satisfaire les nouveaux besoins de l'utilisateur. Pour cela, de nombreux langages de description de l'évolution d'un système existent. Par exemple, PRAXIS est un outil d'analyse de l'évolution de modèles [Blanc 2009]. Plus proche de notre domaine d'application, ADORE est un langage d'évolution de processus métier basé sur les techniques de tissage de modèle [Mosser 2013]. Ces langages expriment une évolution en proposant principalement des modifications structurelles du système. Il s'agira de pouvoir déclarer l'ajout, la suppression ou la mise à jour de concepts du système étudié.

Nous proposons ici de reprendre les concepts communs aux langages d'évolution pour définir notre propre langage d'évolution. Nous avons défini des opérateurs permettant de manipuler des variables, des activités et des relations d'ordre. Notre langage d'évolution permet l'ajout, la suppression et la mise à jour des différents éléments du système. Il est constitué d'un jeu de dix opérations destiné à ajouter, supprimer des éléments propres aux processus métiers. La TABLE 8.1 dresse la liste des différentes opérations disponibles.

Dans le cadre de notre exemple fil rouge, nous rappelons que suite à la première version de PicWeb, les utilisateurs ont émis le souhait de bénéficier d'une source de données supplémentaire, Flickr, pour les photos retournées par le processus métier. L'architecte de l'évolution conçoit alors une évolution répondant à ce besoin. Elle se découpe en trois étapes :

- **Ajout de l'appel à Flickr :** afin de faire évoluer le processus métier, l'architecte de l'évolution propose d'ajouter en parallèle de l'appel à Picasa un appel au service de Flickr. Il s'agit donc ici d'ajouter une nouvelle activité au processus métier, ayant en entrée le mot-clé recherché. Afin de pouvoir récupérer des données en sortie, une nouvelle variable, *PicsFlickr*, a besoin d'être créée et ajoutée en temps que valeur de sortie de l'activité Flickr.
- **Ajout de l'appel à Concat :** désormais, le processus métier doit manipuler deux ensembles de données, que nous souhaitons concaténer. De manière similaire à l'ajout de l'activité Flickr, l'architecte ajoute une activité, *concat*, prenant en entrée les deux ensembles, pour produire en sortie un nouvel ensemble concaténant le deuxième ensemble à la suite du premier.
- **Mise à jour de l'ordre d'exécution :** la création de deux nouvelles activités dans notre processus métier implique de rendre de nouveau cohérent l'ordre partiel d'exécution. Pour que Flickr s'exécute en parallèle de Picasa, l'architecte introduit une relation temporelle entre l'activité receive et l'appel à Flickr. Les deux services de récupération de données sont suivis désormais par *concat* qui termine la modification en reprenant la suite du processus métier.

Pour effectuer cette évolution, l'architecte de l'évolution décrit en Figure 8.2 l'ensemble des opérations à effectuer. L'évolution est composée de quatre parties : les lignes 2 à 4 mettent à jour l'activité Picasa en changeant le nom de sa valeur de retour par une nouvelle variable, *PicsPicasa*. Les lignes 6 à 9 ajoutent l'appel à l'activité Flickr. Les lignes 11 à 14 sont responsables de l'appel à l'activité Helper, qui va concaténer *PicsPicasa* et *PicsFlickr* dans *Pics*. Enfin, les lignes 16 à 20 rétablissent l'ordre partiel entre les activités.

Une fois que l'évolution est décrite, l'architecte de l'évolution peut déclencher l'analyse de l'évolution.

8.3 Spécification de l'analyse de l'évolution

L'analyse de l'évolution est une étape cruciale du processus d'évolution. Elle soulève les objectifs suivants :

- **Minimisation de la re-vérification :** nous cherchons à effectuer le moins de vérifications inutiles possibles, particulièrement dans le cas où la re-vérification doit être effectuée à l'exécution, pouvant potentiellement affecter les utilisateurs du système. Il s'agira de cerner au plus près l'ensemble des éléments impactés, en écartant les éléments neutres vis à vis de l'évolution, sans pour autant écarter d'éléments affectés.
- **Qualification de l'évolution :** afin de pouvoir caractériser l'évolution, nous cherchons à savoir ici si l'évolution est bénéfique, neutre ou néfaste du point de vue d'une propriété de qualité de service. À la fin de cette étape, l'architecte de l'évolution a besoin de connaître les valeurs de propriétés pour l'ensemble du système, afin d'être en mesure de comparer les valeurs avant évolution et après évolution. Au delà d'un rapport qualitatif, l'architecte de l'évolution doit être en mesure de pouvoir quantifier son apport, afin de déterminer si elle est acceptable ou non.

```
1  Evolution{
2          addVariable(PicsPicasa)
3          delOutParameter(a2,Pics)
4          addOutParameter(a2, PicsPicasa, 0)
5
6          addVariable(PicsFlickr)
7          addActivity(a22,Flickr)
8          addInParameter(a22,Tag,0)
9          addOutParameter(a22,PicsFlickr,0)
10
11         addActivity(a23, join)
12         addInParameter(a23, PicsPicasa,0)
13         addInParameter(a23, PicsFlickr,1)
14         addOutParameter(a23, Pics,0)
15
16         addRelation(receive, a22)
17         addRelation(a22,a23)
18         addRelation(a23,a3)
19         addRelation(a2,a23)
20         delRelation(a3,a2)
21 }
```

FIGURE 8.2 – Évolution de PICWEB.

Le but de cette section est de présenter les différentes étapes constituant l'analyse de l'évolution, tout en motivant leur intérêt. Notre analyse a pour objectif de montrer à l'architecte de l'évolution les effets développés sur les propriétés de qualité de service du système. Elle prend en entrée un système, une évolution, et des propriétés à considérer, afin de lister en sortie le ou les contrats de qualité de service violés, accompagnés d'une trace d'exécution montrant la violation de contrat. Nous présentons chacune des trois étapes la constituant, et l'illustrons en les appliquant sur PICWEB.

8.3.1 Aperçu des différentes étapes

Nous présentons dans cette partie les différentes étapes de l'analyse d'évolution décrite dans la FIGURE 8.3, en précisant leurs entrées et sorties, et en expliquant l'objectif et l'intérêt de chacune des étapes.

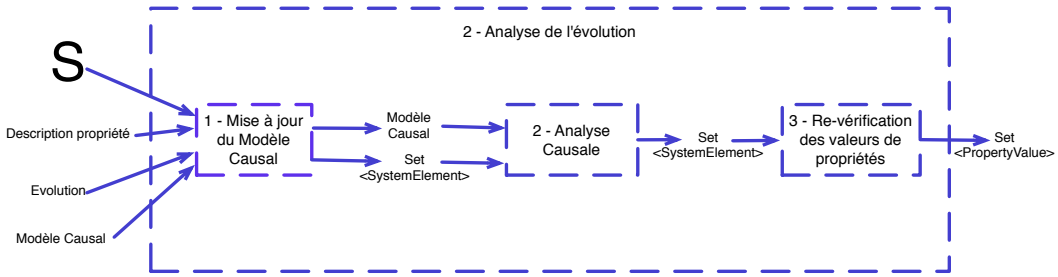


FIGURE 8.3 – Étapes de l'analyse de l'évolution.

– **Étape 1 : mise à jour du modèle causal**
Evolution × *CausalModel* ⟶ *CausalModel*

Une fois que l'évolution est décrite, l'analyse de l'effet de l'évolution sur le système débute. Puisque cette analyse se base sur le modèle causal de l'application, la première

étape de l'analyse consiste à mettre à jour le modèle causal. Il s'agit ici de défaire les relations n'ayant plus de sens, et d'ajouter les nouvelles relations causales correspondant à l'évolution en cours d'application. Similairement, des nœuds du modèle causal sont créés ou détruits, afin de refléter au plus près les causalités entre les éléments du système tel qu'il serait si l'évolution était effectivement appliquée.

- **Étape 2 : analyse Causale**
  $CausalModel \times Set<SystemElement> \longrightarrow Set<SystemElement>$
  Prenant en entrée les éléments ajoutés ou supprimés, l'analyse causale consiste en un parcours du modèle causal, afin d'identifier les éléments du système affectés par l'évolution. En sortie, l'analyse causale produit une liste des tous les éléments du système ayant une relation de causalité établie avec l'évolution, et pour lesquels une nouvelle vérification est nécessaire.
- **Étape 3 : re-vérification des valeurs de propriétés**
  $Set<SystemElement> \longrightarrow Set<PropertyValue>$
  La dernière étape de l'analyse détermine les nouvelles valeurs de propriétés des éléments affectés, afin de connaître l'effet concret de l'évolution sur les propriétés de qualité de service étudiées. Ici, l'analyse déclenche alors les outils de re-détermination des valeurs de propriétés adéquats, et déploie si besoin sur un serveur de test une version du système évolué et enrichi de moniteurs afin de pouvoir collecter les valeurs de propriétés modifiées. Le résultat de cette étape produit un différentiel entre les valeurs du système pré et post-évolution. À partir de ce différentiel, l'analyse vérifie les contrats de QoS afin de détecter une potentielle violation.

Nous présentons dans la suite le comportement de l'analyse de l'évolution, et expliquons en quoi les contraintes énumérées ci-dessus sont respectées.

8.3.2 Étape 1 : mise à jour du modèle causal

L'analyse de l'évolution repose sur un parcours du modèle causal, afin de déterminer quels éléments seront affectés. Pour cela, il est nécessaire d'établir l'hypothétique modèle causal du système, en considérant que l'évolution a déjà eu lieue, afin de pouvoir en analyser les effets. La première étape de l'analyse consiste à mettre à jour le modèle causal pour obtenir les causalités du système post-évolution. Il s'agit d'une étape cruciale, car c'est elle qui garantit l'exactitude du modèle causal, permettant par la suite à l'analyse de l'évolution de produire un résultat précis.

La mise à jour du modèle causal se résume par une mise à jour de ses nœuds et de ses relations causales. Cette étape se réalise en deux temps : d'un point de vue fonctionnel, où Smile traduit les opérations d'évolution en opérations de modification du modèle causal, et d'un point de vue qualité de service, où Smile déduit de la mise à jour fonctionnelle et de la description des propriétés les différentes relations causales de qualité de service La Figure 8.4 est une représentation schématique du processus de mise à jour. Cette étape prend en entrée la description des propriétés de QoS, l'évolution à appliquer, et le modèle causal du système. En sortie, nous trouvons un modèle causal futur, tel qu'il serait si l'évolution était appliquée. En supplément, l'étape de mise à jour retourne également un ensemble d'éléments constituant les éléments explicitement affectés par l'évolution. C'est à partir de cette liste que l'étape d'analyse causale détermine quels autres éléments du système sont indirectement affectés par l'évolution.

Pour effectuer le traitement attendu, la mise à jour du modèle causal s'effectue en trois sous-étapes :

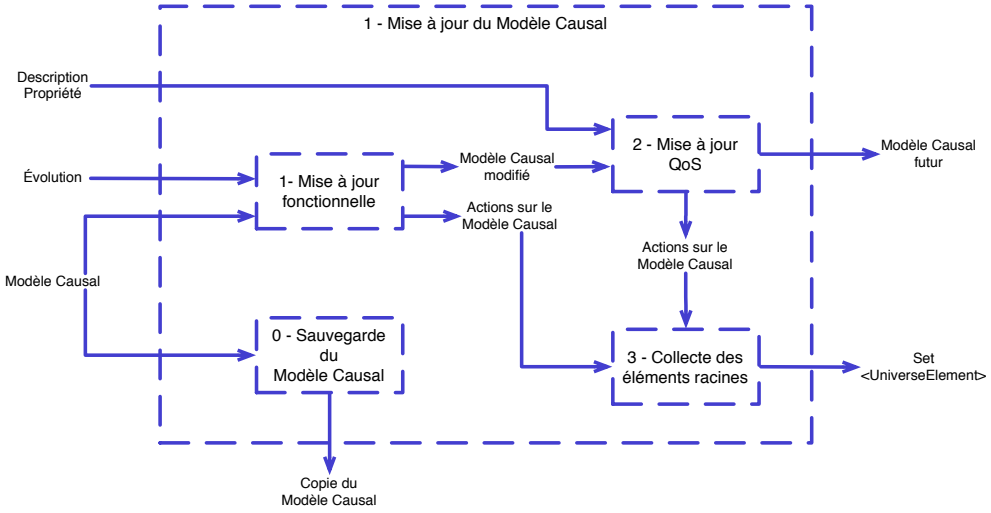


FIGURE 8.4 – Traduction de l'évolution en actions sur le modèle causal.

TABLE 8.2 – Correspondance entre opérations d'évolution et actions du modèle causal.

Opération d'évolution	**Action(s) sur les nœuds**	**Action(s) sur les relations**
addActivity(act)	addNode(act)	–
delActivity(act)	delNode(act)	$\forall var \in input(act)$, delCausalRel(var,act) $\wedge$ $\forall var \in output(act)$, delCausalRel(act,var)
addVariable(var)	addNode(var)	–
delVariable(var)	delNode(var)	–
addInPar(act,var,pos)	–	addCausalRel(var,act)
delInPar(act,var)	–	delCausalRel(act,var)
addOutPar(act,var,pos)	–	addCausalRel(act,var)
delOutPar(act,var)	–	delCausalRel(var,act)
addRelation(act,act2)	–	–
delRelation(act,act2)	–	–

1 - Mise à jour fonctionnelle : la mise à jour fonctionnelle du modèle causal consiste à ajouter/supprimer les nœuds modifiés par l'évolution, et à ajouter/supprimer les relations causales sous-jacentes. Cette étape est nécessaire car elle permet d'avoir un modèle causal reflétant le système une fois que l'évolution a été appliquée.

La mise à jour des nœuds du modèle causal consiste à identifier au sein de la description de l'évolution les éléments du système ajoutés ou supprimés. Pour cela, SMILE traduit les opérations de l'évolution modifiant la structure du système en actions à effectuer sur le modèle causal. Nous avons défini une correspondance entre les différentes opérations du langage d'évolution, et les actions à effectuer sur le modèle causal, consignée dans la TABLE 8.2. Cette correspondance est valable pour n'importe quelle évolution. À titre d'illustration, nous établissons la correspondance entre les opérations d'évolution utilisées dans l'évolution de PICWEB et les actions effectuées sur son modèle causal dans la TABLE 8.3 (écrites en bleu). Ici, nous ajoutons simplement deux nœuds pour les variables *PicsPicasa* et *PicsFlickr* et deux nœuds pour les activités *Flickr* et *Join*.

Evolution{	CausalEvolution{
addVariable(PicsPicasa)	addNode(PicsPicasa)
delOutParameter(a2,Pics)	delCausalRel(a2,Pics)
addOutParameter(a2, PicsPicasa)	addCausalRel(a2,PicsPicasa)
addVariable(PicsFlickr)	addNode(PicsFlickr)
addActivity(a22,Flickr)	addNode(Flickr)
addInParameter(a22,Tag)	addCausalRel(tag,a22)
addOutParameter(a22,PicsFlickr)	addCausalRel(a22,PicsFlickr)
addActivity(a23, join)	addNode(join)
addInParameter(a23, PicsPicasa)	addCausalRel(PicsPicasa,a23)
addInParameter(a23, PicsFlickr)	addCausalRel(PicsFlickr,a23)
addOutParameter(a23, Pics)	addCausalRel(a23,Pics)
addRelation(receive, a22)	—
addRelation(a22,a23)	—
addRelation(a23,a3)	—
addRelation(a2,a23)	—
delRelation(a3,a2)	—
}	}

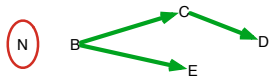
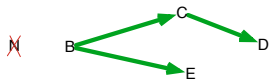
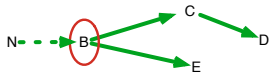
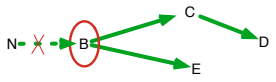
TABLE 8.3 – Génération de la mise à jour du modèle causal de PICWEB.

De manière similaire à la construction initiale du modèle causal (voir chapitre 6), la mise à jour des relations causales fonctionnelles s'effectue en appliquant les règles causales fonctionnelles sur les nouveaux éléments de l'évolution. À titre d'exemple, nous considérons les règles causales introduites dans le chapitre 7. L'application de ces règles sur les éléments manipulés au cours de l'évolution permet de reconstruire les relations causales manquantes. Pour cela, nous avons établi la correspondance entre les opérations d'évolution et les actions sur le modèle causal du point de vue de la construction des relations causales fonctionnelles. Cette correspondance est décrite dans la TABLE 8.2 : elle permet de générer pour l'évolution de PICWEB la mise à jour décrite dans la partie droite de la TABLE 8.3 (écrite en rouge).

2 - Mise à jour de la QoS : de manière similaire, la mise à jour du modèle causal d'un point de vue qualité de service consiste à introduire les nouvelles valeurs de propriétés et les nouvelles relations causales de QoS. Compte tenu de sa similitude avec l'étape de mise à jour fonctionnelle, nous décrivons brièvement cette étape : ici, il s'agit dans un premier temps d'assurer la cohérence du modèle, en ajoutant ou en supprimant les *PropertyValues* correspondant aux modifications effectuées du point de vue fonctionnel. Par exemple, si une nouvelle activité est ajoutée, il s'agit d'ajouter également dans le modèle causal l'ensemble de ses *PropertyValues*. Une fois la cohérence retrouvée d'un point de vue structurel, il est nécessaire de construire les nouvelles relations causales propres aux propriétés de QoS étudiées. Cette tâche s'effectue simplement en appliquant de nouveaux les règles causales de QoS déduites, comme présenté dans le chapitre 7.

3 - Collecte des éléments racines : afin de pouvoir établir les conséquences liées à l'évolution, nous avons besoin de déterminer les causes racines [Rooney 2004], qui seront dans ce contexte les éléments fonctionnels directement manipulés par l'évolution. L'étape de collecte des éléments racines analyse chacune des actions effectuées sur le modèle causal. Pour chaque action, la collecte consiste à capturer le ou les éléments du système, facteurs d'influence. Pour cela, nous considérons chacun des quatre types d'opération de mise à jour du modèle causal pour déterminer les éléments-causes racines de l'effet de l'évolution. La recherche des éléments racine est réalisée à l'aide de l'opérateur *collecte(CausalOperation)*. Nous décrivons sa spécification pour chacune des opérations de mise à jour du modèle causal

TABLE 8.4 – Description de l'opération *collecte*.

Modèle Causal	Description de l'opération de collecte
	**collecte(addNode(N)) :** l'ajout d'un nœud dans le modèle causal, considéré séparément, n'a d'influence que sur lui-même. En effet, à ce stade, le nœud n'est relié causalement à aucun autre nœud ; le seul élément racine ici est donc N. $collecte(addNode(N)) = N$.
	**collecte(delNode(N)) :** pour la suppression d'un nœud, nous considérons comme pré-condition que l'ensemble des relations causales dont N est la source ou la cible a déjà été supprimé avant d'effectuer la suppression du nœud. Dans cette situation, N n'est lié causalement à aucun autre élément, et ne peut donc pas être une cause racine. Le résultat de *collecte(delNode(N))* est donc égal à $\emptyset$.
	**collecte(addCausalRel(N, B)) :** l'ajout d'une relation causale signifie que N a une influence sur B. Toutefois, il existe peut-être d'autres influences provenant de N qui, elles, n'ont pas changé. il s'agit donc de considérer B comme une cause racine. $collecte(addCausalRel(N, B)) = B$.
	**collecte(delCausalRel(N, B)) :** la suppression d'une relation entraîne potentiellement la détermination de nouvelles causes racines. En effet, si une relation causale entre N et B est supprimée, cela veut dire que N n'opère plus d'influence sur B. De ce fait, l'état de B libéré de cette influence changera comparé à son état dans la version pré-évolution. Il est donc nécessaire de considérer B comme une cause racine. $collecte(delCausalRel(N, B)) = B$.

dans la TABLE 8.4.

Dans le cas de PICWEB, les différents opérations de mise à jour du modèle causal consignées dans les mises à jour du modèle causal déduites sont analysées pour collecter les causes racines. Cela produit l'ensemble de nœuds *<pics, a2, Tag, a22, PicsPicasa, PicsFlickr,a23,Flickr,Join>*.

Nous venons de voir comment, à partir de la description d'une évolution, il était possible de mettre à jour le modèle causal. Dans l'étape suivante, il s'agit d'utiliser le modèle causal mis à jour pour déterminer quelles valeurs de propriétés de QoS sont affectées par l'évolution.

8.3.3 Étape 2 : analyse causale

L'objectif de cette étape est d'identifier les valeurs de propriétés qu'il est nécessaire de re-vérifier. En utilisant les éléments racines identifiés dans l'étape précédent, l'analyse causale consiste à visiter le modèle causal pour établir l'influence de l'évolution sur le reste du système et sur sa qualité de service. Le modèle causal est parcouru afin de capturer les différents nœuds qui seraient directement ou indirectement affectés.

Principe

Pour analyser l'effet d'une évolution sur le système, l'analyse causale consiste à effectuer un parcours de graphe, où chaque nœud rencontré est collecté. En effet, nous partons du principe que les éléments racines sont des sources de causalité. Il s'agit donc de considérer les éléments liés aux causes racines par une relation causale comme étant influencés. De plus, selon notre définition de la causalité, l'état de ces éléments est potentiellement affecté : de la même manière que pour les éléments racines, ce changement d'état a une influence sur d'autres éléments du système. C'est ce que Judea Pearl appelle *l'inférence de causalité* [Pearl 2000]. Pour prendre en compte cette influence, l'analyse causale consiste en

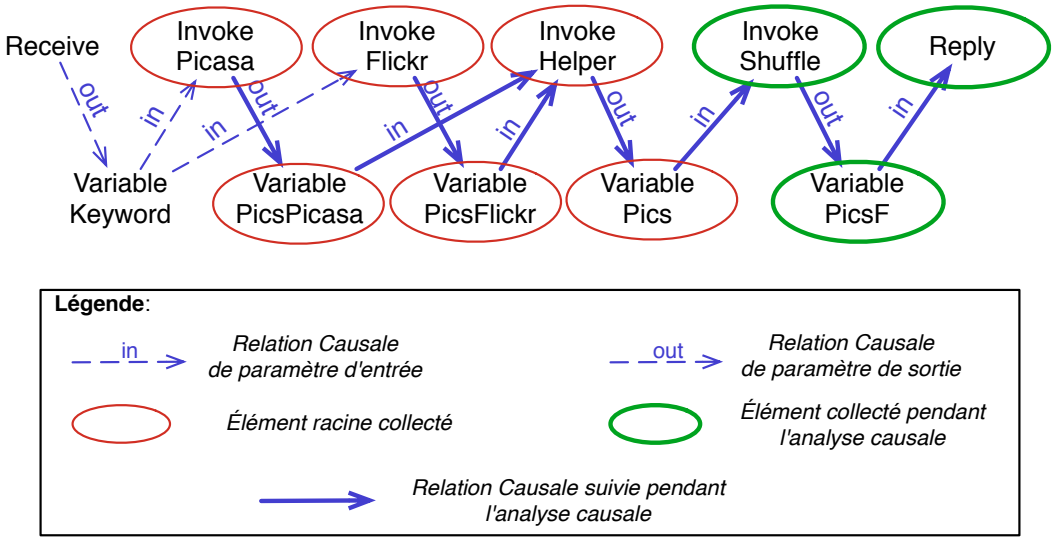


FIGURE 8.5 – Application à PICWEB de l'analyse causale.

une clôture transitive des relations causales. En suivant les relations causales partant des éléments-racines, l'analyse causale détermine l'ensemble des éléments affectés par l'évolution.

Application à PICWEB

Nous illustrons l'application de l'analyse causale à PICWEB en l'observant du point de vue des relations causales fonctionnelles, puis des relations de qualité de service. La FIGURE 8.5 représente le parcours effectué par l'analyse causale sur la partie fonctionnelle du modèle causal ; l'analyse démarre des éléments cerclés en rouge (causes racines), pour suivre les relations causales représentées en gras, pour identifier une série d'éléments causalement affectés, cerclés en vert. Ici, l'analyse causale détermine donc que l'évolution a impacté les éléments *Shuffle, PicsF et Reply*.

Afin d'illustrer le même principe sur les éléments propres à la qualité de service, nous raisonnons sur une sous-partie du modèle causal, décrite dans la FIGURE 8.6. Dans cette figure, nous étudions la partie terminale du processus métier, où l'ensemble des éléments fonctionnels présents a été déterminé comme affecté par l'évolution. Ici, l'analyse causale continue son parcours en suivant les relations d'environnement vers les valeurs de propriété de temps de transmission, puis vers les temps de réponse des activités simples par dérivation, et enfin vers la valeur de propriété de temps de réponse de la séquence entière. Nous pouvons souligner que les valeurs de propriétés de temps de transmission n'ont pas été sélectionnées, grâce à la spécialisation des relations causales de QoS. Dans le cas où nous avions limité notre modèle causal aux seuls éléments fonctionnels, il aurait fallu re-contrôler les valeurs de toutes les propriétés. Nous économisons ainsi la re-vérification de certaines valeurs de propriétés, comme le temps de transmission.

8.3.4 Étape 3 : re-vérification

L'analyse causale effectuée lors de l'étape précédente a identifié un ensemble d'éléments causalement reliés aux changements opérés par l'évolution. Maintenant que nous savons quels éléments ont été affectés, il est nécessaire de savoir à quelle hauteur le changement

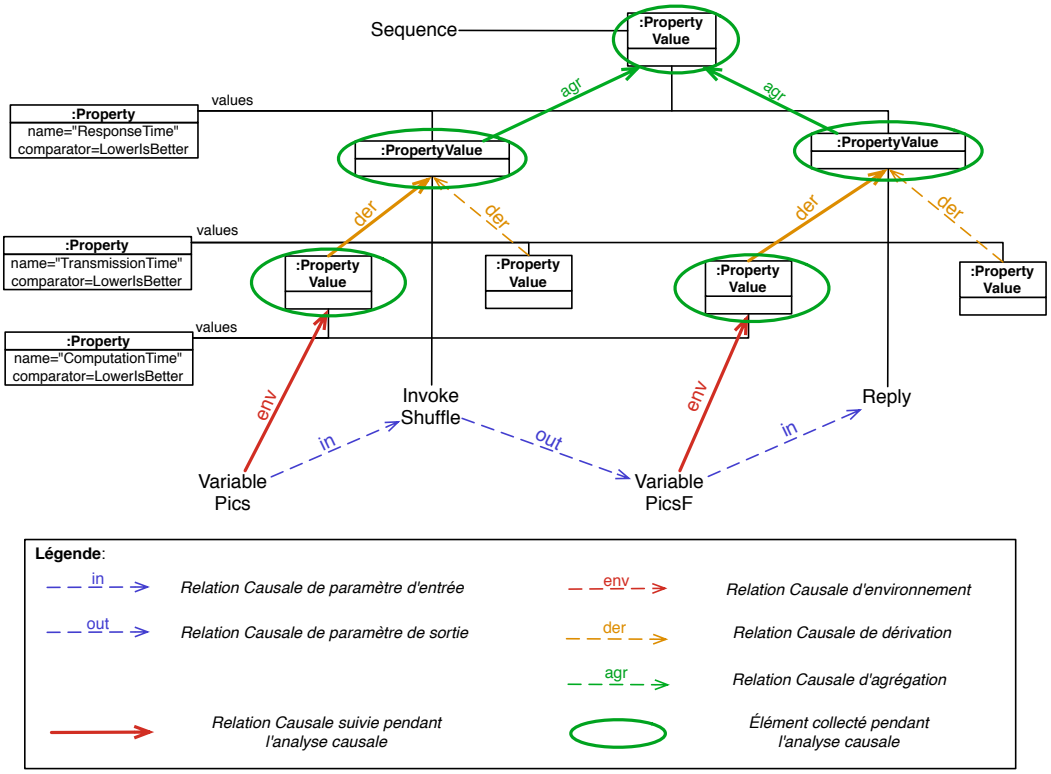


FIGURE 8.6 – Application à PICWEB de l'analyse causale.

effectué sur les éléments améliore ou détériore la qualité de service. L'étape de re-vérification a pour rôle de re-déterminer uniquement les valeurs de propriétés affectées, en générant une version outillée du système post-évolution, afin de pouvoir effectuer les analyses, contrôles et calculs nécessaires à l'architecte de l'évolution pour l'obtention de nouvelles valeurs de propriétés. Ces valeurs aideront par la suite à savoir si le système, ayant évolué, continue de respecter ses contrats de qualité de service.

8.3.4.1 Analyses et contrôles ciblés

L'objectif de cette partie est de re-vérifier de manière minimale les valeurs de propriété indiquées lors de l'analyse causale, afin de dresser un diagnostic qui servira de base à l'architecte de l'évolution pour prendre une décision quant à l'applicabilité de l'évolution. Nous expliquons dans un premier temps comment les valeurs de propriétés sont re-vérifiées, avant de montrer comment de ces valeurs il est possible de poser un premier diagnostic. Afin de déterminer ces valeurs, le processus de détermination présenté dans le chapitre 7 est ré-utilisé. Nous nous intéressons dans un premier temps à des valeurs de propriétés **élémentaires**, *i.e.*, des valeurs de propriétés qui ne peuvent être composées ou déduites d'un calcul. Ce sont des valeurs déterminées par analyse ou par contrôle de l'application à l'exécution. La mesure en environnement de test produit un ensemble de données brutes, caractérisant les parties du système suspectées d'avoir changé pour une propriété donnée. De ces données, le challenge est de déterminer leur impact à l'échelle du système.

La FIGURE 8.7 décrit le procédé permettant d'obtenir les instance de PropertyValue. Partant de la liste des éléments affectés, SMILE outille le système à l'aide des analyses et des contrôleurs décrits dans la description de la propriété considérée, pour obtenir de nouvelles

valeurs. Cette phase est nécessaire pour connaître les nouvelles valeurs de propriétés.

Une fois l'outillage effectué, les analyses et les mesures sont effectuées. Si l'outillage est automatisé, la phase de mesures reste manuelle, afin de laisser la possibilité à l'architecte de l'évolution d'exécuter les cas de tests qu'il désire. Le processus de re-vérification consiste à exécuter plusieurs fois une activité afin de prendre plusieurs échantillons de la même mesure. Chacune des exécutions du système produit des mesures, caractérisant la valeur de propriété sur le système évolué. Nous appelons échantillon l'ensemble des mesures effectuées.

Cette phase, partiellement outillée, joue un rôle important dans l'analyse d'une évolution. Elle quantifie de quelle manière l'évolution a affecté les éléments pointés par l'analyse causale.

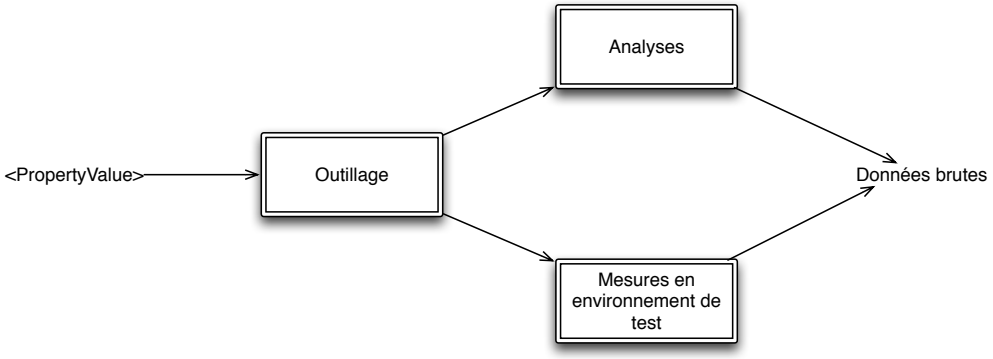


FIGURE 8.7 – Obtention des données brutes.

8.3.4.2 Déduction des métriques

Les mesures produites lors de la phase de test constituent un ensemble de données. Nous cherchons à caractériser l'évolution, à savoir déterminer si après une évolution, les PropertyValues sont améliorées ou dégradées. Pour cela, notre approche consiste à comparer cet ensemble avec les données pré-évolution, afin de pouvoir caractériser, pour une activité donnée, si l'évolution a amélioré, dégradé ou n'a pas changé la valeur de propriété d'une activité donnée. Pour effectuer cette comparaison, nous préconisons à l'architecte de l'évolution de constituer un ensemble de données post-évolution conséquent, afin d'avoir un échantillon représentatif.

Nous cherchons ici non pas à comparer deux valeurs discrètes (pré-et post évolution), mais à comparer deux ensembles de données. Pour cela, nous proposons de caractériser chaque ensemble à l'aide de trois valeurs : le meilleur cas, le pire cas, et la moyenne. Ces trois valeurs correspondent respectivement au maximum, au minimum et à la moyenne de l'ensemble, dans le cas où le maximum représente le meilleur cas. Une fois que ces métriques sont calculées, nous pouvons les comparer avec les mêmes métriques établies pré-évolution en utilisant l'opérateur de comparaison (**comparisonType**), disponible dans la description de la propriété. Si pour une métrique, la valeur a changé, il est nécessaire de reporter ce changement en continuant de parcourir la chaîne de causalité vers les valeurs de propriété de dérivation et d'aggrégation.

8.3.4.3 Dérivations et agrégations ciblées

Si pour une métrique donnée, le différentiel pré-post évolution montre un changement, il est nécessaire de le propager au niveau des valeurs de propriétés dérivées, le cas échéant.

Nous nous plaçons dans le cas où la propriété étudiée rentre dans la détermination d'une propriété dérivée, comme c'est le cas pour le temps d'exécution qui rentre dans la détermination du temps de réponse. Pour une activité donnée, si la métrique maximum du temps de calcul a changé, il est nécessaire de re-calculer la métrique maximum du temps de réponse pour la même activité. Il s'agit donc ici de limiter le re-calcul des propriétés dérivées uniquement pour celles dont les sous-propriétés ont changé. À titre d'illustration, le temps maximal de calcul de l'activité *Format* de PicWeb a changé. Il est donc nécessaire de re-calculer le temps maximal de réponse de la même activité.

De manière similaire à la dérivation, l'aggrégation de valeurs de propriétés est influencée par les valeurs des activités prise en compte dans leurs détermination. Par exemple, pour une activité structuré S, contenant les activités A, B et C, il est nécessaire de ré-aggréger la valeur de propriété de S si l'une des valeurs de propriétés de A, B ou C a changé. Pour cela, en suivant les relations causales depuis les activités contenues, nous provoquons le re-calcul de la PropertyValue de S.

8.3.5 Discussion

Nous venons de présenter les différentes étapes constituant l'analyse de l'évolution. Ces étapes permettent de répondre aux enjeux à relever :

- **Minimisation de la re-vérification :** notre méthode consiste à identifier le sous-ensemble du système affecté par l'évolution, et à regrouper l'ensemble des analyses et contrôles des éléments désignés par l'analyse causale. En ré-exécutant uniquement ces vérifications, nous minimisons les déploiements. Notre approche réduit donc le nombre de vérifications à effectuer, comparé à la re-vérification de l'ensemble du système, grâce à l'identification des éléments impactés par l'analyse causale.
- **Qualification de l'évolution :** La politique de différentiel entre le système pré et post-évolution permet d'arrêter les calculs au plus tôt, dès qu'une causalité est détectée comme étant neutre. De plus, les métriques de maximum, minimum et de moyennes établies avant et après l'évolution permettent de quantifier la différence entre les deux versions. Enfin, la chaîne de causalité établie par l'analyse causale renseigne l'architecte de l'évolution en lui donnant un premier élément de diagnostic, dans le cas d'une violation de contrat.

8.4 Résultat de l'analyse et prise de décision

Une fois que les nouvelles mesures provenant des contrôleurs ont été récupérées de l'exécution, l'ensemble des informations nécessaires est disponible pour faire un différentiel complet entre l'état de la qualité de service avant et après l'évolution. Il est ainsi maintenant possible de savoir si une évolution a été bénéfique ou néfaste pour une propriété donnée. En cas de dégradation, la détermination ciblée permet de savoir concrètement quelles sont les valeurs qui ont changé, permettant de restreindre le champ d'investigation. Mieux encore, la notion de causalité dans le système permet de poser un premier diagnostic en remontant la chaîne de conséquences. Nous montrons dans cette section comment les contrats de qualité de service peuvent être vérifiés pour voir si aucun d'entre eux n'a été violé. Puis, nous expliquons comment le modèle causal peut aider à déterminer, en cas de dégradation, l'origine de la dégradation en suivant les relations causales dans le sens inverse. Enfin, nous présentons l'ultime étape de l'analyse, laissant à l'architecte le choix de déployer la version évoluée du système, ou de revenir à la version précédente.

Dans cette phase, l'analyse cherche avant tout à caractériser si l'évolution est bénéfique, neutre, ou néfaste aux propriétés de qualité de service (voir chapitre 7), et de déterminer si des contrats ont été violés. L'architecte de l'évolution n'aura qu'à prendre connaissance du diagnostic de l'analyse pour savoir si oui ou non l'évolution peut être appliquée de manière effective.

8.4.1 Enjeux

L'ultime étape de l'analyse d'évolution fait intervenir l'architecte de l'évolution, afin qu'il prenne la décision finale de savoir si l'évolution doit effectivement être appliquée, ou bien si un retour en arrière doit être effectué. C'est une étape lourde de conséquences où il faut être sûr qu'aucun contrat de qualité de service ne sera violé. Cela entraîne les enjeux suivants :

- **Disponibilité des informations essentielles :** afin de pouvoir prendre la décision d'appliquer ou non l'évolution, l'architecte de l'évolution a besoin de l'ensemble des informations collectées tout au long de l'analyse. Nous devons déterminer quelles informations sont nécessaires afin de l'accompagner dans la phase de diagnostic.
- **Retour en arrière :** Au delà de l'annulation de l'évolution sur le modèle de données et sur le modèle causal, nous devons également nous assurer que la suite du processus, à savoir la conception d'une nouvelle évolution, sera accompagnée du diagnostic de l'itération précédente. Cet élément semble essentiel pour éviter de réaliser les mêmes erreurs lors de la conception de la nouvelle évolution.

8.4.2 Vérification des contrats

Afin de garantir le maintien de la qualité de service au fil des évolutions, l'architecte de l'évolution doit s'assurer que les contrats de QoS sont toujours respectés. La détermination des trois métriques à l'échelle du système (effectuée lors de l'étape précédente) permet d'avoir un aperçu de l'état du système pour une propriété donnée. Partant de ces valeurs, il est maintenant possible de se positionner selon les contrats pré-établis. Nous pouvons classer le résultat de la comparaison en quatre catégories :

- **Meilleur :** la moyenne, le maximum et le minimum définis post-évolution sont chacun meilleur que les mêmes métriques pré-évolution.
- **Pire :** la moyenne, le maximum et le minimum définis post-évolution sont chacun moins bons que les mêmes métriques pré-évolution.
- **Neutre :** les valeurs des métriques pré et post-évolution sont identiques.
- **Différents :** cas particuliers pour lesquels une règle générale n'est pas applicable pour chacune des métriques.

Pour chacune des quatre catégories, le comportement de l'architecte vis-à-vis de l'évolution sera différent : dans le cas où l'évolution est établie comme étant meilleure, ou neutre, dans l'hypothèse où le contrat de qualité de service était respecté auparavant, nous pouvons dire que le maintien de la qualité de service est garanti pour cette évolution. En effet, si la qualité de service pour une propriété donnée n'a pas changé ou s'est améliorée, le contrat continue d'être respecté. Dans la situation où la valeur de propriété pour le système est pire, il est nécessaire de comparer pour savoir si les conditions du contrat continuent d'être respectées ou non. Finalement, dans la situation où la valeur de propriété est différente, il est nécessaire de vérifier au cas par cas si les contrats sont toujours respectés, et d'isoler les cas où une violation est détectée.

Application à PicWeb

Dans le cas de PicWeb, le contrat établi pour la propriété du temps de réponse ne caractérise que le pire temps. La Figure 8.8 est une partie de notre modèle représentant ce contrat. Ici, le processus métier ne doit pas voir son temps d'exécution dépasser les cinq secondes. Ici, en comparant la nouvelle valeur de propriété de PicWeb avec la valeur de contrat, nous pouvons nous rendre compte que le contrat a été violé.

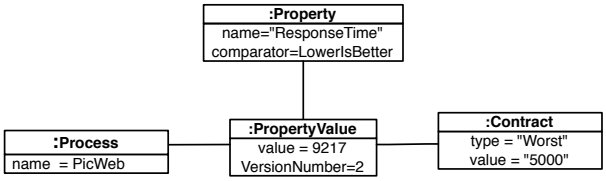


Figure 8.8 – Contrat de QoS qualifiant le temps de réponse de PicWeb.

8.4.3 Détermination des causes racines

De prime abord, si la cause d'une dégradation est bien évidemment l'évolution, il est plus complexe de savoir exactement pourquoi un contrat a été violé. En effet, l'évolution est constituée de plusieurs opérations. Restreindre la cause de la dégradation à une seule opération permettrait de faciliter le diagnostic. Notre approche permet toutefois à l'architecte de l'évolution d'obtenir plus d'informations : la violation d'un contrat est détectée sur une valeur de propriété, qui a été déclarée par l'analyse causale comme étant modifiée : cette *PropertyValue* a été détectée au cours du parcours du modèle causal comme étant potentiellement affectée. Il existe donc une chaîne de causalité reliant un des nœuds de l'évolution à la *PropertyValue*. La chaîne de causalité explique comment d'un nœud de l'évolution, différentes causalités ont été établies pour mener à la violation d'un contrat par la *PropertyValue*. Cette information est une aide pour l'architecte de l'évolution, dans son investigation visant à comprendre pourquoi un contrat a été violé. Par la chaîne causale, nous aidons l'architecte à poser un premier diagnostic.

Nous proposons dans Smile une méthode permettant de réduire par comparaison de valeurs les causes possibles. Pour cela, notre approche consiste à démarrer de la *PropertyValue* correspondant au contrat violé, et de remonter successivement les relations causales ayant servi à construire l'ensemble des éléments affectés. Pour chaque *PropertyValue* rencontrée, deux cas sont à envisager :

- **la *PropertyValue* qualifie un élément du système existant dans la version précédente :** dans ce cas là, il est possible de comparer la valeur actuelle avec la valeur déterminée avant que l'évolution soit appliquée. Si la valeur s'est dégradée après l'évolution, il s'agit de la considérer comme une cause possible. Dans le cas contraire, nous pouvons l'exclure de l'ensemble des causes racines, ainsi que toutes les *propertyValues* l'ayant influencée.
- **la *PropertyValue* qualifie un élément du système créé par l'évolution :** dans cette situation, il n'est pas possible de se comparer avec une valeur précédente. Smile considère donc cette valeur comme une cause potentielle.

Application à PicWeb

Afin de déterminer la cause de la violation du contrat de PicWeb, appliquons notre approche au modèle causal établi. La Figure 8.9 représente le parcours inverse de notre

modèle causal. Nous pouvons ici distinguer quatre cas :

- **Cas 1** : lorsque l'analyse de causes racines traite la *PropertyValue* liée à la séquence, la comparaison avec la version précédente montre que la QoS s'est détériorée. En conséquence, l'analyse se poursuit en suivant en sens inverse les relations causales d'agrégation.
- **Cas 2** : ici, la *PropertyValue* liée à l'activité structurante *flow* a été créée par l'évolution. En conséquence, aucune comparaison n'est possible. L'analyse identifie cet élément comme étant une cause potentielle, et poursuit son analyse en suivant la relation causale d'agrégation vers le temps de réponse de l'activité *Flickr*.
- **Cas 3** : lorsque l'analyse traite la *PropertyValue* de Shuffle, la comparaison détecte une dégradation du temps de calcul. De plus, cet élément est la dernière *PropertyValue* avant de raisonner du point de vue du fonctionnement du système. Celle-ci est identifiée comme étant une cause racine.
- **Cas 4** : au niveau de l'activité *reply*, la comparaison des *PropertyValues* pour le temps de réponse montre qu'il n'y a pas de dégradation une fois l'évolution appliquée. En conséquence, l'élément n'est pas identifié comme une cause racine de la violation du contrat.

Au final, le modèle causal de PicWeb est constitué de vingt et une *PropertyValues* qui, dans une hypothèse pessimiste, auraient toutes dues être re-vérifiées. La méthode que nous avons employée re-vérifie quinze *PropertyValues*, et désigne neuf potentielles causes racines réparties sur trois branches.

8.4.4 Déploiement et retour en arrière (roll-back)

Dans le cas précis où aucun contrat n'a été violé et que l'architecte de l'évolution est satisfait, l'évolution est intégrée à part entière dans le système. Celui-ci est déployé dans les différents serveurs de production. Toutefois, si l'évolution ne convient pas à l'architecte, il est nécessaire de déclencher un retour en arrière. Le modèle du système, le modèle de données des valeurs de propriétés, mais également le modèle causal doivent être restaurés à l'état pré-évolution. Notre politique de sauvegarde des modèles avant l'analyse permet à Smile de restaurer facilement l'état du système, comme si rien ne s'était passé. L'architecte de l'évolution peut alors définir une nouvelle évolution, en utilisant les informations fournies sur le diagnostic pour éviter de commettre la même erreur.

Application à PicWeb

Suite à la détection de la violation du contrat, l'architecte de l'évolution prend en compte les causes racines de cette violation. Il conçoit alors une nouvelle évolution après avoir effectué un roll-back de PicWeb, évolution pour laquelle l'implémentation de *shuffle* fut modifier de sorte à ce que le traitement soit parallélisé. Au final, la nouvelle évolution ne violant plus de contrat, elle put être appliqué au système déployé.

8.4.5 Discussion

Nous venons de présenter l'interaction finale entre l'architecte de l'évolution et Smile pour prendre la décision de l'applicabilité de l'évolution. Cette interaction s'est soldée par les challenges suivants :

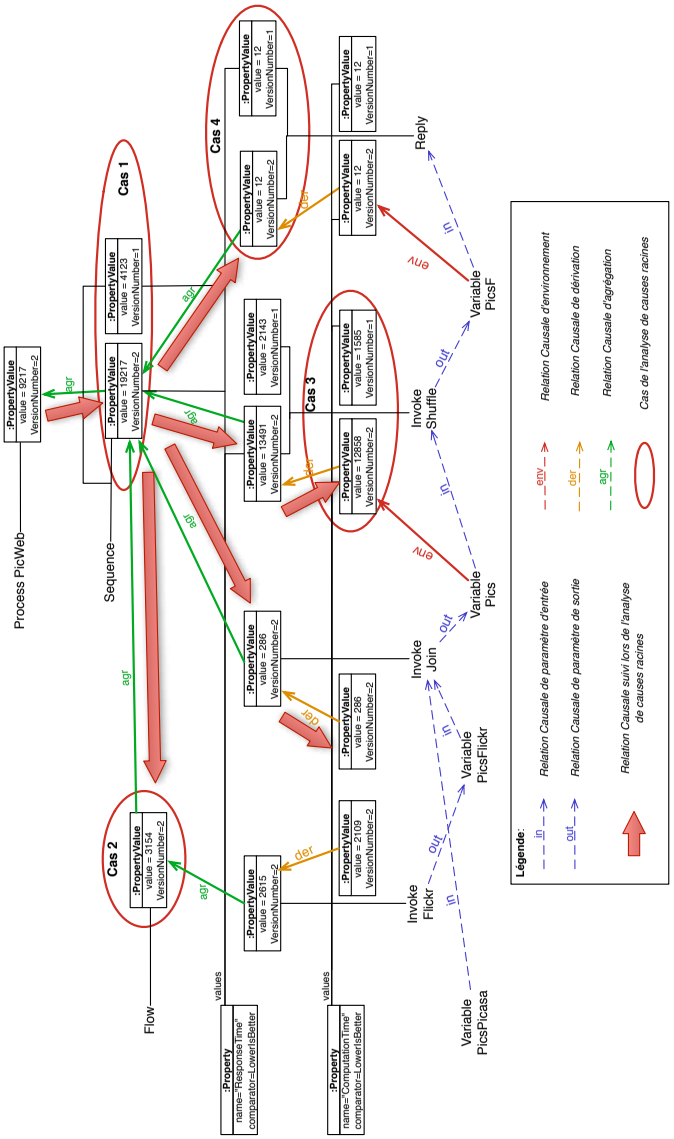


FIGURE 8.9 – Détermination des causes racines de la violation du contrat de PICWEB.

– **Disponibilité des informations essentielles :** l'architecte de l'évolution est guidé dans sa prise de décision par les informations fournies par SMILE, à savoir la chaîne de causalité incriminée, et le différentiel des PropertyValues. En prenant connaissance de la chaîne de causalité et à l'aide des différentiels entre les métriques pré et post-évolution, l'architecte de l'évolution possède les premiers éléments de réponse quant à une possible violation de contrat. Si ces informations ont besoin d'être interprétées par un humain, la mise en évidence d'un sous-ensemble du système responsable de la violation, et la synthèse des données de contrôle en métriques permet toutefois à l'architecte de comprendre de manière plus aisée les effets de l'évolution sur la QoS que s'il avait du analyser manuellement l'ensemble des données de contrôle du système dans sa globalité.

– **Retour en arrière :** en proposant la possibilité de rétablir l'état du système tel

qu'il était avant l'évolution, SMILE assure la cohésion du système à chaque instant du cycle de vie, et permet ainsi à l'architecte de l'évolution de tester facilement des possibilités d'évolution pour satisfaire les nouveaux besoins de l'utilisateur.

8.5 Conclusion du chapitre

Ce chapitre a couvert l'ensemble des étapes pour analyser les effets d'une évolution sur la qualité de service d'un système à base de processus métiers. Constitué d'un langage d'évolution, d'un analyse et d'une étape de prise de décision, l'analyse a de nombreuses propriétés. Parmi elles, nous mettons en avant le caractère d'**automatisation** de l'approche, où la construction du modèle causal et le déploiement d'une version de test pour effectuer des mesures sont totalement automatisés. Notre approche est également remarquable en terme de **réduction du nombre de vérification à effectuer** : en réduisant le nombre d'éléments du système à re-vérifier, nous améliorons les performances de la phase de vérification, et facilitons le diagnostic avec une quantité de données inférieure à traiter. Enfin, grâce à ces données, nous mettons à disposition de l'architecte de l'évolution une **trace de violation de contrats** permettant de le guider dans son diagnostic en cas de violation.

Toutefois, nous pouvons également identifier un certain nombre de limitations à notre approche. L'analyse de l'évolution est un **procédé non-concurrent**. En effet, nous avons émis l'hypothèse simplificatrice qu'une seule évolution était appliquée à la fois. Toutefois, dans un éco-système où le système développé est de plus en plus grand, il n'est pas déraisonnable de penser que plusieurs équipes travaillent en parallèle, et peuvent être amenées à faire évoluer le système en même temps. Il faudrait alors pouvoir déterminer des zones d'évolution, afin d'être en mesure de limiter l'aspect séquentiel à cette seule zone. Ainsi, deux évolutions n'opérant pas sur la même partie du système pourraient s'opérer en concurrence. C'est un défi à résoudre dans le futur. De plus, notre approche repose en grande partie sur l'établissement et la mise à jour du modèle causal. Nous avons vu dans les chapitres précédents comment établir le modèle causal. Notre approche considère que le modèle est consistant et exhaustif en termes de relations causales, si l'on considère uniquement celles que nous avons défini. Toutefois, si d'autres relations causales venaient à être déterminées, il s'agirait alors de les intégrer dans notre module d'évolution.

Dans le but d'améliorer notre approche, nous préconisons de réfléchir aux possibilités d'évolution concurrentes du système. Une piste supplémentaire serait d'améliorer la fiabilité des relations causales. En effet, notamment au niveau de l'influence des ressources, peu d'informations sont fournies par l'expert en QoS. Il serait intéressant de renforcer la description des propriétés pour obtenir des relations causales plus précises. Enfin, de la phase d'analyse, nous pensons que davantage d'informations pourraient être fournies à l'architecte en cas de violations dans l'optique de l'aider à diagnostiquer les causes d'une violation de contrat.

Troisième partie

Validation

Chapitre 9

Mise en œuvre et utilisation de Smile

Sommaire

9.1 Réalisation de Smile 105
9.1.1 Présentation du canevas 106
9.1.2 Besoins des utilisateurs de Smile 106
9.1.3 Architecture de Smile 106
9.2 Smile pour la description du système 109
9.2.1 Import de système 109
9.2.2 Description des règles causales 111
9.3 Smile pour la qualité de service 113
9.3.1 Principe 113
9.3.2 Description d'une propriété 113
9.3.3 Gestion des règles causales de propriété 114
9.3.4 Collecte des données 115
9.3.5 Description d'un contrat 116
9.4 Smile pour l'évolution 117
9.4.1 Description d'une évolution 117
9.4.2 Analyse causale de l'évolution 118
9.4.3 Contrôle à l'exécution 119
9.5 Limites 119
9.6 Conclusion du chapitre 120

Ce chapitre présente la réalisation du canevas Smile, conçu pour mettre en pratique les différentes contributions de la thèse. L'implémentation du canevas [1] propose les outils nécessaires aux différents acteurs de l'équipe de développement pour réaliser les tâches nécessaires à une évolution contrôlée. Notamment, les langages de description de propriétés de qualité de service et d'évolution sont pris en charge, de même que l'analyse causale de l'évolution. Nous détaillons dans ce chapitre les différents choix technologiques, et présentons les fonctionnalités de Smile pour chacun des acteurs identifiés.

9.1 Réalisation de Smile

Dans cette section, nous présentons l'architecture de Smile. Après avoir introduit le contexte du canevas, nous énumérons les besoins des différents utilisateurs, avant de présenter l'architecture de Smile et son découpage en plug-ins.

1. disponible à l'adresse https://github.com/smileFramework/SMILE

9.1.1 Présentation du canevas

Le canevas Smile a été conçu dans le but de fournir les outils nécessaires aux différents acteurs de l'équipe de développement, pour suivre la méthodologie Blink. Nous avons cherché à implémenter Smile de façon modulaire, afin de faciliter sa maintenance, et de façon extensible, afin de pouvoir faire évoluer les fonctionnalités présentes. Pour cela, nous avons fait le choix d'étendre une plate-forme existante, *Eclipse*[2]. Eclipse est un environnement de développement intégré, dont la réalisation a débuté par IBM en 2001. Gérant de nombreux langages de programmation tels que Java ou C++, Eclipse est doté d'un environnement de conception de modèles ainsi que de différents outils propres aux architectures orientées services, tel qu'un éditeur BPEL ou un outil de description d'interfaces WSDL pour les web services. La plate-forme repose sur une construction par plug-ins, rendant possible son extension. Pour toutes ces raisons, Eclipse est devenu un standard *de facto*, faisant d'elle la plate-forme à notre connaissance la plus adaptée pour le développement de Smile.

9.1.2 Besoins des utilisateurs de Smile

Le processus de développement Blink et sa réalisation via Smile ont été pensés pour mettre au centre des préoccupations les trois profils d'acteurs, à savoir *(i)* l'expert en qualité de service, *(ii)* l'architecte du système et *(iii)* l'architecte de l'évolution. Chacun de ces profils ont des besoins spécifiques, besoins devant être satisfaits par une fonctionnalité de Smile. La Figure 9.1 est un diagramme de cas d'utilisation écrit selon le formalisme d'UML. Ce diagramme représente les différents besoins de chaque acteur :

- L'architecte du système a pour tâche d'importer ou de modéliser un système, et de déclencher les analyses de QoS du système, afin de vérifier si les contrats sont respectés (voir chapitre 6). Cela pré-requiert que les contrats et les propriétés de QoS aient été préalablement définis par l'expert en QoS. La section 9.2 présente le support de Smile pour l'architecte du système.
- L'expert en QoS a pour tâche de gérer la QoS du système. Il doit donc pouvoir modéliser une propriété caractérisant le système, définir des contrats de qualité de service pour le système donné et associer les outils de vérification correspondants (comme décrit dans le chapitre 7). Nous verrons dans la section 9.3 les outils fournis par Smile pour réaliser ces tâches.
- L'architecte de l'évolution doit pouvoir définir une évolution en réponse à de nouveaux besoins, analyser les effets de cette évolution sur la qualité de service du système, et vérifier si les contrats sont toujours respectés une fois l'évolution appliquée (voir chapitre 8). La description des fonctionnalités de Smile pour réaliser ses tâches est consignée dans la section 9.4.

Nous présentons par la suite l'architecture de Smile, avant de voir comment les besoins des utilisateurs sont réalisés par Smile.

9.1.3 Architecture de Smile

Afin de pouvoir faciliter la maintenance et l'évolution du canevas, nous avons pensé notre canevas comme un ensemble de modules, réalisés sous formes de plug-ins dans la plate-forme *Eclipse*. La Figure 9.2 représente l'architecture de notre canevas, constitué de plusieurs modules dont leur description est consignée dans le tableau Table 9.1.

2. http://www.eclipse.org/

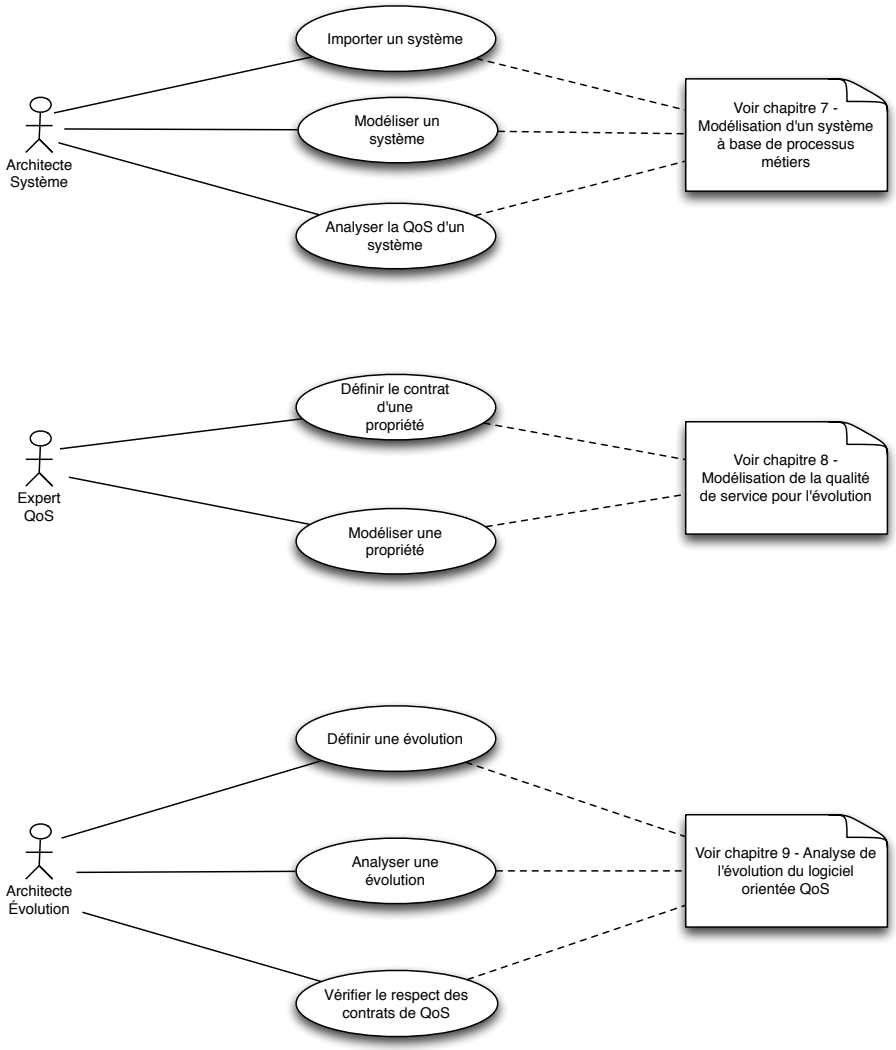


FIGURE 9.1 – Diagramme des cas d'utilisation de SMILE.

Le processus de développement BLINK suit les principes des architectures dirigées par les modèles [OMG 2003]. En se basant sur le concept de développement par raffinements successifs, notre processus a été pensé comme une succession d'étapes où chacune a pour tâche d'enrichir le modèle. De façon similaire, l'implémentation de SMILE suit la logique de BLINK, en présentant le système comme un modèle centralisé, enrichi successivement par les différents acteurs de l'équipe de développement. Pour cela, l'implémentation du canevas repose sur des outils mettant en œuvre l'ingénierie dirigée par les modèles, notamment la librairie *Eclipse Modeling Framework (EMF)* [Steinberg 2008]. EMF est une bibliothèque qui permet, en partant d'une description d'un méta-modèle, de générer les classes java correspondant à son implémentation. La bibliothèque gère notamment le chargement et l'enregistrement de modèles, et permet de fusionner plusieurs modèles entre eux.

La FIGURE 9.3 est une représentation schématique du flot de données de SMILE par rapport aux différents utilisateurs. Chacun des trois profils interagit avec le canevas, pour enrichir le modèle de l'application développé avec leurs connaissances propres. *L'expert en*

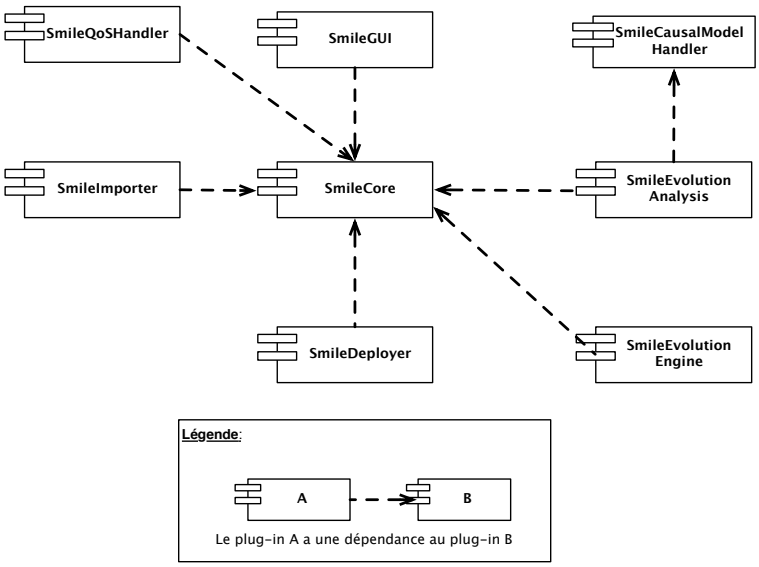


Figure 9.2 – Architecture de Smile.

Module	Description	Lignes écrites	Lignes générées
SmileCore	plug-in central, orchestrant l'interaction entre les différents plug-ins	6853	0
SmileGUI	gère l'interface graphique du canevas	1355	0
SmileImporter	gère l'import des processus métiers écrits en BPEL et des fichiers Composite, en les transposant dans le formalisme de Smile	731	0
SmileQoSHandler	gère l'infrastructure liée aux propriétés de qualité de service : description de propriétés et de contrats, déduction des règles causales	537	2808
SmileCausalModelHandler	gère la synchronisation entre le système et le modèle causal	258	732
SmileEvolutionEngine	applique les opérations d'évolution sur le système étudié	837	0
SmileEvolutionAnalysis	analyse les effets de l'évolution sur le système étudié en s'appuyant sur le modèle causal	1162	0
SmileDeployer	gère le déploiement du système sur le serveur de test ou sur le serveur de production	291	0

Table 9.1 – Description des fonctionnalités par module.

QoS écrit la *description d'une propriété*, qui est utilisé par le plug-in *SmileQoSHandler* pour produire les *règles causales* permettant de construire le modèle causal. *L'architecte du système* utilise le plug-in *SmileImporter* pour produire un *modèle du système* conforme à notre méta-modèle. Ce modèle est alors transformé par le plug-in *SmileCausalModelHandler* pour produire le *modèle causal du système* en utilisant les règles causales obtenues précédemment. Enfin, *l'architecte de l'évolution* décrit une *évolution*, qui est appliquée sur le système importé par le plug-in *SmileEvolutionEngine*. Cette même *description de l'évolution* est enfin analysée par le plug-in *SmileEvolutionAnalysis* : en s'appuyant sur le modèle causal du système, le plug-in réalise l'analyse causale de l'évolution, pour produire un *rap-*

port d'analyse contenant les informations sur le maintien de la QoS, à savoir quel contrat a été violé, et quelles causes racines ont engendré cette violation.

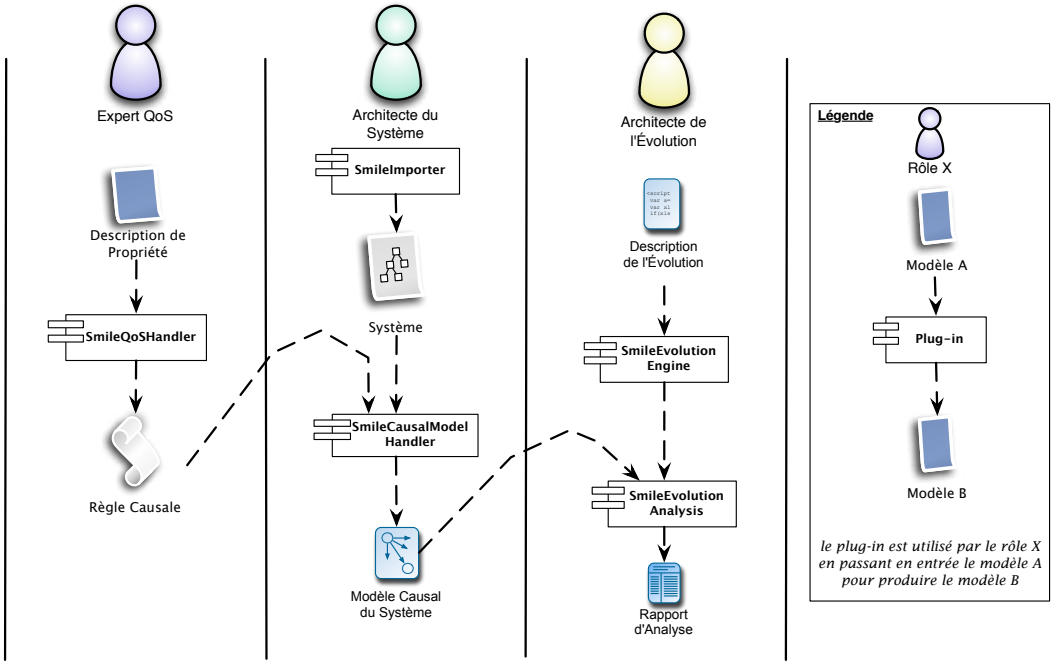


FIGURE 9.3 – Procédé dirigé par les modèles de l'approche SMILE.

Nous présentons dans la suite comment SMILE peut être utilisé en fonction de chacun des profils d'acteurs.

9.2 SMILE pour la description du système

Dans cette section, nous présentons l'ensemble des fonctionnalités disponibles pour l'architecte du système. Nous détaillons dans un premier temps les différentes méthodes pour initialiser l'étude d'un système, à savoir la définition d'un système depuis sa création, l'import d'un processus métier, ou encore l'import d'un fichier de description d'architecture. Puis, nous expliquons comment mettre en œuvre les règles causales pour pouvoir déduire le modèle causal du système.

9.2.1 Import de système

Pour étudier un système avec notre canevas, nous avons voulu donner la possibilité à l'architecte du système de pouvoir construire un système directement dans SMILE, ou d'en importer un existant. Dans le cas d'une construction depuis le début, nous considérons que chaque étape de la construction peut être vue comme une évolution, où le système original serait un système vide. Dans le cas d'un import, nous distinguons deux possibilités : l'import d'un simple processus métier, et l'import d'un système constitué de plusieurs processus métiers. Nous détaillons ces deux possibilités dans la suite.

Import d'un processus métier

L'import d'un processus métier consiste en la traduction d'un processus en concepts définis dans notre méta-modèle (voir FIGURE 6.8). Cette étape de traduction, implémentée par le plug-in **BPEL2SMILE**, prend en entrée un processus métier écrit dans un langage dédié (ici par exemple, en BPEL), pour produire en sortie un modèle de processus métier conforme à notre méta-modèle. Le plug-in réalise la correspondance entre les éléments écrits en BPEL et les éléments décrits dans le formalisme du méta-modèle de SMILE.

Nous avons fait le choix de représenter nos processus métiers comme des graphes pour pouvoir faciliter l'écriture des évolutions, mais également en conservant une structure arborescente afin de pouvoir représenter la notion d'agrégation. Il s'agit ici de traduire la structure arborescente d'un processus BPEL en un graphe représentant l'ordre partiel des activités. Par exemple, la FIGURE 9.4 est une représentation schématique de l'import de PICWEB dans SMILE. Ici, chaque activité est représentée ; l'ordre partiel des activités du flot est retranscrit par le biais des méta-éléments de type *Order*, permettant de créer deux branches se divisant au début du flot, et se rejoignant après l'exécution des dernières activités de chaque branche.

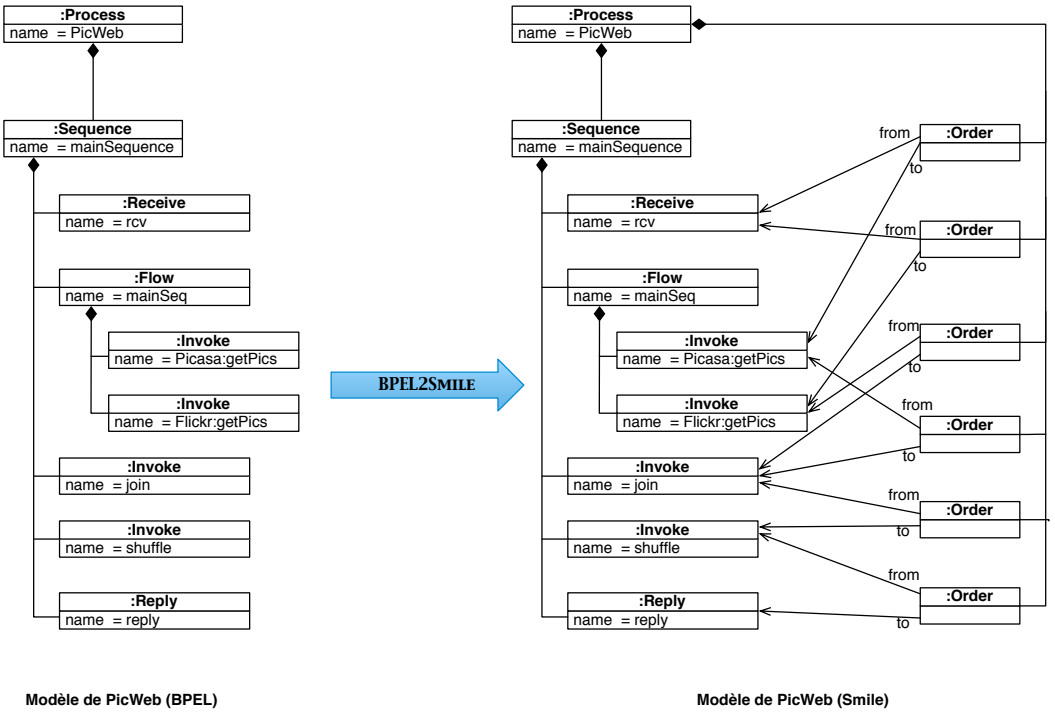


FIGURE 9.4 – Import de PICWEB.

Import d'un système à plusieurs processus métiers

Lorsque l'étude de l'évolution porte sur un système à plusieurs processus métiers, il est nécessaire de pouvoir établir la portée de l'effet de l'évolution sur l'ensemble des processus métiers, ces derniers pouvant s'appeler les uns les autres. Pour cela, nous donnons la possibilité dans SMILE d'importer un système constitué de plusieurs processus métiers. Pour rappel, nous raisonnons sur des systèmes à base de services et de processus métiers, s'exécutant sur une plate-forme. S'il existe plusieurs types de plate-forme d'exécution, nous

avons choisi ici le contexte technologique SCA [Marino 2009], où l'architecture est décrite dans un fichier composite, et où les services sont matérialisés par des composants (voir chapitre 2). Nous avons choisi de nous concentrer sur les architectures correspondant au modèle SCA, car elles proposent un support pour les processus métiers de type BPEL, ainsi que le contrôle à l'exécution.

Dans SMILE, l'import de tels systèmes s'effectue en prenant en entrée le fichier composite du système correspondant. Nous décrivons cette étape dans la FIGURE 9.5 : le fichier composite, au format XML, est analysé pour parcourir chacun des composants décrits. Pour chaque composant, l'analyse du fichier composite détermine si la propriété *implementation.BPEL* est présente ou non. Si tel est le cas, cela veut dire que l'implémentation du composant est réalisée par le biais d'un processus métier. Le processus métier au format BPEL est alors à son tour importé, comme décrit dans la section précédente. La différence majeure par rapport à un import d'un processus métier simple se situe au niveau du lien entre processus. En effet, il s'agit ici de créer des liens entre les appels des différents processus métiers. Le but ici est de pouvoir créer un modèle causal commun, regroupant les différents processus métiers, en construisant le graphe d'appels entre les différents processus métiers du système.

```
<component name="Process">
  <implementation.bpel process="ns:Picweb.bpel"/>
  <service name="picweb" requires="Time">
    <interface.wsdl interface="Picweb.wsdl#wsdl.interface(Picweb)"/>
  </service>
  <reference name="PicasaPartner" target="Picasa/picasa">
    <interface.wsdl interface="Picasa.wsdl#wsdl.interface(Picasa)"/>
  </reference>
  <reference name="FlickrPartner" target="Flickr/flickr">
    <interface.wsdl interface="Flickr.wsdl#wsdl.interface(Flickr)"/>
  </reference>
  <reference name="HelperPartner" target="Helper/helper">
    <interface.wsdl interface="Helper.wsdl#wsdl.interface(Helper)"/>
  </reference>
</component>
```

FIGURE 9.5 – Extrait de fichier composite.

9.2.2 Description des règles causales

Nous avons vu dans le chapitre 6 que la déduction du modèle causal du système s'opérait en appliquant les règles causales du moteur d'exécution sur un système donné. Dans cette section, nous présentons comment les règles causales sont mises en œuvre dans SMILE.

Nous avons développé un méta-modèle permettant d'exprimer les modèles causal. Celui-ci est représenté dans la FIGURE 9.6, où un *CausalModel* est composé de *CausalRelations*. Il est important de noter ici que le méta-modèle est construit pour être extensible : il est ainsi possible de rajouter un nouveau type de relation causale, en héritant simplement de la méta-élément *CausalRelation*.

Pour rappel, nous avons établi qu'une règle causale était un système de règles de production permettant de représenter un motif à trouver dans le système (la situation), afin de créer une ou plusieurs relation(s) causale(s) (l'action). Du point de vue de l'implémentation, nous avons choisi de représenter les règles causales comme des transformations de modèles. En effet, il s'agit ici de règles de réécriture entre le modèle du système, et le modèle causal.

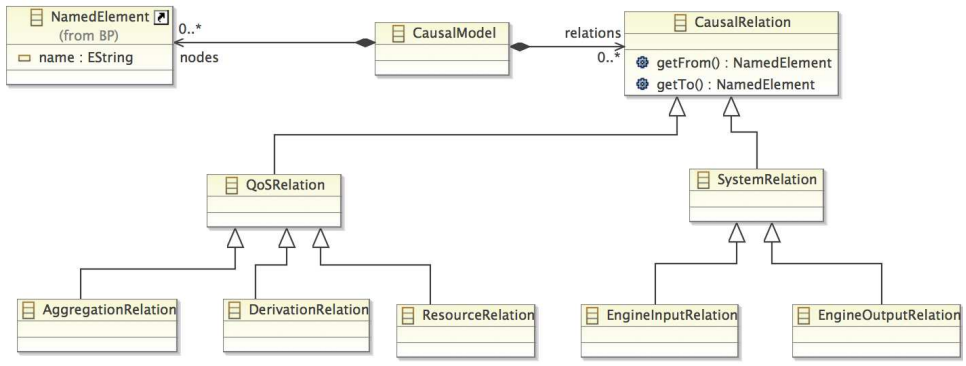


FIGURE 9.6 – Méta-modèle du modèle causal.

Pour cela, l'implémentation des règles causales a été effectuée en s'appuyant sur le langage de transformation de modèles *ATL*[3].

```
1  rels : distinct CM!CausalRelation foreach(a in syst.processes->
2      collect( x | x.activities  -> select( y | y.oclIsTypeOf(SYSMM!Invoke) = true ) ).flatten()->
3      iterate( invoke ; rels: Set(TupleType(left: SYSMM!SystemElement, right: SYSMM!SystemElement))=Set{} |
4          rels -> union( invoke.input.content ->
5          iterate(inVars;relsInv:Set(TupleType(left:SYSMM!SystemElement,right:SYSMM!SystemElement))=Set{} |
6             relsInv.including(Tuple{left=inVars,right=invoke})
7  ))))
8  (
9      from <-model.getMatchingNode(a.left),
10     to <- model.getMatchingNode(a.right),
11     container<-model,
12     type<-'Functional'
13 )
```

FIGURE 9.7 – Extrait d'une règle causale écrite en ATL.

La FIGURE 9.7 est un extrait de la transformation de modèles en charge de construire le modèle causal fonctionnel. Ici, l'extrait a pour tâche de construire les relations causales de paramètres d'entrée. En ligne 1, la règle débute par définir un filtre d'application pour tous les processus métiers du système. Pour chaque processus, la ligne 2 collecte l'ensemble des activités de type *Invoke*. Pour chaque activité, la ligne 3 construit un ensemble de paires de *SystemElement* : il s'agit là de l'ensemble des relations causales de l'ensemble du système. Il est construit comme étant l'union des ensembles de relations causales de chaque activité (ligne 4). Les lignes 5 et 6 construisent les paires d'éléments à proprement parler : selon la définition de la règle causale, il s'agit ici de construire une relation entre chaque variable en entrée de l'activité, et l'activité considérée. Ainsi, la ligne 5 effectue une itération sur toutes les variables en entrée, tandis que la ligne 6 ajoute à l'ensemble des relations une nouvelle paire constituée de la variable itérée, et l'activité considérée. De ces paires, les lignes 9 à 13 en déduisent un méta-élément de type *CausalRelation*, en initialisant les différents champs correspondants.

Pour obtenir le modèle causal d'un système, il suffit alors d'exécuter la transformation de modèle décrite ci-dessus, en passant en entrée le modèle du système.

3. http://www.eclipse.org/atl/

9.3 SMILE pour la qualité de service

Dans cette section, nous présentons les apports de SMILE pour l'expert en qualité de service. D'abord, nous montrons comment la description d'une propriété de qualité de service est mise en œuvre. Puis, nous expliquons comment SMILE déduit les règles causales de QoS à partir de la description d'une propriété. Nous présentons alors le fonctionnement de SMILE pour, à partir du système à étudier et d'une propriété de QoS, déterminer les valeurs de QoS du système. Enfin, nous expliquons le mécanisme de collectes de valeurs de QoS à l'exécution et la possibilité d'exprimer un contrat de QoS.

9.3.1 Principe

Nous présentons dans la FIGURE 9.8 le processus permettant d'analyser les effets d'une évolution sur la qualité de service. Dans cette figure, cinq étapes (cerclées en gras) concernent directement la QoS. La description d'une propriété est la première étape, où l'expert décrit une propriété en utilisant notre langage **QoS4Evol**. Puis, la déduction des règles causales consiste en la génération d'une transformation de modèles représentant notre règle causale. La déduction du modèle causal est l'exécution de cette transformation générée. Elle prend en entrée la description du système, pour produire en sortie le modèle causal correspondant. Après avoir effectué l'analyse de l'évolution (dont la réalisation est présentée plus loin), il s'agit de collecter les données manquantes. Enfin, une fois que toutes les informations sont disponibles, l'étape de vérification des contrats est exécutée, dans le but de savoir si un contrat préalablement décrit a été violé. Nous revenons en détail sur chacune de ces étapes dans les paragraphes suivants.

9.3.2 Description d'une propriété

Dans le chapitre 7, nous avons présenté notre langage permettant de décrire une propriété de qualité de service. Ce langage a pour rôle de consigner dans un support centralisé l'ensemble des informations nécessaires pour calculer une valeur de propriété pour une activité donnée. Il est ainsi possible de définir quel outil d'analyse, quel contrôleur ou quelle formule est nécessaire pour calculer une valeur de propriété.

Nous avons mis en œuvre dans SMILE notre langage de description des propriétés. Ce langage a été réalisé à l'aide du canevas *XText*[4], permettant de décrire la grammaire d'un langage, et d'implémenter le comportement associé. Ce canevas fournit également un ensemble de fonctionnalités facilitant l'élaboration de programmes, telles que l'auto-complétion ou la vérification syntaxique du programme écrit. Nous utilisons *XText* pour décrire le langage de description des propriétés de QoS, dans le but d'en déduire les règles causales et de gérer les valeurs de propriétés au sein d'un système donné. Pour cela, la description d'une propriété est enregistrée sous la forme d'un modèle, utilisé par SMILE pour l'appliquer au système étudié ou pour le compiler afin d'obtenir les règles causales. Notamment, ce modèle permet également de pouvoir réutiliser la description de propriété dans le but de constituer une bibliothèque de propriétés. La FIGURE 9.9 décrit la chaîne d'outillage permettant de décrire une propriété : en utilisant notre langage décrit, l'expert QoS est en mesure d'écrire des propriétés en étant conformes à la grammaire du langage, pour pouvoir ensuite les appliquer sur n'importe quel système.

4. http://www.eclipse.org/Xtext/

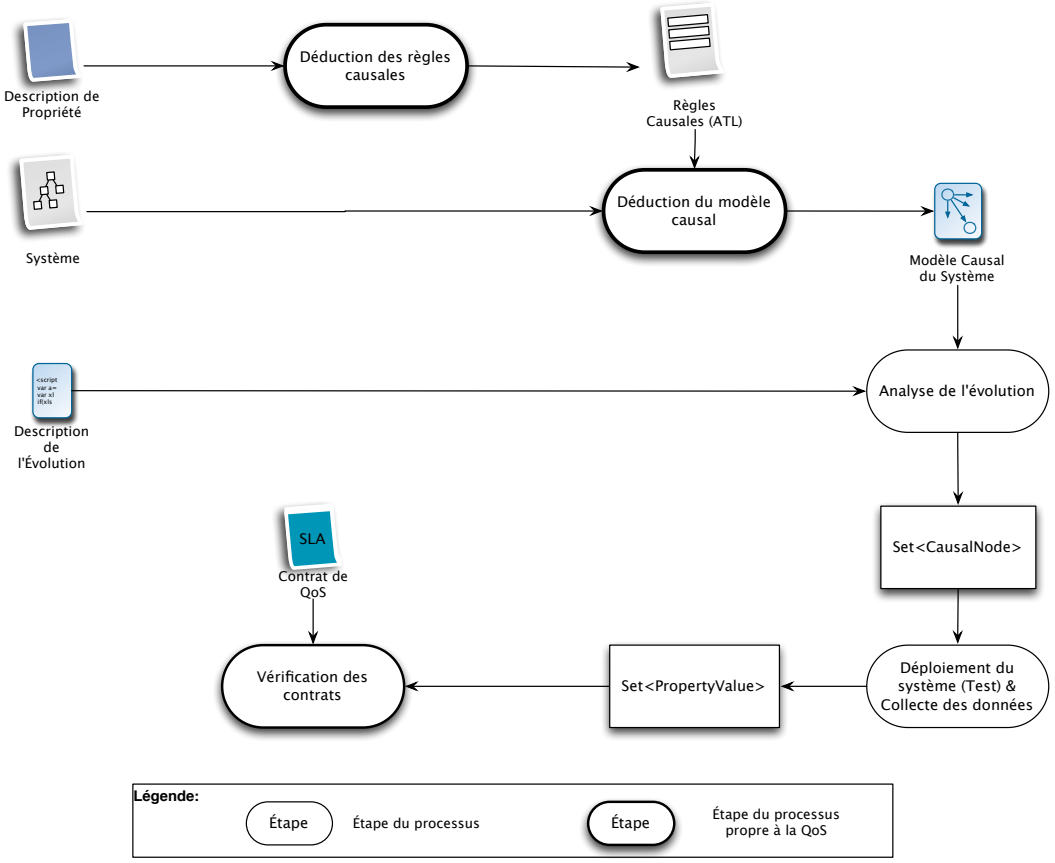


FIGURE 9.8 – Processus de la partie QoS de SMILE.

9.3.3 Gestion des règles causales de propriété

Nous cherchons dans cette partie à obtenir le modèle causal à partir de la description d'une propriété et du système. Cela implique dans un premier temps de déduire les règles causales de la description d'une propriété, pour ensuite appliquer ces règles sur le système pour obtenir le modèle causal.

La déduction des règles causales consiste concrètement en la génération d'une transformation de modèle *ATL*, similaire aux règles présentées en section 9.2.2. De par le caractère exhaustif des types de relations causales déductibles d'une description de propriété, nous avons pré-écrit les modèles de règles causales en nous basant sur les règles de production du chapitre 7. L'implémentation de cette déduction est réalisée à partir de la bibliothèque *Acceleo*[5]. Il s'agit d'une autre technologie de transformation de modèles. À la différence d'*ATL*, la vocation d'*Acceleo* est de s'appuyer sur le modèle en entrée de la transformation pour produire du code, ou de manière plus générale, du texte. Dans notre contexte, nous cherchons ici à générer les règles de transformation au format ATL. En d'autres termes, nous écrivons ici une transformation de modèle écrivant une autre transformation de modèle. Cette bibliothèque fonctionne selon un principe de gabarit structurant le code final, dans lequel sont insérées des informations provenant du modèle en entrée. En exécutant cette transformation de modèle, prenant en entrée la description de la propriété de temps

5. http://www.eclipse.org/acceleo/

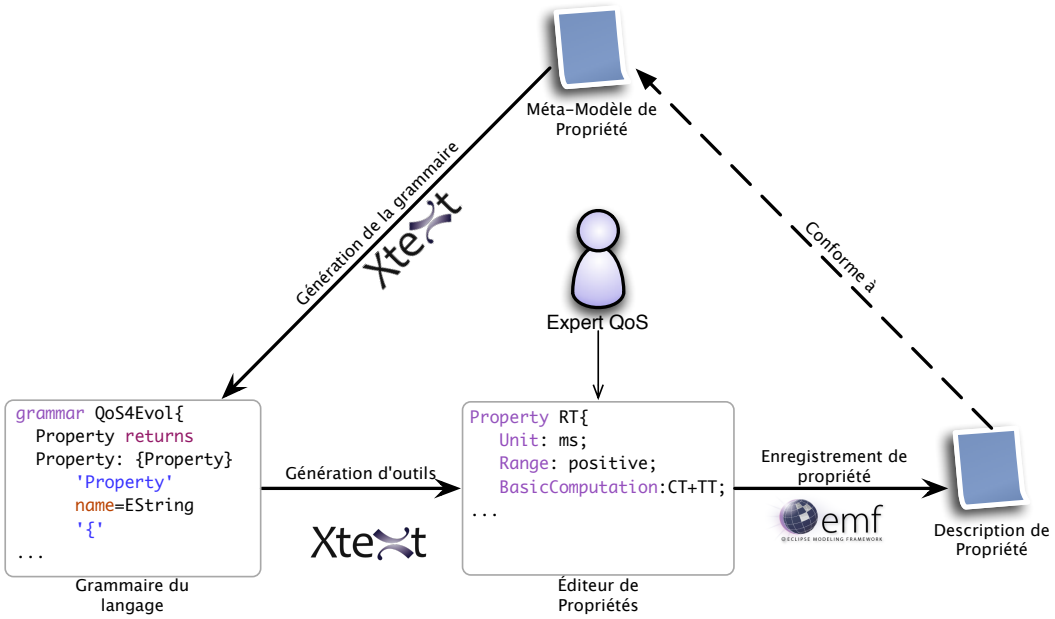


FIGURE 9.9 – Chaîne d'outillage pour la description d'une propriété.

de réponse, nous générons une transformation ATL permettant d'obtenir les relations causales de propriété. C'est cette transformation générée que l'expert en QoS utilise par la suite pour déduire le modèle causal enrichi des relations de QoS du système.

9.3.4 Collecte des données

Afin de pouvoir connaître les valeurs de qualité de service du système, il est nécessaire de pouvoir collecter les données de contrôle et d'analyse, et de pouvoir organiser leur stockage. Nous avons présenté dans le chapitre 7 notre méta-modèle dédié à la qualité de service. Nous rappelons ici simplement la partie en charge de représenter les mesures et valeurs capturées par les différents contrôleurs et analyses, consigné dans la FIGURE 9.10. Pour un élément du système donné, une *PropertyValue* représente sa valeur pour une propriété donnée. Cette valeur représente la synthèse de toutes les mesures effectuées pour l'élément du système caractérisé. Chacune des valeurs enregistrées est conservée à l'aide du méta-élément *PropertyValueInstance*.

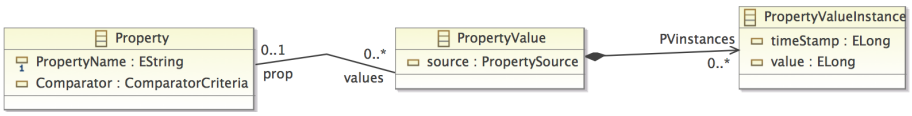


FIGURE 9.10 – Méta-Modèle de traces de SMILE.

La collecte des données est réalisée en exécutant les différents outils d'analyse et de contrôle à l'exécution spécifiés par l'expert en qualité de service dans sa description d'une propriété. Lorsqu'il s'agit d'analyses, il est souvent nécessaire de déclencher un programme extérieur à l'environnement de SMILE. Par exemple, le programme UPPAAL [6] est un outil

6. http://www.uppaal.org/

d'analyse de systèmes temps réels, permettant notamment de vérifier des propriétés temporelles et de raisonner autour des types de données des messages échangés. Dans le but de pouvoir donner la possibilité à SMILE d'exécuter les analyses automatiquement, nous proposons une interface générique *PropertyAnalysis*. Cette interface est nécessaire pour pouvoir harmoniser et automatiser le lancement des analyses. Ainsi, l'analyse peut être enregistrée dans l'annuaire des analyses de SMILE, et être déclenchée automatiquement. Dans le cas d'un outil tiers, il est conseillé d'écrire une enveloppe, implémentant l'interface *PropertyAnalysis*, dont le rôle sera de faire appel à l'outil externe, et de récupérer les données générées par cet outil pour les transformer en méta-éléments conforme au méta-modèle de traces que nous avons présenté dans la FIGURE 7.3 du chapitre 7.

9.3.5 Description d'un contrat

La définition de contrat de qualité de service permet à l'expert en qualité de service d'exprimer des contraintes de temps, de sécurité pour qualifier l'ensemble du système, une opération, ou tout élément du système. Nous proposons dans SMILE d'écrire des contrats de qualité de service pour un système donné en s'inspirant des travaux du langage *CQML+* [Rottger 2003] et *WSLA* [Keller 2003]. Nous apportons pour cela un incrément à la modélisation d'une propriété, en ajoutant la notion de contrat.

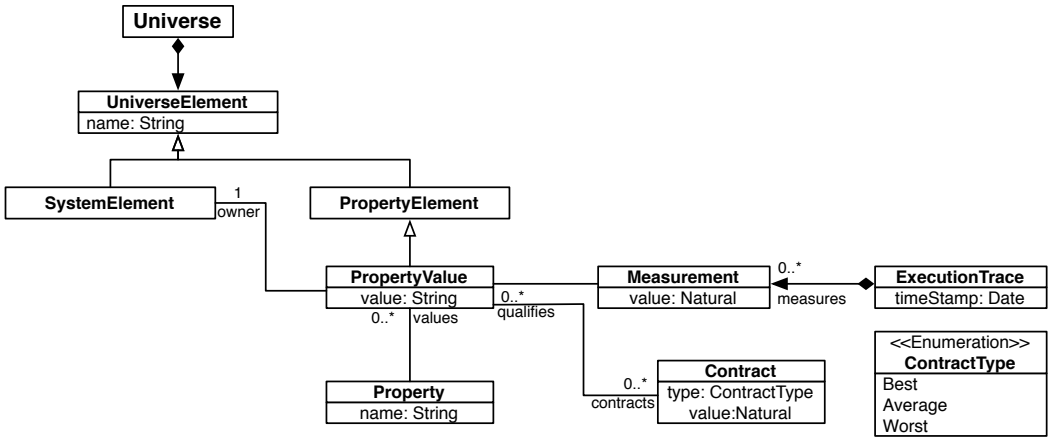


FIGURE 9.11 – Modélisation des contrat de qualité de service.

La FIGURE 9.11 représente la partie du méta-modèle responsable de représenter la notion de *contrat*. Ici, nous représentons par le méta-élément *Contract* une entité spécifiant un type de contrat (*ContractType*) et une valeur (*Value*). Le type de contrat qualifie si la valeur spécifiée correspond au meilleur cas, au pire cas ou à la moyenne.

Nous avons choisi de lier un contrat au méta-élément *PropertyValue*. En effet, ce dernier étant associé à l'élément du système qu'il qualifie, et à une propriété donnée, c'est cette association *SystemElement-Property* qu'un contrat qualifie. Lors de chaque évolution, les contrats associés à une PropertyValue désignée comme étant affectée sont analysés par SMILE. Pour cela, SMILE détermine la nouvelle valeur de propriété pour l'élément qualifié, et la compare à la valeur fixée dans le contrat afin de savoir si ce dernier est violé ou non. Cette comparaison s'appuie sur le paramètre *ComparisonType* de la description de propriété.

9.4 SMILE pour l'évolution

Dans cette section, nous présentons les différents outils fournis par SMILE pour permettre de garantir le maintien de la qualité de service au cours des différentes évolutions. Nous introduisons dans un premier temps la mise en œuvre de l'écriture de l'évolution ; puis nous expliquons comment l'analyse causale de l'évolution est implémentée, avant de terminer par les différents outils aidant l'architecte de l'évolution à diagnostiquer l'effet d'une évolution sur la qualité de service du système étudié.

9.4.1 Description d'une évolution

De manière similaire à la description d'une propriété, l'architecte de l'évolution a la possibilité de décrire une évolution dans SMILE. Comme pour l'implémentation du langage de description de la QoS, nous proposons ici un langage écrit à l'aide de XTEXT permettant de décrire des évolutions. Ce langage, dont nous avons présenté les instructions dans le chapitre 8, a été réalisé à partir du méta-modèle d'évolution décrit dans la FIGURE 9.12.

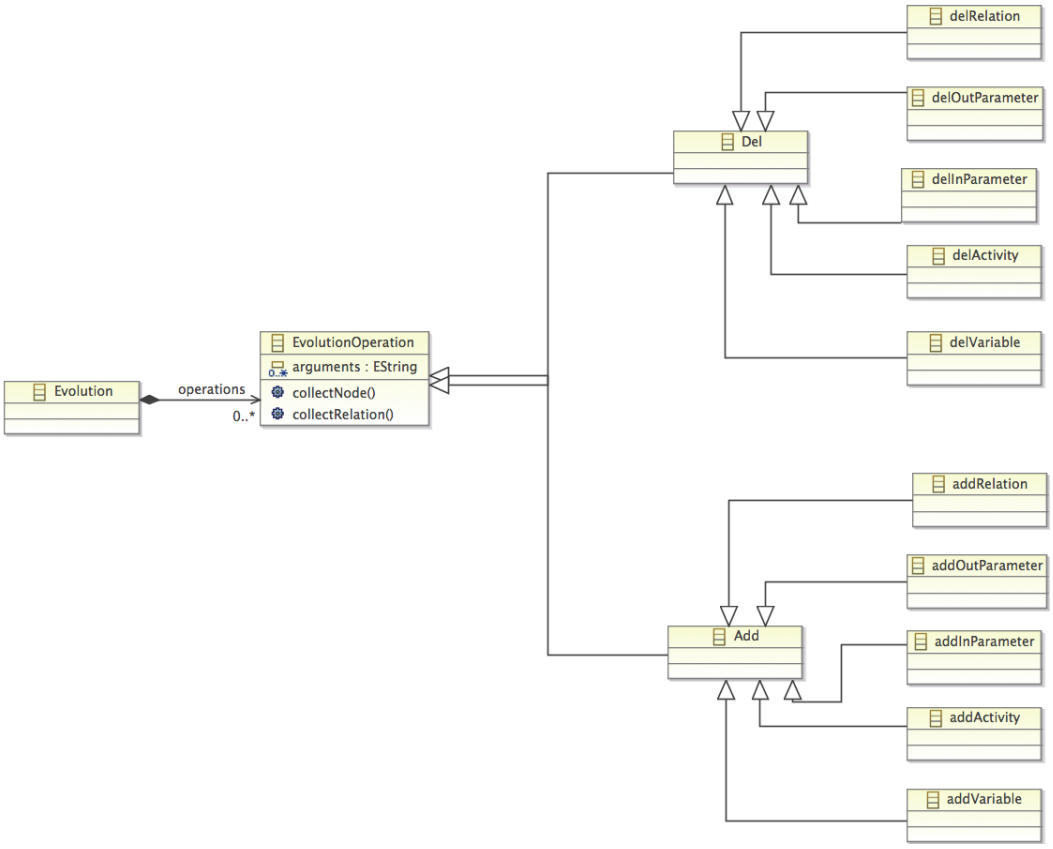


FIGURE 9.12 – Méta-Modèle d'évolution de SMILE.

En se basant sur ce méta-modèle, le canevas XTEXT permet de générer un éditeur d'évolution dont l'utilisation est illustrée dans la FIGURE 9.13. Afin de pouvoir garantir l'extensibilité de notre moteur, SMILE offre la possibilité de pouvoir enrichir le moteur d'évolution de nouvelles opérations. Pour cela, il suffit d'étendre le méta-modèle d'évolution en créant un méta-élément héritant de *EvolutionOperation*. Par exemple, nous pourrions

enrichir notre moteur d'une nouvelle opération « *AjoutActivitéParNom()* », qui, au lieu de créer une activité selon un identifiant, l'ajouterait selon son nom.

FIGURE 9.13 – Utilisation de l'éditeur d'évolutions de SMILE.

9.4.2 Analyse causale de l'évolution

Afin de pouvoir déterminer les effets de l'évolution sur la QoS du système, l'analyse causale reprend les concepts présentés dans le chapitre 8 pour parcourir le modèle causal et identifier les éléments affectés. Pour rappel, l'analyse causale consiste à appliquer l'évolution sur le système, à collecter les éléments du système directement affectés par l'évolution, pour enfin parcourir le modèle causal afin d'identifier les éléments impactés par l'évolution par un effet de causalité. Nous détaillons dans la suite la réalisation de cette analyse.

L'analyse causale débute par l'application de l'évolution. Pour cela, SMILE modifie le modèle du système, et met à jour le modèle causal. Cette étape est effectuée par la classe *SmileEvolutionAnalysis*, où il s'agit d'ajouter ou de supprimer des éléments au système en fonction de l'évolution, tout en préservant la cohérence du point de vue de la QoS. Cela se traduit par la création de nouvelles valeurs de propriété pour les nouveaux éléments, et la suppression des valeurs de propriété orphelines. Une fois le système dans un état cohérent et correspondant à l'application de l'évolution, SMILE modifie le modèle causal en s'appuyant sur la correspondance entre une opération d'évolution et une action sur le modèle causal, décrite dans le chapitre 8 dans la TABLE 8.2. Cette correspondance est implémentée par le biais des méthodes collectNode() et collectRelation(). Dans le cas où nous aurions enrichi le moteur d'évolution avec d'autres opérations d'évolution, il s'agira ici d'hériter du méta-élément *EvolutionOperation*, et donc d'implémenter ces deux méthodes.

Afin de pouvoir amorcer le parcours du modèle causal, l'analyse se poursuit par l'identification au sein des différentes opérations constituant l'évolution des éléments du système directement affectés. Pour cela, l'implémentation de *SmileEvolutionAnalysis* se base sur les différents cas décrits dans le chapitre 8 (voir la TABLE 8.4) pour effectuer cette collecte. Enfin, l'analyse en tant que tel consiste dans le parcours d'un graphe orienté, où chaque nœud rencontré est collecté. Une fois l'analyse causale terminée, l'ensemble des éléments du système pour lesquels une nouvelle analyse ou un nouveau contrôle est nécessaire a été collecté. Il s'agit maintenant de déployer une nouvelle version de test pour déterminer de

nouvelles valeurs. Cette étape est réalisée par le plug-in *RuntimeManager*.

9.4.3 Contrôle à l'exécution

Afin de pouvoir déterminer si la nouvelle version du système respecte les contrats de QoS, il est nécessaire de pouvoir mettre en place des contrôles pendant l'exécution. Dans Smile, le module **RuntimeManager** a pour tâche de mettre en œuvre les mécanismes de contrôle, de gérer la collecte des données et le déploiement des différentes versions de production et de test. Ce plug-in est une partie importante de Smile dans le sens où il est le trait d'union entre le canevas et l'environnement d'exécution. Pour parvenir à récupérer ces données, le module a pour tâche de s'interfacer entre l'architecture du logiciel, le modèle de QoS et les différents processus métiers. Il effectue dans un premier temps la génération du logiciel déployable, embarquant les contrôleurs de QoS nécessaires. Puis, il joue le rôle d'interface entre les contrôleurs et le modèle Smile de l'application, en se chargeant de la persistance des valeurs de propriétés collectées. Nous nous intéressons à chacun de ces deux points dans la suite de ce paragraphe.

Nous avons vu précédemment les différentes étapes pour faire évoluer un système au sein de Smile. Par dualité à l'importation d'un système, notre canevas propose en complément de la phase d'évolution la fonctionnalité d'exporter le système post-évolution, dans l'optique d'une mise en production, ou pour terminer l'analyse de l'évolution en exécutant une version de test. Nous nous intéressons ici tout particulièrement à la version de tests, où il est nécessaire de mettre en place des contrôleurs dans l'application.

La mise en œuvre du contrôle à l'exécution peut reposer sur des mécanismes propres à la plate-forme d'exécution. Dans notre cas, nous préconisons le déploiement des applications à base de services sur une plate-forme SCA, telle que FraSCAti[7]. La spécification SCA a pour objectif de permettre la description d'un modèle architectural, simplifiant l'écriture d'une application SOA. Dans ce contexte technologique, la spécification SCA propose un concept nommé *Intent*, permettant de réaliser des traitements (le plus souvent à vocation non-fonctionnel), avant, en parallèle, et/ou après l'exécution d'un service. Ici, nous utilisons ce mécanisme pour effectuer nos contrôles à l'exécution. Pour cela, l'expert en QoS décrit pour une propriété de QoS donnée l'outil de détermination au moyen d'un Intent. Cet outil, dont le nom est utilisé dans la description de la propriété (voir chapitre 7), peut être utilisé sur n'importe quelle activité, de n'importe quel système. Une fois intégré à Smile, le **RuntimeManager**, dans sa phase d'export du système, incorpore automatiquement l'utilisation de l'intent pour les activités concernées, avant de le déployer.

9.5 Limites

Dans la réalisation de notre canevas Smile, nous avons réalisé une correspondance entre certaines activités du formalisme BPEL vers notre méta-modèle. Toutefois, nous avons fait le choix de ne supporter que les activités rencontrées le plus fréquemment. Certaines activités, telles que les activités propres à la gestion du flot d'exceptions, ne sont pas gérées. De même, nous avons fait le choix de déployer nos applications sur la plate-forme FraSCAti. Notre implémentation dépendant fortement du mécanisme d'*Intent*, notre implémentation est restreinte pour une projection vers les plate-formes SCA implémentant ce mécanisme. Enfin, une limitation de Smile réside dans le manque de généricité de nos outils d'analyse. Il serait intéressant de pouvoir mettre en œuvre un mécanisme permettant d'intégrer facilement n'importe quel contrôleur ou n'importe quelle analyse statique. Ce point sera traité

7. http://frascati.ow2.org/

dans de futurs travaux.

9.6 Conclusion du chapitre

Dans ce chapitre, nous avons présenté l'implémentation de Smile pour répondre aux attentes des différents acteurs de l'équipe de développement du système. L'implémentation de Smile repose fortement sur une approche dirigée par les modèles, en se basant sur un modèle central du système, successivement enrichi par les utilisateurs du canevas. Notre canevas permet l'import d'un système sous différentes formes. Il offre la possibilité de définir des propriétés de QoS, dans l'optique de déterminer les valeurs de propriétés d'un système. Il permet enfin de faire évoluer le système, et d'en analyser les conséquences en terme de qualité de service en utilisant notre implémentation de l'analyse causale et en ré-exécutant analyses et contrôles sur les parties affectées.

Dans le chapitre suivant, nous montrons sur le cas d'un système de gestion de crises comment Smile peut être utilisé pour analyser les effets de ses différentes évolutions.

Chapitre 10

Evaluation

Sommaire

10.1 Défis 122
10.2 Cas d'étude : le Système de gestion de crises 122
10.2.1 Contexte du cas d'étude 122
10.2.2 Description du scénario 123
10.2.3 Cas d'utilisation 124
10.3 Implémentation 127
10.4 Évolutions du scénario 128
10.4.1 Évolutions pour la construction du système 128
10.4.2 Évolutions du processus *Gestion de la Mission* 129
10.5 Évaluation quantitative des contributions 132
10.5.1 Comparaison des éléments à re-vérifier 132
10.6 Discussion 135
10.7 Conclusion du chapitre 135

Dans ce chapitre, nous introduisons un autre cas d'étude nommé *Système de gestion de crise : accident de voiture*, que nous utilisons pour montrer l'apport de Smile pour l'analyse des évolutions, dans le but de maintenir la qualité de service. Notre objectif ici est de montrer que Smile peut être utilisé sur des systèmes composés de plusieurs processus métiers, en accordant une importance sur le gain en terme de précision du résultat, et sur les limites dont notre approche pourrait souffrir. Pour cela, nous évaluons notre approche en se comparant à d'autres approches pour étudier l'effet d'une évolution sur la qualité de service. Nous évaluons également le sur-coût engendré par l'utilisation de l'analyse causale du point de vue du temps d'exécution de l'analyse, et montrons dans quelle mesure ce sur-coût est acceptable comparé aux gains apportés par notre approche.

Ce chapitre est constitué comme suit : dans un premier temps, nous présentons les défis traités dans ce chapitre. Puis, nous introduisons le cahier des charges du *système de gestion de crise*, précisant dans quel contexte il a été conçu, les différents acteurs œuvrant dans le système, et ses différents cas d'utilisation. De ce cahier des charges, nous définissons en section 10.3 une implémentation qui nous sert de base pour l'évaluation de Smile. Nous menons alors une première expérience dans la section 10.4, où nous faisons successivement évoluer le système pour le construire, le but étant de quantifier l'augmentation de la taille du modèle causal au fil des évolutions. Puis, nous nous focalisons sur un processus métier donné, et menons une deuxième expérience où nous mesurons l'apport de notre analyse sur différents scénarios d'évolution. Enfin, il s'agit de critiquer ces résultats, afin de mettre en évidence les points forts et les points faibles de l'utilisation de Smile.

10.1 Défis

Ce chapitre a pour but de démontrer les points suivants :

- **Comparaison de l'analyse de l'évolution avec les méthodes classiques d'analyse d'évolution :** nous avons présenté tout au long de ce document une approche permettant d'analyser les effets d'une évolution sur la QoS d'un système. Toutefois, nous pouvons nous demander quel est le bénéfice de notre approche comparée à des approches analytiques, telles que la re-vérification complète de l'ensemble du système, ou la re-vérification des simples éléments impliqués dans une évolution. Nous cherchons à travers notre cas d'étude à comparer notre approche aux approches existantes, afin d'identifier les bénéfices et les désavantages de SMILE.
- **Précision du sous-ensemble déterminé :** notre cas d'utilisation comporte un ensemble d'évolutions nous permettant de vérifier si l'ensemble des éléments affectés par l'évolution correspond bien aux éléments déterminés par SMILE. Plus précisément, nous chercherons à démontrer que l'ensemble calculé comprendra au moins tous les éléments affectés. Il est en effet vital de n'omettre aucun élément affecté, afin de pouvoir garantir que le maintien ou non de la qualité de service est une information fiable.
- **Passage à l'échelle :** nous avons tout au long du document présenté une approche en nous appuyant sur un exemple constitué d'un processus unique. Si cet exemple, par sa simplicité et sa clarté, facilite la compréhension du fonctionnement de notre approche, il ne permet cependant pas de s'assurer si le même genre de résultats pourraient être obtenus sur des systèmes d'un niveau de complexité plus élevé et constitués de plusieurs processus.

Pour cela, nous proposons d'implémenter le cas d'étude nommé *Système de gestion de crise : accident de voiture*, et d'analyser les différentes évolutions. Ce cas diffère de PICWEB de par sa taille et son évolutivité. Il est un bon candidat pour illustrer les points décrits ci-dessus.

10.2 Cas d'étude : le Système de gestion de crises

Pour illustrer les différentes contributions autour de SMILE, nous utilisons le cas d'étude extrait de journal TAOSD [1] [Kienzle 2010]. Nous présentons dans un premier temps le contexte du cas d'étude. Puis, nous présentons les différents acteurs agissant en coopération avec le système, et décrivons leur rôle. Enfin, nous présentons de manière succincte les différents cas d'utilisation du système de gestion de crise.

10.2.1 Contexte du cas d'étude

De nombreux incidents surviennent au quotidien : catastrophes naturelles (e.g tsunamis, séismes, ouragans), accidents de la route et immeubles en feu sont des exemples d'événements dramatiques pouvant mettre en danger des vies humaines. Dans le but de secourir les victimes, différents corps de métier sont amenés à intervenir pour tenter de résoudre ces situations de crises dans les meilleures conditions possibles. Dans un contexte où chaque minute compte, il est vital de faire collaborer les différents acteurs pour optimiser les chances de sauver des vies. Pour cela, nous proposons de mettre en place un système de gestion de crises, permettant de déployer et coordonner des employés sur le terrain lorsque une crise a

1. *Transaction on Aspect Oriented Software Development*

lieu, et de leur communiquer toute information pertinente pour l'accomplissement de leur mission. Il s'agit ici d'un système critique, dans le sens où un défaut dans la coordination ou toute panne liée au système pourrait avoir des conséquences désastreuses sur le plan humain [Knight 2002, Storey 1996]. Plus particulièrement, nous nous centrerons ici sur un type de crise spécifique, à savoir les crises liées aux accidents de voiture. Dans ce cadre, les tâches à accomplir pour gérer une crise consistent à sécuriser le périmètre de la crise, secourir les victimes et rétablir la circulation. Nous détaillons dans la suite le scénario de résolution de crise.

10.2.2 Description du scénario

Le scénario débute lorsqu'un accident de voiture a été détecté par un *témoin*. Celui-ci appelle le centre de gestion de crises, dans lequel un *opérateur* commence le scénario de crise en recueillant des informations de la part du témoin, telles que la localisation de l'accident, le nombre de véhicules impliqués ou encore le signalement des personnes blessées. L'opérateur vérifie ensuite le caractère véridique des informations communiquées, afin d'éviter tout canular. Si les informations s'avèrent véridiques, la résolution de la crise commence alors. Le système détermine en fonction des informations recueillies quelles ressources sont à déployer sur le lieu de l'accident, et communique aux différents organismes de secours les informations nécessaires pour que la mission de sauvetage puisse être menée à bien. Enfin, le scénario se termine lorsque la crise a été résolue, ou que son ampleur est telle qu'elle ne peut être gérée par le centre de gestion de crises. Elle est alors déléguée à un autorité de gestion de crises hiérarchiquement supérieure.

Nous présentons dans la suite les différents acteurs du système, ainsi que les différents cas d'utilisation du système de gestion de crises.

10.2.2.1 Acteurs du système

Les acteurs du système peuvent être classifiés selon leur rôle dans la résolution de la crise. Nous distinguons trois catégories :

- **rôles opérationnels :** les acteurs tenant un rôle opérationnel ont pour tâche d'agir sur le lieu de la crise pour assister les victimes, rétablir la circulation ou corriger tout problème technique. Ils interviennent directement dans l'avancement de la résolution de la crise. La FIGURE 10.1 présente les différents types d'acteurs ayant un rôle opérationnel : tandis que *les secours* ont pour tâche de stabiliser les victimes sur place, *l'ambulance* a pour mission de transporter une fois stabilisée dans l'hôpital le plus proche. Afin d'éviter tout autre accident, *la police* est dépêchée sur place pour effectuer la circulation, en attendant l'arrivée du *camion de remorquage* qui va tenter de libérer la route en remorquant les véhicules endommagés.

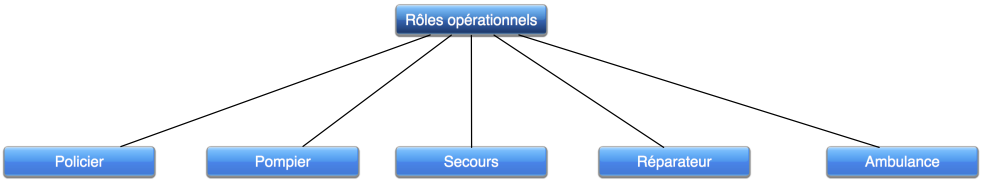


FIGURE 10.1 – Description des différents rôles opérationnels.

Dans un contexte stratégique où plusieurs crises peuvent être gérées en même temps par le système, il est nécessaire de savoir quels acteurs opérationnels peuvent être

potentiellement assignés à la gestion d'une crise. Pour cela, un acteur de cette catégorie peut être soit disponible, soit affecté à une mission de gestion de crise.

– **rôles d'observation :** les acteurs tenant un rôle d'observation sont présents sur le lieu de l'accident. Ils ont pour rôle de faire remonter l'information auprès du centre de commandement, et de coordonner les différentes ressources sur place. Nous distinguons ici deux rôles, rappelés dans la FIGURE 10.2 : *le témoin*, à l'origine de la détection de la crise, fournit des éléments préliminaires de description de la crise. *Le super-observateur* a pour tâche de se rendre effectivement sur le lieu de la crise afin de pouvoir évaluer spécifiquement l'ampleur de la crise, dans le but de définir les missions liées à la résolution de la crise.

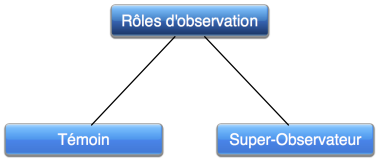


FIGURE 10.2 – Description des différents rôles d'observation.

– **rôles stratégiques :** les acteurs tenant un rôle stratégique ont pour but de fournir des informations clés pour la gestion de la crise. La FIGURE 10.3 en dénombre quatre : *l'opérateur* joue le rôle d'interface entre le système, les rôles d'observation, les rôles opérationnels et les autres rôles stratégiques. Le *coordinateur* gère les acteurs sur le terrain, déploie des acteurs opérationnels ou les rappelle en cas de danger imminent ou de crise plus prioritaire. Enfin, le *système de surveillance* fournit des informations visuelles sur le lieu de la crise, tandis que l'*entreprise de téléphone* permet de vérifier l'identité du témoin signalant la crise en croisant les informations collectées avec leur base de données.

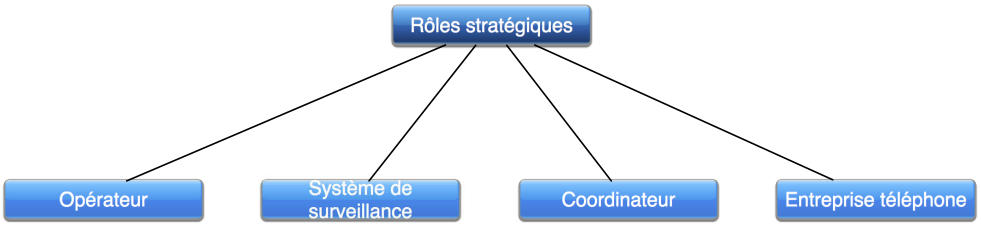


FIGURE 10.3 – Description des différents rôles stratégiques.

Nous voyons dans la suite comment ces différents rôles interagissent dans le contexte de gestion de crises.

10.2.3 Cas d'utilisation

Nous distinguons dans le système de gestion de crises d'accident de voitures plusieurs cas d'utilisation. Nous présentons dans la suite chacun des cas d'utilisation (noté en italique), en mettant l'accent sur les acteurs impliqués, et les données échangées. Si chacun de ces cas est doté dans le cahier des charges de scénario(s) d'évolution [Kienzle 2010], nous faisons le choix de décrire de façon détaillée le cas *Gestion de la Mission* dans la section 10.4.2.

Cas 1 : résoudre une crise

Le scénario du système est mené par le cas principal *"résoudre une crise" (Cas 1)*. Ce dernier est initié par l'opérateur, qui reçoit un appel d'un témoin. Il déclenche alors le cas d'utilisation de *"capture d'un rapport de témoignage" (Cas 2)*. Partant de ce rapport, le système propose un ensemble de missions à mener pour résoudre la crise. L'opérateur sélectionne une ou plusieurs missions pour lesquelles le système va *"allouer des ressources internes" (Cas 3)* et *"solliciter des ressources externes au système" (Cas 4)*. Lorsqu'une ressource est arrivée sur le lieu de la crise, elle *"exécute la mission" (Cas 5)*. Chaque ressource est à même de transmettre des informations concernant l'avancée de la mission et de la résolution de la crise. Lorsque toutes les missions ont été menées à bien, l'opérateur clôt la crise.

Cas 2 : capture d'un rapport de témoignage

Dans ce cas d'utilisation, impliquant un opérateur, un témoin et l'entreprise de télécommunication, le témoin appelle le central pour signaler une crise. L'opérateur saisit dans le système des informations préliminaires rapportées par le témoin, telles que la localisation et le type de crise dont il s'agit. En fonction du type de crise, le système construit une check-list des actions à effectuer pour gérer la crise. L'opérateur saisit alors l'ensemble des informations communiquées par le témoin. En parallèle, partant des informations préliminaires, le système s'assure de l'authenticité de l'information : il sollicite de la part de l'entreprise de télécommunication de vérifier les informations concernant le témoin. Si les informations ne sont pas validées, il s'agit probablement d'un canular. La crise est alors annulée. Dans le cas contraire, le système établit le niveau d'urgence de la crise et définit son statut à *actif*.

Cas 3 : allocation de ressources internes

Ce cas d'utilisation concerne les employés du système de gestion de crises. Ici, il s'agit pour une mission donnée de trouver un employé et de l'affecter sur le lieu de la crise. Pour cela, le système choisit parmi l'ensemble des employés disponibles celui le plus à même d'effectuer la mission. Une fois sélectionné, le système le contacte pour savoir s'il est réellement disponible et s'il accepte la mission. Le cas échéant, l'employé se rendra sur le lieu de la crise pour effectuer la mission spécifique. Si ce dernier n'est pas disponible, le système contacte un autre employé, et ainsi de suite jusqu'à ce qu'il en obtienne un acceptant la mission. Dans le cas où aucun employé n'est disponible, la crise se solde par un échec.

Cas 4 : sollicitation de ressources externes

Ce cas d'utilisation, bien que très similaire au cas précédent, diffère de part la nature de la ressource à solliciter. Ici, le système fait appel à un organisme extérieur pour agir sur la crise : il peut s'agir d'une caserne de pompier, d'un commissariat, ou encore d'un service d'ambulance. Cette caractéristique impacte le comportement du système, par le fait que nous considérons ces ressources comme uniques. Un refus de la part de la ressource entraînerait l'échec de la gestion de la crise, aucune alternative n'étant possible pour effectuer la mission considérée.

Cas 5 : exécution de la mission

Ce cas d'utilisation décrit de manière générale comment se déroule une mission. Nous présentons de manière spécifique chaque type de mission dans les cas d'utilisation suivant.

De manière générale, l'exécution de la mission implique une ressource et/ou un employé. Celui-ci, recevant de la part du système une check-list, réalise l'ensemble des actions à effectuer pour accomplir sa mission.

Cas 6 : exécution de la mission du super-observateur

La mission du super-observateur consiste à se rendre sur les lieux de la crise pour superviser et donner des informations considérées comme fiables au système. Le super-observateur, en collaboration avec le système, définit les missions nécessaires pour enrayer la crise. Le scénario débute lorsque le super-observateur est arrivé sur les lieux de la crise. Le système lui envoie une check-list, dans le but d'obtenir plus d'informations sur la crise. Partant de ces informations, le système suggère un ensemble de missions au super-observateur, qui va en sélectionner une ou plusieurs, et fournir des informations spécifiques aux missions. Par exemple, il indiquera le nombre de victimes que l'ambulance devra transporter, ou la taille des véhicules endommagés qu'il faudra déplacer. Au fur et à mesure de l'avancement des missions, le système tient le super-observateur informé, jusqu'à ce que la crise soit terminée.

Cas 7 : exécution de la mission de secours

Ce cas d'utilisation concerne toute ressource devant effectuer une mission de secours. Le scénario commence lorsque le secouriste arrive sur les lieux de l'accident. Celui-ci, après avoir examiné les victimes, fournit au système les informations concernant les blessures des victimes. Si possible, le secouriste détermine l'identité des victimes pour les communiquer au système. Le système contacte alors les hôpitaux connectés afin d'obtenir tout antécédent médical de la victime. Le secouriste, après avoir administré les premiers soins, préparent les victimes nécessitant une hospitalisation pour les conduire à l'hôpital. Le système communique alors l'adresse de l'hôpital le plus approprié au secouriste, qui va informer en retour le système lorsqu'il aura quitté le lieu de la crise, qu'il se sera rendu à l'hôpital pour y déposer la victime, et lorsque enfin, la mission est terminée.

Cas 8 : exécution de la mission de transport par hélicoptère

Ce cas d'utilisation concerne le pilote d'un hélicoptère. Le système demande au pilote d'aller chercher un employé à l'adresse indiqué. Celui-ci s'y rend, récupère l'employé et l'amène sur le lieu de la crise.

Cas 9 : exécution de la mission d'enlèvement d'obstacle

Lorsqu'un véhicule impliqué dans l'accident est endommagé et bloque la route, il est nécessaire d'affréter une dépanneuse sur les lieux de la crise afin de pouvoir libérer la voie et rétablir la circulation. Ce cas d'utilisation concerne le conducteur de la dépanneuse. Le système contacte le conducteur en lui communiquant des informations sur les véhicules à déplacer et l'emplacement de la crise. S'agissant une fois encore d'une collaboration avec un organisme externe au système, le cas se termine par un échec si ce dernier répond de manière défavorable. Si toutefois la mission est acceptée, le conducteur se rend alors sur les lieux de l'accident pour effectuer sa mission, et tient informé le système de l'avancement de la mission.

10.3 Implémentation

Notre implémentation du système de gestion de crises est structurée comme un ensemble de processus métiers et de services pouvant s'appeler entre eux. Au total, onze processus ont été réalisés pour correspondre au mieux au cahier des charges. La FIGURE 10.4 est une vue architecturale du logiciel réalisé. Notre implémentation est déployé sur la plate-forme SCA *fraSCAti*[2] [Seinturier 2012], permettant de déployer des processus métiers tout en proposant un ensemble d'outils pour insérer des contrôleurs à l'exécution (voir chapitre 9).

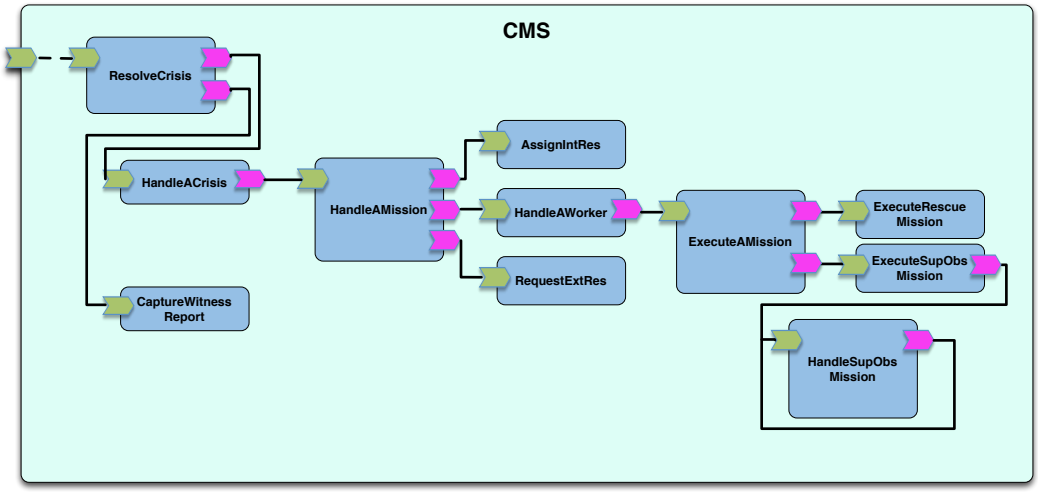


FIGURE 10.4 – Architecture du système de gestion de crises.

Afin d'illustrer les relations entre les différents processus, la FIGURE 10.5 représente leur graphe d'appels. Il est important de noter que chaque processus métier du système est lié à au moins un autre processus métier. L'étude de l'impact d'une évolution pourra donc potentiellement franchir la frontière du processus lui-même, faisant de notre implémentation un bon candidat pour observer les apports de SMILE dans l'étude de l'effet d'une évolution sur l'ensemble du système.

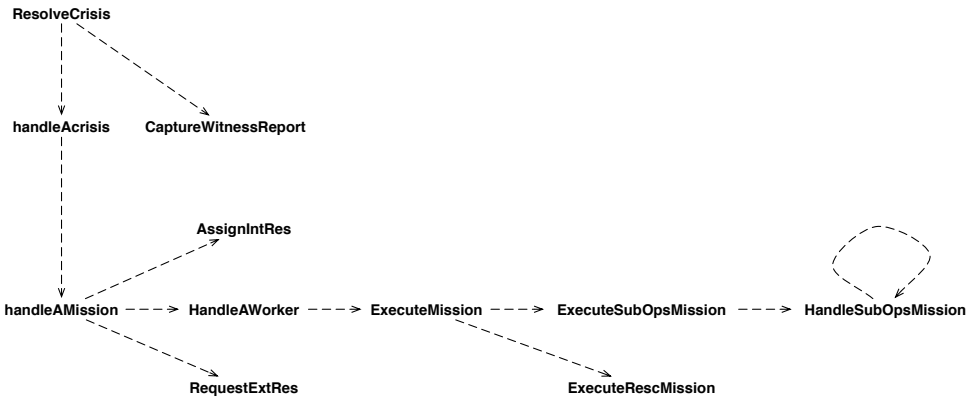


FIGURE 10.5 – Graphe de dépendance des processus métiers du système de gestion de crises.

Notre implémentation est également constituée de différents services implémentés dans

2. http://frascati.ow2.org

un langage généraliste (ici Java). Afin de pouvoir se donner une idée des différents services présents, le graphe d'appel complet du système de gestion de crises est représenté en annexe.

Nous présentons dans la suite de ce chapitre les différentes évolutions pouvant être appliquée au système de gestion de crises, avant de présenter les différents résultats sous-jacents à l'utilisation de SMILE dans ce contexte.

10.4 Évolutions du scénario

Dans cette section, nous présentons de manière succincte les différentes évolutions que nous considérons pour évaluer notre approche. Nous pouvons classifier ces évolutions en deux catégories : les évolutions consistant à construire de manière incrémentale le système, et les évolutions prenant en compte les extensions possibles décrites dans le cahier des charges. Nous traitons dans la suite chacun de ces points.

10.4.1 Évolutions pour la construction du système

Un premier ensemble d'évolution que nous pouvons considérer consiste à prendre la construction du système comme un ensemble d'évolutions successives. Pour cela, nous partons d'un système correspondant à la description architecturale établie dans la FIGURE 10.4, mais où toutes les opérations des services sont implémentées non pas à l'aide de processus métiers, mais dans un langage de programmation classique. Puis, incrémentalement, nous faisons évoluer le système en remplaçant l'implémentation d'une opération par un processus métier. Cette méthode nous permet de pouvoir déterminer l'apport d'informations supplémentaires engendré par l'expression du comportement d'une opération à l'aide d'un processus métier. Nous étudions ainsi un ensemble de dix évolutions, où chaque évolution consiste à la construction d'un processus métier. Dans ce contexte, nous étudions le modèle causal fonctionnel, dépourvu de relations causales de QoS.

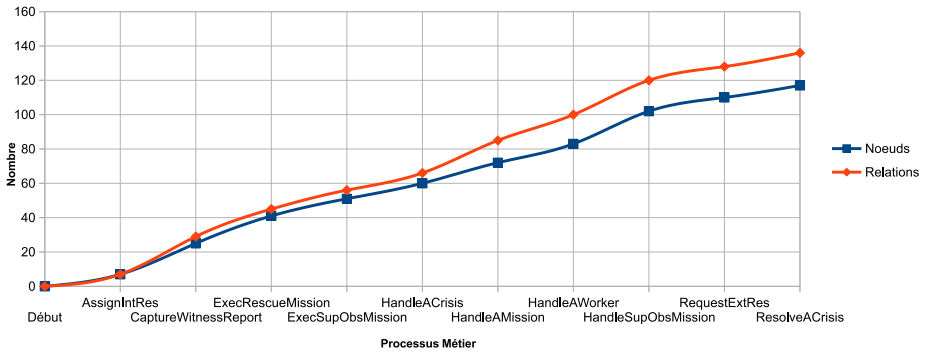


FIGURE 10.6 – Évolution de la taille du modèle causal au cours de la construction du système.

La FIGURE 10.6 est un graphique recensant le nombre de nœuds et le nombre de relations au fil des évolutions, de manière cumulative. Pour nous comparer, nous représentons le même modèle causal, mais sans l'expression de processus métiers. Nous pouvons nous rendre compte ici que, bien évidemment, la richesse supplémentaire d'un processus métier permet d'établir davantage de causalités au sein du système. Si cette information n'est

pas surprenante en soi, il convient cependant de considérer ici le passage à l'échelle de la construction d'un modèle causal. En effet, s'il est appréciable de pouvoir connaître un ensemble d'informations supplémentaires sur le comportement du système, son exploitation doit en rester rapide. Ici, pour dix processus métiers, contenant au total soixante-huit activités, notre modèle causal croît selon une tendance linéaire, pour atteindre un total de cent dix-sept nœuds et cent trente-six relations causales. Si de plus, nous prenons en compte un ensemble de trois propriétés tel que nous l'avons défini, nous obtenons un total de cinq cent vingt-trois nœuds et six cent vingt-et-un relations causales. Il est malgré tout inutile de s'alarmer sur cette croissance. En effet, hormis la taille des données, il convient d'apporter également d'importance au temps d'exécution de l'analyse causale.

10.4.2 Évolutions du processus *Gestion de la Mission*

Dans cette partie, nous nous intéressons particulièrement au processus métier nommé *HandleAMission*. Celui-ci est accompagné dans le cahier des charges d'une description de trois évolutions. Nous présentons dans un premier temps leurs objectifs et leurs réalisations, avant de montrer l'apport de SMILE dans leur mise en œuvre.

10.4.2.1 Description du processus

Comme présenté dans la section 10.2, le processus d'exécution d'une mission consiste à déterminer quelles sont les ressources et employés, qu'ils soient internes au système ou faisant partie d'une organisation externe, à déployer pour effectuer la mission. Une fois leur détermination effectuée, le processus les prévient de leur affectation et suit leur évolution sur le terrain. Nous avons écrit un processus métier correspondant à la description du cas d'étude. Ce processus, constitué de neuf activités, est décrit dans la FIGURE 10.7.

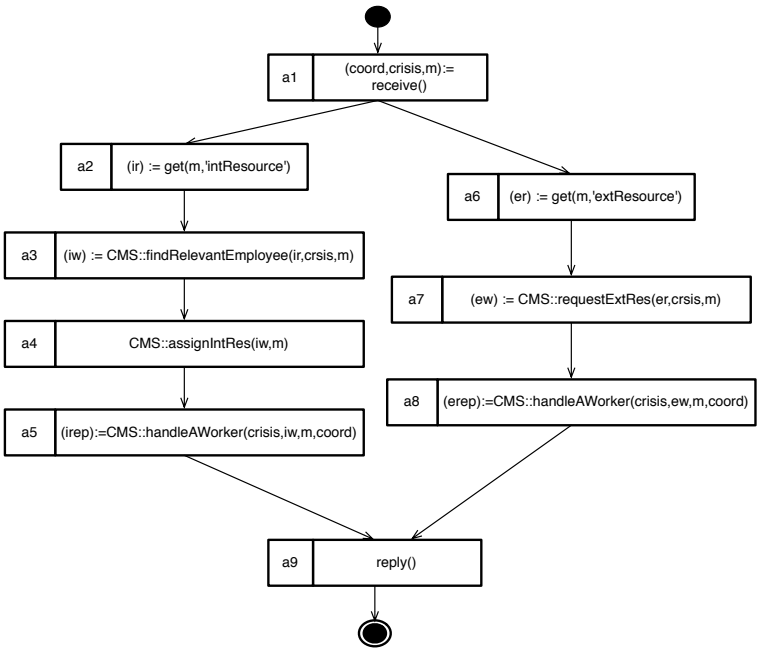


FIGURE 10.7 – Processus métier du cas d'utilisation "Exécution de la mission".

Le processus métier est initié par la réception des coordonnées de la crise, différentes informations, et la mission à gérer (activité *a1*). En parallèle, le processus gère les ressources internes au système, et sollicite les entreprises externes nécessaires. Dans le premier cas, il s'agit de rechercher les ressources et les employés en interne disponibles pour effectuer la mission (activités *a2* et *a3*), de les assigner à la mission (activité *a4*), puis de laisser la gestion propre de l'employé au processus métier *handleAWorker* (activité *a5*). Dans le cas des ressources externes, le processus métier va également solliciter les ressources et employés externes nécessaires (activités *a6* et *a7*), pour déléguer ensuite leur gestion au processus métier *handleAWorker* (activité *a8)*. Enfin, *l'exécution de la mission* se termine (activité *a9*).

10.4.2.2 Évolutions

Dans le cahier des charges, un ensemble de trois évolutions est décrit. Nous les présentons de manière succincte ici.

– **Ressource interne non disponible** : dans le cas où l'*employé* assigné par le système n'est pas disponible, une évolution possible au comportement actuel serait de choisir un nouvel *employé* provenant d'une entreprise externe, à même de remplir la mission. Pour cela, la FIGURE 10.8 représente le processus métier une fois cette évolution appliquée. L'évolution consiste à ajouter une nouvelle activité *a10* invoquant le processus *requestExtRes*, et d'ajouter des gardes dans le flot de contrôle pour conditionner l'exécution en fonction de la disponibilité ou non de l'employé interne.

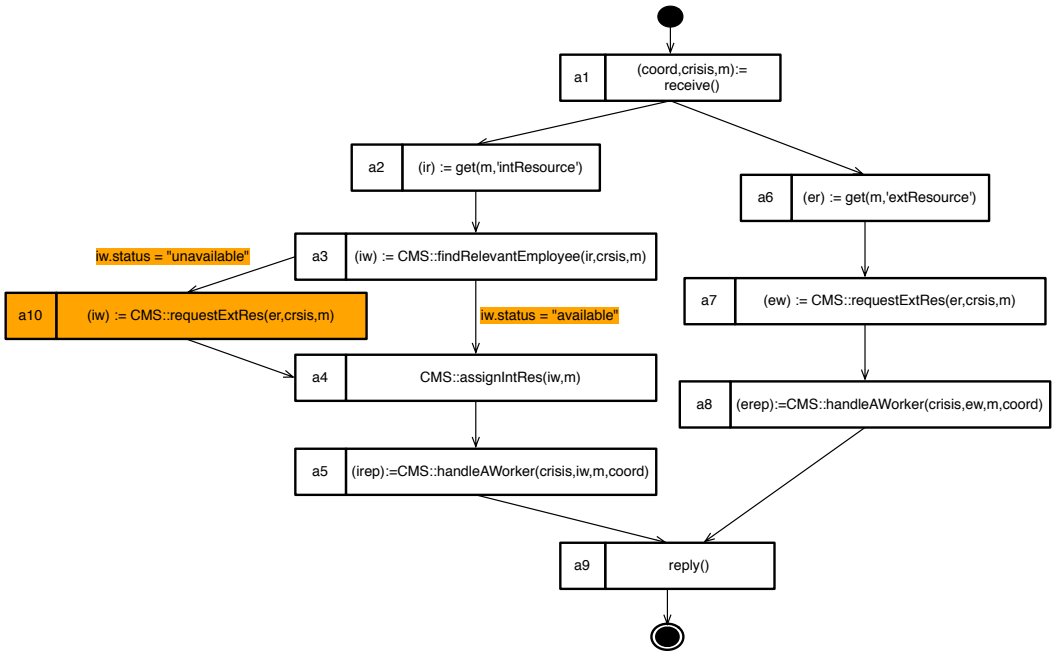


FIGURE 10.8 – Évolution du processus métier *Gestion de la mission* : UnavailableIntRes.

– **Ressource externe non disponible** : dans le cas où la non-disponibilité concerne une ressource externe au système, il n'est pas possible de trouver une nouvelle ressource pour prendre sa place. Dans ce cas bien précis, il s'agit de signaler aux autres

parties du système que quelque chose s'est mal déroulé, et qu'il faut reprendre la gestion de la crise avec, par exemple, la définition d'une mission alternative. Afin de réaliser ce traitement, nous effectuons l'évolution représentée dans la FIGURE 10.9. Il s'agit ici de s'assurer que l'employé externe, matérialisé par la variable *ew*, est dans un statut *indisponible*. Dans ce cas, l'activité *a11* émet sur le support de communication un message signalant un changement nécessaire de statut pour la crise considérée, pour ensuite ré-exécuter le processus métier *handleACrisis*, par le biais de l'activité *a12*.

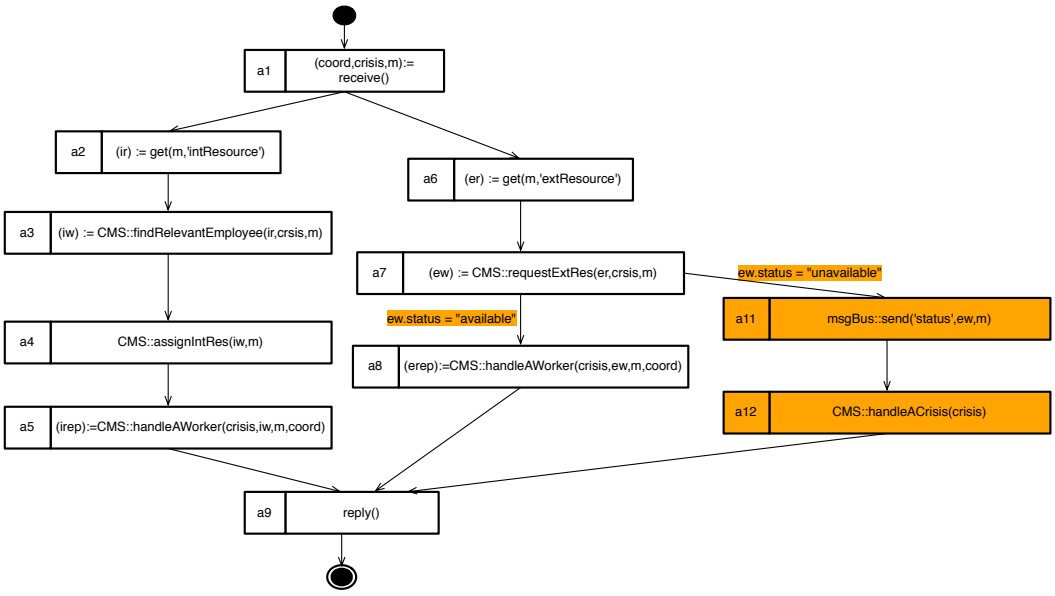


FIGURE 10.9 – Évolution du processus métier *Gestion de la mission* : UnavailableExtRes.

– **Gestion du changement** : dans un contexte où les traitements des différentes informations sont réalisées en parallèle, il est important de pouvoir être réactif face à un changement brusque dans la stratégie de la gestion de crise. Par exemple, il n'est plus utile d'envoyer un camion de remorquage sur les lieux de l'accident si l'officier de police parvient à la déplacer. Il est donc nécessaire de pouvoir réagir si un changement dans la gestion de la crise s'opère. Pour cela, nous faisons évoluer le processus métier pour introduire un mécanisme de réaction au changement. L'évolution consiste à ajouter une écoute sur le bus de communication pour établir si un changement a lieu (activité *a13*. Si tel est le cas, le processus métier se termine en effectuant auparavant un appel au processus métier *handleACrisis* (activité *a14*).

10.4.2.3 Analyses

Afin d'illustrer l'effet de l'analyse causale sur l'évolution, nous présentons dans la FIGURE 10.11 un extrait du modèle causal, pour lequel les relations causales d'agrégation sont représentées, pour la propriété de Temps de Réponse. Comme nous pouvons le voir ici, l'analyse causale permet d'identifier un sous-ensemble d'éléments pour lesquels l'évolution a eu une influence. En l'occurrence, il s'agit ici de l'ensemble des valeurs de propriétés reliées par une relation d'agrégation, permettant de ce fait d'éviter un ensemble de re-vérifications

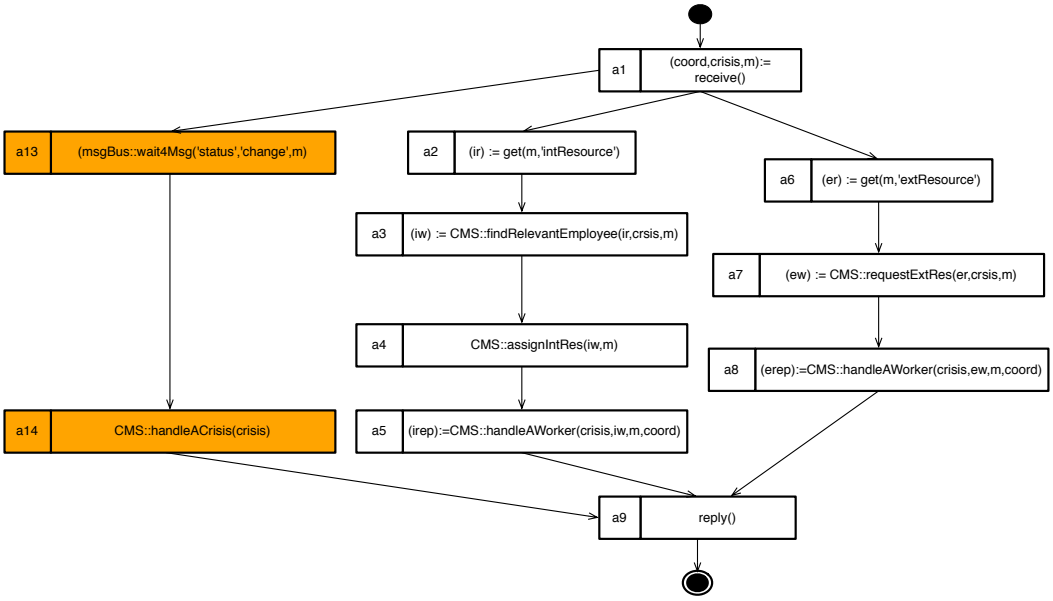


FIGURE 10.10 – Évolution du processus métier *Gestion de la mission* : RehandleOnChange.

inutiles. Sur un ensemble de seize valeurs de propriété, notre analyse a restreint le nombre de vérifications à cinq, pour une seule propriété de QoS étudié. Cela représente un gain de soixante-huit pourcents.

De plus, dans cette situation précise, l'évolution décrite s'opère sur des activités n'ayant aucune influence sur le flot de données du processus métier. Si l'évolution n'engendre pas d'effet indirect d'un point de vue fonctionnel, notre approche permet malgré tout d'obtenir un gain non négligeable dans la re-vérification.

10.5 Évaluation quantitative des contributions

Dans cette section, nous cherchons à évaluer quantitativement notre approche, en comparant son utilisation par rapport à d'autres approches d'analyse de l'évolution, et en quantifiant ses limites. Nous cherchons par exemple à déterminer sur un système plus complexe que PICWEB comment se comporte notre modèle causal en termes de taille, mais également est-ce que notre analyse passe à l'échelle sur un système plus grand. Nous étudions donc le temps d'exécution pour l'analyse, lorsqu'elle est réalisée à la conception, ou lorsqu'elle est réalisée à l'exécution.

10.5.1 Comparaison des éléments à re-vérifier

Dans cette section, nous construisons un cadre de comparaison de l'analyse de l'évolution en terme de précision de l'ensemble des éléments à re-vérifier. Pour cela, nous utilisons l'ensemble des évolutions établies précédemment, et comparons notre analyse à d'autres d'approches permettant de re-vérifier la QoS d'un système lors de son évolution. Nous dénombrons trois approches différentes :

– **Analyse des éléments d'une évolution :** il s'agit ici d'une approche naïve, considérant que le seul effet d'une évolution est circonscrit dans le seul périmètre des

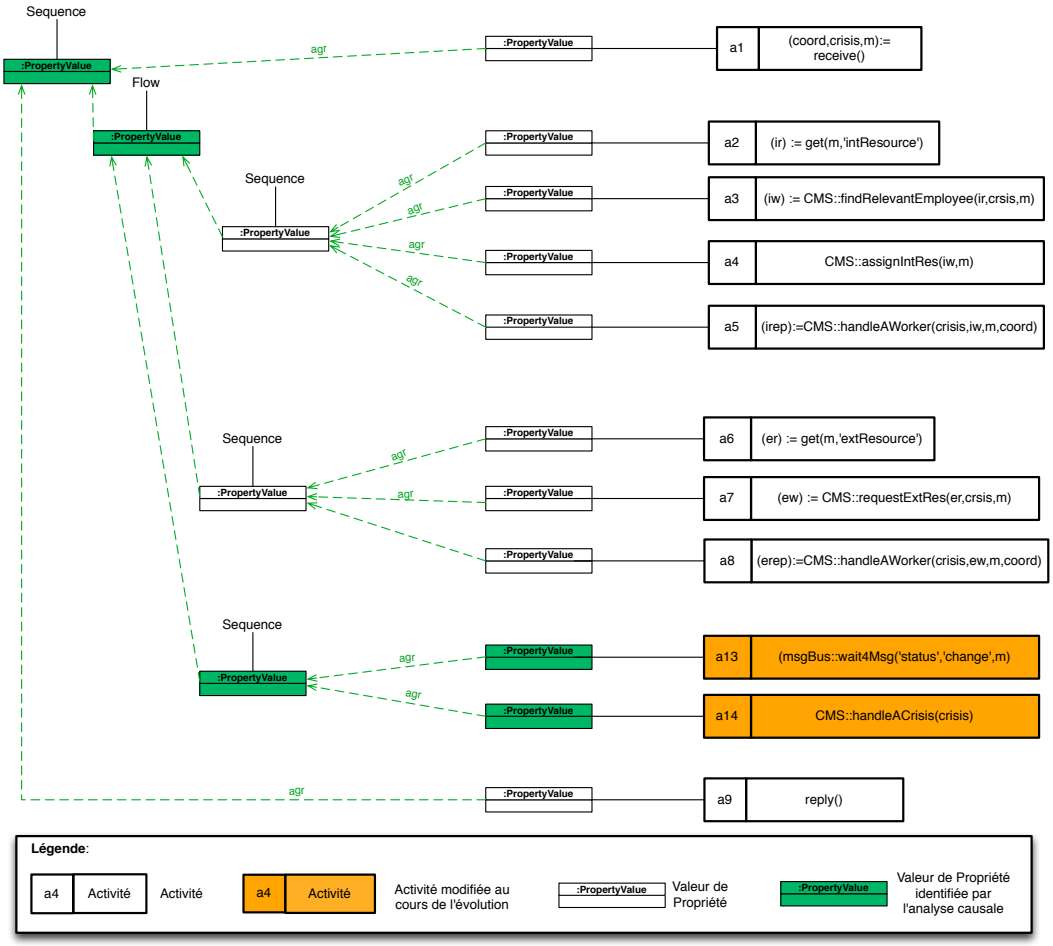


FIGURE 10.11 – Extrait du modèle causal du processus métier *Gestion de la mission* après l'évolution *RehandleOnChange*.

éléments directement manipulés dans l'évolution. Par cette méthode, on considère que l'évolution n'a aucun effet sur le reste du système.

- **Analyse de l'ensemble du système à chaque évolution :** cette approche est précautionneuse, dans le sens où l'on considère ici que l'évolution a une influence potentielle sur l'ensemble du système, et qu'il est donc nécessaire de tout re-vérifier.
- **Utilisation du modèle causal fonctionnel :** dans cette démarche, nous considérons notre approche de détermination de l'effet d'une évolution par le biais d'un modèle causal. Ici, nous considérons uniquement les relations causales fonctionnelles, pour lesquelles un effort de modélisation de la QoS n'est pas nécessaire.

Sur ces approches, en confrontation avec notre méthode d'analyse basée sur un modèle causal associant les relations causales du système et de la QoS, nous comparons le nombre de valeurs de propriété à re-vérifier en fonction de chaque exécution (contrôleurs, et valeurs composées). La TABLE 10.1 consigne ces nombres pour chaque méthode. Il est important de relever un certain nombre d'informations ici :

- **La re-vérification complète du système est consommatrice de ressources.** Comme établi dans cette table, il est évident de constater que la re-vérification com-

TABLE 10.1 – Nombre de valeurs de propriété à re-vérifier pour chaque méthode d'analyse.

BP / evolution	Re-vérification Complète		Re-vérification Évolution		Modèle Causal Fonctionnel		Modèle Causal QoS	
	Contrôle	Calc.	Contrôle	Calc.	Contrôle	Calc.	Contrôle	Calc.
BP resolveACrisis	7 acts							
MustAuthenticate	20	10	6	3	6	3	3	3
BP handleACrisis	5 acts							
ReHandleOnChange	20	10	10	5	10	5	5	5
BP handleAMission	11 acts							
UnavailableIntResource	24	12	2	4	**8**	**4**	**4**	**4**
UnavailableExtResource	30	15	8	4	8	4	4	4
ReHandleOnChange	32	16	10	5	10	5	5	5
BP handleAWorker	10 acts							
UseHelicopter	26	13	6	3	6	3	3	3
LostContact	26	13	6	3	6	3	3	3
Timeout	34	17	14	7	14	7	7	7
MissionFailed	26	13	6	3	6	3	3	3
ReHandleOnChange	30	15	10	5	10	5	5	5

plète du système est celle qui comporte le plus de valeurs de propriété à re-vérifier. De surcroît, nous pouvons également constater que plus le ratio entre le nombre d'éléments manipulés par l'évolution et le nombre d'activités d'un processus métier est petit, plus la sur-charge de cette méthode avec les autres méthodes présentées est importante. Ce sur-coût en terme de re-vérifications est ici le prix à payer pour obtenir une certitude totale sur l'exactitude du résultat de l'analyse.

– **Le modèle causal enrichi des relations causales de QoS fournit de meilleurs résultats que le modèle causal fonctionnel.** La modélisation du temps de réponse que nous avons présenté dans le chapitre 7 permet d'obtenir des relations causales plus précises sur le système. Concrètement, cela veut dire que nous avons établi une dépendance causale directe entre les paramètres en entrée d'une invocation de service et le temps de calcul de ce dernier, écartant par la même occasion leur effet sur le temps de transmission. Ainsi, en ne contrôlant que le temps de calcul et en re-calculant le temps de réponse, nous évitons la re-vérification d'une valeur de propriété sur les trois pouvant caractériser une activité, comme nous pouvons le voir en comparant les colonnes *Contrôle* correspondant au méthodes *"Modèle causal système"* et *"Modèle causal QoS"*, où l'on voit qu'il y a deux fois moins de contrôleurs pour la dernière méthode.

– **La plupart du temps, l'analyse causale de l'évolution reposant sur le modèle causal fonctionnel est aussi efficace que la re-vérification de l'évolution seule.** Cette affirmation repose sur le caractère particulier des évolutions que nous avons considérées. En effet, ici, la plupart des évolutions effectuées sont auto-contenues, dans le sens où elles n'influencent pas directement le flot de données du processus métier d'origine, mais se positionnent comme des branches indépendantes du flot. C'est le cas des évolutions des processus métiers **resolveACrisis**, **handleACrisis** et **handleAWorker**, où le nombre de re-vérifications de la méthode *"Re-vérification évolution"* est le même que pour la méthode *"Modèle causal système"*. Nous pouvons toutefois mettre en avant l'évolution *UnavailableIntResource* qui elle, n'est pas auto-contenue. Dans ce cas là, l'évolution a un effet causal sortant du cadre des éléments directement manipulés dans l'évolution. De ce fait, l'analyse causale dé-

signe un ensemble plus important d'éléments à re-vérifier, avec *huit* contrôleurs pour notre méthode, contre seulement deux pour la re-vérification de l'évolution.

- **La re-vérification seule des éléments de l'évolution ne suffit pas.** Comme établi dans le point précédent, cette méthode peut manquer lors de la re-vérification des éléments qui sont effectivement affectés de manière indirecte, menant à une erreur dans la prédiction de l'effet de l'évolution, et à un non-maintien de la QoS.
- **Les méthodes d'analyse causale facilitent le diagnostic d'une évolution en comparaison avec une re-vérification complète.** En effet, l'analyse causale (reposant sur un modèle causal fonctionnel ou enrichi) fournit de meilleurs résultats qu'une re-vérification complète du système, dans le sens où moins d'éléments sont à re-vérifier. Si nous considérons par exemple l'évolution *UnavailableIntResource* du processus métier *handleAMission*, nous obtenons un gain de soixante-treize pourcents pour la méthode utilisant le modèle causal fonctionnel, et un gain de quatre-vingt-deux pourcents pour la méthode utilisant le modèle causal enrichi avec les relations causales de QoS. Sur l'ensemble des évolutions présentées dans la Table 10.1, nous obtenons un gain moyen de soixante-huit pourcents pour la première méthode, et de soixante-dix-neuf pourcents pour la seconde. En complément, les méthodes d'analyse causale facilitent également la détection du problème, en limitant le domaine de recherche à l'ensemble des éléments qu'elle a identifiés, là où la re-vérification complète n'indique aucun élément potentiellement affecté.

10.6 Discussion

Les expériences que nous venons de présenter montrent que notre outil détermine de manière précise les activités et les valeurs de propriété affectées par l'évolution. Nous avons également comparé notre approche reposant sur un modèle causal fonctionnel enrichi des relations causales de QoS avec une approche pessimiste (re-vérification complète du système), optimiste (re-vérification des éléments de l'évolution), ainsi qu'avec un modèle causal dépourvues de relations causales de QoS. Cette comparaison nous a montré le gain tant sur le plan du nombre de re-vérifications à effectuer, que sur la justesse des re-vérifications comparée à la re-vérification simple de l'évolution. Nous avons montré que plus le modèle causal était riche, plus l'analyse causale était précise. Cependant, cette précision a un coût : il est ici nécessaire que l'expert en QoS fournisse des informations supplémentaires au cours de la définition d'une propriété. De plus, l'augmentation du niveau de détails du modèle causal mène à une augmentation de l'occupation en mémoire, chose qui doit être prise en considération dans le cas de systèmes composés de centaines d'activités.

Enfin, la précision de l'analyse de l'évolution repose sur la justesse de la description de la propriété de QoS. Si des relations causales n'étaient pas prises en compte, il est possible que l'ensemble des éléments affectés que l'analyse calcule diffère de la réalité. Nous ne pouvons que recommander de préférer un analyse et une description des propriétés incluant plus d'éléments que nécessaire, plutôt que de se trouver dans une situation où le modèle causal n'est pas suffisamment précis, et où l'analyse pourrait mener à manquer des valeurs de propriétés à re-vérifier.

10.7 Conclusion du chapitre

Nous venons de montrer à travers le cas d'étude du *Système de gestion de crises* comment l'utilisation de Blink et de Smile permettaient de réaliser différentes évolutions en

s'assurant du maintien de la qualité de service. Nous avons vu que l'utilisation de notre analyse de l'évolution reposant sur un modèle causal permettait d'être plus efficace en termes de re-vérifications qu'une approche effectuant une re-vérification complète. En ce sens, notre analyse permet d'abaisser les coûts de la phase d'analyse de l'évolution.

Nous avons également vu que notre approche, grâce à l'analyse causale, permettait de détecter les effets cachés d'une évolution, en considérant dans la phase de re-vérification des éléments du système qui n'étaient pas directement manipulés par l'évolution. Cela nous permet ainsi de détecter des cas où la méthode de re-vérification de l'évolution produirait un résultat erroné, en omettant des valeurs de propriété.

Au fil des différentes évolutions appliquées, nous avons étudié la progression de la taille du modèle causal. Ce dernier augmente de façon linéaire, nous permettant d'évaluer les limites du passage à l'échelle de Smile. Sur ce dernier point, les limitations pourraient toutefois être levées en découpant le modèle causal en plusieurs sous-modèles de causalité. Cette méthode sera étudiée dans de futurs travaux.

Quatrième partie

Conclusion

Chapitre 11

Conclusion et Travaux Futurs

Sommaire

11.1 Résumé des contributions 139
11.2 Perspectives à court terme 140
11.3 Perspectives à long terme 141
Bibliographie 145

Dans ce chapitre, nous dressons le bilan des différentes contributions présentées dans cette thèse. Après avoir montré en quoi ces contributions parviennent à répondre aux défis présentés dans le chapitre 1, nous élaborons un ensemble de perspectives à court terme permettant de lever certaines limitations de nos travaux, et de perspectives à long terme sur lesquelles les travaux de la thèse pourraient déboucher.

11.1 Résumé des contributions

Dans cette thèse, nous avons présenté un ensemble de quatre défis pour lesquels il était nécessaire de trouver une solution dans le but d'élaborer une réponse à la problématique du maintien de la qualité de service lors de l'évolution d'applications à base de services. Dans la suite de cette section, nous reprenons l'ensemble de nos contributions, afin de montrer en quoi elles répondent aux différents défis identifiés :

- Nous avons défini Blink, un cycle de développement pour l'évolution des logiciels, dans lequel un certain nombre d'acteurs sont définis. Parmi ces acteurs, nous retiendrons essentiellement l'architecte du système, l'architecte de l'évolution, et l'expert en Qualité de Service. Notre processus de développement itératif met au centre l'évolution et les différentes expertises nécessaires, en définissant un ensemble d'étapes pour lesquelles les interactions entre acteurs sont définies, permettant par exemple de partager les informations connues entre l'expert en qualité de service et l'architecte de l'évolution. Par cette contribution, nous avons répondu au **défi n° 1 (Collaboration entre acteurs)** : en identifiant les points d'interaction où un échange d'informations était nécessaire, nous avons défini les responsabilités de chaque acteur.
- Nous avons introduit successivement les rôles d'architecte du système et d'expert en QoS. Ces deux rôles possèdent une connaissance bien précise dans leur domaine, et nécessitent un support d'expression que nous avons fourni en définissant un méta-modèle propre à leurs expertises. Puis, en introduisant la notion de relation causale, nous avons proposé par le biais du modèle causal un médium essentiel pour exprimer les relations de dépendance entre les différents éléments du système et de la QoS. Ces dépendances permettent de rendre explicite les différentes interactions existant dans un logiciel. Cette contribution est notre réponse au **défi n°2 (Interactions au sein d'un logiciel)**.

- Dans le chapitre 8, nous avons défini un outil d'analyse de l'effet de l'évolution. Celui-ci, se basant sur la description de l'évolution et sur le modèle causal, permet de délimiter, en s'appuyant sur les relations causales établies, le sous-ensemble du système affecté de manière directe ou indirecte par l'évolution. De par cette délimitation, nous avons vu que le nombre de re-vérifications, qu'elles soient effectuées par analyse statique ou par contrôle à l'exécution, était minimisé par cette approche. Nous donnons ainsi une solution au **défi n°3 (Minimisation de la vérification)**, en réduisant le nombre d'éléments à re-vérifier.
- Enfin, par le biais de l'analyse causale de l'évolution, le sous-ensemble établi permet de restreindre les causes possibles d'une violation des contrats de QoS. En nous appuyant sur le sous-ensemble des éléments affectés, nous identifions un ensemble de causes racines ayant potentiellement causés cette violation. Cet élément est notre réponse au **défi n°4 (Identification de la cause de la violation d'un contrat)**.

La combinaison de l'ensemble de ces contributions permet de proposer une réponse à notre problématique de départ. En effet, en collectant l'ensemble des informations auprès des acteurs de l'équipe de développement, pour ensuite les utiliser dans le but d'établir le modèle causal du système, puis en utilisant ce modèle causal pour réduire le nombre de re-vérifications à effectuer et pour identifier la cause d'une potentielle violation de contrat, nous avons construit un mécanisme d'évolution s'assurant du maintien de la qualité de service d'un système à bases de processus métiers. Toutefois, si notre approche permet de détecter la violation de contrats de QoS, elle n'est cependant pas adaptée à la détermination de violation au niveau de certains types de contrats. Par exemple, une des contraintes que l'on peut retrouver sur des systèmes concerne la disponibilité du dit système, où l'on exprimera que le système n'a un temps de maintenance que de cinq minutes par an. Notre approche ne permet en aucun cas de prédire ce genre de situation, peu adaptée à la notion d'évolution et sans aucun lien avec le comportement du système.

11.2 Perspectives à court terme

Dans un futur proche, nous envisageons d'améliorer notre approche de différentes manières. Une première piste intéressante consiste à rechercher d'autres relations causales, pouvant potentiellement provenir d'autres supports. Par exemple, certaines propriétés telles que la confidentialité sont exprimées à un niveau de description différent comme par exemple la description de l'architecture de l'application. Ce genre de support pourrait être exploité afin de pouvoir enrichir le plus possible le modèle causal.

Il serait également intéressant de lever la limitation décrite dans la section précédente, en établissant une classification des différentes propriétés de QoS, afin de délimiter le périmètre de celles pour lesquelles notre approche serait profitable. En recherchant des caractéristiques particulières des propriétés, telles que la méthode de détermination où le concept du système qu'elle caractérise, nous pourrions établir quel type d'évolution pourrait affecter ou non cette propriété. Par exemple, un changement des besoins de l'utilisateur engendrant une modification du comportement du système affecte des propriétés telles que la taille des données échangées ou le temps de calcul, qui sont des propriétés très liées au comportement. En revanche, une propriété comme la disponibilité d'un service sera plus disposée à être affectée par un changement dans l'environnement du logiciel.

Nous avons vu dans cette thèse comment il était possible de déduire de manière automatique les règles causales en fonction d'une propriété de qualité de service. Toutefois, une de nos limitations réside dans l'aspect manuel de l'écriture des règles causales du point de vue du système. Afin d'améliorer l'automatisation de notre approche, il serait intéressant

d'établir à l'avance l'ensemble des causalités liées à un moteur d'exécution, ou encore mieux, de pouvoir les déduire de l'implémentation du dit moteur.

Enfin, afin d'améliorer notre canevas SMILE, un effort de développement reste à fournir pour améliorer l'intégration d'autres outils d'analyse, tels que Paladio [Koziolek 2006] ou encore SHARPE [Sahner 2012].

11.3 Perspectives à long terme

En termes de recherche à plus long terme, un certain nombre d'axes pourraient permettre d'améliorer la fiabilité de notre approche dans la construction d'un mécanisme d'évolution fiable en terme de QoS.

- **Auto-réparation de la violation de contrat :** nous avons vu que notre approche permettait de déterminer, en partant d'une évolution déjà écrite, si cette dernière ne violait pas un des contrats de QoS du système. L'objectif ici était de pouvoir dresser un diagnostic. En complément de ce diagnostic, proposer de manière automatique une solution à cette violation est un axe de recherche à explorer. Concrètement, il s'agirait d'une correction automatique de l'évolution appliquée, correction pouvant émaner de plusieurs sources. Dans un premier cas, les principes des architectures orientées services utilisent la QoS comme critère de comparaison entre différentes implémentations d'une même opération. Une solution simple consisterait donc à sélectionner dans un annuaire de services une autre implémentation pour laquelle la valeur de la propriété violée serait meilleure. Dans un second cas, une autre manière de corriger la violation du contrat pourrait s'appuyer sur l'application de patrons d'amélioration de la QoS : il s'agirait ici d'appliquer une deuxième évolution au système pour mettre en œuvre un mécanisme permettant d'améliorer la valeur de certaines propriétés. Par exemple, un patron d'amélioration consiste à migrer un ensemble de services sur une même machine, dans le but de réduire le temps de transmission. Enfin, dans une logique davantage contemplative, il s'agirait, par dualité avec l'application de patrons de performance, d'effectuer une analyse du système permettant de détecter cette fois-ci la présence d'anti-patrons de propriétés de QoS [Smith 2000]. En effectuant ce genre d'analyse, nous pourrions donner davantage d'informations à l'architecte de l'évolution, sur une cause de la violation de contrat.
- **Granularité du modèle causal :** dans ce document, nous avons vu comment établir le modèle causal d'un système, et comment l'enrichir avec les relations causales de QoS. Lors de cet enrichissement, certaines relations causales ont été raffinées en d'autres relations, permettant ainsi un ciblage plus précis des éléments du système impactés. On pensera notamment à l'augmentation de précision qui nous a permis de ne contrôler que le temps de calcul d'une activité, là où auparavant, nous aurions re-vérifié l'ensemble de ses valeurs de propriétés.

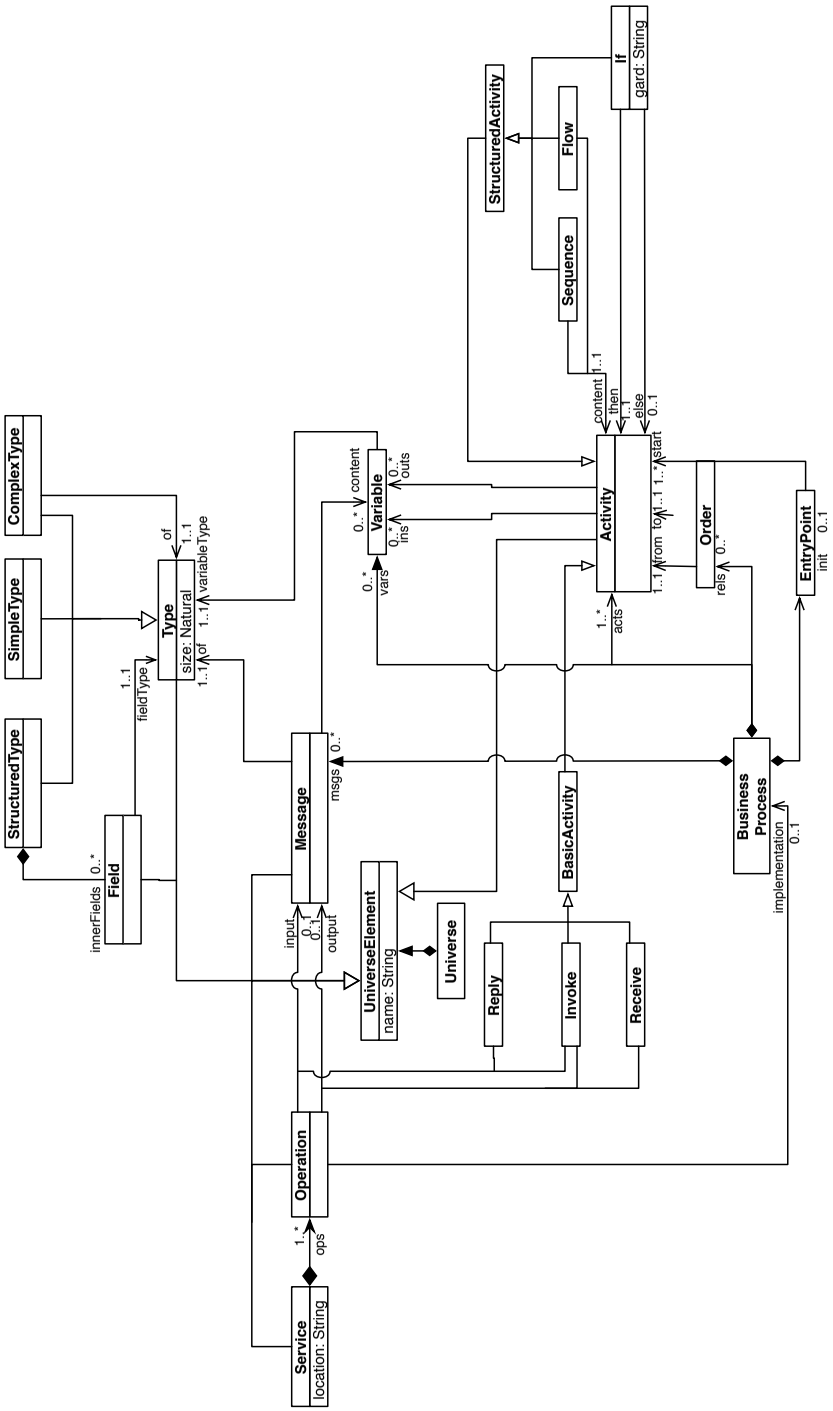
  De fait, selon la granularité de la modélisation choisie et le niveau sémantique des éléments du système exprimé, le niveau de précision des relations causales sera plus ou moins fin. La contrepartie d'une telle richesse réside dans l'augmentation de la taille du modèle causal, et dans la disponibilité d'informations trop détaillées, rendant plus complexe la recherche de la cause réelle. Il s'agit donc dans un premier temps d'explorer ce domaine afin de déterminer quel est le niveau de granularité le plus adapté pour analyser les effets d'une évolution. Dans un second temps, il s'agira également de voir si ce niveau est universel, quels que soient le système et la propriété étudiés. En effet, là où l'étude de certaines propriétés nécessite de représenter le comportement

d'un système au niveau des activités, d'autres propriétés de QoS, *e.g.*, la consommation mémoire, nécessitent d'avoir une granularité plus fine dans la modélisation du comportement du système.

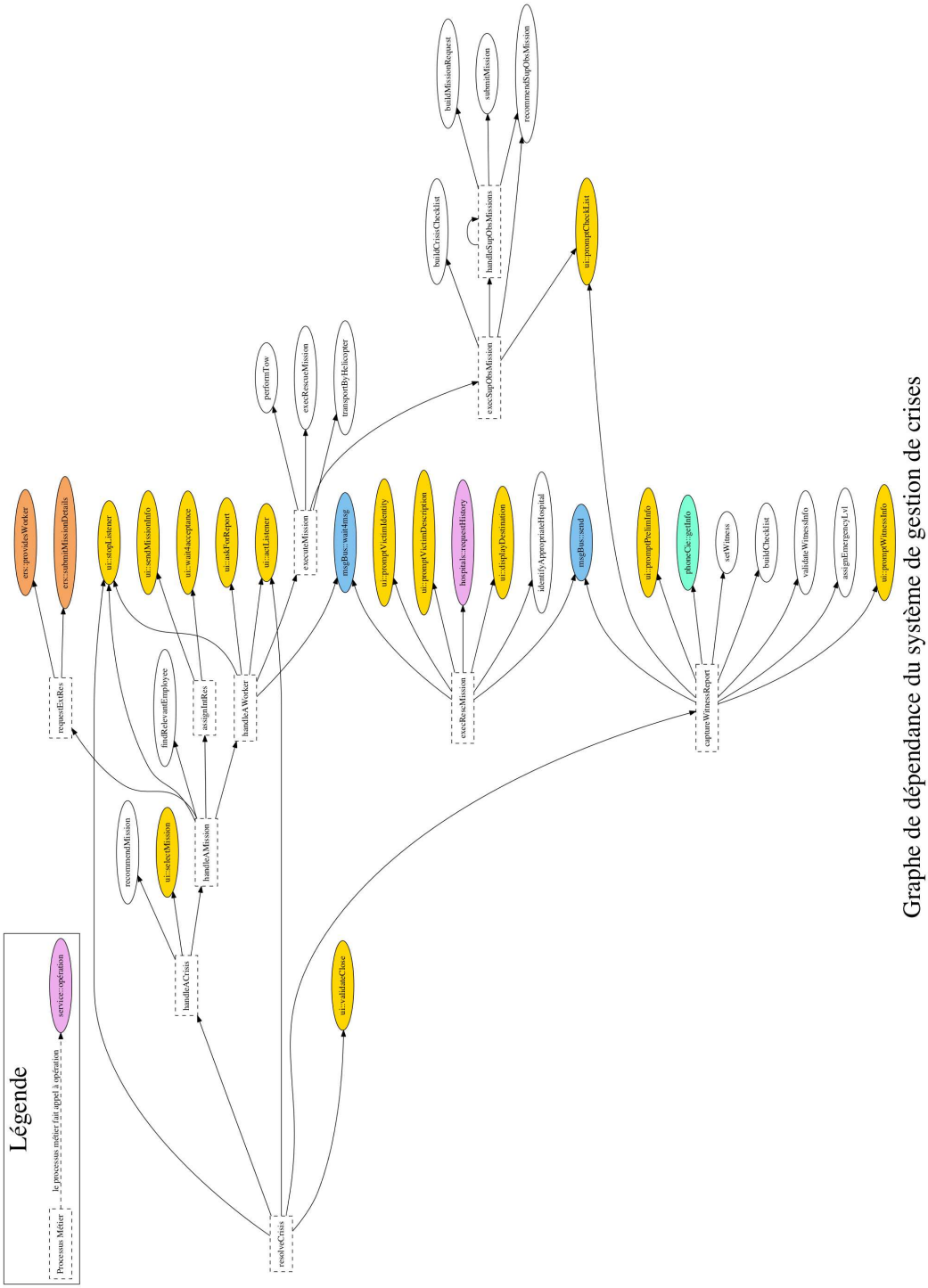
- **Vérification de la minimalité de l'ensemble des éléments affectés :** nous avons décrit dans ce document une méthode permettant de réduire le nombre d'éléments à vérifier lors d'une évolution. Toutefois, cet ensemble, bien que moins grand que l'ensemble des éléments d'un système, n'est pas garanti d'être le plus petit ensemble possible. En effet, dans notre approche, nous avons cherché avant tout à déterminer un ensemble tel qu'il soit aussi réduit que possible, mais pour lequel nous pouvions être sûr qu'il contenait les éléments affectés par l'évolution. Garantir que cet ensemble soit le plus petit ensemble possible est une question de recherche méritant d'être traitée.
- **Détermination empirique de relations de causalité :** tout au long du document, nous avons établi les causalités d'un système en nous basant sur les différentes descriptions disponibles de propriété et du système. Toutefois, une autre approche consiste à établir ces relations en observant les valeurs mesurées, en déterminant par l'observation quel élément du système influence quel autre élément. Pour cela, il s'agira par exemple d'utiliser les principes du processus de développement *Cleanroom* [Mills 1991], reposant sur une étude stricte de la qualité du logiciel selon des méthodes statistiques, afin de pouvoir étudier et corréler les influences des différents éléments du système. Cet axe pourra nécessiter cependant d'élaborer un environnement de test pour lequel des paramètres sur lesquels nous n'avons d'habitude pas la main (tel que la topologie et le comportement du réseau) seraient complètement sous contrôle.

Annexe

Annexe 1 : méta-modèle des systèmes



Annexe 2 : graphe de dépendance complet du système de gestion de crises



Bibliographie

[Arsanjani 2008] Ali Arsanjani, Shuvanker Ghosh, Abdul Allam, Tina Abdollah, Sella Ganapathy and Kerrie Holley. *SOMA : A method for developing service-oriented solutions*. IBM systems Journal, vol. 47, no. 3, pages 377–396, 2008. 27

[Bachmann 2002] Felix Bachmann, Len Bass, Paul Clements, David Garlan and James Ivers. *Documenting software architecture : Documenting interfaces*. Rapport technique, DTIC Document, 2002. 11

[Bass 2003] Len Bass, Paul Clements and Rick Kazman. Software architecture in practice. Addison-Wesley Professional, 2003. 74

[Beck 2001] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries*et al.* *The agile manifesto*, 2001. 26

[Beck 2004] Kent Beck and Cynthia Andres. Extreme programming explained : embrace change. Addison-Wesley Professional, 2004. 26

[Becker 2008] S. Becker, M. Trifu and R. Reussner. *Towards supporting evolution of service-oriented architectures through quality impact prediction*. In Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on, pages 77 –81, sept. 2008. 32, 35

[Bejaoui 2008] Rim Bejaoui. SOA : les impacts sur le cycle de vie du projet de développement logiciel. Master's thesis, Université du Québec à Montréal, 2008. 44

[Ben Halima 2008] R. Ben Halima, K. Guennoun, K. Drira and M. Jmaiel. *Non-intrusive QoS monitoring and analysis for self-healing web services*. In Applications of Digital Information and Web Technologies, 2008. ICADIWT 2008. First International Conference on the, pages 549 –554, aug. 2008. 31, 35

[Blanc 2009] Xavier Blanc, Alix Mougenot, Isabelle Mounier and Tom Mens. *Incremental Detection of Model Inconsistencies Based on Model Operations*. In Proceedings of the 21st International Conference on Advanced Information Systems Engineering, CAiSE'09, pages 32–46, Berlin, Heidelberg, 2009. Springer-Verlag. 86

[Boehm 1976] Barry W Boehm, John R Brown and Mlity Lipow. *Quantitative evaluation of software quality*. In Proceedings of the 2nd international conference on Software engineering, pages 592–605. IEEE Computer Society Press, 1976. 12

[Bohner 1996] Shawn A Bohner. *Software change impact analysis*. 1996. 31

[Breivold 2007] Hongyu Pei Breivold and Magnus Larsson. *Component-based and service-oriented software engineering : Key concepts and principles*. In Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on, pages 13–20. IEEE, 2007. 10

[Brownsword 2004] Lisa L Brownsword, David J Carney, David Fisher, Grace Lewis and Craig Meyers. *Current perspectives on interoperability*. Rapport technique, DTIC Document, 2004. 13

[Busch 1986] David A Busch and Mark W Paradies. *User's guide for Reactor Incident Root Cause Coding Tree*. Rapport technique, Savannah River Lab., Aiken, SC (USA), 1986. 30

[Canfora 2006] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, Francesco Perfetto and Maria Villani. *Service Composition (re)Binding Driven by Application–Specific QoS*. In Asit Dan and Winfried Lamersdorf, editeurs, Service-Oriented Computing – ICSOC 2006, volume 4294 of *Lecture Notes in Computer Science*, pages 141–152. Springer Berlin / Heidelberg, 2006. 10.1007/11948148_12. 16

[Cardoso 2004] Jorge Cardoso, Amit Sheth, John Miller, Jonathan Arnold and Krys Kochut. *Quality of Service for Workflows and Web Service Processes.* Web Semantics : Science, Services and Agents on the World Wide Web, vol. 1, no. 3, pages 281 – 308, 2004. 1, 16

[Cheng 2009] Betty Cheng, Rogrio de Lemos, Holger Giese, Paola Inverardi and Jeff et al. Magee. *Software Engineering for Self-Adaptive Systems : A Research Roadmap.* In Software Engineering for Self-Adaptive Systems. Springer Berlin / Heidelberg, 2009. 15

[Chinnici 2007] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman and Sanjiva Weerawarana. *Web services description language (wsdl) version 2.0 part 1 : Core language.* W3C recommendation, vol. 26, page 19, 2007. 11

[Chung 2009] Lawrence Chung and JulioCesarSampaio Prado Leite. *On Non-Functional Requirements in Software Engineering.* In AlexanderT. Borgida, VinayK. Chaudhri, Paolo Giorgini and EricS. Yu, editeurs, Conceptual Modeling : Foundations and Applications, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer Berlin Heidelberg, 2009. 74

[Cicchetti 2011] Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque and Séverine Sentilles. *Evolution management of extra-functional properties in component-based embedded systems.* In Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering, pages 93–102. ACM, 2011. 32, 35

[Clements 2003] Paul Clements, Rick Kazman and Mark Klein. Evaluating software architectures. 2003. 13

[Coates 2001] Tony Coates, Dan Connolly, Diana Dack, Leslie L Daigle, Ray Denenberg, Martin J Dürst, Paul Grosso, Sandro Hawke, Renato Iannella, Graham Klyne*et al.* *URIs, URLs, and URNs : Clarifications and Recommendations 1.0.* World Wide Web Consortium, 2001. 11

[Comes 2009] Diana Comes, Steffen Bleul, Thomas Weise and Kurt Geihs. *A flexible approach for business processes monitoring.* In Distributed Applications and Interoperable Systems, pages 116–128. Springer, 2009. 68

[Crnkovic 2005] Ivica Crnkovic, Magnus Larsson and Otto Preiss. *Concerning Predictability in Dependable Component-Based Systems : Classification of Quality Attributes.* In Dependable Component-Based Systems, volume 3549, pages 257–278. Springer, 2005. 1, 13, 74

[Dam 2010] Hoa Khanh Dam and A. Ghose. *Supporting Change Propagation in the Maintenance and Evolution of Service-Oriented Architectures.* In Software Engineering Conference (APSEC), 2010 17th Asia Pacific, pages 156 –165, 30 2010-dec. 3 2010. 32, 35

[De Pauw 2005] Wim De Pauw, Michelle Lei, Edward Pring, Lionel Villard, Matthew Arnold and John F Morar. *Web services navigator : Visualizing the execution of web services.* IBM Systems Journal, vol. 44, no. 4, pages 821–845, 2005. 15

[Delerce-Mauris 2009] C Delerce-Mauris, L Palacin, S Martarello, S Mosser and M Blay-Fornarino. *Plateforme SEDUITE : une approche SOA de la diffusion d'informations.* University of Nice, I3S CNRS, Sophia Antipolis, France Research Report, Feb, 2009. 37

[Dumas 2010] Marlon Dumas, Luciano García-Bañuelos, Artem Polyvyanyy, Yong Yang and Liang Zhang. *Aggregate Quality of Service Computation for Composite Services.*

In Paul Maglio, Mathias Weske, Jian Yang and Marcelo Fantinato, editeurs, Service-Oriented Computing, volume 6470 of *Lecture Notes in Computer Science*, pages 213–227. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-17358-5_15. 16

[Elbaum 2001] Sebastian Elbaum, David Gable and Gregg Rothermel. *The Impact of Software Evolution on Code Coverage Information*. In In Intl. Conference on Software Maintenance, pages 170–179, 2001. 31, 35

[Fakhfakh 2012] Nabil Fakhfakh. *Une approche orientée utilisateur pour la supervision des orchestrations de services*. PhD thesis, Université de Grenoble, 2012. 13

[Ferrara 2004] Andrea Ferrara. *Web services : a process algebra approach*. In Proceedings of the 2nd international conference on Service oriented computing, ICSOC '04, pages 242–251, New York, NY, USA, 2004. ACM. 14

[Gatti 2011] Stéphanie Gatti, Emilie Balland and Charles Consel. *A Step-wise Approach for Integrating QoS throughout Software Development*. In FASE'11 : Proceedings of the 14th European Conference on Fundamental Approaches to Software Engineering, pages 217–231, Sarrebruck, Germany, March 2011. 28, 29

[Ghezzi 2007] Carlo Ghezzi and Sam Guinea. *Run-Time Monitoring in Service-Oriented Architectures*. In Luciano Baresi and ElisabettaDi Nitto, editeurs, Test and Analysis of Web Services, pages 237–264. Springer Berlin Heidelberg, 2007. 15

[Giannakopoulou 2001] Dimitra Giannakopoulou and Klaus Havelund. *Automata-based verification of temporal properties on running programs*. In Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on, pages 412–416. IEEE, 2001. 14

[Godfrey 2008] M.W. Godfrey and D.M. German. *The past, present, and future of software evolution*. In Frontiers of Software Maintenance, 2008. FoSM 2008., pages 129 –138, 28 2008-oct. 4 2008. 18

[Haas 2004] Hugo Haas and Allen Brown. *Web services glossary*. W3C Working Group Note (11 February 2004), 2004. 9

[Havelund 2002] Klaus Havelund and Grigore Roşu. *Synthesizing monitors for safety properties*. In Tools and Algorithms for the Construction and Analysis of Systems, pages 342–356. Springer, 2002. 14

[Huhns 2005] M.N. Huhns and M.P. Singh. *Service-oriented computing : key concepts and principles*. Internet Computing, IEEE, vol. 9, no. 1, pages 75 – 81, jan-feb 2005. 10

[IABG 1997] IABG. *The Development Standards for IT Systems of the Federal Republic of Germany. The V-Model*, June 1997. http ://v-modell.iabg.de. 26

[IEEE 2008] IEEE. *Systems and software engineering – Software life cycle processes - Redline*. ISO/IEC 12207 :2008(E) IEEE Std 12207-2008 - Redline, pages 1–195, Jan 2008. 26, 43

[ISO 2001] ISO. *Software engineering – Product quality – Part 1 : Quality model*. Rapport technique ISO/IEC 9126-1, International Organization for Standardization, 2001. 12

[ISO 2004] ISO ISO and IEC FCD. *25000, Software Engineering–Software Product Quality Requirements and Evaluation (SQuaRE)-Guide to SQuaRE*. Geneva, International Organization for Standardization, 2004. 12

[Jaeger 2004] Michael C. Jaeger, Gregor Rojec-Goldmann and Gero Muhl. *QoS Aggregation for Web Service Composition using Workflow Patterns*. Enterprise Distributed Object Computing Conference, IEEE International, vol. 0, pages 149–159, 2004. 16

[Kajko-Mattsson 2007] M. Kajko-Mattsson, G.A. Lewis and D.B. Smith. *A Framework for Roles for Development, Evolution and Maintenance of SOA-Based Systems.* In Systems Development in SOA Environments, 2007. SDSOA '07 : ICSE Workshops 2007. International Workshop on, pages 7–7, 2007. 44

[Keller 2003] Alexander Keller and Heiko Ludwig. *The WSLA Framework : Specifying and Monitoring Service Level Agreements for Web Services.* Journal of Network and Systems Management, vol. 11, pages 57–81, 2003. 10.1023/A :1022445108617. 17, 116

[Kienzle 2010] Jörg Kienzle, Nicolas Guelfi and Sadaf Mustafiz. *Crisis management systems : a case study for aspect-oriented modeling.* Transactions on aspect-oriented software development VII, pages 1–22, 2010. 122, 124

[Kijas 2013] Szymon Kijas and Andrzej Zalewski. *Towards Evolution Methodology for Service-Oriented Systems.* In Wojciech Zamojski, Jacek Mazurkiewicz, Jaroslaw Sugier, Tomasz Walkowiak and Janusz Kacprzyk, editeurs, New Results in Dependability and Computer Systems, volume 224 of *Advances in Intelligent Systems and Computing*, pages 255–273. Springer International Publishing, 2013. 28, 29, 44

[Kim 2010] T. Kim, C.K. Chang and S. Mitra. *Design of Service-Oriented Systems Using SODA.* Services Computing, IEEE Transactions on, vol. 3, no. 3, pages 236–249, July 2010. 28, 29

[Knight 2002] John C Knight. *Safety critical systems : challenges and directions.* In Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on, pages 547–550. IEEE, 2002. 123

[Koziolek 2006] Heiko Koziolek and Jens Happe. *A QoS Driven Development Process Model for Component-Based Software Systems.* In Component-Based Software Engineering, volume 4063, pages 336–343. Springer Berlin / Heidelberg, 2006. 28, 29, 141

[Kruchten 2004] Philippe Kruchten. The rational unified process : an introduction. Addison-Wesley Professional, 2004. 26

[Kuebler 2001] Dietmar Kuebler and Wolfgang Eibach. *Metering and accounting for Web Services.* A dynamic ebusiness solution : IBM Developer Works, 2001. 10

[Larman 2003] Craig Larman and Victor R. Basili. *Iterative and Incremental Development : A Brief History.* Computer, vol. 36, no. 6, pages 47–56, 2003. 26

[Lehman 1969] Meir M Lehman. *The programming process.* internal IBM report, 1969. 17

[Lehman 1985] M. M. Lehman and L. A. Belady. Program evolution : processes of software change. Academic Press Professional, Inc., San Diego, CA, USA, 1985. 17

[Lehman 1997] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry and Wladyslaw M Turski. *Metrics and laws of software evolution-the nineties view.* In Software Metrics Symposium, 1997. Proceedings., Fourth International, pages 20–32. IEEE, 1997. 18

[Lehman 2006] Meir Lehman and Juan C Fernández-Ramil. *Software evolution.* Software evolution and feedback : Theory and practice, vol. 1, pages 7–40, 2006. 18

[Lelli 2007] F. Lelli, G. Maron and S. Orlando. *Client Side Estimation of a Remote Service Execution.* In Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS'07., pages 295–302. IEEE, 2007. 13

[Lewis 2010] Grace A Lewis, Dennis B Smith and Kostas Kontogiannis. *A research agenda for service-oriented architecture (SOA) : Maintenance and evolution of service-oriented systems.* 2010. 28

[Ludwig 2003] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P King and Richard Franck. *Web service level agreement (WSLA) language specification*. IBM Corporation, pages 815–824, 2003. 11

[Marino 2009] Jim Marino and Michael Rowley. Understanding sca (service component architecture). Pearson Education, 2009. 111

[Massuthe 2005] Peter Massuthe, Wolfgang Reisig and Karsten Schmidt. *An operating guideline approach to the SOA*. 2005. 14

[Menasce 2002] D.A. Menasce. *QoS issues in Web services*. Internet Computing, IEEE, vol. 6, no. 6, pages 72 – 75, nov/dec 2002. 11

[Mens 2003] Tom Mens, Jim Buckley, Matthias Zenger and Awais Rashid. *Towards a taxonomy of software evolution*. In Proc. Workshop on Unanticipated Software Evolution, pages 1–18, 2003. 18

[Mernik 2005] Marjan Mernik, Jan Heering and Anthony M Sloane. *When and how to develop domain-specific languages*. ACM computing surveys (CSUR), vol. 37, no. 4, pages 316–344, 2005. 75

[Mills 1991] Harlan D Mills and Richard C Linger. Cleanroom software engineering : Developing software under statistical quality control. Wiley Online Library, 1991. 142

[Milner 1999] Robin Milner. Communicating and mobile systems : the pi calculus. Cambridge university press, 1999. 14

[Moe 2002] Johan Moe and David A Carr. *Using execution trace data to improve distributed systems*. Software : Practice and Experience, vol. 32, no. 9, pages 889–906, 2002. 30

[Mosser 2013] Sébastien Mosser and Mireille Blay-Fornarino. *ADORE, a Logical Metamodel Supporting Business Process Evolution*. Science of Computer Programming (SCP), vol. 78, no. 8, pages 1035–1054, 2013. 86

[Mukherjee 2008] Debdoot Mukherjee, Pankaj Jalote and Mangala Gowri Nanda. *Determining QoS of WS-BPEL Compositions*. In ICSOC 2008, volume 5364, pages 378–393. Springer/ Heidelberg, 2008. 16, 76

[Newcomer 2002] Eric Newcomer. Understanding web services : Xml, wqsdl, soap and uddi. Addison-Wesley Professional, 2002. 14

[OASIS 2006] OASIS. *Reference Model TC," Reference model for Service Oriented Architecture 1.0*. Rapport technique, OASIS, 2006. 54, 57

[OASIS 2007] OASIS. *Web Services business process execution language version 2.0*. Rapport technique, 2007. 38

[OASIS 2010] OASIS. *Web Services Quality Factors Version 1.0*. http ://goo.gl/nm8BH, july 2010. 14

[O'Brien 2007] Liam O'Brien, Paulo Merson and Len Bass. *Quality attributes for service-oriented architectures*. In Proceedings of the international Workshop on Systems Development in SOA Environments, page 3. IEEE Computer Society, 2007. 13

[OMG 2003] OMG. *MDA Guide*, 2003. 107

[OMG 2009a] OMG. *Production Rule Representation (PRR) Specification*, 2009. Version 1.0. 62

[OMG 2009b] OMG. *UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems*, 2009. 14

[Oriol 2008] Marc Oriol, Jordi Marco, Xavier Franch and David Ameller. *Monitoring adaptable soa-systems using salmon*. In Workshop on Service Monitoring, Adaptation and Beyond, pages 19–28, 2008. 17

[Papazoglou 2003] M.P. Papazoglou. *Service-oriented computing : concepts, characteristics and directions*. In Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on, pages 3 – 12, dec. 2003. 10

[Papazoglou 2006] Michael P. Papazoglou and Willem Jan Van Den Heuvel. *Service Oriented Design and Development Methodology*. Int. J. Web Eng. Technol., vol. 2, no. 4, pages 412–442, 2006. 27, 29, 44

[Papazoglou 2007] M Papazoglou, Paolo Traverso, Schahram Dustdar and Frank Leymann. *Service-oriented computing : State of the art and research directions*. Ieee Computer, vol. 40, no. 11, pages 64–71, 2007. 10

[Papazoglou 2011] M.P. Papazoglou, V. Andrikopoulos and S. Benbernou. *Managing Evolving Services*. Software, IEEE, vol. 28, no. 3, pages 49–55, May 2011. 30

[Pearl 2000] J. Pearl. Causality : models, reasoning and inference, volume 29. Cambridge Univ Press, 2000. 29, 59, 92

[Peltz 2003] Chris Peltz. *Web Services Orchestration and Choreography*. Computer, vol. 36, pages 46–52, 2003. 12

[Perez-Palacin 2014] Diego Perez-Palacin, Raffaela Mirandola and José Merseguer. *On the relationships between QoS and software adaptability at the architectural level*. Journal of Systems and Software, vol. 87, no. 0, pages 1 – 17, 2014. 74

[Pnueli 1977] Amir Pnueli. *The temporal logic of programs*. In Foundations of Computer Science, 1977., 18th Annual Symposium on, pages 46–57. IEEE, 1977. 14

[Ramollari 2007] Ervin Ramollari, Dimitris Dranidis and Anthony JH Simons. *A survey of service oriented development methodologies*. In The 2nd European Young Researchers Workshop on Service Oriented Computing, page 75, 2007. 27

[Ravichandar 2008] Ramya Ravichandar, Nanjangud C. Narendra, Karthikeyan Ponnalagu and Dipayan Gangopadhyay. *Morpheus : Semantics-based Incremental Change Propagation in SOA-based Solutions*. In IEEE SCC (1), pages 193–201, 2008. 31, 32, 35

[Reisig 1998] Wolfgang Reisig and Grzegorz Rozenberg. Lectures on petri nets i : basic models : advances in petri nets, volume 149. Springer, 1998. 14

[Rooney 2004] James J Rooney and Lee N Vanden Heuvel. *Root cause analysis for beginners*. Quality progress, vol. 37, no. 7, pages 45–56, 2004. 30, 59, 91

[Rosenberg 2006] Florian Rosenberg, Christian Platzer and Schahram Dustdar. *Bootstrapping Performance and Dependability Attributes ofWeb Services*. In Proceedings of the IEEE International Conference on Web Services, pages 205–212, Washington, DC, USA, 2006. IEEE Computer Society. 13

[Rottger 2003] Simone Rottger and Steffen Zschaler. *CQML+ : enhancements to CQML*. In In Proceedings of the 1st International Workshop on Quality of Service in ComponentBased Software Engineering, pages 43–56. Cepadues-Editions, 2003. 14, 32, 35, 75, 116

[Royce 1970] Winston W Royce. *Managing the development of large software systems*. In proceedings of IEEE WESCON, volume 26. Los Angeles, 1970. 26

[Sahner 2012] Robin A. Sahner, Kishor Trivedi and Antonio Puliafito. Performance and reliability analysis of computer systems : An example-based approach using the sharpe software package. Springer Publishing Company, Incorporated, 2012. 141

[Schwaber 2004] Ken Schwaber. Agile project management with scrum. O'Reilly Media, Inc., 2004. 26

[Seinturier 2012] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni and Jean-Bernard Stefani. *A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures*. Software : Practice and Experience, vol. 42, no. 5, pages 559–583, 2012. 127

[Shahrbanoo 2012] Majlesi Shahrbanoo, Mehrpour Ali and Mohsenzadeh Mehran. *An Approach for agile SOA development using agile principals*. arXiv preprint arXiv :1204.0368, 2012. 27

[Singh 2006] Munindar P Singh and Michael N Huhns. Service-oriented computing : semantics, processes, agents. Wiley. com, 2006. 52

[Smith 2000] Connie U Smith and Lloyd G Williams. *Software performance antipatterns*. In Workshop on Software and Performance, pages 127–136, 2000. 141

[Soffer 2005] Pnina Soffer. *Scope analysis : identifying the impact of changes in business process models*. Software Process : Improvement and Practice, vol. 10, no. 4, pages 393–402, 2005. 32, 35

[Spohn 2001] Wolfgang Spohn. *Bayesian nets are all there is to causal dependence*. 2001. 30

[Steinberg 2008] Dave Steinberg, Frank Budinsky, Ed Merks and Marcelo Paternostro. Emf : eclipse modeling framework. Pearson Education, 2008. 107

[Storey 1996] Neil R. Storey. Safety critical computer systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. 123

[Tague 2004] Nancy R Tague. *Seven basic quality tools*. The Quality Toolbox. Milwaukee, Wisconsin : American Society for Quality, page 15, 2004. 31

[Tarr 1999] Peri Tarr, Harold Ossher, William Harrison and Stanley M. Sutton Jr. *N Degrees of Separation : Multi-dimensional Separation of Concerns*. In Proceedings of the 21st International Conference on Software Engineering, ICSE99, pages 107–119, New York, NY, USA, 1999. ACM. 43

[Taskforce 2011] UML Revision Taskforce. *OMG UML Specification v. 2.4.1*. Object Management Group, 2011. 44

[Turing 1936] Alan Mathison Turing. *On computable numbers, with an application to the Entscheidungsproblem*. J. of Math, vol. 58, pages 345–363, 1936. 58

[Ucla 2001] Algirdas Avizienis Ucla, Algirdas Avizienis, Jean claude Laprie and Brian Randell. *Fundamental Concepts of Dependability*, 2001. 13

[Van der Aalst 2008] Wil MP Van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat and Eric Verbeek. *Conformance checking of service behavior*. ACM Transactions on Internet Technology (TOIT), vol. 8, no. 3, page 13, 2008. 14

[Wetzstein 2009] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Schahram Dustdar and Frank Leymann. *Monitoring and Analyzing Influential Factors of Business Process Performance*. Enterprise Distributed Object Computing Conference, IEEE International, vol. 0, pages 141–150, 2009. 15

[Xiao 2007] Hua Xiao, Jin Guo and Ying Zou. *Supporting Change Impact Analysis for Service Oriented Business Applications*. In Proceedings of the International Workshop on Systems Development in SOA Environments, SDSOA '07, pages 6–, Washington, DC, USA, 2007. IEEE Computer Society. 31

[Zimmermann 2006] Olaf Zimmermann, Jana Koehler and Frank Leymann. *The role of architectural decisions in model-driven SOA construction*. In the 21st Int. Conf. on

Object-Oriented Programming, Systems, Languages, and Applications,, Best Practices and Methodologies in Service-Oriented Architectures, Portland, pages 425 – 432. Citeseer, 2006. 44