HAL
open science

Flexible Quality of Service Management of Web Services Orchestrations

Ajay Kattepur

To cite this version:

Ajay Kattepur. Flexible Quality of Service Management of Web Services Orchestrations. Web. Université Rennes 1, 2012. English. NNT : . tel-00756048v2

HAL Id: tel-00756048

https://theses.hal.science/tel-00756048v2

Submitted on 23 Nov 2012

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

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

N° d'ordre : 4535 **ANNÉE 2012**

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Ajay KATTEPUR

Préparée à l'unité de recherche IRISA-INRIA : UMR 6074
DistribCom, Institut de recherche en informatique et systèmes aléatoires
ISTIC - UFR Informatique et électronique

Gestion Flexible de la Qualité de Service dans les Orchestrations de Services Web.

Flexible Quality of Service Management of Web Services Orchestrations.

Thèse soutenue à Rennes le 8 Novembre 2012

devant le jury composé de :

Jean-Marc JEZEQUEL
Professeur Université de Rennes 1 / *Président*
William R. COOK
Professeur Associé Université de Texas à Austin / *Rapporteur*
Danilo ARDAGNA
Professeur Assistant Polytechnique de Milan / *Rapporteur*
Valérie ISSARNY
Directrice de Recherche, INRIA / *Examinateur*
Fayçal BOUJEMAA
IT Technology Strategist, Orange / *Examinateur*
Albert BENVENISTE
Directeur de Recherche, INRIA / D*irecteur de thèse*
Claude JARD
Professeur, Université de Nantes / *Co-directeur de thèse*

“ In our highly complex organic state, we advanced organisms respond to our environment with an invention of many marvelous analogues. We invent earth and heavens, trees, stones and oceans, gods, music, arts, language, philosophy, engineering, civilization and science. We call these analogues reality. And they are reality. We mesmerize our children in the name of truth into knowing that they are reality. We throw anyone who does not accept these analogues into an insane asylum. But that which causes us to invent the analogues is Quality. Quality is the continuing stimulus which our environment puts upon us to create the world in which we live. All of it. Every last bit of it. ”

– Robert M. Pirsig
Zen and the Art of Motorcycle Maintenance:
An Inquiry into Values

Acknowledgments

This thesis has been influenced by many people, chief of which have been my supervisors *Albert Benveniste* and *Claude Jard*. I am grateful for the extremely conducive and encouraging atmosphere they provided, in order to conduct this research work. They have tolerated and answered my, sometimes naive, queries relating to all aspects of computer science and mathematics. Their passion for research has motivated me to strive for excellence in my work as well.

Further thanks go out to members of the thesis examining committee. Rapporteurs *William Cook* and *Danilo Ardagna* provided detailed feedback on my work and took time off to attend my defense. Examinateurs *Jean-Marc Jezequel*, *Valérie Issarny* and *Fayçal Boujemaa* were kind enough to attend my defense and pose interesting and thought provoking questions.

Working on this thesis has also put me in touch with collaborators: *John Thywissen*, *Sidney Rosario*, *Sagar Sen* and *Benoit Baudry* have contributed to various aspects of this work. My thanks to them for providing guidance in topics that were new and difficult for me. Special thanks to *Carole Hounkonnou* who helped proofread the introduction in French.

I would like to thank all my friends at DistribCom, INRIA and Rennes, in general, for making these years of PhD study a memorable one. I have enjoyed the hours of discussions and banter along with the cricket sessions. My gratitude to *Nambi*, *Srinivas* and *Yogesh* for helping me with arrangements for the "pot de thèse".

Finally, love to my parents and family for standing by me throughout the course of my study.

"We were but stones; Your light made us stars"
– *Pearl Jam*
Light Years

Contents

I Gestion Flexible de la Qualité de Service dans les Orchestrations de Services Web. 7

1 Résumé en Français 9
1.1 Technologies de Services Web . . . 11
1.2 Composition de Services Web . . . 13
1.3 Langages pour Spécifier des Orchestrations . . . 16
1.4 Qualité de Service . . . 17
1.4.1 QoS dans les services Web . . . 17
1.4.2 Composition de QoS . . . 18
1.4.3 Monotonie dans les Orchestrations . . . 20
1.4.4 Service Level Agreements . . . 20
1.4.4.1 Négociation . . . 21
1.4.4.2 Surveillance . . . 22
1.4.5 Optimisation dépendante de la QoS . . . 23
1.5 Organisation de la thèse . . . 25

II Flexible Quality of Service Management of Web Services Orchestrations. 35

2 Introduction 37
2.1 Web Services Technologies . . . 38
2.1.1 XML: Extensible mark-up language . . . 39
2.1.2 UDDI: Universal Description, Discovery and Integration . . . 39
2.1.3 WSDL: Web Services Description Language . . . 41
2.1.4 SOAP . . . 42
2.1.5 REST: Representational State Transfer . . . 43
2.2 Web Service Composition . . . 43
2.2.1 Workflow Management . . . 44
2.2.2 Orchestrations . . . 44
2.2.3 Formal Models for Service Orchestrations . . . 45
2.2.3.1 Statecharts . . . 45
2.2.3.2 Petri Nets . . . 45
2.2.3.3 π-calculus . . . 47
2.2.3.4 Composition Models . . . 47
2.3 Languages to Specify Orchestrations . . . 48
2.3.1 BPEL: Business Process Execution language . . . 48
2.3.2 Orc . . . 49
2.3.2.1 Sites . . . 49
2.3.2.2 Combinators . . . 50
2.3.2.3 Values, Definitions and Time . . . 51
2.3.2.4 Semantics . . . 52

2.3.2.5 Contributions . . . 53
2.4 Quality of Service . . . 54
2.4.1 QoS in Web Services . . . 54
2.4.2 QoS Composition . . . 56
2.4.2.1 Contributions . . . 57
2.4.3 Monotonicity in Orchestrations . . . 58
2.4.3.1 Contributions . . . 58
2.4.4 Service Level Agreements . . . 59
2.4.4.1 Negotiation . . . 60
2.4.4.2 Monitoring . . . 61
2.4.4.3 Contributions . . . 64
2.4.5 QoS dependent Optimization . . . 65
2.4.5.1 Contributions . . . 66
2.5 Thesis Organization . . . 68
2.5.1 Chapter 3: QoS-Aware Management of Monotonic Service Orchestrations . . . 70
2.5.2 Chapter 4: Leveraging Causality for QoS Tracking in Service Oriented Systems . . . 71
2.5.3 Chapter 5: Variability Modeling and QoS Analysis of Web Services Orchestrations . . . 72
2.5.4 Chapter 6: Pairwise Testing of Dynamic Composite Services . . . 73
2.5.5 Chapter 7: Optimizing Decisions in Web Services Orchestrations 74
2.5.6 Chapter 8: Importance Sampling/Splitting of Probabilistic Contracts in Web Services . . . 75
2.5.7 Chapter 9: Negotiation Strategies for Probabilistic Contracts in Web Services Orchestrations . . . 76
2.6 Future Work . . . 77

3 QoS-Aware Management of Monotonic Service Orchestrations 79
3.1 Introduction . . . 80
3.1.1 Running Example . . . 80
3.1.2 Key Issues . . . 82
3.1.2.1 Monotonicity and Consequences for Management . . . 82
3.1.2.2 Handling Probabilistic QoS . . . 84
3.1.3 Our Contribution . . . 85
3.1.3.1 An Abstract Algebraic Framework for QoS composition 85
3.1.3.2 A Careful Handling of Monotonicity . . . 85
3.1.3.3 Support for Separation of Concerns . . . 86
3.1.3.4 A Methodology: Managing QoS by Contracts . . . 86
3.2 Related Work . . . 87
3.3 QoS Calculus . . . 89
3.3.1 An Informal Introduction to the QoS Calculus . . . 89
3.3.2 Some Examples of QoS Domains . . . 91
3.3.3 Formalizing the QoS Calculus . . . 92
3.4 A Theory of QoS for Workflows . . . 93
3.4.1 Petri Nets, Occurrence Nets, and Orchestration Nets . . . 94
3.4.2 OrchNets: Definition and QoS Semantics, Application to QoS Composition . . . 96
3.4.3 Monotonicity: Results . . . 98
3.4.4 Probabilistic OrchNets . . . 100
3.4.5 Ensuring Monotonicity . . . 100
3.5 Summary of the Theory for Practical Use . . . 101

3.6 Implementing Our Approach . . . 102
3.6.1 Weaving QoS in Orchestrations . . . 102
3.6.2 Upgrading Orc for QoS . . . 105
3.6.3 The TravelAgent2/3 Example in Orc . . . 107
3.7 Evaluation of Our Approach . . . 107
3.8 Conclusion . . . 112

4 Leveraging Causality for QoS Tracking in Service Oriented Systems 115
4.1 Introduction . . . 116
4.2 Orc syntax . . . 117
4.3 Causality . . . 118
4.3.1 The algebra of causality . . . 119
4.3.2 Transformation rules . . . 120
4.3.3 Orc with Causality: Examples . . . 121
4.4 Causality and QoS Tracking . . . 121
4.4.1 QoS domain . . . 122
4.4.1.1 The special competition operator . . . 122
4.4.2 Composite QoS, no ambient metrics involved . . . 124
4.4.3 Composite QoS, with ambient metrics involved . . . 125
4.4.4 Extending Orc for QoS . . . 125
4.4.5 Enhancing the algebra of causality to support QoS: first attempt 125
4.4.6 Enhancing the algebra of causality to support QoS: the right solution . . . 126
4.4.7 The rules . . . 127
4.4.8 Orc with Causality and QoS: Examples . . . 127
4.5 Related work . . . 127
4.6 Conclusions . . . 128

5 Variability Modeling and QoS Analysis of Web Services Orchestrations 131
5.1 Introduction . . . 132
5.2 Foundations . . . 133
5.2.1 Modeling Variability in Composite Services . . . 133
5.2.2 Service Orchestration using Orc . . . 134
5.2.3 Configuration Generation from Feature Diagram . . . 135
5.2.4 QoS Aspects of the Orchestration . . . 135
5.3 Methodology . . . 136
5.4 Crisis Management System Case Study . . . 137
5.4.1 Feature Diagram of CMS . . . 137
5.4.2 Service Orchestrations in CMS . . . 138
5.5 Experiments . . . 140
5.5.1 Simulation of QoS Distributions . . . 140
5.5.2 Generating a sample of configurations for CMS . . . 140
5.5.3 Evaluating QoS of a Composite Service . . . 141
5.5.4 Evaluating the Pairwise Sampling Technique . . . 143
5.6 Related Work . . . 145
5.7 Conclusion and Perspectives . . . 147

6 Pairwise Testing of Dynamic Composite Services 149
6.1 Introduction . . . 150
6.2 Foundations . . . 151
6.2.1 Feature Diagrams . . . 151
6.2.2 Service Orchestrations using Orc . . . 152

6.2.3 Feature Diagrams with Orchestrations . . . 153
6.2.4 Combinatorial Interaction Testing . . . 153
6.2.5 QoS Aspects of the Orchestration . . . 155
6.3 Methodology . . . 156
6.4 Case Studies . . . 156
6.4.1 Car Crash Crisis Management System . . . 157
6.4.2 eHealth Management System . . . 158
6.5 Experiments . . . 161
6.5.1 Evaluating QoS of the Car Crash Crisis Management System . . 161
6.5.2 Evaluating QoS of the eHealth System . . . 162
6.5.3 Comparison with Random Sampling . . . 163
6.5.4 Consistency of Pairwise Samples . . . 165
6.5.5 Perspectives due to Analysis . . . 166
6.5.6 Threats to Validity . . . 167
6.6 Related Work . . . 167
6.7 Conclusion . . . 168

7 Optimizing Decisions in Web Services Orchestrations 171
7.1 Introduction . . . 172
7.2 Fundamentals . . . 173
7.2.1 Optimization models . . . 173
7.2.2 QoS in Web Services . . . 173
7.2.3 Analytic Hierarchy Process . . . 174
7.3 Methodology . . . 175
7.4 Formulating Optimization Problems . . . 176
7.5 Optimization Routines in Orc . . . 179
7.5.1 QOrc: Upgrading Orc for QoS management . . . 180
7.5.2 Interfacing QOrc to Optimization Services . . . 181
7.6 Optimal Decision Results . . . 182
7.7 Related Work . . . 185
7.8 Conclusion . . . 186

8 Importance Sampling/Splitting of Probabilistic Contracts in Web Services 187
8.1 Introduction . . . 188
8.2 Foundations . . . 188
8.2.1 QoS in Web Services . . . 188
8.2.2 Probabilistic Contracts . . . 189
8.2.3 Orc . . . 190
8.3 Rare Event Simulation Techniques . . . 190
8.3.1 Importance Sampling . . . 191
8.3.2 Importance Splitting . . . 192
8.4 Dell Supply Chain . . . 193
8.4.1 Contract Composition . . . 195
8.4.2 Forecasting . . . 197
8.5 Upgrading WSLA Specifications . . . 198
8.6 Related Work . . . 200
8.7 Conclusion . . . 200

9 Negotiation Strategies for Probabilistic Contracts in Web Services Orchestrations **201**
9.1 Foundations . . . 203
9.1.1 Web services' QoS . . . 203
9.1.2 Probabilistic Contracts . . . 203
9.1.3 Contract Negotiation . . . 204
9.1.4 Orc . . . 205
9.2 Optimization Formulation . . . 205
9.2.1 Stochastic Optimization . . . 206
9.2.2 Web Services' Negotiation . . . 206
9.3 GarageOnline Example . . . 207
9.4 Composite Contract Re-Negotiation . . . 208
9.4.1 Runtime Negotiation . . . 209
9.4.2 End-to-end QoS . . . 210
9.4.3 Re-negotiation methodology . . . 210
9.5 Negotiation Specification . . . 211
9.6 Negotiation Results . . . 212
9.7 Related Work . . . 213
9.8 Conclusions . . . 215

10 Implementation Overview **217**
10.1 Orc . . . 217
10.2 Weaving QoS algebraic rules . . . 218
10.2.1 Updating OIL . . . 218
10.2.2 *Q-Orc*: Updating Orc with QoS . . . 219
10.2.3 Monotonicity . . . 223
10.3 Platform for QoS Management . . . 224
10.3.1 Management Tasks . . . 224
10.3.2 Experimentation . . . 225

11 Appendices **227**
11.1 Proofs from Chapter 3 . . . 227
11.1.1 Proof of 2 . . . 227
11.1.2 Proof of 3, Sufficiency . . . 228
11.1.3 Proof of 3, Necessity . . . 229
11.2 Pairwise Products in Chapters 5 and 6 . . . 229
11.3 Dell Example in Chapters 7 and 8 . . . 229
11.3.1 Orc code . . . 229
11.3.2 MATLAB code . . . 232
11.4 Importance Sampling/Splitting codes for Chapter 8. . . . 233
11.5 Stochastic Dominance for Chapter 9 . . . 234
11.6 Q-Orc Implementation Outline from Chapter 10. . . . 235

Bibliography **252**

List of figures **253**

Part I

Gestion Flexible de la Qualité de Service dans les Orchestrations de Services Web.

Chapter 1

Résumé en Français

L'internet a eu un impact considérable sur notre vie quotidienne. Ses applications dont le nombre a augmenté à un rythme rapide sont diverses. Parmi elles figurent la banque en ligne, l'achat en ligne, les sites de réservation de voyage ou encore les réseaux sociaux. Le facteur clé dans cette progression a été la révolution logicielle mise en jeu. Les *Services Web* ont pour principal objectif de rendre plus modulaire et moins propriétaire le développement de logiciels et d'applications informatiques. Il peut s'agir d'une application d'agenda ou de traitement de texte hébergée sur un serveur distant accessible via Internet. L'utilisation des technologies web avec des protocoles normalisés réduit l'hétérogénéité tout en facilitant l'interopérabilité.

Les services web émergent dans le monde des systèmes d'information, dans le contexte de la mise en oeuvre d'*Architectures Orientées Services*, en implémentant les fonctions applicatives élémentaires sous forme de modules. Composés de services normalisés et interopérables, ces modules sont susceptibles d'être ensuite combinés et réutilisés pour donner naissance à de nouvelles solutions ou services composites.

L'*orchestration* des services web fournit un contrôle centralisé de la gestion et de la coordination des modules impliqués dans de telles solutions composites. Un exemple typique est celui d'un service de réservation de voyage contrôlant le flux d'information entre les trois modules suivants: la réservation du vol, la réservation d'hôtel et la location de voiture. De par la récursivité de la composition de service, les services composites peuvent eux aussi être combinés de façon à donner de nouveaux services composites.

Le contrôle de flux dans l'orchestration des services web dépend à la fois des données reçues des services invoqués mais aussi de propriétés *non fonctionnelles*. Par conséquent, une description exacte des paramètres de *Qualité de Service* (QoS) est cruciale dans les architectures orientées service. La qualité de service associée à un service peut avoir plusieurs formes. Il peut s'agir du temps de réponse, de la sécurité des informations transmises, du coût de l'invocation, de la renommée du service, etc. La qualité de service pourrait être un facteur déterminant lorsqu'il s'agira de faire un choix entre plusieurs services offrant des fonctionalités similaires.

Vu le besoin d'assurer une interaction fiable et efficace entre les services composites, la notion de contrat de service ou de *Service Level Agreement* (SLA) a pris de l'importance. Si l'un des services invoqués ne répond pas aux exigences de QoS, ce service pourrait avoir des effets néfastes sur la performance des autres services qui est éventuellement meilleure. Ainsi, pour les fournisseurs de service, il est essentiel de modéliser, d'étudier et de surveiller les propriétés non fonctionnelles des services offerts. Une fois cela fait, l'orchestration doit impérativement choisir de façon optimale les services qui vont maximiser la qualité de service perçue par le client.

Dans cette thèse nous étudierons plusieurs aspects de la gestion fiable de la QoS dans les services web. Quelques uns des aspects considérés sont les suivants:

1. *Modélisation précise de la QoS* - Des efforts considérables ont été fournis à la fois dans le milieu universitaire et dans celui de l'industrie pour décrire des modèles sémantiquement précis de composition de service web. Cependant, la question du traitement de la QoS est orthogonale à celle des spécifications de l'orchestration et demeure ouverte dans le cas où le flux d'orchestration dépend des paramètres de QoS comme la latence ou le niveau de sécurité. Les modèles doivent intégrer de façon précise les aspects QoS au sein des orchestrations en particulier la façon dont ces aspects QoS influencent le contrôle de flux.

2. *Accords contractuels ameliorés* - La QoS de l'orchestration dépend de celle des services invoqués. Par conséquent, il est important que ces services soient contractuellement tenus de répondre à certaines exigences de sorte que l'orchestration puisse satisfaire son propre contrat de qualité de service. Pour certains services le niveau de QoS est bien specifié tandis que pour d'autres ce niveau dépend des observations ou doit être accepté tel quel. La QoS des services invoqués et celle de l'orchestration doit être surveillée en temps réel. De sorte à s'assurer qu'ils répondent au niveau de QoS spécifié et qu'ils offrent des solutions de rechange en cas de dégragation des performances. Une notion à considérer ici est celle de la non monotonie dans le comportement, quand l'amélioriation d'un service peut impliquer la détérioration du comportement global. Les contrats de bout en bout peuvent être basés sur des simulations ou des techniques analytiques. Les techniques que nous utilisons sont basées sur celle de Monte-Carlo, elles peuvent être lentes et présenter une grande variance. C'est pourquoi des techniques alternatives permettant d'améliorer le processus de simulation sont nécessaires.

3. *Outils d'optimisation* - A partir d'un ensemble de services et de leur description contractuelle, une orchestration peut faire un choix parmi plusieurs services alternatifs offrant des fonctionalités similaires avec différents niveaux de QoS. Des compromis peuvent être faits de façon à optimiser la latence ou le coût engagé par l'orchestration. De plus, des négociations peuvent être faites pour fournir de «meilleures» obligations contractuelles permettant d'améliorer l'invocation d'un service particulier au sein d'une orchestration. Un autre aspect à considérer est celui de fournir de meilleurs outils mathématiques pour les concepteurs d'orchestration. L'optimisation vue en tant que service peut améliorer de façon considérable le dynamisme de la gestion via le web de processus complexes dans le domaine de la logistique et de la recherche opérationnelle.

4. *Prise en compte de différents aspects* - Dans la mise en place d'une gamme de produit, plusieurs instances de services composites peuvent être développées. Il y a de la variabilité supplémentaire dans le choix des services disponibles mais aussi dans le comportement QoS. Les techniques pour analyser de telles familles de services sont nécessaires. Dans les services transactionnels, fonctionalité et QoS interagissent de façon très étroite, menant à des situations où elles ne peuvent pas être découplées. Il est nécessaire de disposer de techniques permettant de les traiter conjointement, avec des spécifications fonctionnelles prenant en compte la gestion de la QoS.

Résumé du chapitre Ce chapitre vise à introduire brièvement les notions de services web, de composition et de gestion de la QoS. Un aperçu des services web et des technologies associées (XML, UDDI, WSDL and SOAP) est donné en Section 1.1. Différents aspects de la composition des services web sont presentés en Section 1.2. Parmi eux figurent les modèles de flux et les représentations formelles des orchestrations. Deux langages permettant de spécifier les orchestrations : BPEL et Orc sont étudiés en Section 1.3. La Section 1.4 traite en détail la qualité de service dans les services web,

notamment les questions de composition de QoS, d'accords contractuels, de surveillance et d'optimisation y sont traitées. Ce chapitre se termine par un aperçu du reste de la thèse mettant en évidence les principales contributions.

1.1 Technologies de Services Web

Cette section aborde brièvement les Services Web ainsi que quelques-unes des technologies et langages associés. Une courte présentation de WSDL, d'UDDI, de SOAP est fournie. Pour plus d'informations à ce sujet, le lecteur intéressé pourra consulter [ACKM04, TP02].

Comme défini par le World Wide Web Consortium (W3C), un *Service Web* est un système logiciel identifié par un URI, dont les interfaces publiques et les incarnations sont définies et décrites en XML. Sa définition peut être découverte (dynamiquement) par d'autres systèmes logiciels. Ces autres systèmes peuvent ensuite interagir avec le service web suivant la façon décrite dans sa définition, en utilisant des messages XML transportés par des protocoles Internet [W3c04b].

D'une manière générale, les services Web sont sans état (stateless) et peuvent être invoqués un certain nombre de fois par les clients, sans qu'il ne soit nécessaire de partager des protocoles spécifiques. Chaque requête est, en général, traitée individuellement avec différentes caractéristiques de qualité dans les réponses.

L'architecture orientée services (SOA) est un nouveau modèle d'interaction applicative qui met en oeuvre des services (composants logiciels) autonomes et hétérognènes (il peut s'agir par exemple de services s'exécutant sur différentes plates-formes ou détenues par différentes organisations).

Avec l'évolution des technologies, les applications ont tendance à devenir hétérogènes et à se spécialiser par métier (entité, service, etc.). Cela provoque un fonctionnement en silo empêchant certaines formes de transversalité. L'intégration des applications consiste à développer des connecteurs spécifiques permettant de faire communiquer entre-eux les différents silos. Les avantages induits comprennnent entre autres une productivité élevée et une réduction des coûts dans les processus business-to-business (B2B).

Les trois entités principales intervenant dans les architectures de services web sont le client du service, le fournisseur du service et le registre des services. Dans une une telle architecture, il est indispensable de disposer d'une description des services disponibles et d'un serveur d'annuaire mais aussi de permettre une communication entre les entités citées ci-dessus. l'architecture orientée service repose sur des technologies et langages standardisés. En général, XML (Extensible Markup Language) sert à décrire les données, SOAP est le protocole d'échanges des messages au format XML dans une logique de RPC (Remote Procedure Call), WSDL (Web Services Description Language) est utilisé pour décrire le service et UDDI pour lister les services disponibles.

XML: Extensible Mark-up Language

XML (Extensible mark-up language) fournit une syntaxe commune utilisée pour tous les standards de services Web. Dans le contexte des services Web, plusieurs protocoles peuvent être utilisés: TCP / IP, HTTP ou les protocoles de transfert de courrier. Le mécanisme devrait être en mesure de travailler avec une variété de protocoles de transport. En utilisant SOAP, les services peuvent échanger des messages selon un format standard: c'est-à-dire convertis en messages XML, transmis puis reconvertis en une invocation de service. XML est un standard pour définir le contenu d'un message informatique. Si la sortie d'une application logicielle est au format XML et qu'une autre application est capable d'interpréter ce langage, elle (cette autre application)

peut analyser cette sortie et agir en conséquence. XML permet de spécifier des données semi-structurées et de les interroger.

UDDI: Universal Description, Discovery and Integration

Universal Description, Discovery and Integration (UDDI) [Oas04] est un ensemble de registres web qui divulguent des renseignements sur une entreprise ou tout autre entité. UDDI complète les technologies basiques de service web en permettant de créer un annuaire permettant de localiser sur le réseau le services web recherché. Cela place les Services Web au coeur des entreprises qui opèrent entre elles via Internet. La spécification UDDI permet à ces entreprises de se localiser et d'effectuer des transactions rapidement, facilement et dynamiquement. UDDI fournit une API qui permet l'invocation du service Web lui-même. Il peut être utilisé pour:

- trouver des mises en oeuvres de services web qui sont basées sur une définition commune d'interface abstraite.
- interroger les fournisseurs de services web répertoriés suivant un schéma de classification connu
- démarrer une recherche de services basée sur un mot-clé général.
- mettre en mémoire cache des informations sur un service Web, puis les mettre à jour lors de l'exécution.

Il est possible d'utiliser UDDI pour créer des registres privés résidant au sein de réseaux privés, et offrant leur fonctionnalités à un ensemble spécifique d'utilisateurs.

WSDL: Web Services Description Language

Web Services Description Language (WSDL) [W3c01] fournit une grammaire XML pour décrire les services réseau comme des points terminaux de communications capables d'échanger des messages. Les définitions des services WSDL présentent de la documentation pour les systèmes distribués et servent d'outil pour l'automatisation des détails relatifs aux communications entre les applications.

L'un des avantages de WSDL c'est qu'il soit un standard à usage général qui peut être utilisé dans une variété d'environnements. Les normes établies peuvent être enveloppées par des interfaces WSDL pour décrire le protocole de communication et le format de messages requis pour communiquer avec le service. La description orientée fonction que fournit WSDL présente l'inconvénient d'être statique. Cependant, l'ordre des messages ou des flux de données ne peuvent pas être traité. Des langues comme BPEL [OAS07] doivent être utilisés avec WSDL pour spécifier l'ordre de transmission des messages. WSCI (Interface Web Chorégraphie Services) [W3c02] complète WSDL et supporte de telles transmissions de messages. WSDL est également limité puisqu'il ne précise que les aspects syntaxiques des services. Les langages sémantiques comme OWL-S [W3c04a] essaient de combiner ces fonctionnalités pour la découverte automatique, la définition des protocoles à utiliser pour invoquer le service et la composition de services.

SOAP

SOAP [W3c00] sert de base pour toutes les interactions entre services web. Il définit l'organisation des données sous forme texte structuré au format XML pour permettre l'échange entre pairs. Il précise:

- un format de message à sens unique pour encapsuler les informations dans un document XML,
- des conventions pour définir comment les clients peuvent invoquer procédures distantes en envoyant un message SOAP et comment les services peuvent répondre en retournant un message SOAP,
- des règles pour traiter et analyser les messages SOAP au format XML,
- une description de la façon dont les messages SOAP peuvent être transportés au-dessus de HTTP, SMTP et d'autres protocoles.

SOAP permet l'échange d'informations en utilisant des messages encapsulés dans une *enveloppe*. L'enveloppe contient deux parties: un en-tête (*Header*) et un corps (*Body*). L'information principale que l'expéditeur souhaite transmettre au destinataire est fourni dans le corps. Les informations supplémentaires nécessaires aux traitements intermédiaires des services (sécurité, ajout de nouveaux messages, etc) sont situées dans l'en-tête.

REST: Representational State Transfer

REST (Representational State Transfer) [Fie00] est une alternative style architectural services web utilisée dans l'industrie. Les services web de type REST forment un ensemble de ressources Web identifiés par des identificateurs de ressources uniformes (URI). La mise en œuvre sur HTTP/HTTPS, quatre méthodes sont disponibles: `GET`, `PUT`, `POST`, and `DELETE`.
En comparant RESTful en SOAP / WSDL services:

Scalabilité - Dans REST toutes les interactions sont sans état (stateless) - chaque requête doit contenir toutes les informations nécessaires pour que cette demande soit comprise, et elle ne peut tirer profit d'aucun contexte stocké sur un serveur par exemple. Cette caractéristique (sans état) améliore la scalabilité des applications. D'une manière générale, SOAP est aussi sans état. Toutefois, des normes de niveau plus élevé existent pour créer des services web avec état (stateful) via SOAP.

Composants indépendants - Dans REST les composants peut être déployés indépendamment les uns des autres. Le contenus des sites web peuvent être échangés sans adaptation à un protocole spécifique.

Composition de services - au sens strict, il n'y a pas de services REST, seulement des ressources. Les URLs offrent un espace d'adressage global de sorte que les documents fassent référence (de façon simple) à une ressource qui est dans une autre organisation. Les primitives d'interaction (`GET`, `POST`, `PUT`, `DELETE`) découlant de REST peut être utilisé à partir d'un processus BPEL comme nouvel appel de service [Pau09].

1.2 Composition de Services Web

Un service Web mis en oeuvre en combinant les fonctionnalités offertes par des services web individuels est appelé *Service Web Composite* et le processus qui consiste à combiner les services est appelé *composition* de services web. Les applications sont créées en combinant les éléments de base fournis par d'autres services. Les services composites peuvent aussi être combinés suivant un modèle de composition service récursif.

Les études sur les techniques et les outils disponibles pour l'architecture des services composites incluent [DS05, HS05].

Cette section donne un aperçu de la gestion de workflow, des orchestrations, des chorégraphies et des méthodes formelles pour les représenter.

Workflow Management

La notion de *Processus d'affaires* (*Business Process*) fait référence à une collection de tâches exécutées de manière coordonnée entre utilisateurs et applications logicielles pour accomplir une tâche. On peut appeler *workflows* une représentation formelle et exécutable des processus d'affaires. Un système de gestion de workflow fournit un outil logiciel permettant de concevoir, exécuter et analyser des workflows.

Les systèmes de gestion de workflow visent à automatiser le traitement administratif des documents tels que les ordres de mission, les fiches financières et les rapports d'avancement de projet. Plutôt que de traiter ces tâches par courrier en version papier ou électronique, les systèmes de gestion de workflow utilisent des formulaires web en version électronique. Ces systèmes permettent également l'intégration de participants hétérogènes et distribués, ce qui est crucial pour les grandes entreprises. Elles peuvent gérer le flux de contrôle et de données entre les diverse participants et les applications, ce qui permet l'intégration dans la plupart des scénarios d'affaires.

Les systèmes de gestion de workflow ont tendance à être exprimés en langages graphiques de haut niveau plutôt que sous forme de de langage de programmation. C'est dans le but de permettre à des personnes qui ont des compétences variées de modéliser les workflows avec beaucoup d'aisance. Les normes élaborées telles que *Business Process Modeling Notation (BPMN)* [OMG11] ont été implémentées dans des solutions commerciales de système de workflow comme IBM ILOG JViews [IBM11] et Oracle Business Process Management Suite [Ora11]. Celles-ci ont été étendues à des systèmes de gestion de workflow basés sur le web. Les éditeurs de processus d'affaires comme BonitaSoft [Bon11], Signavio [Sig11] et Oryx [DOW08] en sont des exemples.

Orchestrations

Les termes *orchestration* et *chorégraphie* décrivent deux aspects de la création de processus d'affaires à partir de services web composites. Une orchestration représente toujours le contrôle d'un point de vue d'un participant. Dans la plupart des cas, elle est centralisée dans le traitement des flux de contrôle qui peut être dépendant des données reçues d'autres participants. Cela diffère de la chorégraphie, qui est plus collaborative et permet à chaque participant impliqué de décrire son rôle dans l'interaction. La chorégraphie suit les séquences de messages entre de multiples participants et sources plutôt que de suivre ceux d'un processus d'affaire spécifique exécuté par un participant unique.

Les orchestrations dans le contexte des services web doivent gérer plusieurs propriétés d'activités, y compris le langage utilisé pour spécifier les compositions, les données, les transactions et la gestion des exceptions. D'autres techniques proposées en regard des orchestrations utilisent un contrôle décentralisé [CCMN04] ou sont basées sur des sessions [FN08].

Modèles formels pour les orchestrations de services

Les orchestrations de services Web spécifient l'ordre et les conditions dans lesquelles les services web sont invoqués dans une composition cohérente. Bien qu'il existe de nombreux langages tels que BPEL [OAS07], COWS [LPT07], Orc [KQCM09] et YAWL

[tHvdAAR10] pour représenter les workflows, les orchestrations s'appuient sur les modèles suivants :

Les diagrammes d'état [Har87] sont un formalisme basé sur une machine d'état qui spécifie les tâches réalisées lors de l'entrée dans un état, lors de la sortie d'un état ou à l'intérieur d'un état. Le tir de transitions conditionnelles en fonction des données et des flux de contrôle est également intégré. Les états composites, les actions parallèles et la synchronisation sont des actions complémentaires et complexes dans les diagrammes d'états. Ceux-ci possèdent une sémantique formelle pour analyser les spécifications de services composites. Cette sémantique est liée à des langages tels que le langage de modélisation unifié (UML) [DH01]. Les diagrammes d'états offrent la plupart des constructions de flux de contrôle et des dépendances qui sont disponibles dans les langages de modélisation de processus existants. Pour ces raisons, les diagrammes d'états ont été utilisés dans l'architecture AgFlow dans [ZBN+04] pour décrire des compositions dépendantes de la qualité de service.

Un diagramme d'état est constitué d'états et de transitions. Les transitions sont étiquetées avec des événements, des conditions et des opérations. Les états peuvent être simples ou composés. Les états simples sont marqués avec un nom d'opération d'un service donné qui peut être invoqué. Les états composés peuvent être raffinés en utilisant la relation AND (sous états concurrents) ou la relation OU (sous états disjoints). L'état initial d'un diagramme d'état est représenté par un cercle plein, tandis que l'état final est désigné par deux cercles concentriques.

Les réseaux de Petri [Mur89, Rei92] ont été introduits en tant que base pour modéliser des systèmes concurrents. Leur attrait principal est la façon naturelle dont de nombreux aspects fondamentaux des systèmes concurrents sont identifiés à la fois mathématiquement et conceptuellement. Leur facilité de modélisation conceptuelle (en grande partie due à notation graphique simple) a par ailleurs fait des réseaux de Petri le modèle favori dans de nombreuses applications [vdA96]. Ils sont capables de modéliser les comportements concurrents, non-déterministes et asynchrones qui peuvent être appliqués aux services web.

Parmi les alternatives proposées, figure Workflow nets [vdABL08] utilisé pour modéliser les workflows génériques avec une entrée unique et des points de sortie. Dans YAWL [tHvdAAR10], les réseaux de Petri sont étendus pour supporter la modélisation de patterns de workflow [vdAtHKB02] que les réseaux de Petri traditionnels ne sont pas capables d'exprimer. [tHvdAAR10] prend en charge des instanciations multiples, fournit des capacités de synchronisation et d'annulation intégrées que l'on trouve généralement dans les orchestrations.

Le π-calcul [MPW92] est une algèbre de processus qui développe un modèle formel pour décrire les processus. L'utilisation du π-calcul pour la description des processus s'explique par les avantages qu'un modèle formel avec une théorie riche fournit pour la vérification automatique des propriétés exprimées dans un tel modèle. Pour une orchestration, le π-calcul fournit des constructions permettant d'appeler les services de façon séquentielle, simultanée ou dépendantes d'exécutions conditionnelles.

Quand un service à lui tout seul est incapable de répondre à toutes les exigences fonctionnelles d'un client, une solution envisageable consiste à générer un service composite. La composition s'occupe de produire un cahier des charges sur la façon de coordonner les différents services pour satisfaire la demande du client. Les études réalisées pour synthétiser automatiquement de tels services composites comprennent [HS05]:

OWL-S [W3c04a]: Une contribution majeure de OWL-S est la modélisation de la façon dont les services web interagissent avec le «monde réel». La notion de *processus* atomiques et composites est le fondement du modèle OWL-S. Les processus OWL-S sont spécifiés pour avoir des entrées, des sorties, des pré-conditions, et des effets conditionnels.

Roman [BCG+03]: Utilise une notion abstraite de service atomique dans le cadre d'automates à états finis pour décrire les flux de processus. Pour spécifier le déroulement du processus interne d'un service Web, les systèmes de transitions où chaque branche correspond à une séquence d'exécutions permises sont utilisés.

Message Based [BFHS03]: Tandis que les modèles OWL-S et Roman mettent l'accent sur ce qu'un service ou une composition fait, ils ne résolvent pas complètement la question de la façon dont les services, dans une composition, interagissent entre eux. La notion de conversations ou de passage de messages est utilisée dans ce modèle dans un cadre de «pairs à pairs».

1.3 Langages pour Spécifier des Orchestrations

Dans cette section, nous décrivons deux langages utilisés pour spécifier des orchestrations: la norme de l'industrie *BPEL* et le langage académique *Orc*. D'autres langages comme Yet Another Workflow Language [tHvdAAR10], generic web service combinators [CD99] ou Calculus for Orchestration of Web Services [LPT07] existent. Tout lecteur intéressé peut consulter leurs références.

BPEL: Business Process Execution Language

BPEL [OAS07] est devenu la norme de facto pour décrire les compositions de services Web. D'autres langages, comme BPML [Ark02] et WSFL [Ley01] sont des alternatives moins courantes. BPEL peut être utilisé à la fois pour spécifier des orchestrations et des chorégraphies. Elles peuvent définir la séquence de messages envoyés ou reçus par les services ainsi que les contraintes d'ordre sur les actions envoyer ou recevoir.

Un moteur d'orchestration peut alors exécuter cette grammaire BPEL, coordonner des activités et compenser le processus global lorsque des erreurs surviennent. La couche BPEL est située au-dessus de WSDL. L'interface WSDL définit les opérations permises tandis BPEL définit comment les enchainer.

Un langage de spécification supplémentaire a été proposé dans Web Services Choregraphy Interface [W3c02]. WSCI définit l'ensemble de la chorégraphie ou de l'échange de messages entre les services web. WSCI ne décrit que le comportement observable entre les services Web, il ne traite pas la définition de processus d'affaires executables comme le fait BPEL. Le traitement XML des flux de données au sein des chorégraphies a reçu une extension dans [HB09, HB10], où des documents actifs sont créés et modifiés par des pairs distribués.

Orc

Bien que BPEL a été intégré industriellement, sa sémantique informelle donne lieu à des parties du langage ayant différentes interprétations possibles, ce qui n'est pas souhaitable pour modéliser les orchestrations. Un modèle formel pour spécifier et analyser le comportements des orchestrations de services Web est essentiel et très utile. Les études théoriques pour formaliser et étudier mathématiquement le comportement des services Web comprennent [Rei08, BKM07]. BPEL a également été examiné dans le cadre de la sémantique opérationnelle [Fah05] en utilisant comme modèle les réseaux de Petri [HSS05] et le π-calcul [LM05]. La traduction de BPEL dans des modèles de flux de travail dans [BP06] fournit des analyses intéressantes, telles que la comparaison avec YAWL.

Orc [Orc11, KQCM09, Mis10] est utile pour modéliser des calculs distribués, mais est surtout destiné pour modéliser les orchestrations de services Web sur Internet. Ce langage est basé sur un cadre mathématique simple et possède une sémantique formelle.

La simplicité de sa syntaxe et sa grande expressivité le rend très intéressant pour spécifier les orchestrations. Orc implémente le *tree programming* pour des orchestrations où la requête initiale peut être transmise dans des branches parallèles ou enchaînées en séquence de façon à effectuer la tâche demandée. Les résultats des sous-requêtes sont regroupés, transmis à d'autres sous-requêtes et finalement retournés à l'apppelant initial. Cela correspond bien au concept d'orchestration.

Dans langage Orc, le calcul est basé sur l'exécution d'*expressions* Orc. Les expressions sont construites récursivement en utilisant les *combinateurs* Orc concurrents. Lorsqu'elle est exécutée, une expression Orc appelle les services et peut publier des valeurs. Les expressions Orc utilisent des *sites* pour faire référence à des services externes. Un site peut être mis en oeuvre sur la machine du client ou sur une machine distante. Un site peut fournir un service ou jouer le rôle de proxy pour permettre l'interaction avec un utilisateur.

L'expression Orc la plus simple est un appel *site* $F(p)$, où F est un nom de site et p est une liste de paramètres. Ceux-ci qui sont des valeurs ou des variables. L'exécution d'un site appel invoque le service associé à F en lui envoyant les paramètres p. Si le site répond, l'appel publie cette réponse. Un site peut donner au plus une réponse à un appel. Un appel site peut explicitement déclarer qu'il ne donnera jamais de réponse, dans ce cas, nous disons que l'appel est *suspendu*. Certains appels site ne peuvent ni répondre, ni être suspendus.

Orc a quatre combinateurs pour composer des expressions: le combinateur parallèle `|` , le combinateur séquentiel $>x>$, le combinateur réduction $<x<$ et le combinateur sinon `;` . Lorsqu'on constitue des expressions, le combinateur $>x>$ a la plus haute priorité, suivie de `|` , puis de $<x<$, et enfin `;` avec la priorité la plus basse.

D'autres aspects comme les types de données (`tuples`, `records`, `lists`), le traitement de la concurrence (`semaphores`, `channels`) et les constructions permettant de créer de nouveaux sites (`class`) sont mis à disposition des utilisateurs d'Orc. Pour plus de détails, le lecteur est invité à consulter [1].

1.4 Qualité de Service

Cette section examine en détail différents aspects qui doivent être considérer dans le cadre de la qualité de service. Partant du concept général de QoS, nous décrivons la composition de QoS, les SLAs, leur surveillance et leur négociation ainsi que l'optimisation dépendante de la QoS.

1.4.1 QoS dans les services Web

Deux aspects différents doivent être considérés quand on parle de services:

1. la description *fonctionnelle* contient les spécifications formelles des fonctionnalités offertes par le service.

2. la description *non fonctionnelle* de la façon dont le service remplit ses fonctionnalités.

Considérons l'exemple d'un service de réservation de train. Sa fonctionnalité (la réservation d'un billet de train) pourrait être limitée par l'utilisation d'une connexion sécurisée (la sécurité étant une propriété non-fonctionnelle) ou par la tranche horaire dans laquelle les services sont invoqués (la disponibilité étant une propriété non-fonctionnelle). Pour les services individuels, les propriétés fonctionnelles et non fonctionnelles sont traitées de façon orthogonale.

[1] http://orc.csres.utexas.edu/documentation/html/refmanual/refmanual.html#N14CDB

S'agissant des services composites qui invoquent divers services variant suivant leur caractéristiques fonctionnelles et non fonctionnelles, les caractéristiques de bout en bout sont intimement liées au contrôle du flux de l'orchestration. La performance fonctionnelle de bout-en-bout repose sur une grande partie des services qui exécutent des tâches spécifiques et qui transmettent les données nécessaires au sein du contrôle de flux. Cependant, dans la plupart des orchestrations, les contraintes sur les propriétés non-fonctionnelles (expiration de délai, niveaux de sécurité, etc) affectent aussi la performance fonctionnelle de bout en bout. C'est pourquoi elle ne peuvent pas être traitées de façon orthogonale. Les propriétés non-fonctionnelles peuvent également jouer un rôle important dans toutes les tâches associées aux services, en particulier au niveau de la découverte, de la sélection et du remplacement de services. On imagine un scénario où l'on pourrait faire un choix entre plusieurs services, pouvant répondre à la requête d'un utilisateur et offrant essentiellement la même fonctionnalité, en fonction de certaines propriétés non fonctionnelles comme le prix ou la performance.

Une certaine analyse est nécessaire pour faire un lien entre les propriétés fonctionnelles et non fonctionnelles de bout en bout et les propriétés des services invoqués dans une orchestration. La dépendance et la causalité à l'égard des données dans le flux d'une orchestration peut fournir des indications sur la performance fonctionnelle de bout en bout (par exemple la panne d'un certain service peut mener à un interblocage). Des paliers dans la QoS de bout en bout et leur relation avec le flux de contrôle (invocation en parallèle, en séquentiel, avec des contraintes) peuvent préciser la contribution individuelle de chaque service à la performance de bout en bout. Compte tenu de ce fait, une modélisation précise de la QoS pour les services web est importante pour le fonctionnement de l'orchestration et pour la spécification, de façon contractuelle, des propriétés non fonctionnelles des services invoqués.

La QoS couvre toute une gamme de techniques qui répondent aux besoins des demandeurs et des fournisseurs de services sur la base des ressources réseau disponibles. Par QoS, nous faisons référence aux propriétés non fonctionnelles des services web comme la performance, la fiabilité, la disponibilité et la sécurité [TF06]. Les études sur la QoS pour les services Web ont reçu une attention considérable dans les articles tels que [TF06, CSM+04, ACH98, AGM08], qui fournissent un large aperçu des aspects de la QoS et de leur gestion. Des études sur le comportement en temps réel de la latence et du débit des services ont été faites dans [CTB98, LNJV01, XSCT02]. Les aspects comme la sécurité et les certificats numériques ont été abordés dans [KM03, BMR08, KCE+04, BFG05, IM02]. Les stratégies de tarification pour les services web sont mises en évidence dans [HDHA09]. Toutes les propriétés ci-dessus représentent des paramètres QoS pour les orchestrations mais les chorégraphies peuvent bénéficier de modèles élargis prenant en compte l'analyse de la transmission des messages et l'absence d'interblocage [MPR+10, BZ07].

1.4.2 Composition de QoS

Comme les compositions de services exigent de combiner les fonctionnalités de différents services, constituer la QoS de ces services devient tout aussi important. Les contrats ou les valeurs de QoS convenu(e)s avec les sous-traitants doivent être agrégées et comparées pour produire en sortie la QoS de bout en bout. La composition de QoS est le processus d'agrégation de la QoS des services invoqués (éventuellement en sous-traitance) pour que l'orchestration puisse estimer son propre résultat à l'égard des obligations contractuelles.

L'orchestration peut disposer d'informations et de modèles sur ses ressources locales et ses opérations. Cependant, elle ne peut pas disposer d'informations complètes sur la performance des services distants ou sur l'utilisation du réseau. Elle devra s'appuyer sur

les obligations contractuelles pour estimer cette performance ou l'accepter telle qu'elle (par exemple, l'accès à un site web public). La composition de la QoS provenant de services individuels nécessite un ensemble de règles permettant d'agréger les quanta de QoS.

En utilisant les simulations de Monte-Carlo ou des techniques analytiques, on peut déduire la QoS de bout en bout pour une seule exécution du service composite. Toutefois, cela ne prend pas en compte la nature probabiliste de certains paramètres et donc, plusieurs simulations de Monte-Carlo ou données historiques sont nécessaires pour déduire des distributions. L'agrégation des indicateurs probabilistes de QoS peut se faire en utilisant des valeurs moyennes pour la latence, la disponibilité ou en utilisant des modèles mathématiques stochastiques et de files d'attente plus appronfondis. Des approches stochastiques pour la composition et la planification ont été présentées dans [WHK08, WZZF09].

Pour développer davantage cette notion de composition de QoS, nous prenons l'exemple du service web composite de la Fig. 1.1. Il met en évidence le processus typique de la commande des produits informatiques.

Nous détaillons certains aspects du contrôle de flux au sein de l'orchestration. Les services Check Order et Check Credit vérifient l'information fournie par le client ou une base de données traite la commande. Une fois cela fait, le Hardware Supplier qui réagit le plus rapidement et le Software Supplier avec le coût le plus bas sont utilisés pour la composition. Par ailleurs, une opération timeout peut englober toute l'orchestration pour éviter que le client n'attende des services «suspendus».

Figure 1.1: Un ensemble de service composite orchestration démontrant des composants.

Notons qu'il s'agit d'un workflow typique où interagissent les données des services invoqués et la QoS mesurée par l'orchestration. Pour cette raison, les approches utilisant la théorie des files d'attente [ALZH04, DB78] ou les modèles stochastiques [TLY+04] pour les performances des services web ont des difficultés à décrire avec justesse le comportement des services basés sur des transactions. Ces approches sont appropriées pour l'analyse du réseau et l'étude de compositions de services simplistes. Les approches basée sur la simulation de performance des services composites peuvent fournir une description plus réaliste. Des exemples d'études empiriques du comportement de la performance des services Web sont disponibles dans [HM09, LNJV01].

1.4.3 Monotonie dans les Orchestrations

Une propriété importante que nous considérons ici est la monotonie de l'orchestration [BRBH08]. Pour qu'une agrégation ou pour que toute approche contractuelle soit valable, l'orchestration doit être *monotone*, c'est-à-dire «si un service fournit de meilleurs résultats, alors il en sera de même pour l'orchestration». Lorsque plusieurs aspects de la QoS et des données interagissent, une amélioration de la performance d'un des services peut quand même dégrader la performance globale. Cette remarque est illustrée dans une variante, reliant un fournisseur de matériel spécifique à un fournisseur de logiciel spécifique. L'amélioration du temps de réponse de Hardware Supplier 1 peut entraîner un retard de l'orchestration dans son ensemble si le service lent de Software Supplier 1 est invoqué, comme indiqué dans la Fig. 1.2. Les conditions pour éviter que cela se produise sont données dans [BRBH08]. Parmi elles figure le fait de ne pas lier un service particulier à un autre (comme on le voit dans la Fig. 1.1). Il est important de considérer cette remarque dans le cas où l'utilisation de SLAs comme service de performance supérieure (sous-traitant) peut être inutilement pénalisante du fait de la détérioration de performance par un autre service.

Figure 1.2: Une *non-monotone* orchestration.

Les orchestrations qui ne sont pas *dépendantes des données* (c'est-à-dire où le contrôle de flux ne dépend pas des données renvoyées par les services), sont en général monotone. Toutefois, pour les workflows qui sont dépendants des données, une analyse minutieuse des conditions de monotonie est nécessaire. Dans la plupart des cas, les implications de monotonie ont été ignorées, ce qui n'est pas adapté à la gestion basée sur des. Les articles de Ardagna et al. [AP05], Alrifai & Risse [AR09a] et Zeng et al. [ZBN+04] identifient néanmoins les notions d'optimisation globale et locale, ce qui permet de contrôler les conditions de monotonies. Dans une orchestration, l'optimisation globale fournit la situation idéale. Dans une orchestration monotone, cela peut être raffiné avec l'optimisation locale suffisante pour assurer l'optimalité globale.

1.4.4 Service Level Agreements

Afin de fournir un support de QoS dans les services Web, un certain nombre d'extensions aux normes actuelles ont été proposées. Une proposition visant à publier des services Web à QoS imposée en les enregistrant dans l'annuaire UDDI est présentée dans [Ran03, KBT+09, WVKT06]. Une extension similaire, appelée Q-WSDL, est fournie dans [D'A06] avec une approche basée sur des modèles de paramètres de QoS au sein de WSDL.

Le Service Level Agreement (SLA) (également connu sous le nom de contrat de niveau service ou convention de service) définit un ensemble d'attentes du client qui doivent être remplies par un fournisseur [TP05, SRE10]. Il est tout à fait possible que

les fournisseurs offrent différents services à différents clients. C'est pourquoi, les fournisseurs doivent adopter une politique efficace de gestion des ressources qui différencie les clients suivant les gammes de services.

Un accord de niveau de service est un document qui formalise un accord signé entre deux parties: le fournisseur et le client. Il spécifie, généralement en termes mesurables, les services que le fournisseur s'engage à fournir. Les fournisseurs révèlent la performance promise sous forme de contrats. Ces contrats sont de nature statistique. Un contrat typique entre un serveur et un client est le suivant: «le service doit répondre correctement 90% du temps et, quand il répond correctement, dans 90% des cas, il répondra en moins de 2 secondes». Dans [PB08, TP05, SRE10, jJMS02], des limites dures sont utilisées pour les obligations. Une politique de bonus-malus, dans laquelle un service qui dévie du contrat est puni, y est également prévue.

Une alternative à cette approche consiste à utiliser des modèles probabilistes comme décrit dans [HWSP04, HWTS07, RBHJ08]. Dans [RBHJ08, HWTS07], les services spécifient leur comportement QoS sous forme de distribution probabiliste des paramètres de QoS. La distribution peut aussi être spécifiée de façon approximativee par un ensemble de quantiles. Dans certains cas, des mesures peuvent être utilisés pour obtenir le contrat probabiliste. L'utilisation de distributions a l'avantage de supprimer les approches pessimistes vis-à-vis des contrats, en utilisant des contrats plus souples. L'avantage d'utiliser des contrats probabilistes réside dans une gamme d'outils statistiques comme la dominance stochastique [BD03, And96] qui peuvent aisément être intégrés dans la formulation du SLA.

Etant donné que les valeurs de QoS sont des variables aléatoires, l'estimation des contrats de bout en bout se fait généralement au moyen de simulations. Quelques articles théoriques [ZYZB11] proposent d'agréger les distributions de probabilité, ce qui est presque impossible lorsque les orchestrations présentent des dépendances dans les données du contrôle de flux. Des modèles comme SALSA [BHS+10] (Simulated Annealing Load Spreading Algorithm) utilisent la théorie des files d'attente pour respecter les SLAs de façon autonomique, sans un sur-dimensionnement a priori des ressources.

1.4.4.1 Négociation

La négociation de SLA fait référence à une procédure entre deux parties (client et fournisseur) qui se mettent d'accord sur les termes du SLA. Les parties tentent de parvenir à un accord basé sur un consensus après avoir échangé plusieurs devis sans obligation. Les étapes de négociation sont en général les suivantes:

1. le fournisseur publie un modèle décrivant le service et ses conditions éventuelles, y compris la qualité de service et les compensations possibles en cas de violation. Ce modèle laisse plusieurs champs vides ou modifiables qui visent à appliquer les besoins spécifiques de l'utilisateur.

2. le client récupère le modèle et le remplit avec des valeurs décrivant l'utilisation prévue des ressources. Certains termes du modèle peuvent être supprimés, ajoutés ou modifiés.

3. ce nouveau document, qui n'engage aucune des parties, est envoyé au fournisseur. Lorsque le fournisseur reçoit ce document, il se base sur la disponibilité actuelle des ressources et les politiques à l'égard du client pour renvoyer à ce dernier un devis. Ce devis correspond à des valeurs sur lesquelles le fournisseur serait probablement d'accord (même si cela ne l'engage à rien), en fonction des besoins du client.

4. si le client est satisfait du devis il appose sa signature au document qu'il renvoie au fournisseur comme une proposition de SLA. Le client est effectivement déjà en

train de proposer un SLA au fournisseur, néanmoins la signature du fournisseur est manquante.

5. à la réception de la proposition, le fournisseur est libre de la rejeter ou de l'accepter. Dans ce dernier cas, la proposition devient un contrat de service officiellement signé par les deux parties, et commence à être un document juridique valable.

L'échange de devis (étapes 2 et 3) peut être répété un nombre quelconque de fois. L'utilisateur peut modifier les termes de la demande de devis jusqu'à ce que le devis du fournisseur réponde aux attentes de ce que le client est prêt à accepter. La dernière étape pour le client (l'étape 4), qui demande le SLA réel, a une réponse booléenne: le SLA est accepté ou rejeté par le fournisseur. Dans ce dernier cas, le client peut repartir en arrière et redemander un devis, dans l'espoir que le fournisseur ait changé ses conditions. Les étapes 2-3 constituent la partie essentielle de la négociation, puisque chaque partie peut tirer la négocitation dans n'importe quelle direction. Les parties peuvent modifier librement les différents termes: la baisse des frais, la baisse de qualité de service, l'augmentation des intervalles de temps, la baisse des besoins en ressources, la baisse des compensations, etc.

Une fois qu'un contrat a été signé et accepté, la nécessité de le modifier pourrait être envisagée (voir la re-négociation). Dans ce cas, la même structure de devis pourrait être utilisée. La différence principale vient des ressources déjà affectées, et de l'existence d'un premier contrat à modifier.

Les travaux sur l'automatisation des procédures pour générer de telles procédures de négociation comprennent [CCP07, DDK+04, YKL+07]. Le processus automatique de négociation vise à identifier le niveau maximal de qualité admissible selon le budget de l'utilisateur. [ZZ04] spécifie la vitesse à laquelle les sondes affectés aux services peuvent déployées sans affecter les performances d'exécution. Dans [RBHJ08], la procédure de négociation de contrat implique l'accord des deux distributions - une *hypothèse* côté client sur la capacité de débit et une *garantie* côté fournisseur sur la performance. [BS09b] parle de négociation qui se déroule en utilisant des contraintes souples avec des opérateurs algébriques modélisés en semi-anneau.

1.4.4.2 Surveillance

Une fois qu'un utilisateur et qu'un service Web ont été réunis, l'exécution du service et des propriétés connexes doivent être surveillés en permanence. En outre, il est souhaitable que les ajustements nécessaires se fassent en temps réel sans affecter les opérations sur le site de l'utilisateur. C'est une tâche difficile puisque le service Web peut être exécuté sur un système qui n'appartient pas à l'utilisateur, qui n'est pas contrôlé par celui-ci ou sur un système d'exploitation dont l'utilisateur ignore tout. La notion de surveillance peut se rapporter à de nombreux aspects des services Web tels que la charge, la gestion des erreurs [BGG04] néanmoins nous faisons spécifiquement allusion à la surveillance des propriétés non fonctionnelles et des obligations contractuelles correspondantes.

La surveillance de SLA devient plus compliquée dans le cas des services web composites. Les propriétés d'un service Web composite dépendent de celles des services qui le constituent et il peut être nécessaire de faire collaborer les entités responsables de la gestion de chaque service constituant le serveice web composite. Certains paramètres de QoS comme le temps de réponse ou de l'utilisation des ressources peuvent être contrôlés par le fournisseur de services, d'autres comme le débit et la sécurité des données peuvent exiger d'interroger les clients. Les exemples de travaux sur ce sujet comprennent [MRLD09, SMS+01, ZLC07].

La surveillande de QoS peut être appliquée sur les services Web pendant les différentes phases d'un cycle de vie SOA système. Au cours de la découverte de service, la surveillance est utile pour obtenir la qualité de service réelle pour une sélection fiable de service Web. Une fois le service sélectionné, la surveillance peut être utilisée afin d'assurer que le service remplit toujours la QoS promise. En particulier, dans les systèmes SOA d'*auto-réparation*, quand cette violation a lieu, le système SOA peut remplacer le service web défectueux par un autre. De toute évidence, les systèmes de surveillance ne récupérent des données qu'à partir de mesures basiques, étant donné que les mesures dérivées sont calculées à partir des précédentes par l'intermédiaire d'une règle prédéfinie.

Web Service Level Agreement Language [LKD+03] est un langage standard pour spécifier des accords et surveiller les protocoles pour les SLA. Un accord WSLA complète une définition de service. Une description de service (WSDL) définit la relation d'interface de service entre un service et son application. Quant au WSLA, il définit les caractéristiques de performance convenues et la façon de les évaluer et de les mesurer. Le WSLA est utile au système de mesure et de gestion d'une organisation qui vérifie et gère la conformité de l'organisation avec un WSLA. Les fournisseur de services et les clients peuvent exécuter leur propre système de mesure et de gestion. Chaque organisation peut accéder aux paramètres mesurés à partir de sources diverses. Par exemple, les paramètres côté serveur peuvent être accédés à partir du fournisseur et ceux côté client peuvent être accédés à partir du client.

Une solution de substitution au WSLA est présentée dans SLAng [SLE04] et permet de spécifier les caractéristiques non-fonctionnelles des contrats conclus entre des parties indépendantes. [Men02, ZLC07, LZZX10] développent des techniques pour surveiller les déviations du comportement nominal au niveau de la performance de QoS. D'autres propositions ont pour but d'élargir le cadre du WSLA pour offrir, négocier et suivre les accords contractuels dans les compositions de services [DDK+04].

1.4.5 Optimisation dépendante de la QoS

Lorsque la QoS est considérée comme multi-dimensionnelle, ses domaines sont partiellement (pas totalement) ordonnés. Pour résoudre le problème de la comparaison probabiliste de domaines de QoS qui ne sont que partiellement ordonnés, le Théorème 1 de [TKO77] peut être utilisé. Cela permet de simplifier une comparaison stochastique de variables aléatoires en une comparaison ordinaire puisque les fonctions sont définies sur la même affectation expérimentale de probabilité.

Pour optimiser la sélection sur plusieurs domaines de QoS, une sorte d'ordre total est nécessaire pour générer des fonctions de coût minimum. Il est également possible de minimiser, sur un domaine, tout en faisant implicitement des compromis sur d'autres domaines. L'objectif est d'optimiser la QoS de bout en bout (globale) de l'orchestration. Diverses techniques d'optimisation ont été proposées à cette fin. L'optimisation globale nécessite d'examiner tous les chemins alternatifs lors du chaînage des appels de service. C'est donc une tâche à effectuer hors-ligne et qui requiert une algèbre pour évaluer statistiquement la QoS de bout en bout à partir de la QoS de chaque service appelé.

[LB10] propose une architecture permettant la sélection de services logiciels en se basant sur leur réputation. Un algorithme de sélection est conçu pour la recommandation de service, fournissant aux clients SaaS les meilleurs choix possibles. Dans [CCGM06], une architecture basée négociation est utilisé pour la sélection des services. En modélisant cela comme un problème d'optimisation sous contraintes où chaque classe de QoS est modélisée par des contraintes appropriées, on peut maximiser la QoS globale d'un flux de requêtes. Dans [YZL07], l'objectif de la sélection des services est modélisé comme un problème de sac à dos multidimensionnel à choix multiple (MMKP). Des

approches de sélection similaires générant des configurations optimales de services sont présentées dans [HMR10, XFZ08].

Dans [ZBN+04] la comparaison entre l'optimisation locale et la planification globale est étudiée. Différents paramètres de QoS sont étudiés (prix, durée, réputation, disponibilité, taux de réussite) et des règles de composition sont fournies. Une solution de programmation entière est proposée comme dans [KP09]. Des problèmes similaires et leur solutions sont également traités dans où la notion de contraintes locales et d'optimalité globale sur les domaines de QoS est étudiée.

L'utilisation de techniques mathématiques plus intensives comme le contrôle stochastique [SDR09] et la programmation dynamique [Ber07] a également été proposé. Dans [ZNB+08, GNZ+06], ces techniques sont utilisées pour la sélection et pour la planification de la sélection du service composite. Plutôt que de choisir le «meilleur» service avec des informations à court terme, une approche de programmation dynamique plus poussée est utilisée. Cette notion supprime l'effet de la non-monotonie et se concentre sur les objectifs de qualité de service globale du service composite. Cependant, cette technique peut comporter un très grand nombre de calculs et n'est pas appropriée lorsque les paramètres de QoS varient considérablement des distributions présumées.

1.5 Organisation de la thèse

Cette thèse est organisée en plusieurs chapitres, chaque chapitre correspondant à une publication. Comme décrit ci-dessous, ces chapitres sont rangés par thèmes (ou concepts) des plus généraux aux plus spécifiques dans le contexte des services Web.

Théorie de la QoS pour les Orchestrations: Dans les orchestrations où il existe des interactions étroites entre la QoS, les données et la fonction, les architectures classiques pour prendre en compte QoS ne sont pas assez souples. Les Orchestrations dépendantes de données peuvent être non-monotones par rapport à la qualité de service, ce qui signifie que l'amélioration de la QoS d'un service particulier peut diminuer la qualité de service de bout-en-bout de l'orchestration. Dans le Chapitre 3, nous développons un calcul riche pour analyser la qualité de service multidimensionnelle dans un sens probabiliste. En utilisant le modèle *orchnets*, les conditions pour traiter la non-monotonie dans des orchestrations sont également étudiées. Ces techniques sont mises en oeuvre dans Orc en utilisant la QoS comme des aspects qui peuvent être «tissés» dans les spécifications fonctionnelles de l'orchestration. Dans le Chapitre 4, les règles pour tisser une approche de causalité temporelle et la QoS pour les expressions Orc sont présentées. En faisant usage de ces règles au dessus de la forme OIL (Orc Intermediary Language), les publications Orc peuvent être étendues afin de fournir passé causal et des incréments de QoS.

QoS pour les lignes de produit de Services Web: Les lignes de produits de services composites peuvent avoir des configurations multiples présentant un comportement variable du à la fois à l'incorporation/rejet de service et au comportement probabiliste de QoS. Batir les deux formes de variabilité conduit à une explosion combinatoire, en particulier lorsque de multiples services et des combinateurs d'orchestration sont utilisés. Le Chapitre 5 vise à étudier ces deux aspects de la variabilité et à faire comprendre leur effet sur la QoS de bout en bout et le SLA correspondant. Le Chapitre 6 compare l'utilisation de l'échantillonnage combinatoire avec un échantillonnage aléatoire en fonction de l'efficacité et de la stabilité des configurations générées. Ces techniques démontrent, de façon empirique, l'avantage d'utiliser des méthodes d'interaction combinatoires pour échantillonner à la fois l'invocation de variable et la QoS dans les grandes lignes de produits de services Web composites.

Outils QoS pour les Orchestrations: Alors que des langages comme BPEL et Orc ont mis l'accent sur des spécifications fonctionnelles (incorporer des combinateurs pour la concurrence, le choix et ainsi de suite), ils manquent d'outils pour intégrer des aspects plus intensifs mathématiquement. Dans le Chapitre 7, nous démontrons l'incorporation de paquets d'optimisation au sein des spécifications d'orchestration. Une conséquence de cela réside dans la meilleure prise de décision lorsque les flots de contrôle des orchestrations dépendent de modèles de QoS probabilistes multidimensionnels. Nous proposons également l'utilisation d'une procédure de requête de haut niveau et flexible pour intégrer ces optimisations dans des langages tels que Orc.

Amélioration de Service Level Agreements: Pour finir, l'incorporation de ces paramètres QoS peut conduire à des SLA supérieurs. Dans le Chapitre 8, nous nous concentrons sur des simulations Monte-Carlo pour estimer la QoS de bout en bout et les SLA pour les orchestrations. En cas de distributions à «queue lourde», les simulations de Monte-Carlo sont inefficaces pour estimer les quantiles extrêmes de

valeurs qui démontrent une forte variance. Des techniques de réduction de variance telles que importance sampling et importance splitting peut s'avérer être des alternatives plus efficaces, permettant une définition précise d'échantillonnage, de mesure et de tolérance d'écart dans les déclarations de SLA. Dans le Chapitre 9, nous étudions la négociation de SLA dans des orchestrations de services composites, en mettant l'accent sur la sélection optimale de la stratégie de renégociation pour l'amélioration de la QoS de bout en bout. Nous montrons qu'en formulant le problème sous forme de programmation en nombres entiers, les contraintes peuvent être spécifiées pour sélectionner le service qui offre la meilleure stratégie de renégociation.

Chapitre 3: Gestion des orchestrations monotones de services avec la QoS

Nous étudions la gestion des orchestrations de service avec la QoS, en particulier pour les orchestrations ayant un workflow dépendant des données. Notre étude supporte la QoS *multidimensionnelle*. Pour capturer l'incertitude dans la performance et la QoS, nous apportons un support pour la QoS *probabiliste*. Avec les hypothèses ci-dessus, les orchestrations peuvent être *non-monotones* par rapport à la qualité de service. Cela qui signifie que l'amélioration de la QoS d'un service peut réduire la qualité de service globale de l'orchestration. C'est une caractéristique embarrassante pour la gestion de QoS. Nous étudions la monotonie et fournissons les conditions suffisantes pour cela. Nous proposons ensuite une théorie complète et une méthodologie pour les orchestrations monotones. Les règles génériques de composition de QoS sont établies par un *calcul de QoS*, capturant aussi la meilleure liaison de service — cependant, la découverte de service n'est pas dans le cadre de ce travail.

La monotonie permet de justifier l'utilisation d'une approche contractuelle pour la gestion QoS. Bien que la fonctionnalité et la qualité de service ne puissent pas être séparées dans la conception des orchestrations complexes, nous montrons que notre architecture prend en charge la séparation de ces deux préoccupations en permettant de développer la fonctionnalité et la QoS séparément puis de les «tisser» ensemble pour obtenir l'orchestration à QoS améliorée. Notre approche est mise en oeuvre au dessus du langage Orc pour spécifier les orchestrations de service.

Contributions

1. Analyse de la QoS probabiliste dans les orchestrations transactionnelles qui dépendent des données et où la fonctionnalité et la QoS interagissent.
2. Mise en relief de la nécessité d'étudier la *monotonie* lors la définition de la QoS dans de telles orchestrations avec le développement d'un calcul riche basé sur *OrchNets*.
3. Un contexte théorique pour la pour monotonie avec des extensions pour gérer la monotonie probabiliste a été développé en utilisant OrchNets et de la ramification des cellules.
4. L'architecture permet la séparation des préoccupations (inspiré de Développement Orienté Aspect), qui permet de spécifier séparément les aspects fonctionnels et la qualité de service.
5. Mise en oeuvre de la théorie en utilisant le mécanisme de réécriture de règles où la qualité de service est «batie» au sein de la description fonctionnelle des orchestrations.
6. Mise en oeuvre de cette spécification SLA dans Orc avec les spécifications des augmentations de QoS et les opérations algébriques pour générer des contrats de bout en bout.
7. *Logiciel*: Le TravelAgent, par exemple, avec la déclaration de SLA, les spécifications fonctionnelles et le cachier des charges de la QoS «batie» précisés dans Orc est disponible à [2].

Publication

Ce document a été soumis à Springer Formal Methods in System Design (2012) [BJK+12].

[2] http://orc.csres.utexas.edu/papers/bjkrt2012fmsd.shtml

Chapitre 4: Leverage Causalité pour QoS suivi dans les systèmes des service orienté

Dans une orchestration, le contrôle de flux est passé à différents services en fonction de constructions comme le débit séquentiel / parallèle, si-sinon-alors ou timeouts. Dans le langage concurrent de programmation Orc, le flux de contrôle entre les `sites` circule à l'aide l'un des quatre combinateurs: `parallèle`, `séquentiel`, `réduction` et `Sinon`. Une `Expression` Orc combine les définitions de sites et les valeurs de ces combinateurs pour produire une spécification d'orchestration. Dans ce document, nous donnons des règles visant à transformer de tels programmes Orcs dans d'autres programmes qui permettent de suivre la causalité temporelle des évènements. Les évènements dans un programme Orc sont des publications, des appels ou des retours de sites. La causalité temporelle est présentée avec chaque évènement et peut être utilisée pour le diagnostic des pannes ou d'interblocage dans des orchestrations. Cela est étendu pour «bâtir » les valeurs de qualité de service ainsi que les règles pour suivre la causalité. La mise en oeuvre de cela est faite au-dessus d'Orc et la transformation est réalisée au niveau du language OIL (Orc Intermédiary Language).

Contributions

1. Fournit des règles de réécriture utilisant la syntaxe d'origine Orc pour suivre l'historique de causalité.
2. Fournit des règles pour batir la QoS théorique en tant que réécriture d'OIL.
3. Met en oeuvre les règles de réécriture dans Orc pour la causalité et la QoS.
4. *Logiciel*: Un prototype de l'implemntation a été fait au dessus de la version 2.0 d'Orc en Scala.

Publication

Une première version du document devant être soumis est présentée dans [JKTB12].

Chapitre 5: modélisation de la variabilité et analyse de la QoS des orchestrations de services Web

Le choix sans cesse croissant dans les services divers fait de *la variabilité de l'orchestration de service* un aspect essentiel d'un service web composite. L'effet de cette variation sur la qualité de service (QoS) d'un service composite est critique et c'est le centre de notre travail. Dans cet article, nous présentons une méthodologie pour la variabilité du modèle d'orchestration d'abord en utilisant un *feature diagram* (FD). Le FD spécifie une ligne de produits des orchestrations représentés comme des *configurations* de services atomiques invoqués/rejetés. Deuxièmement, en raison de l'ensemble potentiellement important de configurations, nous employons des techniques de test combinatoire pour générer automatiquement des configurations couvrant toutes les *interactions* par paires valides entre les services. Troisièmement, nous analysons la variation de QoS pour chaque configuration en utilisant des modèles probabilistes de QoS. En utilisant un *système de gestion de crise*, nous montrons expérimentalement que la génération de paires couvre toutes les valeurs aberrantes de QoS et élimine l'analyse de plus de 75% de toutes les configurations possibles. L'analyse de QoS des configurations par paires révèle des configurations dangereuses/inefficaces, aide à déterminer des Service Level Agreements (SLA) réalistes, et fournit un commentaire précieux pour aider à remodeler une orchestration.

Contributions

1. Analyse des aspects communs de la variabilité de ligne de produits et des distributions de QoS.
2. le comportement des variables dans l'invocation du service est capturé à l'aide des modèles caractéristiques.
3. Utilisation d'une technique d'échantillonnage par paires pour gérer l'explosion combinatoire.
4. L'échantillonnage par paires réduit de manière significative le nombre de configurations à analyser.
5. Un cas de gestion de crise est fourni puis analysé afin de mieux générer des SLA.
6. *Software*: La nouvelle génération d'échantillons qui satisfont les interactions par paires a été développé en Java et en langage Alloy pour la modélisation. Il est disponible en téléchargement à l'adresse [3]. Les simulations pour les distributions de QoS ont été effectuées dans MATLAB.

Publication

Une version de ce chapitre a été présentée à la Conférence internationale sur les services Web (ICWS) 2010 [KSB+10]. Des détails supplémentaires sur les produits générés et les détails expérimentaux sont inclus.

[3] http://pairwise-models.googlecode.com/svn/trunk/

Chapitre 6: Test par paires de services composites dynamiques

Les services en ligne regroupent les entreprises, les personnes, les systèmes logiciels et fonctionnent souvent dans des environnements mal compris. En utilisant ces services en tandem pour orchestrer de façon prévisible une tâche complexe est l'un des principaux défis de l'informatique orientée service. Une orchestration des services composite sollicitant de multiples services atomiques est en proie à un certain nombre de sources de variation. Par exemple, la disponibilité d'un service atomique et son temps de réponse sont deux importantes sources de variation. En outre, le nombre de variations possibles dans un service composite augmente de façon exponentielle avec l'augmentation du nombre de services atomiques. Le test d'un tel service composite présente un enjeu crucial puisqu'il est souvent très coûteux d'examiner de manière exhaustive l'espace de variation. Peut-on tester efficacement le comportement dynamique d'un service composite en utilisant uniquement un sous-ensemble de ces variations? C'est la question qui nous intrigue. Dans ce document, nous modélisons d'abord la variabilité du service composite sous forme de diagramme caractéristique (FD) qui capture toutes les configurations valides de son orchestration. Deuxièmement, nous appliquons le *test par paire* pour échantilloner l'ensemble des configurations possibles de façon à obtenir un sous-ensemble concis. Enfin, nous testons le service composite pour certaines paires de configurations avec une variété de paramètres QoS tels que le temps de réponse, la qualité des données, et la disponibilité. En utilisant de deux cas d'études, *la gestion des crises d'accident de voiture* et *la gestion de la cybersanté*, nous montrons que la génération par paires échantillone efficacement la gamme complète des variations de QoS dans une orchestration dynamique. La technique d'échantillonnage par paires élimine plus de 99% de redondance dans les configurations, tout en appelant tous les services atomiques au moins une fois. Nous évalueons rigoureusement les tests par paires pour les critères tels que: a) la capacité d'échantilloner des paramètres extrêmes de QoS du service, b) un comportement stable des configurations extraites, c) un ensemble compact de configurations qui peuvent aider à évaluer les compromis de QoS et d) la comparaison avec un échantillonnage aléatoire.

Contributions

1. Utilisation de larges cas d'études pour la situation de crise et de cybersanté où un échantillonnage exhaustif est impossible.

2. Analyse de l'échantillonnage par paires en ce qui concerne la taille, la stabilité et la capacité à générer de multiples familles de SLA.

3. Preuve empirique sur un échantillonnage aléatoire qui ne peut pas garantir une couverture suffisante sur toutes les interactions. Les perspectives sur le développement de plusieurs niveaux de SLA pour les familles de services composites sont également évaluées.

4. *Software*: La nouvelle génération d'échantillons qui satisfont les interactions par paires a été développée en Java et en langage Alloy pour la modélisation. Il est disponible en téléchargement à l'adresse [4]. Les simulations pour les distributions de QoS ont été effectuées sous MATLAB.

Publication

Une version de ce chapitre a été présentée lors du 6e colloque international sur le génie logiciel pour les systèmes adaptatifs et autogestionnaire, 2011 [KSB+11].

[4] http://pairwise-models.googlecode.com/svn/trunk/

Chapitre 7: Optimiser les décisions dans les orchestrations de services Web

Les orchestrations Services Web emploient en général une comparaison exhaustive des paramètres de QoS lors de la prise de décision. La capacité d'incorporer des packages mathématiques plus complexes est nécessaire, notamment dans les cas des workflows pour l'affectation des ressources et des systèmes de files d'attente. En modélisant de telles routines d'optimisation comme les appels de service dans les spécifications d'orchestration, des techniques telles que la programmation linéaire peut être commodément invoquée par des concepteurs de workflow non spécialistes. En s'appuyant sur la théorie de la QoS précédemment développée, nous proposons l'utilisation d'une procédure de requête haut niveau flexible pour intégrer des optimisations dans des langages tels que Orc. Le site *Optima* fournit une extension pour les opérations de tri et de réduction actuellement employées dans Orc. En outre, l'absence d'une technique objective pour consolider les paramètres de QoS constitue un problème dans l'identification des fonctions de coût appropriées. Nous utilisons l'AHP (*analytical hierarchical process*) pour générer un ordre total de paramètres de QoS à travers différents domaines. Avec des constructions pour assurer la *cohérence* sur des jugements subjectifs, l'AHP offre une technique appropriée pour produire des fonctions de coût objectives. En utilisant la chaîne d'approvisionnement Dell (*Dell Supply Chain*) par exemple, nous démontrons la faisabilité de la prise de décision par des routines d'optimisation, en particulier lorsque le flux de contrôle est dépendent de la QoS.

Contributions

1. Les hypothèses sur la monotonie et les obligations contractuelles permettent d'enrichir le meilleur opérateur Orc avec de l'optimisation à travers des domaines multiples de QoS.

2. Le site *Optima* peut être utilisé pour intégrer des solveurs dans Orc.

3. Utilise l'AHP pour générer un ordre total sur les domaines de qualité de service multiples. Item démontré l'intégration de l'optimisation au sein des spécifications de workflows à l'aide de la chaîne d'approvisionnement Dell par exemple.

4. Démontre les conséquences de l'optimisation en temps d'exécution sur la détection des violations de SLA.

5. *Logiciel*: Les simulations ont été réalisées en utilisant MATLAB et des solveurs d'optimisation intégrés. Un exemple du code MATLAB pour l'exemple de Dell dans Orc est indiqué dans l'annexe 11.3.

Publication

Un document correspondant à ce chapitre a été présenté à la Conférence internationale sur l'informatique orientée services, 2011 [KBJ11]. Des détails supplémentaires sur l'appel des bibliothèques mathématiques de Orc sont fournis.

Chapitre 8: Echantillonnage d'importance de types Importance Sampling ou Importance Splitting pour les contrats probabilistes dans les services Web

Avec des services Web de qualité de service (QoS) modélisées comme des variables aléatoires, la précision des valeurs échantillonnées pour des SLA spécifiques est remise en question. Les échantillons dont la répartition est plus faible sont plus précis pour le calcul des obligations contractuelles, ce qui n'est généralement pas le cas pour les services Web de QoS. En outre, les valeurs extrêmes en cas de distributions à queue lourde (par exemple 99,99 percentile) sont rarement observées dans les schémas d'échantillonnage limités. Pour améliorer la précision des contrats, nous proposons l'utilisation de techniques de réduction de variance comme l'échantillonnage d'importance de types Importance Sampling ou Importance Splitting. Nous démontrons cela pour des contrats impliquant les opérations de réclammations et de ravitaillement au sein de la chaîne d'approvisionnement Dell. En utilisant les valeurs mesurées, il est aussi possible d'effectuer des prévisions efficaces de la déviation future des contrats. Une des conséquences est une définition plus précise d'échantillonnage, de mesure et de tolérance de variance dans les déclarations de SLA.

Contributions

1. Démontre l'inefficacité des techniques traditionnelles de Monte-Carlo pour la composition contrat.
2. Utilise des techniques d'échantillonnage d'importance de types Importance Sampling ou Importance Splitting pour étudier la composition contrat dans l'exemple de Dell.
3. Propose d'étendre ces techniques pour la prévision des pannes ou des violations de contrats.
4. Etend les modèles actuels WSLA pour intégrer la variance dans le comportement de l'échantillonnage.
5. *Logiciel*: Les techniques d'échantillonnage d'importance de types Importance Sampling ou Importance Splitting ont été mises en oeuvre sous MATLAB et sont fournies dans l'annexe 11.4 pour une utilisation avec l'exemple de Dell.

Publication

Une version courte de ce chapitre a été présentée à la Conférence internationale sur l'informatique orientée services, 2011 [Kat11]. Ceette version a été étendue avec des détails supplémentaires et l'utilisation des techniques de type Importance Splitting.

Chapitre 9: Stratégies de négociation pour les contrats probabilistes dans les Orchestrations de services Web

Les Service Level Agreements (SLA) ont été proposés dans le cadre de services Web afin de maintenir une performance de QoS acceptable. Ceci est particulièrement crucial pour les orchestrations de services composites qui, pour fournir leur fonctionalité, peuvent invoquer plusieurs services atomiques. Gérer les SLAs implique d'utiliser des protocoles de négociation efficaces entre les orchestrations et les services invoqués. Dans les services composites où les données et la QoS (modélisée dans un cadre probabiliste) interagissent, il est difficile de sélectionner un service atomique individuel avec qui négocier. Une meilleure amélioration dans un domaine négocié (par exemple, la latence) pourrait signifier la détérioration dans un autre domaine (par exemple le coût). Dans ce document, nous proposons comme stratégie de renégociation au dessus de plusieurs services, une formulation de programmation en nombres entiers basée sur Le critère de la dominance stochastique d'ordre 1 (First Order Stochastic Dominance). Une conséquence de cela est une meilleure performance globale de l'orchestration par rapport aux stratégies aléatoires pour la re-négociation. Nous montrons aussi que cette stratégie optimale peut être appliquée à des protocoles de négociation spécifiés dans les langages comme Orc. Ces stratégies sont nécessaires pour les services composites où les contributions, à la QoS, des services atomiques varient de manière significative.

Contributions

1. Etude des stratégies optimales pour la négociation entre services sous-traités.
2. En utilisant une formulation de programmation en nombres entiers, les contraintes du critère de la dominance stochastique d'ordre 1 peuvent être spécifiées pour sélectionner le service qui offre la meilleure stratégie de re-négiociation.
3. La formulation améliore le meilleur opérateur pour la QoS dans les orchestrations monotones où la sélection optimale pour l'amélioration de services concurrentiels ne détériore pas la QoS globale.
4. Ceci est démontré dans l'exemple du GarageOnline, où une meilleure performance globale de l'orchestration est produite par optimum, comparé à la sélection aléatoire de services.
5. *Logiciel*: La comparaison des services en utilisant le critère de la dominance stochastique d'ordre 1 a été mise en oeuvre sous MATLAB et est représentée sous forme abrégée dans l'annexe 11.5.

Publication

Ce document a été soumis à la Conférence internationale de l'IEEE sur les services Web (ICWS) (2012) [KBJ12].

Part II

Flexible Quality of Service Management of Web Services Orchestrations.

Chapter 2

Introduction

The internet has had far reaching effects in making our lives easier. From online banking, *e*shopping, travel bookings to social networks - the number of applications have increased at a rapid pace. A principal catalyst in this has been the transformational software involved. *Web Services* are the industry focus to develop extensible, self-contained, modular applications that can be invoked over the internet. An example is a *Calender* or a *Word Processor* service hosted on a remote server that can be accessed over the internet. Web services expose the functionality of an information system while allowing easy integration due to the use of standard web protocols. The use of standard protocols reduces heterogeneity and promotes easy application integration across organizational boundaries.

A chief advantage is the paradigm of *Service Oriented Architectures*, that allows combining these individual web services into more complex interactions. Composite services can be created through the interaction of functionalities rendered by individual web services. *Web Services Orchestrations* provide a centralized control of the interaction, management and coordination involved in such composite services. An instance is a *travel booking* service that controls information flow between *airline booking*, *hotel booking* and *rental car* services. Service compositions may themselves become services, following a model of recursive service composition.

The control flow in such web services orchestrations is dependent on both data received from invoked services as well as *non-functional* properties. Hence, accurate description of *Quality of Service (QoS)* metrics is crucial in the analysis of such architectures. The QoS of a service can have many dimensions including response time, security of information transmitted, invocation cost, reputation of the service and so on. QoS might be a dominant factor over choosing alternative services offering similar functionalities.

To ensure efficient and reliable interactions between composed services, the notion of contractual obligations such as *Service Level Agreements* come into prominence. If one of the invoked services does not meet its QoS requirements, it can have deleterious effect on the possibly superior performance of sister services. Hence, it is critical for service providers to model, study and monitor non-functional properties of offered services. Once this is done, it is the imperative of the orchestration to optimally choose services that will maximize the QoS experienced by the client.

In this thesis, we study various aspects of reliable QoS management in web services. Some aspects considered are:

1. *Accurate modeling of QoS* - While considerable efforts have been given both by academia and industry to describe semantically accurate models of web services compositions, handling of QoS remains orthogonal to orchestration specifications. This cannot be done in case of orchestration flow dependent on QoS metrics such

as latency or security levels. Models are needed to integrate clearly the QoS aspects within orchestrations and how they influence control flow.

2. *Improved Contractual Agreements* - The QoS of the orchestration is dependent on the QoS of the services that it calls. Hence, it is important to contractually oblige services called to meet requirements in order for the orchestration to satisfy its own contracts. While some services have well studied QoS, others may be dependent on observations or should be accepted as-is. The QoS of the called services and the orchestration needs to be monitored at runtime, to ensure that they meet the desired levels, and have alternatives in case of deteriorating performance. A notion to consider here is non-monotonicity in behavior, when the improvement of one service may mean deteriorating overall behavior. End-to-end contracts can be based on simulation or analytic techniques. We make use of Monte-Carlo based techniques that may be slow and display high variance. Alternative techniques to improve the simulation process are needed.

3. *Optimization tools* - Once a set of services and their contractual descriptions are obtained, an orchestration can select from alternative services offering similar functionalities with varying QoS performance. Tradeoffs can be made to optimize the net latency or cost incurred by the orchestration. Negotiations may be involved to provide "better" contractual obligations to enhance a particular service's invocation within an orchestration. A further angle to be examined is providing better mathematical tools for orchestration designers. Optimization as a service can improve considerably the agility of web based management of complex workflows for logistics and operations research.

4. *Handling multiple aspects* - In product-line settings, multiple instances of composite services may be developed. There is added variability in the choice of services available along with QoS behavior. Techniques to analyze families of such services are needed. In transactional services, function and QoS interact tightly - leading to cases where they cannot be separated. Techniques are needed to handle them jointly, with functional specifications including QoS management.

Chapter Outline: This chapter is intended to serve as an introductory tutorial for web services, compositions and QoS management. An overview of web services and associated technologies (XML, UDDI, WSDL and SOAP) is presented in Section 2.1. Various aspects of web services' compositions are dealt with in Section 2.2 such as workflow models and formal representations of orchestrations. Two languages for specifying orchestrations: BPEL and Orc, are studied in Section 2.3. Section 2.4 studies in detail Quality of Service in web services, including QoS composition, contractual agreements, monitoring and optimization. This chapter ends with an outline of the rest of the thesis by giving the principal contributions in Section 2.5.

2.1 Web Services Technologies

This section briefly introduces web services and some of the technologies that surround them. A brief description of WSDL, UDDI, SOAP and REST are provided. A curious reader is referred to [ACKM04, TP02] for a comprehensive description of associated architecture.

As provided by the World Wide Web Consortium (W3C), a *Web Service* may be defined as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artifacts. A web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols" [W3c04b]. In general, web services are stateless

and can be invoked a number of times by clients without the need to share specific protocols. Each request is, in general, treated separately with differing quality attributes in responses.

While a web service is often seen as an application available to other such application over the web, this might lead to ambiguity over technologies available with a uniform resource locater (URL) or a script or as a application programming interface (API). For web services to be effectively used by multiple parties over the internet, the technologies must be platform and language agnostic. The benefits as compared to today's applications include [LPM+09]:

Easy and fast deployment: Enterprises using the Web Service model can provide new services and products without the investment and delays a traditional enterprise requires. They may develop new web services by reusing and/or combining existing ones.

Interoperability: Any web service can interact with other web services. This is achieved through a XML-based interface definition language and a protocol of collaboration and negotiation. This means that developers do not need to change their development environments in order to produce or consume web services.

Run-Time Integration: Traditional system architectures incorporate relatively brittle coupling between various components in the system. These systems are sensitive to change. A change in the output of one of the subsystems or a new implementation of a subsystem will often cause old, statically bound collaborations to break down. Web Services make it possible for significant decoupling and just-in-time integration of new applications and services, as they are based on the notion of building applications by discovering and orchestrating network-available services.

Reduced complexity: All components are services. What is important is the type of behavior a service provides, not how it is implemented. This reduces system complexity, as application designers do not have to worry about implementation details of the services they are invoking.

Web services architectures are based on three parties - the service requester, the service provider and a service registry. For such a model, there is a requirement to describe available services, provide a directory server and allow communication between the parties. In general, XML is used to tag the data, SOAP to transfer the data, WSDL to describe the web service and UDDI to list the available services as shown in Fig. 2.1. We further discuss some of these technologies.

2.1.1 XML: Extensible mark-up language

A common syntax that is used for all web services standards is provided by XML (Extensible mark-up language). In the context of web services, several protocols may be used: TCP/IP, HTTP or mail transfer protocols. The mechanism should be able to work with a variety of transport protocols. XML is a standard specification for defining the content of a computer message. If a software application writes its output in XML and another application is capable of interpreting XML, then it can read the output and act on it. XML allows specifying semi-structured data and querying them.

2.1.2 UDDI: Universal Description, Discovery and Integration

The Universal Description, Discovery and Integration (UDDI) [Oas04] is a group of web-based registries that expose information about a business or other entity. UDDI provides this extension to the basic Web Services technologies by allowing the means to

Figure 2.1: Clients invoke web services be exchanging SOAP messages. WSDL specifications are compiled into stubs and skeletons. Providers advertise their services on a UDDI registry that can be queried by the clients [ACKM04].

create a registry of web services. This takes web services into the realm of companies doing business with each other over the Internet. The UDDI specification enables companies to quickly, easily, and dynamically find and transact with one another. The UDDI provides an API that allows web service invocation itself. It may be used to:

- Find web services implementations that are based on a common abstract interface definition.
- Query web services providers that are classified according to a known classification scheme.
- Issue a search for services based on a general keyword.
- Cache information about a web service and then update at run-time.

The specifications for UDDI allow the creation and use of a registry containing information about businesses and the services they offer. The information is organized as follows [Mus04].

1. *Business entity* - A business entity represents information about a company. Each business entity contains a unique identifier, a short description of the company, some basic contact information, a list of categories and identifiers that describe the company, and a URL pointing to more information about the company.

2. *Business service* - Associated with the business entity is a list of business services offered by the business entity. Each business service entry contains a description of the service, a list of categories that describe the service, and a list of pointers to references and information related to the service.

3. *Specification pointers* - Associated with each business service entry is a list of binding templates that point to specifications and other technical information about the service. For example, a binding template might point to a URL that supplies information on how to invoke the service. It is also possible to use these pointers to access the service-level agreements that describe the contractual nature of the usage of the service.

4. *Service types* - A service type is defined by a *t*Model. Multiple companies can offer the same type of service, as defined by the *t*Model. A *t*Model specifies information such as the *t*Model name, the name of the organization that published the *t*Model, a list of categories that describe the service type, and pointers to technical specifications for the service type such as interface definitions, message formats, message protocols, and security protocols.

It is possible to use UDDI to create private registries that reside within private networks, offering functionality to a specific set of users.

2.1.3 WSDL: Web Services Description Language

Web Services Description Language (WSDL) [W3c01] provides a XML grammar for describing network services as endpoints capable of exchanging messages. WSDL service definitions provide documentation for distributed systems and describe web services interfaces. In addition, as services can be made available with different protocols, this information is contained in the WSDL service description.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the *abstract* definition of endpoints and messages is separated from their *concrete* network deployment or data format bindings. The following example shows the WSDL definition of a simple service providing stock quotes [W3c01]. The service supports a single operation called `GetLastTradePrice`, which is deployed using the SOAP protocol over HTTP. The request takes a ticker symbol of type string, and returns the price as a float.

Types - a container for data type definitions using some type system.

```
<types>
   <schema targetNamespace="http://example.com/stockquote.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
   <element name="TradePrice">
      <complexType>
         <element name="price" type="float"/>
      </complexType> </element> </schema>
</types>
```

Message - an abstract definition of the data being communicated.

```
<message name="GetLastTradePriceInput">
   <part name="body" element="xsd1:TradePriceRequest"/> </message>
<message name="GetLastTradePriceOutput">
   <part name="body" element="xsd1:TradePrice"/> </message>
```

Operation - an abstract description of an action supported by the service.

Port Type - an abstract set of operations supported by one or more endpoints.

```
<portType name="StockQuotePortType">
<operation name="GetLastTradePrice">
   <input message="tns:GetLastTradePriceInput"/>
   <output message="tns:GetLastTradePriceOutput"/>
</operation> </portType>
```

Binding - a concrete protocol and data format specification for a particular port type.

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
   <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
         <input> <soap:body use="literal"/> </input>
         <output> <soap:body use="literal"/> </output>
   </operation>
</binding>
```

Port - a single endpoint defined as a combination of a binding and a network address.

Service - a collection of related endpoints.

```
<service name="StockQuoteService">
    <documentation>My first service</documentation>
       <port name="StockQuotePort" binding="tns:StockQuoteBinding">
    <soap:address location="http://example.com/stockquote"/>
</port>  </service>
```

An advantage of WSDL is that it is a general purpose standard that can be used in a variety of environments. Established standards can be wrapped by WSDL interfaces to describe protocols and bindings. WSDL provides a function-centric description of web services, which has a drawback of being static. However, the order of messages or flow of data cannot be handled. Languages such as BPEL [OAS07] are needed along with WSDL to specify the order of message passing. WSCI (Web Services Choreography Interface) [W3c02] complements WSDL and can support such message passing. WSDL is also limited by specifying only syntactical aspects or services. Semantic languages such as OWL-S [W3c04a], try to combine these functionalities for automatic discovery, binding and composition of services.

2.1.4 SOAP

SOAP [W3c00] is a standard for sending messages and making remote procedure calls over the Internet. It defines the organization of information in XML in a typed and structured manner to allow exchange between peers. It specifies the following:

- An one-way message format for packaging information into a XML document.
- Conventions to define how clients can invoke remote procedures by sending a SOAP message and how services can reply by sending back a SOAP message.
- Rules to process and understand SOAP messages in XML format.
- A description of how SOAP messages can be transported over HTTP, SMTP and other protocols.

SOAP exchange information using messages encapsulated in an *envelope*.The envelope contains two parts: a *header* and a *body*. The core of the information that

the sender wants to transmit to the receiver is provided in the body. Additional information necessary for intermediate processing of services (security, adding new messages, etc.) are located on the header. An example [W3c00] is given below where a `GetLastTradePrice` SOAP request is sent to a `StockQuote` service with a HTTP request.

```
POST /StockQuote HTTP/1.1
 Host: www.stockquoteserver.com
 Content-Type: text/xml; charset="utf-8"
 Content-Length: nnnn
 SOAPAction: "Some-URI"

 <SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <symbol>DIS</symbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

An interested reader is referred to [W3c00] for further details on implementation, handling and specification of SOAP requests.

2.1.5 REST: Representational State Transfer

Representational State Transfer (REST) [Fie00] architectural style provides an alternative abstraction for publishing information and giving remote access to applications. REST-based or Restful web services are collections of web resources identified by uniform resource identifiers (URIs). A common implementation on HTTP/HTTPS provides four methods: `GET`, `PUT`, `POST`, and `DELETE`.
When comparing RESTful to SOAP/WSDL based services, the differences include:

- *Scalability*: In REST all interactions are stateless - each operation stands for itself. The representations of the resources contain all necessary information. This fact improves the applications' scalability. SOAP is also, in general, stateless; however, higher level standards exist to create stateful web services via SOAP.

- *Services Composition*: Strictly speaking, there are no REST services, only resources referred to by the URL. However, the interaction primitives (`GET`, `POST`, `PUT`, `DELETE`) stemming from REST can be used from within a BPEL process as new service invocation [Pau09].

- *Address Space*: REST uses a global address space with URLs, over which each resource can be addressed. SOAP messages are always addressed to an endpoint, which is implemented by a SOAP router.

- *Interface*: REST offers with the `GET`, `POST`, `PUT`, `DELETE` methods, a generic interface. In SOAP all methods for each application must be defined by the user.

2.2 Web Service Composition

A web service that is implemented by combining the functionality offered by many individual web services are called *composite* web services and the process of combining them is called *web service composition*. Applications are created by combining

the basic building blocks provided by other services. Service compositions may themselves become services, following a model of recursive service composition. Surveys on techniques and tools available for composite service architecture include [DS05, HS05].

This section provides an overview of Workflow management, orchestrations, choreographies and formal methods to represent them.

2.2.1 Workflow Management

Business Process refers to a collection of tasks performed in a coordinated manner between users and software applications to complete a task. A formal and executable representation of business processes may be called *workflows*. A *workflow management system* provides a software tool to design, execute and analyze workflows.

Workflow management systems [GHS95] aim to automate administrative processing of documents such as travel orders, financial statements and project progress reports. These systems also allow integration of heterogeneous and distributed participants, which is a requirement for large corporations. They can handle control and data flow between diverse parties and applications, which allow integration in most business scenarios.

Workflow management systems tend to be expressed in high level graphical languages rather than programming versions. This is to enable people with diverse skill sets to model workflows with considerable ease. Standards developed such as *Business Process Model and Notation (BPMN)* [OMG11] have received implementation in commercial workflow systems such as IBM ILOG Jviews [IBM11] and Oracle Business Process Management Suite [Ora11]. These have been extended to web-based workflow management systems such as Bonitasoft [Bon11], Signavio [Sig11] and Oryx [DOW08] business process editors.

Workflows are typically specified by a directed graph that defines the order of execution among nodes in the process. Nodes can be of the following types [ACKM04]:

1. *Work node* - Represents a work task to be performed by a user or an application.
2. *Routing node* - Defines the order in which the execution can be performed. This can include conditional, parallel and optional control flows.
3. *Start and End node* - Denote the beginning and completion of the workflow.

By combining these nodes, developers can specify the amount and order in which the tasks are executed. Contingency plans for failure may also be developed. Using several *instances* of the same workflow, several concurrent invocations of a similar workflow is also possible.

Examples of large workflows making use of web services' based framework for support include eGovernment [BRMO01], healthcare [LNS06] [AD05], logistics [FH05] and communication [AB06b] [GP07].

2.2.2 Orchestrations

Service-Oriented Architecture (SOA) is a paradigm with services modeled as computational entities that are autonomous and heterogeneous (e.g. running on different platforms or owned by different organizations). Applications both within and across organizational boundaries are integrated, avoiding difficulties due to different platforms, heterogeneous programming languages and so on. Exploiting this kind of ubiquitous network fabric should result in an increased productivity and in a reduction of costs in business-to-business (B2B) processes.

The terms *orchestration* and *choreography* describe two aspects of creating business processes from composite web services. As shown in Fig 2.2, an Orchestration always

represents control from one party's perspective. In most cases, it is centralized in handling of control flow which may be dependent on data received from other parties. This differs from choreography, which is more collaborative and allows each involved party to describe its part in the interaction. Choreography tracks the message sequences among multiple parties and sources rather than a specific business process that a single party executes.

Figure 2.2: An *Orchestration* refers to an executable process while a *Choreography* tracks the message sequences between parties and sources [Pel03].

An orchestration describes how web services can interact with each other, in a coherent manner. Particularly, it specifies the order in which services are invoked and the conditions under which a certain service may or may not be invoked. Orchestrations in the web services context must handle multiple dimensions of activities including the language used to specify compositions, data, transaction and exception handling. Other techniques proposed to look at orchestrations include those using decentralized control [CCMN04] and those based on sessions [FN08].

2.2.3 Formal Models for Service Orchestrations

Orchestrations of web services specify the order and conditions under which individual web services are invoked in a coherent composition. While there are many languages such as BPEL [OAS07], COWS [LPT07], Orc [KQCM09] and YAWL [tHvdAAR10] for representing workflows, they rely on the following models of orchestration:

2.2.3.1 Statecharts

Statecharts [Har87] are a formalism based on state machine that specify the activities performed when entering, exiting and while within a state. Conditional transitions firing dependent on data and control flow are also incorporated. Composite states, parallel actions and synchronization are additional complex actions present in statecharts. Statecharts possess a formal semantics for analyzing composite service specifications are are related to languages such as unified modeling language (UML) [DH01]. Statecharts offer most of the control-flow constructs and dependencies found in existing process modeling languages. For these reasons, statecharts have been used in the AgFlow architecture in [ZBN+04] to describe QoS dependent compositions. Associated formalisms to statecharts include Message Sequence Charts [HT03] for communication behavior in real-time systems.

2.2.3.2 Petri Nets

Petri nets [Mur89, Rei92] were introduced in as a framework to model concurrent systems. They are capable of modeling aspects such as concurrency, nondeterminism

and asynchronous behavior that can be applied to web services. A Petri net is a tuple (P, T, F, M_0) where:

- P is a finite set of places, denoted by circles.
- T is a finite set of transitions, denoted by rectangles. Note that $P \cap T =$
- F is a flow relation such that $F \subseteq (P \times T) \cup (T \times P)$.
- M_0 the initial marking, is a function $M_0 : P \to \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers including 0.

A transition $t \in T$ is enabled if each input place $p \in P$ has tokens in the initial marking M_0. Firing an enabled transition T, removes tokens from each input place p and adds tokens at each output place p'. Petri nets have behavioral properties that may be analyzed, such as:

Reachability - A marking M is said to be reachable from M_0 if there exists a sequence of firings of transitions $\sigma = T_1...T_K$ that transforms M_0 to M. The set of all reachable markings from M_0 is written as $R(M_0)$.

Boundness - A Petri net with initial marking M_0 is said to be k-bounded, if $M(p) \leq k$ for all $p \in P$ and $M \in R(M_0)$. If a net is 1-bounded is said to be *safe*.

Coverability - A marking M is said to be coverable from M_0 if there exists a marking $M' \in R(M_0)$ such that $M(p) \leq M'(p)$ for all places p in the net.

Liveness - A Petri net is said to be *live* if it is deadlock-free. A transition $t \in T$ is said to be: Dead (L0-live) if t can never be fired; L1-live if t can be fired at least once in some firing sequence; L2-live if given any $k \in \mathbb{N}$, then t can be fired at least k-times in some firing sequence; L3-live if t appears infinitely often in some firing sequence; Live (L4-live) if t is L1-live for every marking $M \in R(M_0)$.

Unfolding and analysis of coverability trees are other techniques to study properties of Petri nets.

As the research on petri nets has progressed, further enhancements have been proposed to model more complex systems, such as:

1. Timed Petri Nets [Wan98] - Obtained from Petri nets by associating a firing time to each transition of the net.
2. Stochastic Petri Nets [Mar89, BK02] - Uses probability distributions to model the firing times of transitions.
3. Coloured Petri Nets [JMW07, KCJ98] - Allows colours to be associated with tokens to distinguish them for complex systems.

Petri nets tend to have many advantages to model workflows for web services including partial order semantics to study unfoldings and complex constraints such as timed transitions. However, they cannot handle data dependent aspects of web services and may require coloured / timed Petri nets to model complex orchestrations.

Alternatives suggested include Workflow nets [vdABL08] used to model generic workflows with unique input and output places. In YAWL [tHvdAAR10], Petri nets are extended to support modeling standard workflow patterns [vdAtHKB02] which traditional Petri nets are unable to express. It supports multiple instantiations, provides built-in synchronization and cancellation capabilities, constructs typically found in orchestrations.

2.2.3.3 π-calculus

Process algebras are a popular means to describe and reason about process behaviors. Their underlying semantic foundation is based on labeled transition systems. The most well-known process algebras are Milner's Calculus of Communicating Systems (CCS [Mil89]) and Hoare's Calculus of Sequential Processes (CSP [Hoa04]). Like Petri nets, process algebras are precise and well-studied formalisms that allow the automatic verification of certain properties of their behaviors. Likewise, they provide a rich theory on bisimulation analysis, i.e. one can establish whether two processes have equivalent behaviors. Such analyses are useful to establish whether a service can substitute another service in a composition or to verify the redundancy of a service.

π-calculus [MPW92] is a process algebra that develops a formal model for describing processes. The rationale behind using π-calculus to describe processes lies in the advantages that a formal model with a rich theory provides for the automatic verification of properties expressed in such a model. For an orchestration, π-calculus provides constructs to call services sequentially, concurrently or dependent on conditional executions. For example, consider invocation of two web services A and B. Notations such as $A.B$ imply B is called after invocation of A; $A|B$ implies they are invoked in parallel; $A + B$ imply either of them are non-deterministically invoked; `while [var = value]`A denotes a conditional invocation of A dependent on $var = value$.

2.2.3.4 Composition Models

When a single service is unable to meet all the functional requirements of a client, generating a composite service is a possible option. The composition is concerned with synthesizing a specification of how to coordinate the individual services to fulfill the client request. Studies done to automatically synthesize such composite services include [HS05]:

OWL-S[W3c04a]: A key contribution of OWL-S is the modeling of how web services interact with the "real world". The basic building block of the OWL-S model is the notion of atomic and composite *processes*. OWL-S processes are specified to have inputs, outputs, pre-conditions, and conditional effects.

Roman[BCG+03]: Uses a abstract notion of atomic service in a finite-state automata framework for describing process flows. To specify the internal process flow of a web service, transition systems are used where each branch corresponds to a permitted sequencing of executions.

Message Based[BFHS03]: While the OWL-S and Roman models focus on what a service or composition does, neither completely addresses the issue of how the services in composition interact with each other. The notion of conversations or message passing is used in this model in a peer-to-peer framework.

Related formalisms to the models described above include FLOWS and COLOMBO. FLOWS [BGHM04] atomic processes are based on OWL-S atomic processes with inputs, outputs, pre-conditions and conditional affects. The flow of information between services can occur in two ways: (a) via message passing and (b) via shared access to the same database. Messages have types, which indicate the kind of information that they can transmit. COLOMBO [BCG+05] extends on some of the properties of FLOWS with an automata-based model of the internal behavior of web services, where the individual transitions correspond to atomic processes, message writes, and message reads. Colombo also includes a "local store" for each web service, used manage the data read/written with messages.

In [CH09], the notion of artifacts are developed with a data-centric rather than a control flow view. Such a view of composite services draws inspiration from the database community. Artifacts combine both data aspects and process aspects into a holistic unit, and serve as the basic building blocks from which models of business operations and processes are constructed. Artifacts are business-relevant objects that are created, evolved, and archived as they pass through a business.

2.3 Languages to Specify Orchestrations

In this section, we describe two languages used to specify orchestrations: the industry standard *BPEL* and the academic language *Orc*. Other languages like Yet Another Workflow Language [tHvdAAR10], generic web service combinators [CD99] and Calculus for Orchestration of Web Services [LPT07] exist and an inquisitive reader is referred to their references.

2.3.1 BPEL: Business Process Execution language

BPEL [OAS07] has become the de-facto standard language for describing compositions of web services. Others such as BPML [Ark02] and WSFL [Ley01] are less common alternatives. BPEL can be used for specification of both orchestrations and choreographies. They can define the sequence of messages sent / received by services as well as the ordering constraints on the send / receive actions. BPEL make use of a XML structure to specify the roles that take part in message passing, port types and message routing for orchestration specifications.

BPEL supports many *structured activities* that can have the following constraints among them:

1. `Sequence` - Activities that are executed one after the other in the order they are listed.

2. `Switch` - A set of activities that are specified with conditions. If the first activities' condition is true, it is executed while others are discarded. Using an *otherwise* activity, flow when none of the conditions are satisfied may be executed.

3. `Pick` - When one of the set of available processes are finished, the pick is considered complete. This can include the first responding service or a timeout.

4. `While` - Repeatedly executes while a condition is true.

5. `Flow` - Parallel execution of activities.

An orchestration engine can then execute this BPEL grammar, coordinating activities and compensating the overall process when errors occur. As shown in Fig. 2.3 the BPEL specification supports management of the overall process flow as well as activities that involve interactions with services external to the process itself.

An additional specification language proposed in the Web Services Choreography Interface [W3c02]. WSCI defines the overall choreography or message exchange between web services. WSCI describes only the observable behavior between web services, it does not address the definition of executable business processes as BPEL does. Extensions of XML based handling of data flow within choreographies have been proposed in [HB09, HB10], where active documents are created and modified by distributed peers.

Figure 2.3: A BPEL process flow [Pel03].

2.3.2 Orc

While BPEL has been incorporated industrially, the informal semantics results in parts of the language having different possible interpretations, which is quite undesirable for modeling orchestrations. A formal model for specifying and analyzing behaviors of web service orchestrations is essential and highly useful. Theoretical studies to formalize and study mathematically web services' behavior include [Rei08, BKM07]. BPEL has also received some attentional on operational semantics [Fah05] using Petri net [HSS05] and π-calculus [LM05] models. The translation of BPEL into workflow patterns in [BP06] provides interesting analysis such as comparison with YAWL.

Orc [Orc11, KQCM09, Mis10] is useful for modeling distributed computations but primarily targeted for modeling orchestration of web services over the internet. The language is based on a clean, mathematical framework and has a sound formal semantics. The simplicity of its syntax, yet its high expressibility makes it interesting to specify orchestrations. Orc implements the *tree programming* for orchestrations where in the initial query can be forwarded in parallel branches or cascaded in sequence to carry out the required task. The results from the sub-queries can be collected and forwarded to other sub-queries and finally returned to the initial caller. This paradigm nicely corresponds to the concept of an orchestration.

The Orc calculus is based on the execution of Orc *expressions*. Expressions are built up recursively using Orc's concurrent *combinators*. When executed, an Orc expression calls services and may publish values. Orc expressions use *sites* to refer to external services. A site may be implemented on the client's machine or a remote machine. A site may provide any service or be a proxy for interaction with an user.

2.3.2.1 Sites

The simplest Orc expression is a *site* call $F(p)$, where F is a site name and p is a list of parameters, which are values or variables. The execution of a site call invokes the service associated with F, sending it the parameters p. If the site responds, the call publishes that response. A site may give at most one response to a call. A site call may explicitly report that it will never respond, in which case we say that the call has *halted*. Some site calls may neither respond nor halt. Some examples of site calls are:

- `Println("hello world")` prints hello world to the console and publishes a `signal`.
- `add(3,4)` will add the numbers 3 and 4.
- `Random(10)` publishes a random integer from 0 to 9, uniformly distributed.

- `Prompt("Username:")` requests some input from the user, then publishes the user's response as a string. If the user never responds, the site waits forever.
- `Browse("http:www.google.com")` opens a browser window pointing to the web page and publishes a `signal`.
- `Rwait(420)` waits for 420 milliseconds, then publishes a `signal`.

Orc uses the expressions `signal` and `stop`. The expression `signal` just publishes a signal when executed and is equivalent to `if(true)`. The expression `stop` halts when executed and is equivalent to `if(false)`.

Though the Orc calculus itself contains no sites, there are a few fundamental sites which are so essential to writing useful computations. The site `let` is the identity site; when passed one argument, it publishes that argument, and when passed multiple arguments it publishes them as a tuple. The site `if` responds with a `signal` if its argument is true, and otherwise halts.

2.3.2.2 Combinators

Orc has four combinators to compose expressions: the parallel combinator $|$, the sequential combinator $>x>$, the pruning combinator $<x<$ and the otherwise combinator **;** . When composing expressions, the $>x>$ combinator has the highest precedence, followed by $|$, then $<x<$, and finally **;** with the lowest precedence.

1. **Parallel Combinator**: In $F \mid G$, expressions F and G execute independently. It initiates two independent computations; up to two values will be published depending on the number of responses received. The sites called by F and G are the ones called by $F \mid G$ and any value published by either F or G is published by $F \mid G$. There is no direct communication or interaction between these two computations. The parallel combinator is commutative and associative.

2. **Sequential Combinator**: In $F >x> G$, expression F is evaluated. Each value published by F initiates a separate execution of G wherein x is bound to that published value. Execution of F continues in parallel with these executions of G. If F publishes no values, no executions of G occur. The values published by the executions of G are the values published by $F >x> G$. The values published by F are consumed. The sequential combinator is right associative: $F >x> G >y> H$ is $F >x> (G >y> H)$. When x is not used in G, one may use the short-hand $F \gg G$ for $F >x> G$.

3. **Pruning Combinator**: In $F <x< G$, both F and G execute in parallel. Execution of parts of F which do not depend on x can proceed, but site calls in F for which x is a parameter are suspended until x is bound to a value. If G publishes a value, then x is assigned that value; the execution of G is terminated and the suspended parts of F can proceed. This is the only mechanism in Orc to block or terminate parts of a computation. The pruning combinator is left associative: $F <x< G <y< H$ is $(F <x< G) <y< H$. When x is not used in F, one may use the short-hand $F \ll G$ for $F <x< G$.

4. **Otherwise Combinator**: The execution of F **;** G starts with F. If F publishes no values and then halts, then G executes. We say that F halts if all of the following conditions hold:

   - All site calls in the execution of F have either responded or halted.
   - F will never call any more sites.

- F will never publish any more values.

The otherwise combinator is associative: $(F \;\mathbf{;}\; G) \;\mathbf{;}\; H$ is the same as $F \;\mathbf{;}\; (G \;\mathbf{;}\; H)$.

One of the most common concurrent idioms is a fork-join: evaluate two expressions F and G concurrently and wait for a result from both before proceeding. This is easy to express in Orc as (F, G) and is equivalent to $((x, y) <x< F) <y< G$. This implementation takes advantage of the fact that a tuple is constructed by a site call, which must wait for all of its arguments to become available.

2.3.2.3 Values, Definitions and Time

An Orc expression may be preceded by one or more declarations. Declarations are used to bind values to be used in that expression (or scope).

The declaration **val** $x = G$, followed by expression F, executes G, and binds its first publication to x, to be used in F. This is actually just a different way of writing the expression $F <x< G$. Thus, **val** shares all of the behavior of the pruning combinator. In fact, the **val** form is used much more often than the $<x<$ form, since it is usually easier to read.

The declaration **def** $E(x) = F$ defines a function named E whose formal parameter list is x and body is expression F. A call $E(p)$ is evaluated by replacing the formal parameters x by the actual parameters p in the body F. Unlike a site call, a function call does not suspend if one of its arguments is a variable with no value. A function call may publish more than one value; it publishes every value published by the execution of F. Definitions may be recursive.

Orc is designed to communicate with the external world, and one of the most important characteristics of the external world is the passage of time. Orc implicitly accounts for the passage of time by interacting with external services that may take time to respond. However, Orc can also explicitly wait for a specific amount of time, using the special site `Rwait`. The call `Rwait`(t), where t is an integer, responds with a signal exactly t milliseconds later.

This allows us to define interesting aspects of concurrent execution of sites:

Recursion: Expression definitions in Orc allows us to introduce recursion in programs adding significant expressibility. The expression name can appear in the body of the expression itself, recursively calling the expression. The following example defines a metronome, which publishes a `signal` once every t milliseconds, indefinitely.
def `metronome`(t) = `signal` | `Rwait`$(t) \gg$ `metronome`(t)

Timeout: The ability to execute an expression for at most a specified amount of time, is an essential ingredient of fault-tolerant and distributed programming. Orc accomplishes this using the $<x<$ and the `Rwait` site. The following program runs F for at most one second, publishing its result if available and the value 0 otherwise.
$x \;<x<\; (\; F \;|\; \texttt{Rwait}(1000) \gg 0 \;)$

Other aspects such as data types (`tuples`, `records`, `lists`), concurrency handling (`semaphores`, `channels`) and constructs to create new sites (`class`) are also made available for users of Orc. Details in [1].

[1] http://orc.csres.utexas.edu/documentation/html/refmanual/refmanual.html#N14CDB

2.3.2.4 Semantics

The abstract syntax of Orc is given in Table 2.1. The invocation of an orchestration occurs by calling the unique main *Expression* of an Orc program. It may contain statements on *Definitions*, *Values* and *Parameters* passed to the orchestration. The evaluation could return zero or multiple results.

$D \in$ Definition	$::=$	$\textbf{def}\ y(\bar{x}) = f$
$f, g, h \in$ Expression	$::=$	$p \mid p(\bar{p}) \mid ?k \mid$
		$f \mid g \mid f >x> g \mid f <x< g \mid f\ ;\ g \mid D\ f$
$v \in$ Orc Value	$::=$	$V \mid D$
$w \in$ Response	$::=$	$V \mid D \mid \textbf{stop}$
$p \in$ Parameter	$::=$	$V \mid D \mid \textbf{stop} \mid x$
$n \in$ Non-publication Label	$::=$	$V_k(\bar{v}) \mid k?w \mid \tau \mid \bot$
$l \in$ Label	$::=$	$!v \mid n$

Table 2.1: Abstract syntax of the Orc Calculus.

The internal semantics of Orc is presented in Table 2.2. The semantics is operational, asynchronous, and based on labeled transition systems. As is common in small-step operational semantics, the syntax of Orc must be extended to represent intermediate states. $?k$ is used to denote an instance of a site call that has not yet returned a value, where k is a unique handle that identifies the call instance. A publication event, $!v$, publishes a value v from an expression and τ denotes an internal event.

A site call involves three steps: invocation of the site, response from the site, and publication of the result. The Rule SiteCall specifies that a site call $V(\bar{v})$, where $\bar{v}$ is a value, transitions to $?k$ with event $V(\bar{v})$. The handle k connects a site call to a site return. A site call occurs only when its parameters are values; in $V(x)$, where x is a variable, the call is blocked until x is defined. In SiteRet a pending site call $?k$ receives a result w from the environment and transitions to the expression w. There is no assumption that all site calls eventually respond. The Publish rule generates a publication event $!v$ from its argument value v.

The rules DefDeclare and DefCall are evaluated using call-by-name in the DefDeclare rule. A single global set of definitions D is assumed with parameters $\bar{x}$ in DefCall producing an output g.

The Rules for the combinators are as described earlier. When f publishes a value ($f \xrightarrow{!v} f'$), rule SeqV creates a new instance of the right side; $[v/x]g$, the expression in which all free occurrences of x in g are replaced by v. The publication $!v$ is hidden, and the entire expression performs a τ action. Note that f and all instances of g are executed in parallel. Because the semantics is asynchronous, there is no guarantee that the values published by the first instance will precede the values of later instances. Instead, the values produced by all instances of g are interleaved arbitrarily.

Pruning is similar to parallel composition, except when g publishes a value v. In this case, rule PruneV terminates g and x is bound to v in f. One subtlety of these rules is that f may contain both active and blocked subprocesses: any site call that uses x is blocked until g publishes. In case of Rule OrtherV, if f publishes a value, the expression publishes the resulting evaluation f'. However, if the site f halts ($\bot$), we execute: `stop`; g.

Associated studies on orc include the Tree Semantics [HMM04], Trace Semantics [CM], Event Structure Semantics [RKB+08], Timed semantics [WKCM08] and Secure Information Flow [Thy09]. The translation of Orc into workflow patterns [CPM06a], allows comparison with other languages such as YAWL and BPEL.

$$\text{SiteCall}\ \frac{k \text{ fresh} \quad \bar{v} \text{ closed}}{V(\bar{v}) \xrightarrow{V_k(\bar{v})} ?k}$$

$$\text{SiteReturn}\ ?k \xrightarrow{k?w} w$$

$$\text{Publish}\ \frac{v \text{ closed}}{v \xrightarrow{!v} \textbf{stop}}$$

$$\text{DefDeclare}\ \frac{D \text{ is } \textbf{def}\ y(\ldots) = \ldots}{D\ f \xrightarrow{\tau} [D/y]\, f}$$

$$\text{DefCall}\ \frac{D \text{ is } \textbf{def}\ y(\bar{x}) = g}{D(\bar{p}) \xrightarrow{\tau} [D/y]\,[\bar{p}/\bar{x}]\, g}$$

$$\text{Par}\ \frac{f \xrightarrow{l} f'}{f \mid g \xrightarrow{l} f' \mid g}$$

$$\text{SeqN}\ \frac{f \xrightarrow{n} f'}{f >x> g \xrightarrow{n} f' >x> g}$$

$$\text{SeqV}\ \frac{f \xrightarrow{!v} f'}{f >x> g \xrightarrow{\tau} f' >x> g \mid [v/x]\, g}$$

$$\text{PruneLeft}\ \frac{f \xrightarrow{l} f'}{f <x< g \xrightarrow{l} f' <x< g}$$

$$\text{PruneN}\ \frac{g \xrightarrow{n} g'}{f <x< g \xrightarrow{n} f <x< g'}$$

$$\text{PruneV}\ \frac{g \xrightarrow{!v} g'}{f <x< g \xrightarrow{\tau} [v/x]\, f}$$

$$\text{OtherN}\ \frac{f \xrightarrow{n} f'}{f \,;\, g \xrightarrow{n} f' \,;\, g}$$

$$\text{OtherV}\ \frac{f \xrightarrow{!v} f'}{f \,;\, g \xrightarrow{!v} f'}$$

$$\text{OtherStop}\ \frac{f \xrightarrow{\perp} \textbf{stop}}{f \,;\, g \xrightarrow{\perp} g}$$

Table 2.2: Internal Structural Operational Semantic Rules of Orc.

2.3.2.5 Contributions

This thesis uses Orc to analyze orchestrations and their associated QoS. Orc, with its elegant and simple mathematical model can express a variety of orchestration patterns. In [BJK+12], we focus on extending Orc to handle QoS:

- A comprehensive framework supporting function and QoS together, and allowing for multi-dimensional QoS metrics.
- The framework allows separation of concerns (inspired from Aspect Oriented Development), that allows functional and QoS aspects to be specified separately.
- Making use of the rich QoS algebra, the "weaving" mechanism provides a new Orc program that provides both functional and end-to-end QoS output of the resulting orchestration.
- An improvement to the pruning operator in Orc is developed, that takes a generic QoS domain and returns the "best" QoS output.

The extension to tracking causality and QoS in Orc is presented in [JKTB12]. This information can be used for debugging of large workflows as well as generating partial order structures of executions. It is a generic technique that may be applied to other such concurrent systems with similar transformation rules.

- Using the notion of events (publications, site calls and returns) in Orc, rules are presented to track the causal history of Orc publications.
- Starting with an Orc program, the parsed intermediary form (Orc-intermediary-language (OIL)) is transformed to provide additional causal information.
- Causality can be used for program analysis and debugging of distributed systems.
- A similar technique can also append QoS contributions of event executions.

2.4 Quality of Service

This section reviews in detail, various aspects to consider in QoS. Starting from a general framework from QoS, we describe QoS composition, SLAs, their monitoring and negotiation and QoS dependent optimization.

2.4.1 QoS in Web Services

Two different aspects must be considered when talking about services:

1. The *functional* description contains the formal specification of what exactly the service can do.

2. The *non-functional* descriptions of how well the service performs these functions.

For example, in case of a train booking service, invoking its functionality (booking a train ticket) might be constrained by using a secure connection (security as non-functional property) or by actually performing the invocation of the services in a certain point in time (availability as non-functional property). For individual services, the functional and non-functional properties are treated orthogonally.

Moving on to composite services that invoke multiple services differing in functional and non-functional characteristics, the end-to-end characteristics are intricately linked with the orchestration's control flow. The end-to-end functional performance can rely on most of the services performing specific tasks and passing required data within control flow. However, in most orchestration, constraints on non-functional characteristics (timeout periods, security levels, etc.) also affect the end-to-end functional performance, and so, cannot be treated orthogonally. Non-functional properties might also play an important role in all service related tasks, especially in discovery, selection and substitution of services. It is simple to imagine a scenario in which services which can fulfill a user request and which provide basically the same functionality are selected based on some non-functional properties like price or performance.

Linking the end-to-end functional and non-functional characteristics to those of invoked services in an orchestration requires analysis. Data dependency and causality in the flow of an orchestration can provide insights into end-to-end functional performance (eg. failure of a certain service can lead to a deadlock). Increments in the end-to-end QoS and their relation to the control flow (invocation in parallel, sequentially, with constraints) can specify individual services' contribution to the end-to-end performance. With these in mind, the accurate modeling of QoS for web services is important to both functioning of the orchestration and specifying contractually defined non-functional behavior of invoked services.

QoS covers a whole range of techniques that match the needs of service requesters with those of the service provider's based on the network resources available. By QoS, we refer to non-functional properties of web services such as performance, reliability, availability, and security [TF06]:

Availability: Availability is the quality aspect of whether the web service is present or ready for immediate use. Availability represents the probability that a service is available. Larger values represent that the service is always ready to use while smaller values indicate unpredictability of whether the service will be available at a particular time.

Accessibility: Accessibility is the quality aspect of a service that represents the degree it is capable of serving a web service request. It may be expressed as a probability measure denoting the success rate or chance of a successful service instantiation at a point in time. There could be situations when a web service is available

but not accessible. High accessibility of web services can be achieved by building highly scalable systems. Scalability refers to the ability to consistently serve the requests despite variations in the volume of requests.

Integrity: Integrity is the quality aspect of how the web service maintains the correctness of the interaction in respect to the source. Proper execution of web service transactions will provide the correctness of interaction. A transaction refers to a sequence of activities to be treated as a single unit of work. All the activities have to be completed to make the transaction successful. When a transaction does not complete, all the changes made are rolled back.

Performance: Performance is the quality aspect of web service, which is measured in terms of throughput and latency. Higher throughput and lower latency values represent good performance of a web service. Throughput represents the number of web service requests served at a given time period. Latency is the round-trip time between sending a request and receiving the response.

Reliability: Reliability is the quality aspect of a web service that represents the degree of being capable of maintaining the service and service quality. The number of failures per month or year represents a measure of reliability of a web service. In another sense, reliability refers to the assured and ordered delivery for messages being sent and received by service requesters and service providers.

Regulatory: Regulatory is the quality aspect of the web service in conformance with the rules, the law, compliance with standards, and the established service level agreement. Web services use a lot of standards such as SOAP, UDDI, and WSDL. Strict adherence to correct versions of standards (for example, SOAP version 1.2) by service providers is necessary for proper invocation of web services by service requesters.

Security: Security is the quality aspect of the web service of providing confidentiality and non-repudiation by authenticating the parties involved, encrypting messages, and providing access control. Security has added importance because web service invocation occurs over the public Internet. The service provider can have different approaches and levels of providing security depending on the service requester.

Work by O'Sullivan et al. [SEH02] attempts to clarify non-functional properties and their relation to discovery, negotiation, invocation and execution of services. The non-functional properties covered include Temporal (when) and Spatial (where) availability; Broadcast channels used to deliver content; Cost structures (per request, commission); Settlement models for SLAs (subscription, metered, broker-based; Service quality (reliability, responsiveness, assurance, empathy, tangibles); Security and trust. The use of such non-functional criterion for discovery, composition and substitution of services is also elaborated in [SEH02].

Studies on QoS for web services have received considerable attention in papers such as [TF06, CMP+04, MM10, CMSA02, CSM+04, ACH98, AGM08], which provide a broad overview of QoS domains and their management. Studies on real time behavior of services latency and throughput include [CTB98, LNJV01, XSCT02]. Aspects such as security and digital credentials have been covered in [KM03, BMR08, KCE+04, BFG05, IM02]. Pricing strategies for web services are demonstrated in [HDHA09]. While these represent QoS metrics for orchestrations, choreographies can have extended models including analysis of message passing and deadlock freeness [MPR+10, BZ07].

2.4.2 QoS Composition

As compositions of services require combining functionalities of differing individual services, composing QoS of such services becomes equally important. Contracts or QoS values agreed upon with sub-contractors need to be aggregated and compared to develop an end-to-end QoS output. QoS composition is the process of aggregating the QoS of invoked (possibly sub-contracted) services in order for the orchestration to estimate its own performance for contractual obligations.

The orchestration can have details and models about its local resources and operations. It cannot however have complete information about remote services' performance or network usage. It will have to rely on contractual obligations to estimate performance or accept performance as-is (eg. accessing a public website). The composition of QoS derived from individual services requires a set of rules for aggregating QoS quanta. In [CMSA02, CSM+04, CPEV05a], the aggregation functions are presented using BPEL terminology in Fig. 2.4. Similar models for composing QoS across multiple domains is presented in [ZBN+04, ZBD+03, RLM+09].

QoS Attr.	Sequence	Switch	Flow	Loop
Time (T)	$\sum_{i=1}^{m} T(t_i)$	$\sum_{i=1}^{n} p_{ai} * T(t_i)$	$Max\{T(t_i)_{i \in \{1 \ldots p\}}\}$	$k * T(t)$
Cost (C)	$\sum_{i=1}^{m} C(t_i)$	$\sum_{i=1}^{n} p_{ai} * C(t_i)$	$\sum_{i=1}^{p} C(t_i)$	$k * C(t)$
Availability (A)	$\prod_{i=1}^{m} A(t_i)$	$\sum_{i=1}^{n} p_{ai} * A(t_i)$	$\prod_{i=1}^{p} A(t_i)$	$A(t)^k$
Reliability (R)	$\prod_{i=1}^{m} R(t_i)$	$\sum_{i=1}^{n} p_{ai} * R(t_i)$	$\prod_{i=1}^{p} R(t_i)$	$R(t)^k$
Custom Attr. (F)	$f_S(F(t_i))$ $i \in \{1 \ldots m\}$	$f_B((p_{ai}, F(t_i)))$ $i \in \{1 \ldots n\}$	$f_F(F(t_i))$ $i \in \{1 \ldots p\}$	$f_L(k, F(t))$

Figure 2.4: Aggregating QoS metrics for various domains and operations [CPEV05a].

Making use of Monte-Carlo simulations or analytic techniques, the end-to-end QoS for a single run of the composite service may be derived. However, this does not capture the probabilistic nature of some metrics, which requires multiple Monte-Carlo runs / historical data to derive distributions. Aggregation of probabilistic QoS metrics may be done using averaged values for latency, availability (as in Fig. 2.4) or use more mathematically intensive queuing and stochastic models. Stochastic approaches to composition and planning have been presented in [WHK08, WZZF09].

To further elaborate this notion of composing QoS, we make use of an example of a composite web service shown in Fig. 2.5. It demonstrates a typical ordering process for computer products. We elaborate on some aspects of control flow within the orchestration. The Check Order and Check Credit services check information provided by the client or a database to process the order. Once this is done, the fastest responding Hardware Supplier and the lowest costing Software Supplier are used for composition. Additionally a timeout operation can encompass the whole orchestration to prevent the client waiting for "halted" services.

Note that this is a typical workflow where both *data* from invoked services and *QoS* measured by the orchestration interact. For this reason, approaches using queuing theory [ALZH04, DB78] or stochastic models [TLY+04] for web services' performance have difficulty in accurately portraying behavior of transaction based services. These techniques are suited for network analysis and study of simplistic service compositions. Instead, simulation based treatment of composite services' performance can provide more realistic behavior. Examples of empirical studies of performance behavior of web services' behavior are [HM09, LNJV01].

Figure 2.5: A composite service orchestration demonstrating assembly of components.

When looking at composite services from a modeling point of view, product lines may be used [KDS+08, INPJ09]. Essentially, every concrete instance of a product line can be viewed as a composite service. This is specially true in large systems where parties may/ may not participate.

2.4.2.1 Contributions

Work on QoS composition is one of the areas studied in [BJK+12]:

- Rich theory is developed to handle probabilistic and multi-dimensional QoS composition.
- The abstract algebra supports increments $\oplus$, synchronization $\vee$, partial orders $\preceq$ and competition $\triangleleft$ in QoS metrics among invoked services.
- End-to-end measurements of QoS can be performed using this algebra and Monte-Carlo techniques - leading to distributions of QoS behavior of composite services.

Work on integrating aspects of variability (induced by inclusions/exclusions of services) and the relative impact on end-to-end QoS is studied in [KSB+10]:

- The variable behavior in service invocation is captured using feature models that can provide product lines of composite services.
- Handling both probabilistic QoS and such variability results in a combinatorial explosion of samples that need to be effectively handled.
- The composite services' behavior is analyzed using combinatorial interaction techniques covering pairs of services rather than exhaustive sampling.
- The demonstration on a crisis management system example shows that this can lead to efficient analysis of QoS behavior for families of composite services.

Subsequent work in this area is done in [KSB+11]:

- Tackles much larger case studies for crisis and e-health management where exhaustive sampling is impossible.

- Demonstrates the advantage of pairwise sampling: it guarantees pairwise interaction coverage with just a fraction of runs when compared to random sampling.
- As multiple sets of instances can satisfy a given feature model, analysis of the variance on results across different sets of solutions are also studied.
- Perspectives on developing multiple SLA levels for families of composite services are also evaluated.

2.4.3 Monotonicity in Orchestrations

An important aspect to consider here is monotonicity of the orchestration [BRBH08]. For any aggregation or contract based approach to be valid, the orchestration should be *monotonic*, that is, "if any service performs better, then so will the orchestration". When multiple domains of QoS and data interact, an improvement in the performance in one of the services can deteriorate overall performance nevertheless.

This is seen in a variant of Fig. 2.5, linking a particular hardware supplier with a particular software supplier. Improvement in response time of Hardware Supplier 1 can entail delaying the whole orchestration by being obliged to invoke the slow service Software Supplier 1 as shown in Fig. 2.6. Conditions to prevent this from happening is elaborated in [BRBH08] which include not linking a particular service with another (as seen in Fig. 2.6). It is important to consider this notion when using SLAs as a superior performing service (sub-contractor) may be unnecessarily penalized due to deteriorating performance by another service.

Figure 2.6: A *non-monotonic* orchestration.

Orchestrations that are not *data-dependent* (that is, control flow is not dependent on data returned from services), are in general always monotonic. However, for workflows that are data-dependent, a careful analysis of conditions for monotonicity is needed. In most cases, the implications of monotonicity has been ignored, which is not suitable for contract based management. Papers by except in Ardagna et al. [AP05], Alrifai & Risse [AR09a] and Zeng et al. [ZBN+04] identify the notion of "global versus local optimization", which is a check for monotonic conditions. In a general orchestration, global optimization provides the ideal case. In monotonic orchestrations, this can be refined with local optimization enough to ensure global optimality.

2.4.3.1 Contributions

A careful analysis of monotonicity and implications on QoS management are provided in [BJK+12]:

- Conditions to ensure monotonicity in case of data dependent orchestrations, when dealing with probabilistic QoS are provided.
- Using OrchNets and branching cells, a theoretical background for monotonicity is developed, with extensions to handle probabilistic monotonicity.
- Techniques to aggregate QoS are provided when monotonicity is not ensured (treating as a single transition, pessimistic evaluation). These techniques are crucial for contract composition is demonstrated on the TravelAgent example.

2.4.4 Service Level Agreements

In order to provide QoS support in web services, a number of extensions to current standards have been proposed. A proposal to publish QoS-enabled web services by registering them at the UDDI registry is shown in [Ran03, KBT+09, WVKT06]. A similar extension is provided by [D'A06], with a model-driven extension of QoS metrics within WSDL called Q-WSDL.

Service Level Agreement (SLA) (used synonymously with service contract or contractual obligations) defines a set of consumer expectations which must be met by a provider [TP05, SRE10]. Since providers will potentially be offering many different services to different consumers, they must adopt an efficient policy for resource management which differentiates consumers into service ranges.

A service level agreement is a document which defines the relationship between two parties: the provider and the recipient. A contract specifies, usually in measurable terms, what services the provider will furnish. Some metrics that contracts may specify include:

Service Definition - it describes the services and the manner in which those services are to be delivered. The information on the services must be accurate and contain detailed specifications of exactly what is being delivered.

Performance - It deals with monitoring and measuring service level performance. Essentially, every service must be capable of being measured and the results analyzed and reported. The benchmarks, targets and metrics to be utilized must be specified in the agreement itself.

Problems - To minimize the adverse impact of incidents and problems, there must be an adequate process to handle and resolve unplanned incidents. Formal records and logs must be maintained of all incidents and problems.

Customer Duties - These include the obligations of the customer with respect to the service and might include throughput and security constraints.

Providers would expose their promised performance in the form of contracts. Contracts are statistical in nature. A typical contract between a server and a client could be of the following form : "the service shall respond correctly 90% of the time, and, when responding correctly, in 90% of the cases it will respond with less than 2 seconds". In [PB08, TP05, SRE10, jJMS02] hard bounds are used for obligations. A bonus-malus policy is provided where a service deviating from a contract is punished.

An alternative to this approach is using probabilistic models as described in [HWSP04, HWTS07, RBHJ08]. As an example, we make use of an Orange API [2] to repeatedly test the response time of a *sms* service. The result of the response time distribution is presented in Fig. 2.7.

[2] http://api.orange.com/

Figure 2.7: Response time distribution of an API.

As specified in [RBHJ08, HWTS07], services specify their QoS behavior as a probabilistic distribution of metrics. The distribution can also be approximately specified by a set of quantiles. In some cases measurements can also be used to derive the probabilistic contract. An advantage of using distributions is that pessimistic approaches to contracts are removed with softer contracts used. Consider the services A and B with cumulative distributions F and G, respectively. A is said to first order dominate B iff:

$$F(x) \leq G(x) \quad \forall x \in \mathbb{R} \quad \Leftrightarrow \quad \mathbb{E}_F u(x) \geq \mathbb{E}_G u(x) \quad \forall u \in U \tag{2.1}$$

That is, there are more chances of being less than x if the random variable is drawn according to G than according to F. This allows for some amount of deviation on part of the provider for outlying QoS values that may cause contractual deviations. The advantage of using probabilistic contracts is a range of statistical tools such as stochastic dominance [BD03, And96] can be conveniently incorporated within SLA formulation.

In [FK98, Zsc10], a QoS specification language for distributed object systems is presented. This language, called Quality of service Quality Language (QML), is not restricted to any particular domain (e.g. real-time or multimedia systems) or to any particular quality of service. QML separates specification of QoS aspects from functional specification aspects. A useful mechanism in QML is the refinement mechanism that allows QoS aspects to be defined as refinements of existing ones. This notion is extended to contractual obligations in [FBH05]. The semi ring based Q-automata is another formal language to describe QoS properties of concurrent systems [CK07].

As QoS values are random variables, estimating end-to-end contracts is generally done through simulation runs. There are a few theoretical papers [ZYZB11] that propose aggregating the probability distributions, which is nearly impossible when orchestrations have control flow data dependencies. Models such as SALSA [BHS+10] (Simulated Annealing Load Spreading Algorithm) use queuing theory to autonomously meet SLAs, without a priori over-dimensioning resources.

2.4.4.1 Negotiation

SLA negotiations relates to the procedure of parties (consumer and providers) agreeing on the terms of an SLA. The parties try to reach a deal based on a consensus after exchanging (possibly several) non-binding "quotes". The typical negotiation steps are the following:

1. The provider publishes a template describing the service and its possible terms, including the QoS and possible compensations in case of violation. This template

leaves several fields blank or modifiable, which are meant to hold the user specific needs.

2. The client fetches the template, and fills it in with values which describe the planned resource usage. Some terms of the template may be removed or added or changed.

3. This new document, which engages neither party, is sent to the provider. Receiving this, the provider, based on the current resource availability and customer policies, sends back to the client a quote. This quote corresponds to values on which the provider would probably agree (but this is by no means binding), based on the client's needs.

4. The client, if satisfied with the quote, applies his/her signature to the document, and sends it back to the provider as a SLA proposal. Effectively, the client is already proposing an SLA to the provider, but the provider's signature is missing.

5. The provider, receiving the proposal, is free to reject or accept it. In the latter case, the proposal becomes an SLA officially signed by both parties, and starts to be a valid legal document.

The quotes exchange (steps 2 and 3) can be repeated any number of times. The user can tune the terms in the quote request until the provider's quote is in line with what the client is ready to accept. The last step for the client, step 4, requesting the real SLA, has a Boolean answer: the SLA is either accepted or rejected by the provider. In the latter case, the user can go back to asking for a quote, as the provider might have changed his conditions. The steps 2-3 are the core part of the negotiation, as each party can pull the deal in any direction. The parties may freely modify the different terms: lower fees, lower QoS, longer time slots, fewer resource needs, lower compensations, etc.

Once a contract has been signed and agreed, the necessity of changing it could be envisaged (see re-negotiation). In that case, the same quote framework could be used. The main difference comes from the already allocated resources, and the existence of a first contract to modify.

Work on automating procedures for generating such negotiating procedures include [CCP07, DDK+04, YKL+07]. The automatic negotiation process aims at identifying the maximum quality level admissible with respect to the user budget. [ZZ04] specify the rate at which probes to deployed services can be made without affecting runtime performance. In [RBHJ08], the procedure for contract negotiation entails agreement of two distributions - a client side *assumption* on the throughput rate and the service provider side *guarantee* on performance. In [BS09b], negotiation is said to proceed using relaxed constraints with algebraic operators modeled in a semi ring.

Techniques such as [BAM+08, TZCB08, CP06] combine both analysis of functional as well as performance of web services. They analyze properties such as WSDL description and proper functioning when overly invoked. As the number of services increases, analyzing all these properties can entail more involved testing strategies as demonstrated in [CP06].

2.4.4.2 Monitoring

Once a user and a Web Service have been brought together, service execution and associated properties need to be continuously monitored. Also, required adjustments are desirable to take place in real-time without affecting operations at the user's site. This is a challenging task as the Web Service may be running on a system that is not own or controlled by the user or running on an operating system that the user knows nothing

about. While monitoring may refer to many aspects of web services such as load, error handling [BGG04], we specifically refer to monitoring non-functional properties and corresponding contractual obligations.

The situation with SLA monitoring becomes more complicated in the case of composite Web Services. The properties of a composite Web Service are a function of the properties of its component services and the managers of the component Web Services may need to coordinate in a particular way. Some of the QoS metrics such as response time or resource utilization may be monitored by the service provider; others such as throughput and security of data may require probing the customers. Examples of work on this topic include [MRLD09, SMS+01, ZLC07].

Monitoring QoS can be applied on web services during different stages of a SOA System life cycle. During the Service discovery, monitoring is useful to obtain the real QoS for a reliable Web Service Selection. Once the service has been selected, monitoring can be used in order to ensure that the service is still fulfilling the promised QoS. Particularly, in *self*-healing SOA Systems, when this requirement violation happens, the SOA system might change the failing web service by another one. Obviously, monitor systems retrieve data only from basic metrics, since derived ones are calculated from the previous ones through a predefined rule.

Web Service Level Agreement Language [LKD+03] is a standard language for specifying agreements and monitoring protocols for SLAs. A WSLA agreement complements a service definition. While a service description (WSDL) defines the service interface relationship between a service and its using application, the WSLA defines the agreed performance characteristics and the way to evaluate and measure them. The WSLA provides input to the measurement and management system of an organization that checks and manages an organization's compliance with a WSLA. Both service provider and service customer may run their own instrumentation and measurement and management systems. Each organization may access measured metrics from various sources, such as server-side metrics from the provider and client-side metrics from the customer. The relationship between WSLA, client, provider and WSDL is shown in Fig. 2.8.

Figure 2.8: Role of a Web Service Level Agreement [LKD+03].

A WSLA comprises the following major parts:

Parties - describes the parties involved in the management of the Web Service.

```
<Parties>
     <ServiceProvider name="ACMEProvider">
     <Contact><City>Yorktown, NY 10598, USA</City></Contact>
     <Action xsi:type="WSDLSOAPOperationDescriptionType" name="notification"
      partyName="ZAuditing">
        <WSDLFile>Notification.wsdl</WSDLFile>
        <SOAPBindingName>SOAPNotificationBinding</SOAPBindingName>
        <SOAPOperationName>Notify</SOAPOperationName></Action>
     </ServiceProvider>

     <ServiceConsumername="XInc">
     <Contact><City>Hawthorne, NY 10532, USA</City></Contact>
     <Action xsi:type="WSDLSOAPOperationDescriptionType" name="notification"
      partyName="ZAuditing">
        <WSDLFile>Notification.wsdl</WSDLFile>
        <SOAPBindingName>SOAPNotificationBinding</SOAPBindingName>
        <SOAPOperationName>Notify</SOAPOperationName></Action>
     </ServiceConsumer>
  <\Parties>
```

Service Definitions - describe the services the WSLA is applied to. The service definitions represent the common understanding of the contracting parties of the structure of the service, in terms of operations and the service's parameters and metrics that are the basis of the SLA. It also includes the specification of the measurement of a service's metrics.

```
<ServiceDefinition>
   <Schedule name="hourlyschedule">
   <Period>
      <Start>2001-11-30T14:00:00.000-05:00</Start>
      <End>2001-12-31T14:00:00.000-05:00</End></Period>
   <Interval><Minutes>60</Minutes></Interval>
   </Schedule>

   <SLAParameter name="Availability_CurrentDownTime" type="long" unit="minutes">
      <Metric>CurrentDownTime</Metric>
      <Communication>
          <Source>Measurement</Source>
          <Push>Auditing</Push></Communication>
   </SLAParameter>
<\SeviceDefinition>
```

Obligations - define the service level that is guaranteed with respect to the SLAParameters defined in the service definition section. The promises to perform actions under particular conditions are also represented in this part.

```
<Obligations>
   <ServiceLevelObjective name="ContinuousDowntimeSLO">
      <Obliged>ACMEProvider</Obliged>
      <Validity>
         <Start>2001-11-30T14:00:00.000-05:00</Start>
         <End>2001-12-31T14:00:00.000-05:00</End></Validity>
      <Expression>
      <Predicate xsi:type="Less">
         <SLAParameter>Availability_CurrentDownTime</SLAParameter>
         <Value>10</Value></Predicate>
      </Expression>
      <EvaluationEvent>NewValue</EvaluationEvent>
   </ServiceLevelObjective>
</Obligations>
```

Alternatives to WSLA include SLAng [SLE04] that allows the specification of non-functional features of contracts between independent parties. [Men02, ZLC07, LZZX10] develop techniques for monitoring deviations in QoS performance from nominal behavior. Other proposals include extending the WSLA framework to offer, negotiate and monitor contractual agreements in service compositions [DDK+04].

In [RBHJ08], [Ros09], the one-sided Kolmogorov-Smirnov gap is used for monitoring deviation from contracts:

$$\delta(F_S, G_S) = \sup_{x \in X}(F_S(x) - G_S(x)) \tag{2.2}$$

where F_S is the distribution function according to the contract agreed with site S and G_S is the actual runtime distribution function of site S. If this gap is greater than a tolerance parameter, the contract is said to be deviated from.

Once QoS attributes to be monitored are specified, the composite service can perform runtime diagnosis and adaptation to improve end-to-end performance. As studied in [WWW+07], adaptations can be triggered by violation in QoS contracts or a overload on available resources. Adaptation in this case would mean offering a substitute service or changing some parts of the QoS guarantee until acceptable levels are reached.

2.4.4.3 Contributions

The QoS algebra proposed in [BJK+12] is also used to formally specify contractual obligations:

- The SLA defines the Qos domains and the associated algebra for various Orc combinators which are hen "weaved" with these QoS values to generate a tuple of functional and QoS values.
- The generated distributions (through Monte-Carlo) can be compared using both data-dependent and pessimistic techniques.
- The effect of monotonicity on such contractual obligations is demonstrated using the TravelAgent example.

Improving the efficacy of simulation techniques to estimate the end-to-end QoS is the focus of [Kat11]:

- Traditional Monte-Carlo cannot detect extreme values efficiently, which leads to imprecise contractual obligations on availability or throughput.
- Rare event simulation techniques (importance sampling/splitting) are applied to provide low variance measures of extreme quantiles.
- The use of these techniques are demonstrated on the *Dell supply chain* example for improving contract composition and forecasting of outages.
- Techniques to specify such extreme quantiles with low variance in languages such as WSLA is also provided.

Contributions to improving negotiations in composite services are presented in [KBJ12]:

- Selecting one of the participating services in an orchestration for improving end-to-end QoS behavior is the focus.
- An integer programming formulation based on first order stochastic dominance is proposed to pick such services.
- The formulation improves upon the "best" operator for QoS in monotonic orchestrations, where optimal selection for improvement competing services does not deteriorate end-to-end QoS.
- This is demonstrated on the GarageOnline example, where better end-to-end performance of the orchestration is produced by optimal compared to random selection of services.

2.4.5 QoS dependent Optimization

When considering multi-dimensional QoS, QoS domains are partially, not totally ordered. In order to deal with probabilistic comparison of QoS domains that are only partially ordered, *Theorem 1* of [TKO77] may be used. This allows stochastic comparison of random variables to be simplified to ordinary comparison as functions defined over the same experimental probability setting.

In case of optimized selection over multiple QoS domains, some sort of total ordering is needed to generate a minimizing cost functions. It is also possible to minimize over one domain while implicitly handling tradeoffs across others. An example of a QoS dependent selection can be done as in [PD06]. Grid workflows are used to model web services architectures. Each web service is modeled as a $G/G/k$ queue with infinite customer capacity, meaning the number of jobs that can wait in the queue of a web service is infinite. Table 2.3 provides the necessary parameters for the optimization.

i - web service type
j - index of competing web services
x_{ijy} - binary selection variable: workflow task y to be executed on web service j of service type i

R_{ij} - average response time for web service j
d_{iy} - deadline allocation of workflow task y assigned to web service of type i
c_{ij} - average cost for web service j
e_{iy} - expected cost of a task y allocated to service of type i
f_{ij} - reliability of web service j
g_{iy} - reliability requirement of a task y

Table 2.3: Parameters for web services workflow optimization.

The objective is to minimize the number of jobs failing to meet their QoS requirements. A penalty term h^T is introduced with constraint limits z.

$$\text{minimize} \quad h^T z \tag{2.3}$$

Subject to the following constraints:
Expected average response time must be less than the deadline allocation of the workflow task.

$$(R_{ij} - d_{iy})\ x_{ijy} \leq z^d_{iy} \quad \forall i, j, y \tag{2.4}$$

Expected average cost must be less than the allocated cost.

$$(c_{ij} - e_{iy})\ x_{ijy} \leq z^c_{iy} \quad \forall i, j, y \tag{2.5}$$

Reliability requirement of a task must be less than the web service reliability.

$$(g_{iy} - f_{ij})\ x_{ijy} \leq z^f_{iy} \quad \forall i, j, y \tag{2.6}$$

Constraints to assign a task to one and only one web service.

$$\sum_{j=1}^{N_i} x_{ijy} = 1 \quad \forall i, y \tag{2.7}$$

$$x_{ijy} \in \{0, 1\} \quad \forall i, j, y \tag{2.8}$$

$$z^d_{iy}, z^c_{iy}, z^f_{iy} \geq 0 \quad \forall i, y \tag{2.9}$$

The goal is to optimize the end-to-end (or global) QoS of the orchestration. Various optimization techniques have been proposed to this end. As global optimization requires considering all alternative paths when chaining service calls, it is an off-line task that

requires an algebra to statically evaluate the end-to-end QoS from the QoS of each called service.

[LB10], a framework for reputation-aware software service selection and a selection algorithm is devised for service recommendation, providing SaaS consumers with the best possible choices. In [CCGM06], a broker based architecture is used for service selection. By modeling this as a constrained optimization problem, where each QoS class is modeled by suitable constraints, maximizing overall QoS of a flow of requests is achieved. In [YZL07], the objective of service selection is modeled as a multidimension multichoice knapsack problem (MMKP). Similar selection based approaches generating optimal configurations of services are presented in [HMR10, XFZ08].

In [ZBN+04] local optimization versus global planning is studied. Different QoS parameters are studied (price, duration, reputation, availability, success rate) and composition rules are provided. An integer programming solution is proposed as in [KP09]. Similar problems and solutions are also studied in [AP05, AGI+06, AR09a, QTDC10, AR09b], where the notion of local constraints and global optimality over QoS domains are studied.

[CSM02, CSM+04] proposes a predictive QoS model that allows to compute the QoS of workflows from the QoS of their atomic parts. Individual QoS parameters are estimated for their minimum, maximum, and averaged values based on measurements. [AGM08] studies service selection in composite service using a queuing network paradigm. In [YRB+10] [KKK08] a service query framework is presented to select services with best QoS output from a large service space. The optimization strategy leverages a QoS-based cost model to select the best execution plans for a composite service. An interesting contribution of [CPEV08] is the technique of re-binding that on-line performs a replacement of a service when the latter performance deviates too much from its nominal behavior.

The use of more mathematical intensive techniques such as stochastic control [SDR09] and dynamic programming [Ber07] has also been proposed. In [ZNB+08, GNZ+06], these techniques are used for selection and planning of composite service selection. Rather than selecting the "best" service with short term information, a more involved dynamic programming approach is used. This notion removes the effect of non-monotonicity and focuses on overall QoS goals of the composite service. However, this technique can prove to be computationally intensive and not suitable when QoS metrics vary significantly from assumed distributions.

The use of Dynamic Programming and Divide-and-Conquer approaches for a query optimization framework is proposed in [YB07]. However, the composition of workflows and QoS aspects are treated together. This entails assumptions on the knowledge of web services' QoS values before generating possible service execution paths. Such a selection, composition and QoS querying mechanism is also employed in [USM06]. Web services are modeled as function calls with optimization over select-project-join ordering of the services to reduce total run time. In [AGP08] reconfiguration using dynamic programming constraints are used for resource allocation and load balancing.

2.4.5.1 Contributions

In [BJK+12], the "best" operator and conditions on monotonicity ensure local vs. global optimization. Instead of performing computationally intensive dynamic programming, the monotonic assumptions ensure an approximate optimal selection of services. Local decisions taken for improvement (either through optimization / exhaustive search) will not deteriorate end-to-end QoS as a result of this.

While significant work has been done in the areas of optimal service selection and optimal composition assignments dependent on QoS, the use of optimization packages

within workflows is missing. The use of these tools is important for resource allocation and load balancing in a variety of operational research problems as shown in [KBJ11]:

- The assumptions on monotonicity and contractual obligations allow the "best" operator in Orc to be extended with optimization across multiple QoS domains.
- Incorporation of the `Optima` site is proposed that enables mathematical packages to be called within Orc - this enables complex workflows such as the *Dell* choreography to be specified.
- In order to develop total ordering of QoS domains (required for optimization specifications), a multi-criterion decision process is used to generate objective cost functions from consistent subjective judgments.
- Using such a technique, complex manipulation of resources such as optimizing restocking policies can be developed.

2.5 Thesis Organization

This thesis is organized into multiple chapters, with each chapter corresponding to a publication. The chapters are arranged from general to specific themes in the web services' context as described below:

QoS theory for Orchestrations: In orchestrations where there are tight interactions between QoS, data, and function, classical frameworks for dealing with QoS are not flexible enough. Data dependent Orchestrations may be non-monotonic with respect to QoS, meaning that improving the QoS of an individual service may decrease the end-to-end QoS of the orchestration. In Chapter 3: QoS-Aware Management of Monotonic Service Orchestrations, we develop a rich calculus to analyze multi-dimensional QoS in a probabilistic sense. Using the model *orchnets*, conditions to handle non-monotonicity in orchestrations are also studied. These techniques are implemented in Orc using QoS as aspects that may be "weaved" into functional orchestration specifications. In Chapter 4: Leveraging Causality for QoS Tracking in Service Oriented Systems, rules for weaving causal history and QoS for Orc expressions are discussed. By making use of these rules over the Orc Intermediary Language (OIL) form, Orc publications can be extended to provide causal past and QoS increments.

QoS for Product lines of Web Services: Product lines of composite services can have multiple configurations displaying variable behavior due to both service incorporation/rejection and probabilistic QoS behavior. Tacking both forms of variability leads to a combinatorial explosion, specially when multiple services and orchestration combinators are used. Chapter 5: Variability Modeling and QoS Analysis of Web Services Orchestrations aims to study both these aspects of variability and understand their effect on end-to-end QoS and corresponding SLAs. Chapter 6: Pairwise Testing of Dynamic Composite Services compares the use of combinatorial sampling with random sampling with respect to efficiency and stability of generated configurations. These techniques demonstrate, empirically, the advantage of using combinatorial interaction methods for sampling both variable invocation and QoS in large product lines of composite web services.

QoS tools for Orchestrations: While languages such as BPEL and Orc have focused toward functional specifications (incorporating combinators for concurrency, choice and so on), toolkits to incorporate more mathematically intensive aspects are missing. In Chapter 7: Optimizing Decisions in Web Services Orchestrations, we demonstrate incorporation of optimization packages within orchestration specifications. A consequence of this is improved decision making when orchestrations' control flow are dependent on multi-dimensional, probabilistic models of QoS. We also propose the use of a high-level flexible query procedure for embedding such optimizations in languages such as Orc.

Improving Service Level Agreements: Finally, the incorporation of these QoS metrics can lead to superior SLAs. In Chapter 8: Importance Sampling/Splitting of Probabilistic Contracts in Web Services, we focus on Monte-Carlo runs for estimating end-to-end QoS and SLAs for orchestrations. In case of heavy-tailed QoS distributions, Monte-Carlo runs are inefficient to estimate the extreme quantiles of values that demonstrate high variance. Variance reduction techniques such as importance sampling and importance splitting can prove to be more efficient alternatives, enabling precise definition of sampling, measurement and variance tolerance in SLA declarations. In Chapter 9: Negotiation Strategies for Probabilistic Contracts in Web Services Orchestrations, we study the negotiation of

SLAs in composite service orchestrations, with emphasis on selecting the optimal re-negotiation strategy for end-to-end QoS improvement. We demonstrate that using an integer programming formulation, constraints may be specified to select the service that provides the best re-negotiation strategy.

The pages that follow contain abstracts of individual chapters along with their salient contributions.

2.5.1 Chapter 3: QoS-Aware Management of Monotonic Service Orchestrations

We study QoS-aware management of service orchestrations, specifically for orchestrations having a data-dependent workflow. Our study supports *multi-dimensional* QoS. To capture uncertainty in performance and QoS, we provide support for *probabilistic* QoS. Under the above assumptions, orchestrations may be *non-monotonic* with respect to QoS, meaning that improving the QoS of a service may decrease the end-to-end QoS of the orchestration, an embarrassing feature for QoS-aware management. We study monotonicity and provide sufficient conditions for it. We then propose a comprehensive theory and methodology for monotonic orchestrations. Generic QoS composition rules are developed via a *QoS Calculus*, also capturing best service binding—service discovery, however, is not within the scope of this work. Monotonicity provides the rationale for a *contract-based* approach to QoS-aware management. Although function and QoS cannot be separated in the design of complex orchestrations, we show that our framework supports separation of concerns by allowing the development of function and QoS separately and then "weaving" them together to derive the QoS-enhanced orchestration. Our approach is implemented on top of the Orc script language for specifying service orchestrations.

Contributions

1. Analyzes probabilistic QoS in data-dependent transactional orchestrations with function and QoS interacting.
2. Emphasis the need to study *monotonicity* in defining QoS in such orchestrations with the development of a rich calculus based on *OrchNets*.
3. Using OrchNets and branching cells, a theoretical background for monotonicity is developed, with extensions to handle probabilistic monotonicity.
4. The framework allows separation of concerns (inspired from Aspect Oriented Development), that allows functional and QoS aspects to be specified separately.
5. Implements the theory using the mechanism of rewriting rules where QoS is "weaved" within functional description of orchestrations.
6. Implements this SLA specification into Orc with specifications of increments in QoS and algebraic operations for generating end-to-end contracts.
7. *Software*: The TravelAgent example with the SLA declaration, functional specification and QoS "weaved" specification are specified in Orc and available at [3].

Publication

This paper was submitted to the Springer Formal Methods in System Design (2012) [BJK+12].

[3] http://orc.csres.utexas.edu/papers/bjkrt2012fmsd.shtml

2.5.2 Chapter 4: Leveraging Causality for QoS Tracking in Service Oriented Systems

Service Oriented Architectures (SOA) allow individual software components (*services*) to autonomously describe their functionalities in order to be composed into more complex services. Such compositions can involve many structured interaction paradigms such as concurrency, access control mechanisms and shared memory references. *Causality* of events in such environments is a crucial tool for analysis of event traces, diagnosis of faulty behavior and more generally tracking Quality of Service (QoS) metrics. In this paper, we make use of the concurrent programming language *Orc* to describe interactions in service oriented systems. Building on the formal semantics of Orc, we provide transformation rules to equip Orc events with their causal histories. As a consequence, we demonstrate that QoS increments produced by events can be tracked in service oriented systems. The transformations are implemented as rewriting rules over the Orc Intermediary Language (OIL), which is an abstract syntax tree using only the core Orc calculus.

Contributions

1. Enhances events in distributed executions with causality.
2. Provides rewriting rules using the original Orc syntax to track causal history.
3. Provides rules to weave QoS theory as rewriting of OIL.
4. Implements the rewriting rules in Orc for causality and QoS.
5. *Software*: A prototype of the causality and QoS tracking implementation has been done over the Orc ver 2.0 in Scala with examples shown in the paper.

Publication

A first version of the paper submitted to the 9th International Workshop on Web Services and Formal Methods(2012) is presented in [JKTB12].

2.5.3 Chapter 5: Variability Modeling and QoS Analysis of Web Services Orchestrations

The ever-growing choice in diverse services is making *service orchestration variability* an essential aspect of a composite web service. Influence of this variation on the Quality of Service (QoS) of a composite service is critical and the focus of our work. In this paper, we present a methodology to first model orchestration variability using a *feature diagram* (FD). The FD specifies a product line of orchestrations represented as *configurations* of invoked/rejected atomic services. Second, due to the potentially large set of configurations we employ combinatorial testing techniques to automatically generate configurations covering all valid *pairwise interactions* between services. Third, we analyze QoS variation for each configuration using probabilistic models of QoS. Using a *crisis management system* case study we experimentally show that pairwise generation covers all QoS outliers and eliminates analysis of $> 75\%$ of all possible configurations. The QoS analysis of the pairwise configurations reveals unsafe/ineffective configurations, helps determine realistic Service Level Agreements (SLAs), and provides valuable feedback to help remodel an orchestration.

Contributions

1. Analyses joint aspects of product line variability and QoS distributions.
2. Variable behavior in service invocation is captured using feature models.
3. Uses a pairwise sampling technique to handle the combinatorial explosion.
4. Pairwise sampling significantly reduces the number of configurations to be analyzed.
5. Provides a crisis management case study that is analyzed to better generate SLAs.
6. *Software*: The generation of samples that satisfy pairwise interactions was developed in Java and the Alloy language for modeling. It is available for download at [4]. The simulations for the QoS distributions were performed in MATLAB.

Publication

A version of this chapter was presented at the International Conference on Web Services (ICWS) 2010 [KSB+10]. Additional details on the products generated and the experimental details are included.

[4] http://pairwise-models.googlecode.com/svn/trunk/

2.5.4 Chapter 6: Pairwise Testing of Dynamic Composite Services

Online services encapsulate enterprises, people, software systems and often operate in poorly understood environments. Using such services in tandem to predictably orchestrate a complex task is one of the principal challenges of service-oriented computing. A composite service orchestration soliciting multiple atomic services is plagued by a number of sources of variation. For instance, availability of an atomic service and its response time are two important sources of variation. Moreover, the number of possible variations in a composite service increases exponentially with increase in the number of atomic services. Testing such a composite service presents a crucial challenge as its often very expensive to exhaustively examine the variation space. Can we effectively test the dynamic behavior of a composite service using only a subset of these variations? This is the question that intrigues us. In this paper, we first model composite service variability as a feature diagram (FD) that captures all valid configurations of its orchestration. Second, we apply *pairwise testing* to sample the set of all possible configurations to obtain a concise subset. Finally, we test the composite service for selected pairwise configurations for a variety of QoS metrics such as response time, data quality, and availability. Using two case studies, *Car crash crisis management* and *eHealth management*, we demonstrate that pairwise generation effectively samples the full range of QoS variations in a dynamic orchestration. The pairwise sampling technique eliminates over 99% redundancy in configurations, while still calling all atomic services at least once. We rigorously evaluate pairwise testing for the criteria such as: a) ability to sample the extreme QoS metrics of the service b) stable behavior of the extracted configurations c) compact set of configurations that can help evaluate QoS tradeoffs and d) comparison with random sampling.

Contributions

1. Unlike Chapter 5, uses large case studies for crisis and e-health management where exhaustive sampling is impossible.
2. Provides analysis of pairwise sampling with respect to size, stability and ability to generate multiple families of SLAs.
3. Empirically provides evidence over random sampling that cannot guarantee sufficient coverage over all interactions.
4. Perspectives on developing multiple SLA levels for families of composite services are also evaluated.
5. *Software*: The generation of samples that satisfy pairwise interactions was developed in Java and the Alloy language for modeling. It is available for download at [5]. The simulations for the QoS distributions were performed in MATLAB.

Publication

A version of this chapter was presented at the 6th international symposium on Software engineering for adaptive and self-managing systems, 2011 [KSB+11]. Additional figures and experiments are included.

[5] http://pairwise-models.googlecode.com/svn/trunk/

2.5.5 Chapter 7: Optimizing Decisions in Web Services Orchestrations

Web services orchestrations conventionally employ exhaustive comparison of runtime quality of service (QoS) metrics for decision making. The ability to incorporate more complex mathematical packages are needed, especially in case of workflows for resource allocation and queuing systems. By modeling such optimization routines as service calls within orchestration specifications, techniques such as linear programming can be conveniently invoked by non-specialist workflow designers. Leveraging on previously developed QoS theory, we propose the use of a high-level flexible query procedure for embedding optimizations in languages such as Orc. The *Optima* site provides an extension to the sorting and pruning operations currently employed in Orc. Further, the lack of an objective technique for consolidating QoS metrics is a problem in identifying suitable cost functions. We employ the *analytical hierarchy process (AHP)* to generate a total ordering of QoS metrics across various domains. With constructs for ensuring *consistency* over subjective judgments, the AHP provides a suitable technique for producing objective cost functions. Using the *Dell Supply Chain* example, we demonstrate the feasibility of decision making through optimization routines, specially when the control flow is QoS dependent.

Contributions

1. The assumptions on monotonicity and contractual obligations allow the "best" operator in Orc to be extended with optimization across multiple QoS domains.
2. The site *Optima* can be used to integrate optimization solvers within Orc.
3. Uses the AHP to generate a total ordering over multiple QoS domains.
4. Demonstrated integrating optimization within workflow specifications using the *Dell* supply chain example.
5. Demonstrates the implications of runtime optimization on detection of SLA violations.
6. *Software*: The simulations were performed using MATLAB and integrated optimization solvers. An example of the code for the Dell example in Orc and MATLAB is shown in Appendix 11.3.

Publication

A paper corresponding to this chapter was presented at the International Conference on Service Oriented Computing, 2011 [KBJ11]. Additional details on invoking mathematical libraries from Orc are provided.

2.5.6 Chapter 8: Importance Sampling/Splitting of Probabilistic Contracts in Web Services

With web services quality of service (QoS) modeled as random variables, the accuracy of sampled values for precise service level agreements (SLAs) come into question. Samples with lower spread are more accurate for calculating contractual obligations, which is typically not the case for web services QoS. Moreover, the extreme values in case of heavy-tailed distributions (eg. 99.99 percentile) are seldom observed through limited sampling schemes. To improve the accuracy of contracts, we propose the use of variance reduction techniques such as importance sampling and importance splitting. We demonstrate this for contracts involving *demand* and *refuel* operations within the *Dell* supply chain example. Using measured values, efficient forecasting of future deviation of contracts may also be performed. A consequence of this is a more precise definition of sampling, measurement and variance tolerance in SLA declarations.

Contributions

1. Demonstrates inefficiencies of traditional Monte-Carlo techniques for contract composition.
2. Applies available importance sampling / splitting techniques to study contract composition in the *Dell* example.
3. Proposed extending these techniques to forecast outages or contract violations in future.
4. Extends current WSLA models to incorporate variance in sampling behavior.
5. *Software*: The importance sampling/splitting techniques were implemented in MATLAB and are provided in Appendix 11.4 for use with the Dell example.

Publication

A short version of this chapter was presented at the International Conference on Service Oriented Computing, 2011 [Kat11]. This has been extended with additional details and the use of importance splitting techniques.

2.5.7 Chapter 9: Negotiation Strategies for Probabilistic Contracts in Web Services Orchestrations

Service Level Agreements (SLAs) have been proposed in the context of web services to maintain acceptable quality of service (QoS) performance. This is specially crucial for composite service orchestrations that can invoke many atomic services to render functionality. A consequence of SLA management entails efficient negotiation protocols among orchestrations and invoked services. In composite services where data and QoS (modeled in a probabilistic setting) interact, it is difficult to pick an individual atomic service to negotiate with. A superior improvement in one negotiated domain (eg. latency) might mean deterioration in another domain (eg. cost). In this paper, we propose an integer programming formulation based on first order stochastic dominance as a strategy for re-negotiation over multiple services. A consequence of this is better end-to-end performance of the orchestration compared to random strategies for re-negotiation. We also demonstrate this optimal strategy can be applied to negotiation protocols specified in languages such as Orc. Such strategies are necessary for composite services where QoS contributions from individual atomic services vary significantly.

Contributions

1. Studies optimal strategies for negotiation among sub-contracted services.
2. Using an integer programming formulation, first order stochastic dominance constraints may be specified to select the service that provides the best re-negotiation strategy.
3. The formulation improves upon the "best" operator for QoS in monotonic orchestrations, where optimal selection for improvement competing services does not deteriorate end-to-end QoS.
4. This is demonstrated on the GarageOnline example, where better end-to-end performance of the orchestration is produced by optimal compared to random selection of services.
5. *Software*: The comparison of services based on first order stochastic dominance was implemented in MATLAB and is shown in a shortened form in Appendix 11.5.

Publication

This paper was accepted at the IEEE International Conference on Web Services (ICWS) (2012) [KBJ12].

Chapter Dependencies

Though most chapters are self contained, a reader is suggested some background reading:

- Chapter 3: QoS-Aware Management of Monotonic Service Orchestrations - self contained.
- Chapter 4: Causality and QoS Management in Orc - requires Chapter 3 to understand QoS domains and weaving within Orc.
- Chapter 5: Variability Modeling and QoS Analysis of Web Services Orchestrations - self contained.
- Chapter 6: Pairwise Testing of Dynamic Composite Services - self contained; useful to read Chapter 5 to understand some of the experiments.
- Chapter 7: Optimizing Decisions in Web Services Orchestrations - self contained.
- Chapter 8: Importance Sampling/Splitting of Probabilistic Contracts in Web Services - self contained.
- Chapter 9: Negotiation Strategies for Probabilistic Contracts in Web Services Orchestrations - requires Chapter 3 to understand monotonicity.

2.6 Future Work

Listed here are some topics that may be investigated in future in line with some of the contributions of the thesis:

QoS in Choreographies - While techniques to study QoS in orchestrations (Chapters 3, 4) have been examined in this thesis, the extension to choreographies involves other aspects (deadlock freeness, heterogeneous QoS measures, resource allocation). Extending some of the concepts developed in this thesis (such as causality) may be done for such applications. Contract compositions for large scale choreographies is also a topic that needs analysis - as the interactions are more complex with message passing, more stringent monitoring techniques may be needed. The techniques outlined in Chapter 8 can be upgraded to suit such contract composition techniques.

Security - While security has been studied in Chapter 3, the further study of protocols needs emphasis. In a composite service scenario, access control frameworks would need to be developed to select the *types* of data that can be accessed by particular classes of services. A natural extension of security policies will be reputation with companies offering such protocols becoming reputed "brands" of service providers.

Resource Management - With cloud computing frameworks coming into the forefront, the relationship between Software-as-a-Service (SaaS) and lower layers such as databases (Platform-as-a-Service: PaaS) or servers (Infrastructure-as-a-Service: IaaS) needs analysis. Policies on QoS management at the SaaS level would translate to queue management at PaaS and efficient energy usage at the IaaS levels. Such policies require a deeper understanding of tradeffs and optimized resource management with general tools such as in Chapter 7 proving useful.

Industrial Scenarios - Studies of actual contractual agreements for large companies (IBM, Amazon web services) could lead to realistic models of QoS - mimicking subjective measures used by clients. Using monitored contractual data (cost of contract, recovery time since downtime, availability), better strategies for resource allocation to prevent contract violations may be envisioned. Some aspects of productline variability (Chapter 5) and contract composition (Chapter 8) can be extended to support these requirements.

Automated Negotiations - Automating some features of the negotiation protocols described in Chapter 9 can be useful for runtime integration of discovered services. The called service should have contractual obligations that can be easily described and added on to the current orchestration specification.

Chapter 3

QoS-Aware Management of Monotonic Service Orchestrations

Albert Benveniste, Ajay Kattepur
IRISA/INRIA, Campus Universitaire de Beaulieu,
Rennes-Cedex, France.

Claude Jard
ENS Cachan, IRISA, Université Européenne
de Bretagne, Bruz, France.

Sidney Rosario, John Thywissen
Dept. of Computer Science,
The University of Texas at Austin, U.S.A.

Abstract

We study QoS-aware management of service orchestrations, specifically for orchestrations having a data-dependent workflow. Our study supports *multi-dimensional* QoS. To capture uncertainty in performance and QoS, we provide support for *probabilistic* QoS. Under the above assumptions, orchestrations may be *non-monotonic* with respect to QoS, meaning that improving the QoS of a service may decrease the end-to-end QoS of the orchestration, an embarrassing feature for QoS-aware management. We study monotonicity and provide sufficient conditions for it. We then propose a comprehensive theory and methodology for monotonic orchestrations. Generic QoS composition rules are developed via a *QoS Calculus*, also capturing best service binding—service discovery, however, is not within the scope of this work. Monotonicity provides the rationale for a *contract-based* approach to QoS-aware management. Although function and QoS cannot be separated in the design of complex orchestrations, we show that our framework supports separation of concerns by allowing the development of function and QoS separately and then "weaving" them together to derive the QoS-enhanced orchestration. Our approach is implemented on top of the Orc script language for specifying service orchestrations.

3.1 Introduction

Quality of Service (QoS) is becoming increasingly important for the use of composite services in business processes and workflows [CSM02, CSM+04]. QoS-aware management of workflows and business processes proves to be advantageous in four composite service management activities [CSM+04]:

1. QoS-based design,
2. QoS-based on-line service selection,
3. QoS monitoring, and
4. QoS-based adaptation or reconfiguration.

Various approaches [ST07, HWSP04, HWTS07, MCD08, ZYZB11, CGK+11] can be used to address the above activities depending on the assumptions made:

Process Definition The workflow process underlying the composite service can be statically defined and independent of the data exchanged while the service is being executed. Alternatively, the workflow can be data-dependent, e.g., by involving "if-then-else" branching, also referred to as "switch" (the BPEL construct) or "conditional" by some authors. A number of authors abstract this branching as a probabilistic choice.

QoS Dimensions Quite often, several dimensions for QoS must be handled, leading to the consideration of multi-dimensional QoS. Also, the QoS response of a requested service may or may not depend on the parameters of the request—for instance, simple versus complex queries may result in different response times by the service.

Uncertainty in QoS For simple basic QoS policies, QoS guarantees exposed by the service or expected by the user are typically stated as fixed bounds. QoS is, however, generally subject to uncertainties, due to the numerous hidden sources of nondeterminism (servers, OS, queues, and network infrastructure). Therefore, a number of authors have agreed that QoS should be characterized in probabilistic terms.

In this paper, we consider Activities 1, 2, and 3, under the following 1. Activity 4—QoS-based adaptation and reconfiguration—is not considered:

Assumption 1

- Regarding Process Definition: *Workflows may be data-dependent and may involve unbounded but terminating loops.*
- Regarding QoS Dimensions: *The considered composite service may involve multi-dimensional QoS; QoS domains are thus partially, not totally ordered. QoS values may be data-dependent and dynamically defined for each service call.*
- Regarding Uncertainty in QoS: *QoS is considered probabilistic.*

3.1.1 Running Example

The following example will be used throughout this paper. We begin with a simple form for it and then develop some variations.

Fig. 3.1(a) depicts a simple orchestration for a travel agent. The user enters the location of a place to visit. Two Airline sites are invoked in parallel with the one offering "best" cost being selected. Next, two Hotel reservation sites are invoked and selection

Figure 3.1: TravelAgent1: Simple travel agent; (a) informal diagram, and (b) Petri net form, where rectangles figure transitions and rounded rectangles figure places. This orchestration has a data-independent workflow.

occurs on the basis of cost and category, seen both as QoS dimensions. The selection on cost/category may be done through lexicographic or weighted ordering. The results are presented as an invoice. This orchestration exhibits a data-independent control flow. It has a two-dimensional QoS, with the two dimensions being cost and hotel category. Multi-dimensional QoS is used in this and the following examples and is indeed often encountered. Observe that the two QoS dimensions in this example are correlated.

This diagram is reformulated into that Fig. 3.1(b), to be interpreted as a Petri net, where rectangles figure transitions and rounded rectangles figure places. Each query to the orchestration is modeled by a token traversing the input transition. Upon entering the first place, the transition to traverse must be chosen. This choice is based on best cost among offers by airline companies. Subsequently, the token enters the second place, where choice among different hotel booking services (shown as transitions) occurs based on both cost and category. This alternative Petri net description is a formalization of the previous description. We shall follow this Petri net modeling style hereinafter.

Fig. 3.2 shows a variation of Fig. 3.1(b) with a control flow dependent on returned *data* and *QoS* values. A loop is introduced in the decision process that checks if the total Cost is within the budget and can ask the user to specify preferences again. The presentment of the Invoice is guarded by a timer. The choice at the place labeled with "best(latency)" depends on which subsequent transition fires first. Thus, if the Invoice is ready before **Timeout** occurs, then it is emitted, otherwise a "timeout" message is returned. This timeout mechanism ensures that the loop terminates within a pre-specified time bound, possibly with a failure. This orchestration has a three-dimensional

Figure 3.2: TravelAgent2: A variation of TravelAgent1 having a data-dependent workflow.

QoS, with the dimensions being cost, hotel category, and latency (due to timers).

3.1.2 Key Issues

We now review some issues that we think are important in addressing Activities 1 (process design), 2 (on-line service selection), and 3 (process monitoring).

3.1.2.1 Monotonicity and Consequences for Management

A basic assumption underpinning the management of composite services is that QoS improvements in component services can only be better for the composite service. For example, referring to Activity 1, once service selection in QoS-based design has been performed (e.g., using optimization), a selected service is not expected to get deselected if it improves its QoS performance. The same remark holds for Activity 2: Once services have been selected on the basis of QoS performance, reconfiguration will not occur unless some requested service's performance degrades. Similarly, monitoring (Activity 3) consists of checking components for possible degradations in QoS. In general, we believe that the term "Quality of Service" presupposes that "better QoS is indeed better overall". In other words, the better the involved services[1] perform, the better the composite service performs.

This general property is important, so we give it a name—*monotonicity*. If a composite service fails to be monotonic, the common understanding of QoS is no longer valid and negotiations between the service provider and service requester regarding QoS issues become nearly unmanageable. We see monotonicity as a highly desirable feature, so we make it a central topic of this work.

Monotonicity always holds for orchestrations having a data-independent workflow—the orchestration shown in Fig. 3.1 is an example. A careful inspection shows that the orchestration of Fig. 3.2, which possesses a data-dependent workflow, is also monotonic.

However, monotonicity is not always satisfied under 1. Consider the example in Fig. 3.3. This orchestration performs late binding of service by deciding on-line and

[1] *Involved services* include all services that can potentially be requested by the composite service. For example, if the composite service involves an if-then-else branch, only one branch will actually be executed, but both are involved in the composite service.

Figure 3.3: A simple orchestration where monotonicity does not hold; the two alternatives are selected on the basis of best cost of air fare.

Figure 3.4: TravelAgent3: A variation of TravelAgent2 lacking monotonicity.

based on the cost of the airline ticket, which company to select. The two companies then propose different sets of hotels, shown by the two steps HotelBooking(Company1/2). Let c_1 and c_2 be the cost of ticket for Companies 1 and 2, and $h(c)$ be the optimum cost of the hotel booking if company c was selected. Suppose $c_1 < c_2$ and $h(c_1) < h(c_2)$ both hold. Then, the left branch is preferred and yields a total cost QoS for the orchestration equal to $c_1 + h(c_1)$. Now, suppose Company 2 improves its offer beyond Company 1: $c_2 < c_1$. Then, the right branch will be selected and total cost $c_2 + h(c_2)$ will result. Now, it may very well be that $c_2 + h(c_2) > c_1 + h(c_1)$ still holds, meaning that the improvement in QoS of Company 2 has resulted in a degradation of total QoS. The orchestration of Fig. 3.3 is non-monotonic. Differences in "local" versus "global" optimization due to lack of monotonicity were identified in [AP05, AR09a, ZBN+04]—"monotonicity" was not mentioned in the referred works but the concept was identified. The above example may seem trivial and obvious. However, TravelAgent3 in Fig. 3.4, is not monotonic either, despite it being a quite minor modification of TravelAgent2.

To summarize, this issue of monotonicity is essential. However, it seems underestimated in the literature, with the exception of [AP05, AR09a, ZBN+04], as our discussion of related work will show.

Assume that monotonicity is addressed, either by enforcing it, or by dealing with

the lack of it. Then, new avenues for composite service management under 1 can be considered, by taking advantage of monotonicity:

(a) A requested service improving its QoS can only improve the QoS of the orchestration. Therefore, it is enough for the orchestration to monitor QoS degradations for each requested service. Negotiations and penalties occur on the basis of understandable rules. Whenever acceptable, relations between the orchestration and its requested services can rely on *QoS contracts*. It is then the duty of the orchestration (or of some third party) to monitor such contracts for possible violation (Activity 3).[2]

(b) Since we build on a contract-based philosophy, the orchestration itself must be able to offer QoS contracts to its customers. This brings to the table the need for relating the contracts the orchestration has with its requested services, to the overall contract it can offer to its customers (Activity 1). We refer to this as *contract composition*.

(c) Thanks to monotonicity, it is possible to perform QoS-based late binding of services (Activity 2) by selecting, at run time, the best offer among a pool of possible candidates—by "possible" we mean candidates offering some given function or satisfying some given property.

3.1.2.2 Handling Probabilistic QoS

To handle uncertainty in QoS, probabilistic frameworks have been favored by several authors [ST07, HWSP04, HWTS07, MCD08, ZYZB11, CGK+11, ZBN+04, ZNB+08, ZBD+03, RBHJ07, RBHJ08]. When the workflow of the orchestration is statically defined regardless of data, rules for composing QoS probability distributions of the requested services have been proposed by several authors for various QoS domains [AGM08, ACP11, AGI+06, AP05, AR09a, CCGP10, YL05]. These serve as a basis to perform Activities 1 and 2. Optimal service selection among different options has been solved by efficient optimization methods, by using, e.g., Markov models [ACP11, CGK+11].

For orchestrations exhibiting data-dependent workflow or QoS values, however, such methods do not apply. The QoS-aware model of the orchestration combines *probability* and *non-determinism* — non-determinism arises from the data-dependent selection among alternatives. Markov models do not apply and Markov Decision Process models must be considered instead. The successive data-dependent choices performed are referred to as the *scheduler* of the MDP. Optimization can then be stated in two different ways. In most approaches [ST07, HWSP04, HWTS07, MCD08, ZYZB11, CGK+11], the scheduler itself is also randomized, thus resulting in a larger Markov model (assuming that sources of randomness are all independent). Alternatively, a max-min optimization can be performed, where the min is computed among the different service alternatives for a given fixed scheduler, and then the max over schedulers is computed. These methods have been widely used for Activity 1 of off-line orchestration design. Activity 2 of optimal on-line service selection or binding is much more demanding. Mathematically speaking, this activity amounts to solving a *stochastic control* problem [Ala92], in which, at each decision step, the expected remaining overall QoS is optimized and best decision is taken. Stochastic control is computationally demanding unless the considered orchestration is very small—this approach has not been considered in the literature.

[2] Contracts may not be established with services having huge customer basis (e.g., Google services). Such services are then used on the basis of estimated QoS based on measurements.

In the previous paragraph, we have advocated the importance of monotonicity and have discussed its (good) consequences for QoS-aware management of composite services. Can we lift these considerations to probabilistic QoS? To compare random variables, *stochastic ordering* has been proposed in various forms and extensively used in the area of economics and operations research [Tet77, Mos94, Mos07]. Using this concept, monotonicity was lifted to the probabilistic setting for the particular case of latency in [BRBH09]. Assuming that monotonicity can be lifted to the probabilistic setting for general (possibly multi-dimensional) QoS, the approach outlined in (a), (b), and (c) above becomes applicable and simple techniques can be developed to perform Activities 1, 2, and 3, based on QoS contracts. This agenda was developed by a subgroup of authors of this paper in [RBHJ07, RBHJ08, BRBH09], for the restricted case of latency.

3.1.3 Our Contribution

In this paper we extend our previous work on QoS-aware management of composite services under 1, to generic, possibly multi-dimensional, QoS. Our approach proceeds through the three steps (a), (b), and (c). Overall, we see our main contribution as being *a comprehensive and mathematically sound framework for contract based QoS-aware management of composite services*, relying on monotonicity. This framework consists of the following.

3.1.3.1 An Abstract Algebraic Framework for QoS composition

As QoS composition is the primary building block of QoS-aware management, it is of interest to develop abstract algebraic composition rules. We propose such an *abstract algebraic framework* encompassing key properties of QoS domains and capturing how the QoS of the orchestration follows from combining QoS contributions by each requested service. This algebraic framework relies on an abstract dioid $(\mathbb{D}, \max, \oplus)$, where $\mathbb{D}$ is the (possibly multi-dimensional) QoS domain. The abstract addition of the dioid identifies with the "max" operation associated with the partial order of the QoS domain; it captures both the preference among services in competition and the cost of synchronizing the return of several services requested in parallel. The increment in QoS caused by the different service calls is captured by the abstract multiplication of the dioid, here denoted by $\oplus$. A dioid framework for QoS was already proposed in [BS09a] and an algebraic framework on top of process algebras was proposed in [GGK+11]. With comparison to the above references, we propose in addition a new *competition operator* that must be considered when performing late binding; this competition operator captures the additional cost of on-line comparing the QoS within a pool of competing services. We show how our abstract algebraic framework can be specialized to encompass known QoS domains.

3.1.3.2 A Careful Handling of Monotonicity

We then study *monotonicity* in this generic QoS context, by proposing conditions ensuring it for both non-probabilistic and probabilistic QoS frameworks. Guidelines for how to enforce monotonicity are derived and ways are proposed to circumvent a lack of monotonicity. The mathematical justification of the extension required to deal with probabilistic QoS domains that are only partially, not totally ordered is non-trivial. Due to lack of space, we omit this mathematical justification here and refer the reader to [Ros09, RBJ09b]. We see our in-depth treatment of monotonicity as a major contribution, as this issue has been scarcely recognized and was not properly handled. Due to lack of space, probabilistic contract monitoring and composition is not detailed here.

However, once monotonicity is extended to general QoS, the extension of the techniques developed in [RBHJ08] raises no particular problem, see [Ros09] for details.

3.1.3.3 Support for Separation of Concerns

QoS-aware management of composite services requires developing a QoS-aware model of a service orchestration, which can be cumbersome. At least it is a significant increase in difficulty, compared to the only specification of the function of the orchestration. It is thus desirable to offer means to develop function and QoS in most possible orthogonal ways. By taking advantage of our abstract algebraic QoS framework, we are able to address this need. More precisely, we have developed an implementation of our approach in which QoS-aware orchestration models are automatically generated, from a specification of the function only, augmented with the declaration of the QoS domains and their algebra. This model can be executed to analyze the orchestration and perform QoS contract composition. We have implemented this technique on top of the Orc language for orchestrations [KCM06, MC06].[3]

3.1.3.4 A Methodology: Managing QoS by Contracts

By building on top of monotonicity, we advocate the use of *contract based* QoS-aware management of composite services, in which the considered orchestration establishes QoS contracts with both its users and its requested services. We briefly review how the Activities 1–3 are performed in this context.

Regarding Activity 1, contract based design amounts to performing *QoS contract composition* [RBHJ08], which is the activity of estimating the tightest end-to-end QoS contract an orchestration can offer to its customer, from knowing the contract with each requested service. QoS composition is developed in Section 3.4.2.

Consider Activity 2 for a monotonic orchestration. *Late service selection or binding* is performed on the basis of run-time QoS observations, by simply selecting, among different candidates, the one offering best QoS. Monotonicity ensures that this short-sighted policy will not lead to a loss in overall QoS performance of the orchestration. Best service binding is a built-in mechanism in our model, see 1 in Section 3.4.2.

Focus now on Activity 3 for a monotonic orchestration. To ensure satisfaction of the QoS contract with its users, it is enough to *monitor* the conformance of each requested service with respect to its contract, since a requested service improving its QoS can only improve the overall QoS of the orchestration. This was developed in [RBHJ08] for the case of latency and the techniques developed in this reference extend to multi-dimensional QoS.

To account for uncertainty in QoS, QoS is considered probabilistic. More precisely, *soft probabilistic QoS contracts* were first proposed in [RBHJ07, RBHJ08] for the case of latency and are extended in this paper to multi-dimensional QoS. Such contracts consist of the specification of a probability distribution for the QoS dimensions. Performing this requires formalizing what it means, for a service, to perform *better* than its contract. We rely for this on the notion of *stochastic ordering* [Tet77, Mos94, Mos07] for random variables, a concept that is widely used in econometrics. All our results regarding monotonicity extend to the case in which ordering of QoS values is replaced by stochastic ordering. This allows us to lift our handling of Activities 1–3 to probabilistic contracts. In particular, the *statistical monitoring* techniques proposed in [RBHJ08] allow for monitoring the violation of contracts in this context.

To validate our approach, we illustrate the use of this tool in performing contract composition for the example TravelAgent2.

[3] http://orc.csres.utexas.edu/

The paper is organized as follows. Related work is discussed in Section 3.2. Our QoS calculus is developed in Section 3.3; it provides the generic basis for QoS composition. Section 3.4 develops our theory of QoS for services orchestrations. Algebraic rules for QoS composition and best service binding are developed. Monotonicity is studied. Support for probabilistic QoS is presented. Section 3.4 is technical; a summary of its important results for practical use is provided in Section 3.5. In Section 3.6 we present the implementation of our approach on top of the Orc language. Evaluation of this implementation on the TravelAgent2/3 is discussed in Section 3.7.

3.2 Related Work

We restrict ourselves to papers dealing with QoS-aware management of composite services and addressing one or several of the Activities 1–4. We focus on papers dealing with probabilistic QoS (or at least uncertainty in QoS). In addition, we have included a few more papers, because either the issue of monotonicity is relevant to them, or they address QoS composition in an interesting way. With few exceptions, all papers address multi-dimensional QoS. Tables 3.1 and 3.2 categorize these papers according to the following criteria:

- support for probabilistic QoS;
- type of algorithm proposed to address Activities 1–3.

As the literature reported is already considerable, we focus (with a few exceptions) our detailed discussion on the papers addressing at least one of the Activities 1–3 *under 1*. As we believe mathematical soundness is a serious issue, we discuss it carefully.

Our bibliographical study divides into two parts. Table 3.1 collects papers that are relevant according to our criteria but do not face the issue of monotonicity, because restrictions ensuring built-in monotonicity are clearly stated on the control flow of the considered orchestration. More precisely, the underlying control flow is data-independent, which by itself guarantees monotonicity. To save space, we refer the reader to the comments in the table regarding these papers and do not discuss them any further.

Table 3.1: Literature survey: Papers dealing with orchestrations exhibiting a *data-independent* workflow

Paper	**Probabilistic QoS?**	**Algorithms to address Activities 1, 2, and 3**
Ardagna et al. (2008) [AGM08]	Probabilistic QoS supported Analytic techniques for QoS composition, queuing networks models	Service selection by NL Programming Probabilistic Model Checking for verifying guarantee of service
Abundo et al. (2011) [ACP11]	Probabilistic QoS supported Analytic techniques for QoS composition	MDP formulation of admission control subject to QoS requirements
Ardagna et al.(2006) [AGI+06]; Cardellini et al. (2010) [CCGP10]; Yu and Lin (2005) [YL05]	Probabilistic QoS supported Simulation techniques for QoS composition	QoS-aware service selection solved via linear programming
Cao et al. (2005) [CCL05]	Probabilistic QoS not supported	Optimizing some QoS parameters subject to QoS constraints using Genetic Algorithms
Limam and Boutaba (2010) [LB10]	Probabilistic QoS supported Simulation techniques for QoS composition	Service selection based on reputation and estimated QoS (with feedback to reputation)

We next review the literature collected in Table 3.2, where issues of monotonicity are relevant.

We begin with the work of Yu and Bouguettaya [YB08]. Built-in monotonicity is still ensured, due to proper restrictions on the control flow of the considered orchestrations. We nevertheless discuss it because specific issues of interest are studied. A Service Query Algebra is proposed in which composite services are seen as graphs. They can be further composed. QoS composition is one aspect of this service composition. QoS is treated in a fully algebraic style, very much like our present approach. Probabilistic aspects are not extensively developed, however. The work by Bistarelli and Santini [BS09a, BS09b] is discussed here because it explicitly refers to monotonicity in its title. This is, however, misleading in that this term is used in the totally different setting of "belief revision", a kind of logic in which facts can get falsified (thus the world is not monotonic in this sense). Frolund and Koistenen [FK98] propose QML, a general purpose language for QoS specification for distributed object-oriented systems. The language aspect is emphasized, not the underlying orchestration model. As the language is general, monotonicity cannot be guaranteed. However, we do not see this study as really addressing service orchestrations.

For the next group of papers, the authors seem unaware of the issue of monotonicity for the type of orchestration they consider (we do not repeat this fact for the different papers). Sato and Trivedi (2007) [ST07] provide a precise evaluation of reliability and performance characteristics in the presence of failures and retries, using Continuous-Time Markov Chain models. Marzolla and Mirandola (2007) [MM07] develop accurate queuing network models and evaluation techniques to derive performance bounds for the quick identification of bottlenecks in service orchestrations. Gilmore et al.[GGK+11] is an interesting paper, in which a rich UML-related formalism, called UML4SPA, is proposed to capture flexible QoS definitions in general composite services. This work is restricted to time as the only QoS dimension. The entire chain, from the UML modeling to the formal analysis tool PEPA based on timed process algebras is developed, with great technical details (except for the probabilistic aspects, which are not detailed). Ivanovic [ICH10] studies in detail how to deal with data-dependent QoS to perform run-time adaptation and monitoring. Service selection in composite service optimization and (re)binding is analyzed in [CPEV05b, CPEV05a, CPE+06, CPEV08]. An interesting contribution of [CPEV08] is the technique of re-binding that on-line performs a replacement of a service when the latter performance deviates too much from its nominal behavior. Cardoso et al.[CSM02, CSM+04] propose a predictive QoS model that allows to compute the QoS of workflows from the QoS of their atomic parts. Individual QoS parameters are estimated for their minimum, maximum, and averaged values based on measurements. Rules to compute QoS composition incrementally are used (the SWR rules published in the first author's PhD), with a special attention paid to fault-tolerant systems. Probabilistic QoS is possibly supported, with, however, little technical details. The work by Hwang et al. [HWSP04, HWTS07] is very interesting in its study of probabilistic QoS composition via analytic techniques. To avoid the computational cost resulting from state explosion in composite services, heuristic approximations are proposed. In a similar direction, the work of Zheng et al. [ZYZB11] proposes a very interesting comparison between analytic and simulation approaches for probabilistic QoS composition. The former are advocated, due to considerations of computational cost. The work by Menascé et al. [MCD08] gives a mathematically precise development of optimal service selection with cost and latency as QoS dimensions. The BPEL constructs are supported, including the "switch", which is a source of possible lack of monotonicity; alternative branches of the switch are assigned a probability. A very interesting heuristic is provided to perform near-to-optimal selection at a reasonable computational cost. The long and rich paper by Calinescu et al. [CGK+11] presents a methodology and extensive toolkit for performing Activities 1–4 listed in our introduction. Markov types of models are used in this toolkit, ranging from discrete and

continuous Markov chains to Markov Decision Processes to deal with non-deterministic choices or data-dependent branching. QoS analyses are supported thanks to a formulation using probabilistic temporal logic and associated model checkers. The methodology and toolkit reuses existing tools and did not need the development of any new engine. The paper lacks mathematical details, however, regarding the models and algorithms used.

The issue of monotonicity is identified in only three papers from our list, albeit under a different wording than ours. Ardagna et al. [AP05] discuss local versus global QoS guarantees and explain why optimizing QoS guarantees of local execution paths may not lead to the satisfaction of global QoS guarantees. Alrifai & Risse [AR09a] propose a similar approach using MMKP for computationally efficient selection over global and local constraints. In Zeng et al. [ZBN+04], a thorough comparison is made between local versus global optimization in service selection. It is argued that performing local optimization may not lead to optimal selection; indeed, the beginning of Section 3.4.2 in this paper explains exactly our example of Fig. 3.3. The paper explains that global optimization always provides a relevant selection, which is certainly correct. We have, however, explained in our introduction why we believe that not having monotonicity leads to a strange understanding of QoS management. Now, referring to our taxonomy, in monotonic orchestrations, local optimization is enough to ensure global optimality. Other major features of this paper are summarized in the table.

To conclude on this bibliographical study, we notice that the issue of monotonicity is mostly ignored in the literature on composite Web services, whereas it is known in the area of performance studies for general computer architectures. Our work focuses on monotonicity, its conditions, and its consequences for QoS-aware management of composite Web services.

3.3 QoS Calculus

In this section we develop our QoS calculus as a basis for QoS composition. A toy example allows us to motivate our abstract algebra. Then we illustrate how this algebra can encompass concrete QoS domains. Finally the algebra itself is formalized.

3.3.1 An Informal Introduction to the QoS Calculus

We begin by reviewing the algebra needed for composing QoS. In dealing with multi-dimensional QoS, several approaches can be taken. First, one can see QoS as only partially, not totally ordered. In this case QoS outcomes q and q' satisfy $q \leq q'$ if and only if $q(i) \leq q'(i)$ holds for any dimension $i = 1 \ldots n$ of the QoS. Alternatively, one could prioritize dimensions and then take the lexicographic (total) order $q \leq q'$ iff there exists some i such that $q(j) = q'(j)$ for $j < i$ and $q(i) \leq q'(i)$. Finally, different dimensions could be weighted by considering $\sum_i w_i q(i)$ with its total order, where the w_i's are weights to be selected, e.g., by using AHP (Analytical Hierarchy Process) [Tho90]. Finally, recall that dealing with uncertainty is by regarding QoS outcomes as random variables.

We use colored Petri nets to model the executions of a service orchestration. Queries are represented by tokens that circulate throughout the net and service calls are represented by transitions. To represent QoS parameters and how they evolve while the query is being processed by the orchestration, we equip the tokens with a color, consisting of a pair

$$(v, q) = (\text{data, QoS value}). \tag{3.1}$$

Fig. 3.5 shows such a net. Each query is represented by a token entering the net at the top place. The marking shown figures the reception of such a query by the net:

Figure 3.5: A simple example. Only QoS values are mentioned — with no data. Each place comes labeled with a QoS value q which is the q-color of the token if it reaches that place.

it results in the launching of three sub-queries in parallel. The first two sub-queries re-synchronize when calling t_2. The third sub-query branches toward either calling t'_1 or calling t''_1 and then confluences. The processing of the query ends when the token reaches the exit place. With reference to this figure, the different operators needed to compute the evolution of QoS parameters are introduced next. In the following discussion, we only consider choices governed by QoS (data-driven choices play no role in QoS evaluation).

Basic Operators on QoS

The objective is to capture, via generic operators, how QoS parameters get modified when calling a service (traversing a transition), when synchronizing the responses of services (figured by several tokens consumed by a same transition), or when different services compete against each other (such as t'_1 and t''_1 in Fig. 3.5).

Incrementing QoS When traversing a transition, each token gets its QoS value incremented, which is captured by operator $\oplus$. For example, the left most token has initial QoS value q_0, which gets incremented as $q_1 = q_0 \oplus \delta q_1$ when traversing transition t_1.

Synchronizing tokens A transition t is enabled when all places in its preset have tokens. For the transition to fire, these tokens must synchronize, which results in the "worst" QoS value, denoted by the supremum $\vee$ associated to a given order $\leq$, where smaller means better. For example, when the two input tokens of t_2 get synchronized, the resulting pair of tokens has QoS $q''_0 \vee q_1$. This is depicted in Fig. 3.5 by the shaded area.

QoS composition policy Focus on the conflict following place q'_0. The QoS alters the usual semantics of the conflict by using a *QoS composition policy* that is reminiscent of the classical race policy [MBB+89]. The competition between the two conflicting transitions in the post-set is solved by using order $\leq$ also used for token synchronization:

test whether $q'_o \oplus \delta q'_1 \leq q'_o \oplus \delta q''_1$ holds, or the converse. The smallest of the two wins the competition—nondeterministic choice occurs if equality holds.

However, comparing $q'_o \oplus \delta q'_1$ and $q'_o \oplus \delta q''_1$ generally requires knowing the two alternatives, which in general can affect the QoS of the winner. This is taken into account by introducing a special operator "$\lhd$": If two transitions t and t' are in competition and would yield tokens with respective QoS values q and q' in their post-sets, the cost of comparing them to set the competition alters the QoS value of the winner in that—assuming the first wins—q is modified and becomes $q \lhd q'$, where $\lhd$ denotes a new operator called the *competition function*. For the case of the figure, we get

$$\begin{array}{rl} \text{if} & (q'_o \oplus \delta q'_1) \leq (q'_o \oplus \delta q''_1) \\ \text{then} & t'_1 \text{ fires and } q'_1 = (q'_o \oplus \delta q'_1) \lhd (q'_o \oplus \delta q''_1) \\ \text{if} & (q'_o \oplus \delta q'_1) \geq (q'_o \oplus \delta q''_1) \\ \text{then} & t''_1 \text{ fires and } q''_1 = (q'_o \oplus \delta q''_1) \lhd (q'_o \oplus \delta q'_1) \end{array} \tag{3.2}$$

3.3.2 Some Examples of QoS Domains

We now instantiate our generic framework by reviewing some examples of QoS domains, with their associated relations and operators $\oplus, \leq$, and $\lhd$.

Latency QoS value of a token gives the accumulated latency d, or "age" of the token since it was created when querying the orchestration. Corresponding QoS domain is $\mathbb{R}_+$, equipped with $\oplus_d = +$, and $\leq_d =$ the usual order on $\mathbb{R}_+$. Regarding operator $\lhd_d$, for the case of latency with race policy [MBB+89], comparing two dates via $d_1 \leq_d d_2$ does not impact the QoS of the winner: answer to this predicate is known as soon as the first event is seen, i.e., at time $\min(d_1, d_2)$. Hence, for this case, we take $d_1 \lhd_d d_2 = d_1$, i.e., d_2 does not affect d_1. This is the basic example of QoS parameter, which was studied in [BRBH09].

Security level QoS value s of a token belongs to $(\{\text{high, low}\}, \leq_s)$, with high $\leq_s$ low. Each transition has a security level encoded in the same way, and we take $\oplus_s = \vee_s$, reflecting that a low security service processing a high security data yields a low security response. Regarding operator $\lhd_s$, again, comparing two values via $s_1 \leq_s s_2$ does not impact the QoS of the winner: QoS values are strictly "owned" by the tokens, and therefore do not interfere when comparing them. Hence, we take again $s_1 \lhd_s s_2 = s_1$, i.e., s_2 does not affect s_1. More complex partially ordered security domains can be handled similarly.

We do not claim that this solves security in orchestrations. It only serves a more modest but nevertheless useful purpose, namely to propagate and combine security levels of the requested services to derive the security level of the orchestration. How security levels of the requested services is established is a separate issue, e.g., by relying on reputation or through the negotiation of security contracts.

Reliability Reliability is captured similarly as follows. The QoS attribute of a token takes its value in the set $(\{\text{valid, invalid}\}, \leq_r)$, with valid $\leq_r$ invalid. Other operators follow as for the case of Security level. A service returning "invalid" is an indication of a failure. By equipping this QoS domain with probability distributions we capture reliability in our setting.

Cost QoS value c captures the total cost of building a product by assembling its parts. Referring to Fig. 3.5, costs are accumulated when tokens get synchronized. When a token traverses a transition, its cost is incremented according to the cost of the action

being performed. A natural definition for the corresponding QoS domain would thus be $(\mathbb{D}_c, \leq_c, \oplus_c) = (\mathbb{R}, \leq, +)$ or $(\mathbb{Z}, \leq, +)$. Unfortunately, when taking this definition, synchronizing tokens using $\vee_c$ amounts to taking the worst cost, which is not what we need. We need instead the sum of the costs of incoming tokens, an operation different from $\vee_c$.

The right idea is to encode the cost by using multi-sets. The overall cost held by a token is obtained by adding the costs of the constituting parts plus the costs of successive assembly actions. Parts and actions are then handled as "quanta of cost" and the token collects them while traversing the orchestration. This leads to defining the QoS domain as a multi-set of *cost types*: $\mathbb{D}_c = \mathbf{Q} \mapsto \mathbb{N}$, where $\mathbf{Q}$ is a set of cost types equipped with a cost labeling function $\lambda : \mathbf{Q} \mapsto \mathbb{R}_+$. Each $\mathbf{q} \in \mathbf{Q}$ corresponds to either a part or an assembly action and has a unique identifier. Domain $\mathbb{D}_c$ is equipped with the partial order of functions and $\vee_c$ follows as the corresponding least upper bound. Recall that operator $\vee_c$ is used to synchronize tokens, see Fig. 3.5. In this context, it makes sense to assume that cost types held by the tokens for synchronization are different. For this case, $\vee_c$ coincides with the addition of multi-sets and costs get added as wished. Traversing a transition amounts to adding the corresponding quantum in the set, hence, identifying singletons with the corresponding element, $\oplus_c$ is again the addition of multi-sets. Finally, $(\mathbb{D}_c, \leq_c, \oplus_c) = (\mathbf{Q} \mapsto \mathbb{N}, \subseteq, +)$. As before, the competition function is $c_1 \lhd_c c_2 = c_1$ when $c_1 \leq_c c_2$, i.e., c_2 does not affect c_1.

Composite QoS, first example we may also consider a composite QoS parameter consisting of the pair (s, r), where s is as above and r is some *Quality of Response* with domain $\mathbb{D}_r$, equipped with $\leq_r$ and $\lhd_r$. Since the two components s and r are similar in nature, we simply take $\leq = \leq_s \times \leq_r$ and $\lhd = (\lhd_s, \lhd_r)$.

Composite QoS, second example So far the special operator $\lhd$ did not play any role. We will need it, however, for the coming case, in which we consider a composite QoS parameter (s, d), where s and d are as above. We want to give priority to security s, and thus we now take $\leq$ to be the lexicographic order obtained from the pair $(\leq_s, \leq_d)$ by giving priority to s.

Focus on operator $\lhd$. Consider the marking resulting after firing t_1 and t'_1 in Fig. 3.5, enabling t_2 and t'_2, which are in conflict. Let the QoS value of the token in postset of t_2, i.e. $q_2 = (low, d_2)$. (Recall that $q_2 = (q''_o \vee q_1) \oplus \delta q_2$.) Similarly, let $q'_2 = (low, d'_2)$ where $d'_2 >_d d_2$. From the competition rule, transition t_2 wins the conflict and the outgoing token has QoS value $q_2 = (low, d_2)$. However, the decision to select t_2 can only be made when q'_2 is known, that is, at time d'_2. The reason for this is that, since at time d_2 a token with security level *low* is seen at place following t_2, it might be that a token with security level *high* later enters place following t'_2. The latter would win the conflict according to our QoS composition policy — security level prevails. Observing that the right most token indeed has priority level *low* can only be seen at time d'_2. Thus it makes little sense assigning $q_2 = (low, d_2)$ to the outgoing token; it should rather be $q_2 = (low, d'_2)$. This is why a non-trivial operator $\lhd$ is needed, namely, writing $\leq$ for short instead of $\leq_d$:

$$\begin{array}{l} (s, d) \lhd (s', d') \;\; = \\ \textbf{if } d \leq d' \;\; \text{and } s = low \;\; \textbf{then}(s, d') \;\; \textbf{else} \;\; (s, d) \end{array} \tag{3.3}$$

3.3.3 Formalizing the QoS Calculus

Dï£¡finition 3.1 (QoS domains) *A* QoS domain *is a tuple* $\mathbb{Q} = (\mathbb{D}, \leq, \oplus, \lhd)$ *where:*

- *$(\mathbb{D}, \leq)$ is a partial order that is a complete upper lattice, meaning that every subset $S \subseteq \mathbb{D}$ has a unique least upper bound denoted by $\bigvee S$. By convention, we interpret synchronization order $\leq$ as "better". Hence operator $\vee$ amounts to taking the "worst" QoS and is used while synchronizing tokens.*

- *Operator $\oplus : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ captures how a transition increments the QoS value; it satisfies the following conditions:*

  1. *there exists some neutral element 0 satisfying $\forall q \in \mathbb{D} \Rightarrow q \oplus 0 = 0 \oplus q = q$;*
  2. *$\oplus$ is* monotonic*: $q_1 \leq q'_1$ and $q_2 \leq q'_2$ together imply $(q_1 \oplus q_2) \leq (q'_1 \oplus q'_2)$.*
  3. *$\forall q, q' \in \mathbb{D}$, $\exists \delta q \in \mathbb{D}$ such that $q \leq q' \oplus \delta q$.*

  Condition 3) is a technical condition expressing that order $\leq$ is "rich enough".

- *The* competition function *$\lhd : \mathbb{D} \times \mathbb{D}^* \to \mathbb{D}$, where $\mathbb{D}^* = \biguplus_{k=0}^{\infty} \mathbb{D}^k$ and $\mathbb{D}^0 = \emptyset$, maps a pair consisting of 1/ the QoS resulting from the synchronization of the input tokens, and 2/ the tuple of the QoS of other tokens that must be considered when applying competition. We require the following regarding $\lhd$:*

  1. *$q \lhd \epsilon = q$ where ϵ denotes the empty tuple, that is, if no competition occurs, then q is not altered;*
  2. *$\lhd$ is monotonic:*

$$q \leq q' \quad \textit{and} \quad q_1 \leq q'_1, \ldots, q_n \leq q'_n$$
$$\Downarrow$$
$$(q \lhd (q_1, \ldots, q_n)) \leq (q' \lhd (q'_1, \ldots, q'_n))$$

Examples were given in Section 3.3.1. It is easily checked that axioms are met by these examples. The actual size of the second component of competition function $\lhd$ is dynamically determined while executing the net, this is why the domain of $\lhd$ is $\mathbb{D} \times \mathbb{D}^*$.

If some QoS parameter q of the orchestration is irrelevant to a service it involves, we take the convention that this service acts on tokens with a 0 increment on the value of q. With this convention we can safely assume that the orchestration, all its requested services, and all its tokens use the same QoS domain. This assumption will be in force in the sequel.

3.4 A Theory of QoS for Workflows

This section collects the technical material in support of our theory and developments.

We first recall the needed background on Petri nets as a basic model for service orchestrations. Then we develop our formal QoS-aware model of orchestrations, together with the rules for QoS composition and best service binding. We then study monotonicity. The above material is then lifted to probabilistic QoS. We conclude by some methodological discussion.

The OrchNets we propose as a model to capture QoS in composite services are a special form of *colored occurrence nets (CO-nets)* with *read arcs*. Executions of Workflow Nets [vdA98, vdA97] are also CO-nets. The reader can compare our approach with the graph-based approach of [YB08]. Before providing the formal definition of OrchNets, we need some background on Petri nets and occurrence nets.

3.4.1 Petri Nets, Occurrence Nets, and Orchestration Nets

A *Petri net with read arcs* [Mur89] is a tuple $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{R}, M_0)$, where: $\mathcal{P}$ is a set of *places*, $\mathcal{T}$ is a set of *transitions* such that $\mathcal{P} \cap \mathcal{T} = \emptyset$, $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$

is the *flow relation*, $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{T}$ is the *read relation*, and $M_0 : \mathcal{P} \to \mathbf{N}$ is the *initial marking*. We require that $\mathcal{F} \cap \mathcal{R} = \emptyset$. For $x \in \mathcal{P} \cup \mathcal{T}$, we call ${}^\bullet x = \{y \mid (y, x) \in \mathcal{F}\}$ the *preset* of x, $x^\bullet = \{y \mid (x, y) \in \mathcal{F}\}$ the *postset* of x, ${}^\circ t = \{p \mid (p, t) \in \mathcal{R}\}$ the *context* of t, and we set ${}^{\bullet\!-} t = {}^\bullet t \cup {}^\circ t$.

A *marking* is a map $M : \mathcal{P} \to \mathbf{N}$. *Firing* transition t at marking M requires $M(p) > 0$ for every $p \in {}^{\bullet\!-} t$ and yields the new marking M' such that $M'(p) = M(p) - 1$ for $p \in {}^\bullet t \setminus t^\bullet$, $M'(p) = M(p) + 1$ for $p \in t^\bullet \setminus {}^\bullet t$, and $M'(p) = M(p)$ otherwise. In words, tokens are consumed from the preset, read but not consumed from the context, and produced in the postset.

For a net $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{R}, M_0)$ the *causality relation* $\preceq$ is the transitive and reflexive closure of $\mathcal{F}$ and we set $\prec = \preceq \cap \neq$. For a node $x \in \mathcal{P} \cup \mathcal{T}$, the set of *causes* of x is $\lceil x \rceil = \{y \in \mathcal{P} \cup \mathcal{T} \mid y \preceq x\}$. For two transitions t and t', say that t *weakly causes* t', written $t \nearrow t'$ (or, sometimes, $t' \nwarrow t$), if either $t \prec t'$ or ${}^\circ t \cap {}^\bullet t' \neq \emptyset$ (if t occurs, then it must occur before t'). Say that a set $T \subseteq \mathcal{T}$ of transitions is in *conflict*, written $\#T$, if there exist $t, t' \in T$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ or there exists a cycle $t_0 \nearrow t_1 \nearrow \cdots \nearrow t_n = t_0$ where $t_0, \ldots, t_{n-1} \in T$.[4]

Occurrence nets A Petri net is *safe* if all its reachable markings M satisfy $M(\mathcal{P}) \subseteq \{0, 1\}$. A safe net $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{R}, M_0)$ is an *occurrence net (O-net)* iff

1. $\preceq$ is a partial order and $\lceil t \rceil$ is finite for any $t \in \mathcal{T}$;
2. for each place $p \in \mathcal{P}$, $|{}^\bullet p| \leq 1$;
3. for each $t \in \mathcal{T}$, $\neg\#\lceil t \rceil$ holds;
4. $M_0 = \{p \in \mathcal{P} | {}^\bullet p = \emptyset\}$ holds.

Occurrence nets are a good model for representing the possible executions of a concurrent system. Executions are formalized via the notion of configuration.

A *configuration* of N is a subnet κ of nodes of N such that: 1/ κ is *causally closed*, *i.e*, if $x \preceq x'$ and $x' \in \kappa$ then $x \in \kappa$; and, 2/ κ is *conflict-free*. For convenience, we require that the maximal nodes in a configuration are places. A configuration κ_2 is said to extend configuration κ_1 (written as $\kappa_1 \preceq \kappa_2$) if $\kappa_1 \subseteq \kappa_2$ and $\nexists t \in \kappa_2 \setminus \kappa_1, t' \in \kappa_1$ such that $t \nearrow t'$. Two configurations κ and κ' are said to be *compatible* if 1/ $\kappa \cup \kappa'$ is a configuration, and 2/ $\kappa \preceq \kappa \cup \kappa'$ and $\kappa' \preceq \kappa \cup \kappa'$. Node x is called compatible with configuration κ if $\lceil x \rceil$ and κ are compatible. Transition t is *enabled* by κ if $t \notin \kappa$ and $\kappa \cup \{t\} \cup t^\bullet$ is a configuration. For κ a configuration, its *future* N^κ is defined as

$$\begin{aligned} N^\kappa = {} & maxPlaces(\kappa) \cup \\ & \{x \in \mathcal{P} \cup \mathcal{T} \mid x \notin \kappa \text{ and } x \text{ is compatible with } \kappa\} \end{aligned} \tag{3.4}$$

where $maxPlaces(\kappa)$ is the set of maximal nodes of κ (which are all places). Two nodes x and y are said to be *concurrent* if they are compatible and neither $x \nearrow y$ nor $x \nwarrow y$ holds. Two sets of nodes X and Y are said to be concurrent if x and y are concurrent for any two $x \in X$ and $y \in Y$.

[4] Tuple $(\mathcal{T}, \preceq, \nearrow)$ is a pre-Asymmetric Event Structure in the sense of [Win86].

Unfoldings and Orchestration nets The executions of a safe Petri net N can be represented by its *unfolding* U_N, which is an occurrence net collecting all executions of N in such a way that common prefixes are represented once. For example, Fig. 3.5 shows a net, the unfolding of which is obtained by removing the maximal (exit) place and attaching a different copy of this exit place to each exit transition. Formally, unfolding U_N is derived from N [ERV02] in the following way. For $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{R}, M_0)$ and $N' = (\mathcal{P}', \mathcal{T}', \mathcal{F}', \mathcal{R}', M_0')$ two safe Petri nets, a *morphism* $\varphi : N \to N'$ is a function from $\mathcal{P} \cup \mathcal{T}$ to $\mathcal{P}' \cup \mathcal{T}'$, mapping $\mathcal{P}$ to $\mathcal{P}'$ and $\mathcal{T}$ to $\mathcal{T}'$, preserving the initial marking: $\varphi(M_0) = M_0'$, and preserving the flow and read relations: $\varphi({}^\bullet t) = {}^\bullet\varphi(t)$, $\varphi(t^\bullet) = \varphi(t)^\bullet$, and $\varphi({}^\circ t) = {}^\circ\varphi(t)$. If N' is another occurrence net and $\psi' : N' \to N$ is a morphism, then there exists a third morphism $\psi : N' \to U_N$ such that ψ' factorizes as $\psi' = \varphi \circ \psi$, where $\circ$ is the composition of functions. This property characterizes the unfolding U_N.

We call *Orchestration net* a safe Petri net possessing a finite unfolding. We insist that Petri nets with loops can still possess a finite unfolding. An example of this is the Petri net modeling the examples TravelAgent of Fig. 3.2 and Fig. 3.3, which involve successive retries guarded by a timeout. In the sequel we only consider Petri nets that are orchestration nets. Examples of Orchestration nets are the loop-free and 1-safe WorkFlow nets (WFnets). WF-nets were proposed by van der Aalst [vdA97, vdAtHKB03, vdAvH02] and are Petri nets with a special initial place (where the initial tokens are provided) and a special final place (from which tokens exit the net).

Branching cells The discussion of the example in Section 3.3.1 revealed the need to consider, dynamically while execution progresses, the set of transitions that are both enabled and in conflict with a considered transition. This was studied by Abbes and Benveniste with the notion of branching cell [AB06a], for symmetric event structures or nets without read arcs. This notion was subsequently extended to nets with read arcs by Rosario [Ros09, Chapter 4]. We recall this notion now. Let N be an occurrence net with read arcs. For two transitions $t, t' \in \mathcal{T}$, say that t is in *weak conflict* with t', written $t \nearrow_{\#} t'$, if either $\#\{t, t'\}$ or $[t \nearrow t'] \wedge \neg[t < t']$, expressing that t' preempts t from occurring. Then, say that t' is in *minimal asymmetric conflict* with t, written $t \nearrow_{\#m} t'$, if:

1. $t \nearrow_{\#} t'$, i.e., t is in weak conflict with t';
2. $(\{t\} \times \lceil t' \rceil) \cap \nearrow_{\#} = \{(t, t')\}$, i.e., t is in weak conflict with no cause of t';
3. $\big((\lceil t \rceil \times \lceil t' \rceil) \times (\nearrow_{\#} \cap {}_{\#}\nwarrow)\big) \subseteq \{(t, t')\}$, that is, t and t' are the only pair (x, x') of nodes in the respective causes of t and t' such that both $x \nearrow_{\#} x'$ and $x \,{}_{\#}\nwarrow\, x'$ possibly occur.

A prefix M of N is a causally closed subnet of N whose maximal nodes are places; formally, M is closed under operations $t \to \lceil t \rceil$ and $t \to t^\bullet$. Prefix M is called a *stopping prefix* if it is symmetrically closed under minimal asymmetric conflict: formally $t \in M$ and $t' \nearrow_{\#m} t$ or $t \nearrow_{\#m} t'$ imply $t' \in M$. *Branching cells* of occurrence net N are inductively defined as follows:

1. every minimal (for prefix relation) stopping prefix of N is a branching cell, and,
2. let B be any such branching cell and κ any maximal configuration of it, then any branching cell of N^κ is a branching cell of N,

where N^κ, *the future of* κ, is defined in (3.4). In the example of Fig. 3.5, transitions t_1' and t_1'', along with their pre and post sets form one of the branching cells of the net.

We will need the following results regarding branching cells of finite occurrence nets:

$$\textit{configurations are tiled by branching cells;} \tag{3.5}$$

$$\textit{the minimal branching cells of an occurrence net are pairwise concurrent,} \tag{3.6}$$

where "minimal" refers to the causality order.

3.4.2 OrchNets: Definition and QoS Semantics, Application to QoS Composition

Throughout this section we assume a QoS domain $(\mathbb{D}, \leq, \oplus, \lhd)$. OrchNets formalize the notion of an orchestration with its QoS. The mathematical semantics of OrchNets formalizes QoS contract composition, i.e., the process of deriving end-to-end QoS of the orchestration from the QoS of its involved services.

Dï£¡finition 3.2 (OrchNet) *An* OrchNet *is a tuple* $\mathcal{N} = (N, V, Q, Q_{\text{init}})$ *consisting of*

- *A finite occurrence net* N *with token attributes*

$$c = (v, q) = (\text{data, QoS value})$$

- *A family* $V = (\nu_t)_{t \in \mathcal{T}}$ *of* value functions, *mapping the data values of the transition's input tokens to the data value of the transition's output token.*
- *A family* $Q = (\xi_t)_{t \in \mathcal{T}}$ *of* QoS functions, *mapping the data values of the transition's input tokens to a QoS increment.*
- *A family* $Q_{\text{init}} = (\xi_p)_{p \in \min(\mathcal{P})}$ *of* initial QoS functions *for the minimal places of* N.

Values, assumptions, and QoS functions can be nondeterministic. We introduce a global, invisible, daemon ω *that resolves this nondeterminism and we denote by* Ω *its domain. That is, for* $\omega \in \Omega$, $\nu_t(\omega), \xi_t(\omega)$, *and* $\xi_p(\omega)$ *are all deterministic functions of their respective inputs.*

We now explain how the presence of QoS values attached to tokens affects the semantics of OrchNets. Any place p of occurrence net N has a pair $(v_p, q_p) =$ (data, QoS value) assigned to it, which is the color held by a token reaching that place. In the following QoS composition policy, the role of data in the semantics has been abstracted—taking it into account would only increase the notational burden without introducing changes worth the study.

Competition step 3 formalizes on-line service binding based on best QoS.

Step 4 of QoS composition policy simplifies for all examples of Section 3.3.1 except for the last one, see formula (3.3). Since occurrence net N is finite, the QoS composition policy terminates in finitely many steps when $N^{\kappa(\omega)} = \emptyset$. The total execution thus proceeds by a finite chain of nested configurations: $\emptyset = \kappa_0(\omega) \prec \kappa_1(\omega) \cdots \prec \kappa_n(\omega)$. Hence, $\kappa_n(\omega)$ is a maximal configuration of $\mathcal{N}$ that can actually occur according to the QoS composition policy, for a given $\omega \in \Omega$. We generically denote this maximal configuration by

$$\kappa(\mathcal{N}, \omega). \tag{3.9}$$

For the example of latency, our QoS composition policy boils down to the classical *race policy* [MBB+89]. In general, our QoS composition policy bears some similarity with

Algorithm 1: QoS composition policy

1 Let $\omega \in \Omega$ be any value for the daemon. The continuation of any finite configuration $\kappa(\omega)$ is constructed by performing the following steps, where we omit the explicit dependency of $\kappa(\omega)$, $\nu_t(\omega)$, and $\xi_t(\omega)$, with respect to ω:

1. Choose nondeterministically a minimal branching cell B in the future of κ — this is possible by (3.6).

2. For t any minimal transition of B, compute:

$$q_t = \left(\bigvee_{p' \in {}^{\bullet}_{-}t} q_{p'}\right) \oplus \xi_t(v_{p'} \mid p' \in {}^{\bullet}_{-}t) \tag{3.7}$$

   where we recall that ${}^{\bullet}_{-}t = {}^{\bullet}t \cup {}^{\circ}t$ is the union of the preset of t and the context of t.

3. Competition step: select nondeterministically a minimal transition t_* of B such that no other minimal transition t of B exists with $q_t < q_{t_*}$. The set Ω of daemons is extended to resolve this additional nondeterminism.

4. Augment κ to $\kappa' = \kappa \cup \{t_*\} \cup t_*^{\bullet}$, and assign, to every $p \in t_*^{\bullet}$, the pair (v, q), where

$$\begin{aligned} v &= \nu_t(v_{p'} \mid p' \in {}^{\bullet}_{-}t) \\ q &= q_{t_*} \lhd (q_t \mid t \in B, t \text{ minimal}, t \neq t_*) \end{aligned} \tag{3.8}$$

Observe that the augmented configuration κ' as well as the pair (v, q) depend on ω. □

the "preselection policies" of [MBB+89], except that the continuation is selected based on QoS values in our case, not on random selection. We will also need to compute the QoS for *any* configuration of N, even if it is not a winner of the competition policy. We do this by modifying 1 as follows: We are now ready to define what the QoS value of an OrchNet is:

Dï£¡finition 3.3 (End-to-end QoS) *For κ any configuration of occurrence net N, and ω any value for the daemon, the* end-to-end QoS *of κ is defined as*

$$E_\omega(\kappa, \mathcal{N}) = \bigvee_{p \in maxPlaces(\kappa)} q_p(\omega) \tag{3.10}$$

The end-to-end QoS $E_\omega(\mathcal{N})$ *and* loose end-to-end QoS $F_\omega(\mathcal{N})$ *of OrchNet $\mathcal{N}$ are respectively given by*

$$E_\omega(\mathcal{N}) = E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N}) \tag{3.11}$$

$$F_\omega(\mathcal{N}) = \max\{E_\omega(\kappa, \mathcal{N}) \mid \kappa \in \mathcal{V}(N)\} \tag{3.12}$$

where function max *picks one of the maximal values in a partially ordered set, $\kappa(\mathcal{N}, \omega)$ is defined in* (3.9)*, and $\mathcal{V}(N)$ is the set of all maximal configurations of net N.*

Observe that $E_\omega(\mathcal{N}) \leq F_\omega(\mathcal{N})$ holds and $E_\omega(\mathcal{N})$ is indeed observed when the orchestration is executed. The reason for considering in addition $F_\omega(\mathcal{N})$ will be made clear in the next section on monotonicity.

So far formulas (3.11) and (3.12) provide the composition rules for deriving the end-to-end QoS for each individual call to the orchestration. Monte-Carlo simulation techniques can then be used on top of (3.11) and (3.12) to derive the end-to-end probabilistic QoS contract from the contracts negotiated with the requested services [RBHJ07, RBHJ08]. See also [Kat11] for fast Monte-Carlo simulation techniques.

Algorithm 2: QoS of an arbitrary configuration

1 Let $\kappa_{\max}$ be any maximal configuration of N and $\kappa \preceq \kappa_{\max}$ a prefix of it. With reference to 1, perform: step 1 with B any minimal branching cell in $\kappa_{\max} \setminus \kappa$, step 2 with no change, and then step 4 for any t as in step 2. Performing this repeatedly yields the pair (v_p, q_p) for each place p of $\kappa_{\max}$. □

3.4.3 Monotonicity: Results

The monotonicity of an orchestration with respect to QoS is studied in this section, for the non-probabilistic setting. Extension to the probabilistic setting is discussed in Section 3.4.4. We provide sufficient and structurally necessary conditions for monotonicity, when QoS is measured in terms of tight end-to-end QoS—missing proofs are deferred to Appendix. When these conditions fail to hold, then loose end-to-end QoS can be considered when dealing with contracts, as monotonicity is always guaranteed when using it. Monotonicity is assumed in the rest of the paper. Also, to simplify the presentation, the following assumption will be in force:

Assumption 2 *QoS functions ξ_t can be increased at will within their respective domain of values, independently for each transition t.*

This assumption rules out cases in which one requires, e.g., that QoS functions ξ_t and $\xi_{t'}$ can be modified at will, but subject to the constraint $\xi_t = \xi_{t'}$. The general case yields similar results, at the price of more complex notations [RBJ09b].

For two families Q and Q' of QoS functions, write $Q' \geq Q$ and $Q'_{\text{init}} \geq Q_{\text{init}}$ to mean:

$$\begin{array}{rcl} \forall \omega \in \Omega, \forall t \in \mathcal{T} & \Rightarrow & \xi'_t(\omega) \geq \xi_t(\omega) \\ \text{respectively } \forall t \in \mathcal{T} & \Rightarrow & Q_{\text{init}}(t) \geq Q_{\text{init}}(t) \end{array} \tag{3.13}$$

For $\mathcal{N}' = (N, V, Q', Q'_{\text{init}})$ (observe that N and V are unchanged), write

(i) $\mathcal{N}' \geq \mathcal{N}$;

(ii) $E(\mathcal{N}') \geq E(\mathcal{N})$;

(iii) $F(\mathcal{N}') \geq F(\mathcal{N})$;

to mean that, respectively

(i) $Q' \geq Q$ and $Q'_{\text{init}} \geq Q_{\text{init}}$ both hold;

(ii) $\forall \omega \in \Omega,\ E_\omega(\mathcal{N}') \geq E_\omega(\mathcal{N})$ holds;

(iii) $\forall \omega \in \Omega,\ F_\omega(\mathcal{N}') \geq F_\omega(\mathcal{N})$ holds.

Dï£¡finition 3.4 *Call OrchNet $\mathcal{N}$* monotonic *if*

$$\forall \mathcal{N}' : \mathcal{N}' \geq \mathcal{N} \implies E(\mathcal{N}') \geq E(\mathcal{N})$$

Call OrchNet $\mathcal{N}$ loosely monotonic *if*

$$\forall \mathcal{N}' : \mathcal{N}' \geq \mathcal{N} \implies F(\mathcal{N}') \geq F(\mathcal{N})$$

The following immediate result justifies considering also the loose end-to-end QoS:

Theorem 1 *Any OrchNet is loosely monotonic.*

Consequently, it is always sound to base contract composition and contract monitoring [RBHJ08] on loose end-to-end QoS. This, however, has a price, since loose end-to-end QoS is pessimistic compared to (actual) end-to-end QoS. The next theorem gives conditions ensuring monotonicity—based on $E(\mathcal{N})$:

Theorem 2 *OrchNet $\mathcal{N} = (N, V, Q, Q_{\text{init}})$ is monotonic if and only if:*

$$\begin{aligned} & \forall \omega \in \Omega, \forall \kappa \in \mathcal{V}(N) \\ \implies & E_\omega(\kappa, \mathcal{N}) \geq E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N}) \end{aligned} \tag{3.14}$$

where $\mathcal{V}(N)$ is the set of all maximal configurations of net N and $\kappa(\mathcal{N}, \omega)$ is defined in (3.9).

Condition (3.14) expresses that 1 implements globally optimal service selection. It is costly to verify and may not even be decidable in general.

Thus, we develop a structural condition for monotonicity for *Orchestration nets* N (Section 3.4.1). Orchestration net N induces an OrchNet $\mathcal{N}_N = (U_N, \nu_N, Q_N, Q_{\text{init}})$ by attaching, to each transition t of the unfolding U_N of N, the value and QoS inherited from N through the unfolding $N \mapsto U_N$.

Theorem 3 *Let N and U_N be as before. A sufficient condition for OrchNet $\mathcal{N}_N = (U_N, \nu_N, Q_N, Q_{\text{init}})$ to be monotonic is that every branching cell B of U_N satisfies the following condition:*

$$\forall t_1, t_2 \in B, t_1 \neq t_2 \implies \varphi(t_1)^\bullet = \varphi(t_2)^\bullet. \tag{3.15}$$

If, in addition, every transition of N is reachable and partial order $(\mathbb{D}, \leq)$ is such that for every $q \in \mathbb{D}$, there exists $q' \in \mathbb{D}$ such that $q' > q$, then (3.15) is also necessary.

The notion of branching cell is dynamic in that it is defined on the unfolding U_N. We can propose a simpler structural condition by using the known notion of "cluster" [Mur89] directly defined on N. For a net N, a *cluster* is a minimal set $\mathbf{c}$ of places and transitions of N such that

$$\begin{aligned} \forall t \in \mathbf{c} &\implies & {}^\bullet t &\subseteq \mathbf{c} \\ \forall p \in \mathbf{c} &\implies & p^\bullet &\subseteq \mathbf{c} \\ \forall p \in \mathbf{c} &\implies & \{t \in \mathcal{T} \mid {}^\circ t \ni p\} &\subseteq \mathbf{c} \end{aligned} \tag{3.16}$$

The third condition accounts for read arcs, whereas the other conditions characterize clusters for standard Petri nets (without read arcs). It is easily seen that, for B any branching cell of the unfolding U_N (see Section 3.4.1), $\varphi(B) \subseteq \mathbf{c}$, for some cluster $\mathbf{c}$ of N. The following is an immediate consequence of 3:

Corollary 1 *A sufficient condition for the OrchNet $\mathcal{N}_N = (U_N, \nu_N, Q_N, Q_{\text{init}})$ to be monotonic is that every cluster $\mathbf{c}$ of N satisfies the following condition:*

$$\forall t_1, t_2 \in \mathbf{c}, t_1 \neq t_2 \implies t_1^\bullet = t_2^\bullet. \tag{3.17}$$

In words, a sufficient condition for monotonicity is that, each time branching has occurred in net N, a join occurs right after.

Figure 3.6: Enforcing monotonicity through service aggregation, mid diagram, with $\delta q'_{12} = \delta q'_1 \oplus \delta q'_2$ and $\delta q''_{12} = \delta q''_1 \oplus \delta q''_2$. Pessimistic QoS evaluation, right diagram, with $\delta q_{12} = \delta q'_{12} \vee \delta q''_{12}$.

3.4.4 Probabilistic OrchNets

To account for uncertainties in QoS performance, soft probabilistic contracts were proposed in [RBHJ07], with associated composition and monitoring procedures, for the particular case of response time. In [RBJ09a, RBHJ08] the above approach was extended to more general QoS. In this section, we describe the corresponding model of *probabilistic OrchNets*, an extension of OrchNets supporting probabilistic behavior of QoS parameters. Details are found in [RBJ09b].

In probabilistic OrchNets, the nondeterministic QoS functions ξ_t and initial QoS ξ_p are now random, and so are the non-deterministic selections of minima in competition step of 1. Equivalently, the set Ω for the values of the daemon is equipped with some probability $\mathbf{P}$. In [RBJ09b], we also study monotonicity of probabilistic OrchNets. To define monotonicity, we need to give a meaning to (3.13) when ξ_t is random. This is achieved by considering the *stochastic partial order* [Tet77] induced by partial order $\leq$ defined on $\mathbb{D}$. We briefly recall this notion next—see [RBJ09b] for details. Consider *ideals* of $\mathbb{D}$, i.e., subsets I of $\mathbb{D}$ that are downward closed: $x \in I$ and $y \leq x \implies y \in I$. Examples of ideals are: for $\mathbb{R}_+$, the intervals, $[0, x]$ for all x; for $\mathbb{R}_+ \times \mathbb{R}_+$ equipped with the product order, arbitrary unions of rectangles $[0, x] \times [0, y]$. Now, if ξ has values in $\mathbb{D}$, we define its *distribution function* by $F(I) = \mathbf{P}(\xi \in I)$, for I ranging over the set of all ideals of $\mathbb{D}$. For ξ and ξ' two random variables with values in $\mathbb{D}$, with respective distribution functions F and F', define $\xi' \geq^s \xi$ iff for any ideal I of $\mathbb{D}$, $F'(I) \leq F(I)$ holds. With this new interpretation of the order, we show in [RBJ09b] that Theorems 1–3 are still valid.

3.4.5 Ensuring Monotonicity

1 in Section 3.4.3 provides guidelines regarding how to enforce monotonicity. Consider again the workflow of Fig. 3.5 and the two alternative branches beginning at the place labeled with QoS q'_0 and ending at the place labeled with the QoS q'_2. This pattern is a source of non-monotonicity as we have seen. One way of enforcing monotonicity is by invoking 1 in Section 3.4.3. Aggregate the two successive transitions in each branch and regard the result as a single transition (t'_{12} for the left branch and t''_{12} for the right branch). The QoS increments of t'_{12} and t''_{12} are equal to $\delta q'_{12} = \delta q'_1 \oplus \delta q'_2$ and $\delta q''_{12} = \delta q''_1 \oplus \delta q''_2$, respectively. The resulting Orchestration net satisfies the condition of 1 and thus is monotonic. This process of aggregation is illustrated on Fig. 3.6, mid diagram.

An alternative to the above procedure consists in not modifying the orchestration

but rather changing the QoS evaluation procedure. Referring again to Fig. 3.5, isolate the part of the workflow that is a source of non-monotonicity, namely the subnet shown on Fig. 3.6, left. For this subnet, use pessimistic formula (3.12) to get a pessimistic but monotonic bound for the QoS of this subnet. For this example, the pessimistic bound is equal to $\delta q_{12} = \delta q'_{12} \vee \delta q''_{12}$. We then plug the result in the evaluation of the QoS of the overall orchestration, by aggregating the isolated subnet into a single transition t_{12}, with QoS increment δq_{12}. This is illustrated on Fig. 3.6, right diagram.

The above two procedures yield different results. By aggregating service calls performed in sequence, the first procedure delays the selection of the best branch. The second procedure does not suffer from this drawback. In turn, it results in a pessimistic evaluation of the end-to-end QoS. Both approaches restore monotonicity.

3.5 Summary of the Theory for Practical Use

So far we have introduced a rich body of technical material to support contract based QoS management of data-and-QoS dependent orchestrations. Some of this material was needed for the soundness of our approach but need not be implemented to perform QoS management. In this section we collect the key concepts we really need to implement.

Monotonicity Orchestrations must be monotonic in that

$$\begin{array}{l}\text{if any requested service performs better,}\\ \text{then so will the orchestration.}\end{array} \tag{3.18}$$

Our study of monotonicity used the framework of OrchNets in its full depth. For a high level understanding, the reader may ignore this study and simply remember that we have effective ways to check or ensure (3.18).

OrchNets as a Practical Framework for QoS The actual use of the framework of OrchNets for the practical enhancement of an orchestration language is, indeed, limited. To implement rich QoS management on top of an orchestration language, we only need to be able to perform the following:

- To take the proper decision based on QoS regarding competing events, actions, or service calls while executing an orchestration. Key here is to identify which events, actions, or service calls are in competition when making this decision.
- To compute the end-to-end QoS of a given execution of an orchestration by composing the QoS of the different services.

To perform the above, we only need to perform the following tasks, while running an orchestration:

(a) Since the QoS algebra relies on the knowledge of causality relations between events, actions, or service calls, we need to *keep track of causal dependencies while executing the orchestration.*

(b) We must *identify which events, actions, or service calls are in competition at each stage of a given execution of the orchestration.*

(c) We need to *implement the QoS algebra with its relations and operators*

- $\oplus$ (incrementing QoS),
- $\leq$ (comparing QoS), and

– $\lhd$ (resolving competition based on QoS).

(d) Then, we need to be able to *compute the end-to-end QoS* of an execution of the orchestration, following Sections 3.3.3 and 3.4.2.

3.6 Implementing Our Approach

We have implemented our approach and we now present two aspects of this implementation. We first explain how our approach supports separation of concerns in QoS-aware orchestration modeling. We also illustrate contract composition as a method for QoS-based design of composite services.

3.6.1 Weaving QoS in Orchestrations

Separation of concerns has been advocated as a recommended design discipline in the development of complex software systems. The consideration of QoS in composite services is a source of significant increase in complexity. On the other hand, tight interaction between QoS and the function performed makes QoS a crosscutting concern. Aspect Oriented Programming (AOP) has been advocated as a solution to support separation of crosscutting concerns in software development [KLM+97, Kis02]. In AOP, the different *aspects* are developed separately by the programmer. Their *weaving* is performed using *joinpoints* and *pointcuts*, and by having *advice* refining original pointcuts. In this section we develop a compile-time weaving of QoS aspects in composite services.

Observe first that our formal model of OrchNets offers by itself support for separation of concerns in QoS management. Once the involved QoS domains have been specified with their algebraic operations, the execution policy of OrchNets (1) entirely determines how QoS interferes with the execution of the orchestration, see the discussion of the example of Fig. 3.5 in Section 3.3.1.

Van der Aalst's WF-nets (WFnets) [vdA97, vdAtHKB03, vdAvH02] are a Petri net formalism and are thus closely related to the functional part of our OrchNets. The compile-time weaving of QoS into WF-nets is best illustrated by the example of Fig. 3.7, where the XML-like specification explains how a functional description of a composite service can be complemented with its QoS specification. The original functional specification is BPEL-compliant and is written in **boldface**. Add-ons for QoS are written in *`italics`* and consist of the WSLA specification of the Interface, playing the role of a rich SLA specification. Two QoS domains are declared: *`RTime`* (for ResponseTime) and *`Cost`*. These domains come up with the declaration of their associated operators following Section 3.3.2, namely *`Cost.leq, Cost.oplus, Cost.vee, Cost.compet`* and similarly for *`RTime`*—this is not shown on the figure since such QoS domains should be predefined and available from a library. The Interface also contains, for each called site, the declaration of the QoS parameters that are relevant to it—*`site1`* knows only *`RTime`* whereas *`site2`* knows the two. The functional part of this specification (shown in **boldface**) collects four site calls or returns, each of which constitutes a pointcut.

The QoS-enhanced orchestration is automatically generated from the specification shown in Fig. 3.7—to save space, we do not show it but we only discuss the steps performed in generating it. The added code is written in `roman`. The first step is to initialize the metrics relevant to the orchestration:

```
<assign>
   <$orch.RTime = 0 />
   <$orch.Cost = 0 />
</assign>
```

```
<SLA>
  <SLAParameter name = "ResponseTime"
  type = "float" unit = "milliseconds">
        <Metric>ResponseTime</Metric>
     <Function>
        <Metric>ResponseTimeOplus</Metric>
        <Metric>ReponseTimeCompare</Metric>
        <Metric>ReponseTimeCompete</Metric>
     </Function>
  </SLAParameter>
  <SLAParameter name = "Cost"
  type = "integer" unit = "euro">
        <Metric>Cost</Metric>
     <Function>
        <Metric>CostOplus</Metric>
        <Metric>CostCompare</Metric>
        <Metric>CostCompete</Metric>
     </Function>
  </SLAParameter>
  <ServiceDefinition name="orch">
  <MetricURI http://orch.com/getMetric
   ?ResponseTime />
  <MetricURI http://orch.com/getMetric?Cost />
  </ServiceDefinition>
  <ServiceDefinition name="site1">
  <MetricURI http://site1.com/getMetric
   ?ResponseTime />
  </ServiceDefinition>
  <ServiceDefinition name="site2">
  <MetricURI http://site2.com/getMetric
   ?ResponseTime />
  <MetricURI http://site2.com/getMetric?Cost />
  </ServiceDefinition>
</SLA>

<process>
  <sequence>
    <invoke name = "site1(-)" ... />
    <receive name = "site1(-)" ... />
    <invoke name = "site2(-)" ... />
    <receive name = "site2(-)" ... />
  </sequence>
</process>
```

Figure 3.7: Separation of concerns in QoS-aware specification. The functional specification is depicted last in **boldface**, whereas the QoS part is shown in *italics* on top in the form of a rich SLA specification.

The sequence begins with the initialization of the response time carried by the token using the `<assign>` declaration. Concurrent invocation of the `site(-)` and `clock = site.clock.store` follow, using the `<flow></flow>` declaration. Once the `site(-)` returns, the difference between the current `clock` and `site.clock.store` is assigned to `site.RTime`. Resulting weaving is obtained by applying the generic rewriting rule shown on Fig. 3.8. The same mechanism is used for the response time of `site2` and the end-to-end response time of the orchestration follows by adding the above two. Each pointcut shown in **boldface** in this figure is refined by the corresponding advice (in roman) following it.

The end-to-end evaluation of `Cost` for the orchestration is computed in a different way, because this kind of QoS is individually carried by the tokens representing the queries while being processed by the orchestration. Since `Cost` is relevant to `site2` by interface declaration in Fig. 3.7, the call to `site2` is augmented with the return of the cost of calling `site2`. This weaving is obtained by applying the generic rewriting rule

```
  <sequence>
<invoke name = ”site(-)” />
...
<receive name = ”site(-)” />
</sequence>
```

rewrites as:

```
  <sequence>
   <flow>
        <invoke name = ”site(-)” />
      <sequence>
         <invoke "clock()"/>
         <receive "clock()"
          outputVariable = "clock"/>
         <assign>
            <$site.clock.store = $clock />
         </assign>
      </sequence>
   </flow>
   ...
   <flow>
        <receive name = ”site(-)” />
      <sequence>
         <invoke "clock()"/>
         <receive "clock()"
          outputVariable = "clock"/>
         <assign>
            <$site.RTime =
             $clock - $site.clock.store />
            <$orch.RTime =
             $orch.RTime + $site.RTime />
         </assign>
      </sequence>
   </flow>
  </sequence>
```

Figure 3.8: Rewriting rule for weaving response time.

```
  <sequence>
<invoke name = ”site(-)”/>
<receive name = ”site(-)”/>
</sequence>
```

rewrites as:

```
  <sequence>
     <invoke name = ”site(-)” />
     <receive name = ”site(-)”
    outputVariable = "site.Cost"   />
   <assign>
      <$orch.Cost =
       $orch.Cost + $site.Cost />
   </assign>
  </sequence>
```

Figure 3.9: Rewriting rule for weaving cost.

of Fig. 3.9. Here, the **`invoke`** pointcut is not refined, only the **`receive`** is refined, by the advice code (in roman) following it.

The automatic generation of the augmented program from the original specification is a direct coding of the 1. Rules for other constructions such as the firing of a transition with several input places and the competition when a token exits a place with possible choices, are derived similarly, following 1. For general WFnets, we must keep track of the different tokens and attach QoS values to them. This amounts to keeping track of causalities between site calls that result from the WFnet. To support the weaving,

pointcuts need not be explicitly declared by the programmer. They are instead obtained by pattern matching searching for keywords **invoke** and **receive** in the functional specification.

Instead of developing a tool implementing the above technique for WFnets, we have performed a prototype implementation on top of the Orc orchestration language. This is explained in the next section and subsequently illustrated using the TravelAgent2 example of Fig. 3.2.

3.6.2 Upgrading Orc for QoS

Background on Orc Orc [CPM06b] is a general purpose language aimed to encode concurrent and distributed computations, particularly workflows and Web service orchestrations. An orchestration described in Orc is essentially an Orc expression. An Orc expression is either a *site* or is built recursively using any of the four Orc combinators. A site models any generic service. A site can be *called* with a list of parameters, and all these parameters' values have to be defined before the call can occur. A call to a site returns (or *publishes*) at most one value; it may also *halt* without returning a value. The identity site, which publishes the value x it receives as a parameter, is denoted by x (the name of its parameter). Orc allows composing service calls or actions by using a predefined small set of combinators that we describe next. In the *parallel composition* $f \mid g$, expressions f and g run in parallel. There is no direct interaction between parts of f and g and the returns are merged by interleaving them. The *sequential composition* $f >x> g$ starts by running f. For every value v published by f, a *new* instance of g is run in parallel, with the value of x bound to v in that instance. As a particular case, $f \gg g$ performs f and then g, in sequence. The *pruning composition* $f <x< g$ runs f and g in parallel. When g publishes its *first* value v, the computation of g is terminated, and occurrences of x in f are replaced by v. Since f is run in parallel with g, site calls in f that have x as a parameter are blocked until g publishes a value. Finally, the *otherwise* combinator $f; g$ runs f first. If f publishes a value, g is entirely ignored. However if the computation of f *halts*, then g is run. Orc also has built in sites to track passage of time (Rclock, Rwait), deal with data structures (tuples, lists, records), handle concurrency (semaphores, channels) and define new sites (class). An interested reader is referred to the Orc documentation [5] for details.

Upgrading Orc We now describe how we integrate our QoS framework into the Orc language. In particular, we explain how we perform the four tasks (a), (b), (c), and (d), of Section 3.5 within Orc.

The first task (a) is to track the causal relations between execution events in the Orc interpreter. This was straightforward for WFnets, since causality is revealed by the graph structure of the net. It is not immediate for Orc programs, however. The event structure semantics of Orc [RKB+07] served as a formal specification for this. It turns out that causality can be cast into our generic algebraic framework for QoS developed in Section 3.3.3. Causalities are represented as pairs $x = (e, C)$, where e is the considered event and $C = \{x_1, \ldots, x_k\}$ is the set of its direct causes, recursively encoded as pairs of the same kind. The QoS domain encoding causalities is defined similarly to the QoS domain "Cost" of Section 3.3.2. Consequently, the generic technique developed to weave QoS into an Orc program can be instantiated to generate causalities. Details will be reported elsewhere. As a small illustration example, consider the computation of causalities for the following Orc program:

$$((2 \gg x) <x< (1 \gg 3)) \gg print(4)$$

[5] http://orc.csres.utexas.edu/documentation.shtml

We apply our generic weaving method by seeing causality as a QoS domain. We make use of two data structures in Orc : tuples, such as (f, g) and finite lists, such as $[f, g]$. The causal history is stored as a list of lists with the tuple (publication, causal past) published in the transformed program. The weaving yields the following causality-enhanced Orc program:

$$
\begin{array}{l}
(\\
\quad (\\
\quad\quad ((2, []) >t> (x >(x0, _)> (x0, union([x], [t])))) <x< \\
\quad\quad ((1, []) >t> (3, [t])) \\
\quad) >t> (("print", [t]) >x0> (print(4), [x0])) \\
)
\end{array}
$$

The first event has an empty causal past (represented by []). Through pattern matching, this is propagated to the next event with causal history accumulated. The output of its execution yields the partial order of causes of the publication of $print(4)$:

$$4(signal, [(print, [(3, [(3, [(1, [])]), (2, [])])])])$$

So far for task (a). Focus now on task (b). In its basic form, Orc does offer a way to select one publication among several candidate ones, namely by using the pruning operator. Indeed, in the Orc expression

$$f <x< (E_1 \mid E_2 \mid \cdots \mid E_n) \tag{3.19}$$

the first publication by E_1, E_2, ..., or E_n, preempts any future publication of the parallel composition $g \triangleq E_1 \mid E_2 \mid \cdots \mid E_n$. Since only one publication of g is picked, all possible publications of g are in mutual conflict when in the context of (3.19). One can regard (3.19) as implementing task (b) for the particular case when the conflict is resolved on the basis of the time of occurrence of the conflicting publications, seen as a QoS parameter—only the earliest one survives. We propose to lift the Orc pruning operator by resolving the conflict on the basis of an arbitrary QoS parameter q given as a parameter of the generalized pruning:

$$f <x<_q (E_1 \mid E_2 \mid \cdots \mid E_n) \tag{3.20}$$

Expression (3.20) is macro-expanded in core Orc for the case where f is the identity:

$$\begin{aligned} & x <x<_q (E_1 \mid E_2 \mid \cdots \mid E_n) \\ \triangleq \quad & x <x< \ \mathbf{sort}_q(E_1 \mid E_2 \mid \cdots \mid E_n) \end{aligned} \tag{3.21}$$

where expression $\mathbf{sort}_q(E_1 \mid E_2 \mid \cdots \mid E_n)$ stores as a stream all publications of $(E_1 \mid E_2 \mid \cdots \mid E_n)$ upon termination and then reorders this stream according to the partial order defined by QoS parameter q.[6] For the special case where QoS parameter q is just the response time d, then $x <x<_d (E_1 \mid E_2 \mid \cdots \mid E_n)$ boils down to $x <x< (E_1 \mid E_2 \mid \cdots \mid E_n)$, the original pruning operator.

Task (c) of implementing the QoS algebra is handled as in the SLA declarations of Section 3.6.1.

Finally, task (d) is much less obvious than for WFnets. The reason is that Orc does not handle explicitly states, transitions, and causality. Rewriting rules are needed that automatically transform functional Orc code by enhancing it for QoS, structurally. This resembles what we briefly presented regarding causality. Details will be presented elsewhere.

[6]Since QoS values may be partially ordered, this choice could be non-deterministic.

3.6.3 The TravelAgent2/3 Example in Orc

The TravelAgent2 orchestration is specified in Orc following the informal specification of Section 3.1. In addition, declarations for contractual specifications of the overall orchestration (Fig. 3.2) are included. The QoS-weaved description as in Section 3.6.2 is also provided to efficiently handle data and QoS.

SLA Declaration

For the TravelAgent2 example, the chosen QoS metrics are:

1. *Inter-Query Time*: This metric specifies the ordering $\leq$ of inter-query intervals for each site. For this metric the $\oplus$ operator is not specified, since QoS is not incremented when traversing sites.

2. *Response Time*: This metric specifies the order $\leq$, increment $\oplus$ and synchronization $\vee$ of response time values. The simulation clock is used to compute the latency between site call and return, which is then passed through a QoS enhanced token. The bestQoS operator is used to choose among the competing values according to the order specified.

3. *Cost*: This class is known by site Cost() treated as described in Section 3.3.2. The partial ordering $\leq$, competition $\lhd$ as bestQoS, increment $\oplus$ and synchronization $\vee$ are specified.

The declaration of the QoS operators and algebra is presented in Fig. 3.10 for the TravelAgent2 orchestration. We make use of the `def class` declaration to implement new sites to track QoS metrics. Data structures and operations on records $\{. \quad .\}$, lists $[f, g]$ and tuples (f, g) are used. Other general sites available in Orc such as real time (`Rtime`) rewritable storage locations (`Ref` and FIFO channels (`Channel`) are also invoked.

QoS Weaving

The QoS-weaved specification of the TravelAgent2 orchestration is provided with the original functional specification in **bold**(Fig. 3.11) and the QoS specification in roman (Fig. 3.12). The algebra specified in Fig. 3.10 is added as advice code to sites specifying the necessary QoS domains. Increments to domains Cost and ResponseTime are accumulated as the control flow progresses in the orchestration.

The general purpose `Dictionary` site is used as a mutable map from field names to values, references to which are accessed using . access. Values held by references are obtained using `x?` (equivalent to `x.read()`) and set using `x := y` (equivalent to `x.write(y)`).

The advantage of abstracting away such QoS manipulation is that the end-to-end QoS may be extracted as an aspect of the functional specification. The code is installed on the Orc site at url `http://orc.csres.utexas.edu/papers/bjkrt2012tse.shtml`, from where it can be run.

3.7 Evaluation of Our Approach

In this section, we make use of our implementation for performing contract composition, that is, estimating the end-to-end QoS of the TravelAgent2 and TravelAgent3 examples. The former is monotonic whereas the latter is not. Our study illustrates the effect of monotonicity and substantiates the need for the rich theory developed in this paper.

```
def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))

def class InterQueryTime()=
  def QoS(sitex) =
    val s = {. r = Ref(0), c = Channel() .}
    val curTime = Rclock().time()
    s.r?  >p>
    (s.c.put(curTime-p) | s.r := curTime)  >>
    Dictionary()  >sitex>
    sitex.InterQueryTime := s  >>
    stop
  def QoSCompare(it1, it2) = it1 >= it2
  def QoSCompete(it1, it2) = bestQoS(QoSCompare, [it1, it2])
  stop

def class ResponseTime() =
  def QoS(sitex, d) = Rclock().time()-d >q> q
  def QoSOplus(rt1, rt2) = rt1+rt2
  def QoSCompare(rt1, rt2) = rt1 <= rt2
  def QoSCompete(rt1, rt2) = bestQoS(QoSCompare, [rt1, rt2])
  def QoSVee(rt1, rt2) = max(rt1, rt2)
  stop

def class Cost() =
  def QoS(sitex, c)=
    val s = Ref([])
    s?  >x>  QoSOplus(x, [])  >q>  s := q  >>  Dictionary()  >sitex>
    sitex.Cost := s
  def QoSOplus(c1, c2) =
    def Oplus([], []) = []
    def Oplus(x:xs, y:ys) = (x+y):Oplus(xs, ys)
    Oplus(c1, c2)
  def QoSCompare(c1, c2) =
    def Compare([], []) = true
    def Compare(x:xs, y:ys) = (x <= y) && Compare(xs, ys)
    Compare(c1, c2)
  def QoSCompete(c1, c2) = bestQoS(QoSCompare, [c1, c2])
  def QoSVee(c1, c2) =
    def Vee([], []) = []
    def Vee(x:xs, y:ys) = max(x, y):Vee(xs, ys)
    Vee(c1, c2)
  stop
```

Figure 3.10: Declaration of the SLA for the TravelAgent2 orchestration.

The experiments

Each orchestration is specified as a QoS weaved specification such as explained in Appendix. For each trial, QoS values for each called site are drawn according to their specified contracts and then our automatic QoS evaluation procedure applies—we will actually use both the normal QoS evaluation from (3.11) and the "pessimistic" QoS evaluation from (3.12) and compare them. Drawing $20,000$ successive trials yields, using Monte-Carlo estimation, an estimate of the end-to-end contract in the form of a probability distribution. QoS dimensions considered here are *latency, cost*, and *category*. When choices are performed according to two dimensions or more (e.g., cost and category), we make use of a weighing technique following AHP [Tho90].

Fig. 3.13 displays the results of two experiments, corresponding to two different sets of contracts exposed by the called sites, shown on diagrams (a) for latency, and (c) for cost. In order to evaluate the end-to-end QoS of the TravelAgent 2/3 orchestrations in a realistic setting, the AirlineCompany and HotelBooking sites are modeled as distributed applications hosted on a GlassFish 3.1 server on the INRIA local area network—each call to AirlineCompany or HotelBooking results in a parallel call to one of the above mentioned GlassFish applications and the corresponding latency is recorded and used for end-to-end QoS evaluation. Other sites are assumed to react much quicker and are

```
val AirlineList = ["Airline 1", "Airline 2"]
val HotelList = ["Hotel A", "Hotel B"]

--BestQoS and simulation utilities
def bestQ(comparer, publisher) = head(sortBy(comparer, collect(publisher)))
def cat() = if (Random(1) = 1) then "Economy" else "Premium"
val simElapsedTime = Rclock()

--TravelAgent definition
def TravelAgent(SalesOrder, Budget) =

  def inquireCost(List) = each(List) >sup> Dictionary() >ProductDetails>
    ProductDetails.Company := sup >> ProductDetails.cost := Random(100)
    >> ProductDetails
  def inquireCategory(List) = each(List) >sup> Dictionary() >ProductDetails>
    ProductDetails.Company := sup >> ProductDetails.cost := Random(100) >>
    ProductDetails.category := cat() >> ProductDetails

  def compareCost(x, y) = x.cost? <= y.cost?
  def compareCategory(x, y) = if x.category ?= "Economy" then false else
    if y.category ?= "Economy" then true else compareCost(x, y)

  def SubmitOrder(SalesOrder, Budget) = Dictionary() >GenerateInvoice>
    GenerateInvoice.TravelAgent := SalesOrder.ordernumber? >>
    GenerateInvoice.acceptedTime := simElapsedTime.time() >>
    Println("Order "+GenerateInvoice.TravelAgent?+" accepted at time "
    +GenerateInvoice.acceptedTime?) >> (GenerateInvoice, Budget)

  def AirlineCompany(GenerateInvoice) = bestQ(compareCost,
    defer(inquireCost, AirlineList)) >q> GenerateInvoice.AirQuote := q
  def HotelBooking(GenerateInvoice) = bestQ(compareCategory,
    defer(inquireCategory, HotelList)) >q> GenerateInvoice.HotelQuote := q

  def CheckBudget(GenerateInvoice, Budget) = if (GenerateInvoice.AirQuote?.cost? +
    GenerateInvoice.HotelQuote?.cost? <: Budget) then GenerateInvoice else
    (Println("Resubmit Order "+GenerateInvoice.TravelAgent?)>> Dictionary()
    >SalesOrder> SalesOrder.ordernumber := GenerateInvoice.TravelAgent? >>
    (SalesOrder, SubmitOrder(SalesOrder, Budget)))

  def timeout(x, t, SalesOrder) = Let(Some(x) |
    (Rwait(t) >> notifyFail(SalesOrder, "Timeout") >> None()))
  def notifyFail(SalesOrder, reason) =
    Println("Order "+SalesOrder.id?+" failed: "+reason) >> stop

  timeout((SubmitOrder(SalesOrder, Budget) >(GenerateInvoice, Budget)>
  AirlineCompany(GenerateInvoice) >> HotelBooking(GenerateInvoice) >>
  CheckBudget(GenerateInvoice, Budget)) , 2000, SalesOrder)
  >Some(GenerateInvoice)> GenerateInvoice

--Simulation
def simulateOrders(n) = Dictionary() >SalesOrder>
  SalesOrder.ordernumber := n >> Println("Order "+n+" created") >> SalesOrder
  | Rwait(Random(100)) >> simulateOrders(n+1)

simulateOrders(1) >SalesOrder> TravelAgent(SalesOrder, 50) >GenerateInvoice>
  Println("Invoice for order "+SalesOrder.ordernumber?+"
  presented at time "+simElapsedTime.time()) >>stop
```

Figure 3.11: Orc functional specification.

drawn from a *Student-t* distribution, not shown in the figures. Costs, on the other hand, are drawn from some Gaussian distributions (with small variance/mean ratio); note that we could as well have costs deterministic, this would not change our method. The random numbers and distributions were generated using the built-in functions in the MATLAB statistics toolbox[7]. For N Monte-Carlo runs of the orchestration, the simulation time considering the response time δ, MATLAB proccessing time μ and timeout T would be $\sum_{i=1}^{N} (\max(\delta_i, T) + \mu_i)$.

Fig. 3.13 displays the estimated end-to-end QoS in diagrams (b) for latency and (d) for cost. The results are shown for both the normal QoS evaluation from (3.11) and the "pessimistic" QoS evaluation from (3.12). Not surprisingly, pessimistic evaluation yields larger end-to-end QoS estimates.

Now, recall that TravelAgent2 is monotonic, whereas TravelAgent3 is not. What are the consequences of this? In Fig. 3.13-right, the cost for AirlineCompany2 has been reduced as compared to Fig. 3.13-left. For the monotonic orchestration TravelAgent2, this reduction results in a reduction of the overall cost. For the non-monotonic orchestration TravelAgent3, however, this reduction gives raise to an *increase* in overall cost. On the other hand, pessimistic QoS evaluations are always monotonic, see 1; the results shown

[7] http://www.mathworks.com/products/statistics/

```
--include the SLA Declaration sites
include "SLADeclaration.orc"

val AirlineList = ["Airline 1", "Airline 2"]
val HotelList = ["Hotel A", "Hotel B"]

--BestQoS and simulation utilities
def bestQ(comparer, publisher) =  head(sortBy(comparer, collect(publisher)))
def cat() = if (Random(1)=1) then "Economy" else "Premium"
val simElaspedTime = Rclock()
val RTimer = Ref(0)

--TravelAgent definition
  def TravelAgent(SalesOrder, Budget, ResponseTime, Cost  ) =

  def inquireCost(List) = each(List) >sup> Dictionary() >ProductDetails>
    ProductDetails.Company := sup >> ProductDetails.cost := Random(100) >>
    ProductDetails
  def inquireCategory(List) = each(List) >sup> Dictionary() >ProductDetails>
    ProductDetails.Company := sup >> ProductDetails.cost := Random(100) >>
    ProductDetails.category := cat() >> ProductDetails

  def compareCost(x, y) = x.cost? <= y.cost?
  def compareCategory(x, y) = if x.category ?= "Economy" then false else if
    y.category ?= "Economy" then true else compareCost(x, y)


  def SubmitOrder(SalesOrder, Budget) = (Dictionary() >GenerateInvoice>
    GenerateInvoice.TravelAgent := SalesOrder.ordernumber? >>
    GenerateInvoice.acceptedTime := simElaspedTime.time() >> Println("Order "
    +GenerateInvoice.TravelAgent?+" accepted at time "+GenerateInvoice.acceptedTime?)
    >> (GenerateInvoice, Budget), Rclock().time() )
     >((GenerateInvoice, Budget), d)> ((GenerateInvoice, Budget),
     ResponseTime().QoS(SubmitOrder, d))

  def AirlineCompany(GenerateInvoice, Cost ) = (bestQ(compareCost,
    defer(inquireCost, AirlineList)) >q> GenerateInvoice.AirQuote := q >>
     Cost().QoS(AirlineCompany, [q]), Rclock().time()  ) >(q, d)>
     (q, ResponseTime().QoS(AirlineCompany, d))
  def HotelBooking(GenerateInvoice, Cost) = (bestQ(compareCategory,
    defer(inquireCategory, HotelList)) >q> GenerateInvoice.HotelQuote := q >>
     Cost().QoS(AirlineCompany, [q]), Rclock().time()   ) >(q, d)>
     (q, ResponseTime().QoS(HotelBooking, d))

  def CheckBudget(GenerateInvoice, Budget, Cost  ) =   (if
    (GenerateInvoice.AirQuote?.cost? + GenerateInvoice.HotelQuote?.cost? <: Budget)
    then GenerateInvoice else (Println("Resubmit Order " +GenerateInvoice.TravelAgent?+
    " with bigger budget")>> Dictionary() >SalesOrder> SalesOrder.ordernumber :=
    GenerateInvoice.TravelAgent? >> (SalesOrder, SubmitOrder(SalesOrder, Budget)))
     >r> Cost().QoSOplus(GenerateInvoice.AirQuote?, GenerateInvoice.HotelQuote?) >v>
     Cost().QoSCompete(v, Budget) >> r, Rclock().time()  ) >(r, d)>
     (r, ResponseTime().QoS(CheckBudget, d))

  def timeout(x, t, SalesOrder) = Let(Some(x) | (Rwait(t) >>
    notifyFail(SalesOrder, "Timeout") >> None()))
  def notifyFail(SalesOrder, reason) = Println("Order "+SalesOrder.id?+"
    failed: "+reason) >> stop

  SubmitOrder(SalesOrder, Budget) >((GenerateInvoice, Budget), RT )>
  timeout((AirlineCompany(GenerateInvoice, Cost ) >(q, RT1  )> (q,
   ResponseTime().QoSOplus(RT, RT1)  ) >(q, RT  )>
  HotelBooking(GenerateInvoice, Cost ) >(q, RT2 )> (q,
   ResponseTime().QoSOplus(RT, RT2) ) >(q, RT )>
  CheckBudget(GenerateInvoice, Budget, Cost)) >(r, RT3)> (r,
   ResponseTime().QoSOplus(RT, RT3)) >(r, RT)>
   RTimer := RT >> r, 2000, SalesOrder) >Some(r)>
  (GenerateInvoice, ResponseTime().QoSCompete(RTimer?, 2000))
  >(GenerateInvoice, RT)> (GenerateInvoice, RT)
     | InterQueryTime().QoS(TravelAgent)

--Simulation
def simulateOrders(n) = Dictionary() >SalesOrder> SalesOrder.ordernumber := n
  >> Println("Order "+n+" created") >> SalesOrder |
  Rwait(Random(100)) >> simulateOrders(n+1)

simulateOrders(1) >SalesOrder> TravelAgent(SalesOrder, 50, ResponseTime,
   Cost ) >GenerateInvoice> Println("Invoice for order "+SalesOrder.ordernumber?+"
  presented at time "+simElaspedTime.time()) >> stop
```

Figure 3.12: Orc QoS-weaved specification

conform to this theorem.

Once these end-to-end measurements are taken, the negotiation of contracts and their monitoring may be done as in [RBHJ07, RBHJ08]. This follows the Monte-Carlo procedure explained in [RBHJ07, RBHJ08] and is thus omitted.

(a)

(b)

Figure 3.13: We show results from two experiments. For each experiment we display cumulative densities of: (a) Measured latency of invoked services (b) End-to-end latency for TravelAgent 2/3 orchestrations through two evaluation schemes (c) Measured cost of invoked services (d) Returned cost invoice of TravelAgent 2/3 orchestrations.

Discussion

When dealing with monotonic orchestrations, our contract composition procedure performs at once, both QoS evaluation and optimization. Competing alternatives are captured by the different choices occurring in the orchestration. According to 1, choice among competing alternatives is by local optimization, which implements global optimization since the orchestration is monotonic. Despite the use of Monte-Carlo simulations, this simple policy is cheaper than global optimization, even if analytic techniques are used for composing probabilistic QoS. Furthermore, when applied at run time, 1 implements late binding of services with optimal selection in a very cheap way.

Of course, there is no free lunch. If the considered orchestration is not monotonic,

the above approach does not work as such, as already pointed out in [ZBN+04, AP05, AR09a], see Section 3.2. The bypasses developed in Section 3.4.5 must be used. The aggregation procedure results in aggregating sites that are called in sequence, which increases granularity of the orchestration. When applied in the context of late binding, the decision is delayed until alternatives have all been explored—thus, it is hard to claim that late binding has been achieved by doing so. If pessimistic evaluation is followed, then immediate choices can be applied but, as we said, the end-to-end QoS evaluation that results is pessimistic in that the evaluation accumulates worst QoS among alternatives. So, none of the above techniques is fully satisfactory for non-monotonic orchestrations. In turn, global optimization always applies and implements best service selection—however, we question the meaning of QoS aware management when orchestrations are non-monotonic.

3.8 Conclusion

We have studied the QoS aware management of composite services, with emphasis on QoS-based design and QoS-based on-line service selection. We have advocated the importance of monotonicity—a composite service is monotonic if a called site improving its QoS cannot decrease the end-to-end QoS of the composite service. Monotonicity goes hand-in-hand with QoS, as we think. For monotonic orchestrations, "local" and "global" optimization turn out to be equivalent. This allowed us to propose simple answers to the above tasks. Corresponding techniques are valid for both deterministic and probabilistic frameworks for QoS. We have proposed techniques to deal with the lack of monotonicity. We have observed that the issue of monotonicity has been underestimated in the literature.

To establish our approach on firm bases, we have proposed an abstract QoS calculus, whose algebra encompasses most known QoS domains so far. How QoS based design and on-line service selection are performed in our approach is formalized by the model of OrchNets. Our framework of QoS calculus and OrchNets supports multi-dimensional QoS parameters, handled as partial (not total) orders. To account for high uncertainties and variability in the performance of Web services, we support probabilistic QoS.

QoS and function interfere; still, the designer expects support for separation of concerns. We provide such a support by allowing for separate SLA declaration and functional specification, followed by weaving to generate QoS-enhanced orchestrations. Our weaving techniques significantly clarifies the specification. Finally, we have proposed a mild extension of the Orc orchestration language to support the above approach—the principles of our extension could apply to BPEL [BPE07] as well.

We believe that our approach opens new possibilities in handling orchestrations with rich QoS characteristics.

Table 3.2: Literature survey: Papers dealing with orchestrations allowing for a *data-dependent* workflow (thus exhibiting a risk of non-monotonicity). The issue of monotonicity is ignored, except in the work of the authors of this paper and in Ardagna et al. [AP05], Alrifai & Risse [AR09a] and Zeng et al. [ZBN+04] where it is identified through the discussion on global versus local optimization.

Paper	**Probabilistic QoS?**	**Algorithms to address Activities 1, 2, and 3**
Yu and Bouguettaya (2008) [YB08]	QoS parameters can be defined as "the prob. of something", composition rules are proposed	Extensive study of QoS algebra Optimization of service selection by Dynamic Programming applied to the orchestration modeled as a directed graph
Bistarelli and Santini (2009, 2009) [BS09a],[BS09b]	Probabilistic QoS supported Analytic techniques for composing component QoS to get overall service QoS	Formal language based on semirings used to aggregate QoS However, composition rules for QoS are not detailed
Frolund and Koistenen (1998) [FK98]	Probabilistic QoS supported presents the QML language for QoS specifications in component based systems	Rich specification of contracts No formal mathematical analysis of QoS aspects
Sato and Trivedi (2007) [ST07]	Probabilistic QoS supported for performance and reliability	Precise evaluation of reliability and performance characteristics in the presence of failures
Marzolla and Mirandola (2007) [MM07]	Probabilistic QoS supported for response time and throughput using queuing network models	Performance bounds for quick identification of bottlenecks in orchestrations
Gilmore et al. (2011) [GGK+11]	Probabilistic QoS supported through UML MARTE profile for latency/throughput only	UML4SOA supports rich and extensible QoS definitions Analysis of QoS via mapping of UML activity diagrams to PEPA timed process algebra
Ivanovic et al. (2010) [ICH10];	Probabilistic QoS not supported	Data-aware QoS estimations used in run-time adaptation and monitoring
Canfora et al. (2006, 2008) [CPEV05b, CPEV05a, CPE+06, CPEV08]	Probabilistic QoS not supported	Techniques to "re-bind" alternative services are extensively studied.
Cardoso et al. (2002, 2004) [CSM02, CSM+04]	Probabilistic QoS supported but with little details Composition of QoS values explained but composition of QoS distributions not explained	Generic formulae presented with rules for composing workflows' QoS and tested on a genome based workflow.
Hwang et al. (2004,2007) [HWSP04, HWTS07]	Probabilistic QoS supported Analytic techniques for QoS composition	Efficient approximations for the analytic evaluation of Probabilistic QoS composition are proposed
Menascé et al. (2008) [MCD08]	Probabilistic QoS supported Analytic techniques for QoS composition, mathematical details provided	Optimal service selection precisely formulated and solved with an efficient heuristic
Zheng et al. (2011) [ZYZB11]	Probabilistic QoS supported Both Simulation and Analytic techniques for QoS composition	Simulation and Analytic techniques are compared
Calinescu et al. (2011) [CGK+11]	Probabilistic QoS supported Analytic techniques for QoS composition (Markov models, DMC, CMC, MDP)	Using probabilistic temporal logic, formally specifies QoS Extensive toolkit and model checkers used to implement Activities 1–4 but little details on algorithms
Zeng et al. (2004, 2008, 2003) [ZBN+04],[ZNB+08, ZBD+03]	Probabilistic QoS supported (restricted to Gaussian distributions) Analytic techniques for QoS composition	Integer programming formulation Compares global constraints and local optimization in dynamic environments Issue of monotonicity pinpointed through discussion of local vs. global QoS optimization
Ardagna et al.(2005) [AP05]; Alrifai & Risse(2009) [AR09a]	Probabilistic QoS not supported	QoS-aware service selection solved via Mixed Integer Linear Programming / Multi-dimension Multi-choice 0-1 Knapsack Problem (MMKP) Issue of monotonicity pinpointed through discussion of local vs. global QoS guarantees
Rosario et al. (2007, 2008, 2009) [RBHJ07, RBHJ08, BRBH09]	Probabilistic QoS supported, Soft Probabilistic contracts restricted to latency Monte-Carlo simulation for QoS composition	In-depth study of monotonicity Contract composition and optimal service binding Statistical QoS contract monitoring
Rosario et al. (2009) [RBJ09b, RBJ09a]	Probabilistic multi-dimensional QoS supported, Soft Probabilistic contracts Monte-Carlo simulation for QoS composition	Probabilistic monotonicity Preliminary version of this paper

Chapter 4

Leveraging Causality for QoS Tracking in Service Oriented Systems

Claude Jard
ENS Cachan, IRISA, Université Européenne
de Bretagne, Bruz, France.

Ajay Kattepur, Albert Benveniste
IRISA/INRIA, Campus Universitaire de Beaulieu,
Rennes-Cedex, France.

John Thywissen
Dept. of Computer Science,
The University of Texas at Austin, U.S.A.

Abstract

Service Oriented Architectures (SOA) allow individual software components (*services*) to autonomously describe their functionalities in order to be composed into more complex services. Such compositions can involve many structured interaction paradigms such as concurrency, access control mechanisms and shared memory references. *Causality* of events in such environments is a crucial tool for analysis of event traces, diagnosis of faulty behavior and more generally tracking Quality of Service (QoS) metrics. In this paper, we make use of the concurrent programming language *Orc* to describe interactions in service oriented systems. Building on the formal semantics of Orc, we provide transformation rules to equip Orc events with their causal histories. As a consequence, we demonstrate that QoS increments produced by events can be tracked in service oriented systems. The transformations are implemented as rewriting rules over the Orc Intermediary Language (OIL), which is an abstract syntax tree using only the core Orc calculus.

4.1 Introduction

Service Oriented Systems [Erl05] have received considerable attention due to the abilities of individual components to be platform agnostic while allowing integration with other such components. A key driver is the use of *web services* [ACKM04] to integrate individual functionalities with structured control flows, thus leading to *composite* services.

The execution of such systems can have many concurrent parties and threads interacting. In such systems, the number of execution paths can be large, leading to indeterminate outcomes. Moreover, shared resources such as references can be a source of indeterminacy which requires access control privileges; absence of this can lead to deadlock in many cases. Techniques to understand events and their causes are needed both to understand the global states and for fault diagnosis.

More generally, some knowledge about causal dependencies is critical in computing Quality of Service (QoS) metrics. As an example, latency of a service obviously depends of the causal structure of its internal events. The latency of the execution of concurrent events will result in the maximum latency of each event, while the latency of causally related events will be the addition of latency values. Tracking causality can also be leveraged to compute security levels, reliability and cost increments associated with each events - useful metrics for characterizing QoS in SOA systems.

Causality [SM94] has been studied the context of distributed systems by making use of partial orders in local clocks. While other techniques such as logical time [Lam78] or event structures [RKB+08] have been used in distributed environments, practical tools for programmers of service oriented systems needs some work. The reasons for extracting the causality between events are:

1. *Concurrency Aware Debugging* - Trace executions in distributed systems can have many possible outcomes due to various sources of non-determinacy introduced at the program/compiler level. However, the partial order executions provided by the causality are not prone to this and may be reproduced on multiple engines. Hence, debugging may be performed more accurately in concurrent systems using causality rather than traces.

2. *QoS aware evaluation* - The causality produced can be extended to tracking and aggregating QoS. While a causally aware event produces its causal past, a QoS aware event provides the QoS increment for each associated domain.

3. *Execution platform constraints* - Though many distributed applications may have multi-threaded semantics, at the token level this may be converted to a single thread (for token/program optimization). Such executions are not observed unless the causality is carefully observed between events in multi-threaded systems.

In this paper, we begin with the formalism proposed in the concurrent programming language *Orc* [KQCM09]. It is an elegant way to describe interactions in service oriented environments. We propose to extend the outputs produced by such Orc programs to include causal histories given in terms of partial orders of events (site calls, site returns and publications).

We give rules to transform a given Orc program into one that can track the causality of executed events. Causal history is presented along with each event and can be used for failure diagnosis and thread management in distributed systems. This is extended to tracking QoS in addition to causality with similar transformation rules. The implementation of this is done in the Orc Intermediary language (OIL) level, that makes use of the core Orc calculus in XML form. We envision this approach to be generic and

one that can be applied to other concurrent formalisms for tracking causality / QoS increments.

The rest of this chapter is arranged as follows: An overview of the Orc semantics and OIL language is provided in Section 4.2. The transformation rules and examples of causality aware Orc are presented in Section 4.3. Extension of these rules for QoS tracking are provided in Section 4.4.

4.2 Orc syntax

The abstract syntax of Orc is given in Table 4.1. The invocation of an orchestration occurs by calling the unique main *Expression* of an Orc program. It may contain statements on *Definitions*, *Values* and *Parameters* passed to the orchestration. The evaluation could return zero or multiple results.

$D \in$ Definition	$::=$	$\mathbf{def}\ y(\bar{x}) = f$
$f, g, h \in$ Expression	$::=$	$p \mid p(\bar{p}) \mid ?k \mid$
		$f \mid g \mid f >x> g \mid f <x< g \mid f\ ;\ g \mid D\ f$
$v \in$ Orc Value	$::=$	$V \mid D$
$w \in$ Response	$::=$	$V \mid D \mid \mathbf{stop}$
$p \in$ Parameter	$::=$	$V \mid D \mid \mathbf{stop} \mid x$
$n \in$ Non-publication Label	$::=$	$V_k(\bar{v}) \mid k?w \mid \tau \mid \bot$
$l \in$ Label	$::=$	$!v \mid n$

Table 4.1: Abstract syntax of the Orc Calculus.

Internal semantics of Orc is presented in Table 4.2. The semantics is operational, asynchronous, and based on labeled transition systems. As is common in small-step operational semantics, the syntax of Orc must be extended to represent intermediate states. $?k$ is used to denote an instance of a site call that has not yet returned a value, where k is a unique handle that identifies the call instance. A publication event, $!v$, publishes a value v from an expression and τ denotes an internal event.

A site call involves three steps: invocation of the site, response from the site, and publication of the result. The Rule SiteCall specifies that a site call $V(\bar{v})$, where $\bar{v}$ is a value, transitions to $?k$ with event $V(\bar{v})$. The handle k connects a site call to a site return. A site call occurs only when its parameters are values; in $V(x)$, where x is a variable, the call is blocked until x is defined. In SiteRet a pending site call $?k$ receives a result w from the environment and transitions to the expression w. There is no assumption that all site calls eventually respond. The Publish rule generates a publication event $!v$ from its argument value v.

The rules DefDeclare and DefCall are evaluated using call-by-name in the DefDeclare rule. A single global set of definitions D is assumed with parameters $\bar{x}$ in DefCall producing an output g.

The Rules for the combinators are as described earlier. When f publishes a value ($f \xrightarrow{!v} f'$), rule SeqV creates a new instance of the right side; $[v/x]g$, the expression in which all free occurrences of x in g are replaced by v. The publication $!v$ is hidden, and the entire expression performs a τ action. Note that f and all instances of g are executed in parallel. Because the semantics is asynchronous, there is no guarantee that the values published by the first instance will precede the values of later instances. Instead, the values produced by all instances of g are interleaved arbitrarily.

Pruning is similar to parallel composition, except when g publishes a value v. In this case, rule PruneV terminates g and x is bound to v in f. One subtlety of these rules is that f may contain both active and blocked subprocesses: any site call that

uses x is blocked until g publishes. In case of Rule OrtherV, if f publishes a value, the expression publishes the resulting evaluation f'. However, if the site f halts ($\perp$), we execute: `stop`; g.

$$\text{SiteCall}\ \frac{k \text{ fresh} \quad \bar{v} \text{ closed}}{V(\bar{v}) \xrightarrow{V_k(\bar{v})} ?k}$$

$$\text{SiteReturn}\ ?k \xrightarrow{k?w} w$$

$$\text{Publish}\ \frac{v \text{ closed}}{v \xrightarrow{!v} \textbf{stop}}$$

$$\text{DefDeclare}\ \frac{D \text{ is } \textbf{def}\ y(\ldots) = \ldots}{D\ f \xrightarrow{\tau} [D/y]\, f}$$

$$\text{DefCall}\ \frac{D \text{ is } \textbf{def}\ y(\bar{x}) = g}{D(\bar{p}) \xrightarrow{\tau} [D/y]\,[\bar{p}/\bar{x}]\, g}$$

$$\text{Par}\ \frac{f \xrightarrow{l} f'}{f \mid g \xrightarrow{l} f' \mid g}$$

$$\text{SeqN}\ \frac{f \xrightarrow{n} f'}{f >x> g \xrightarrow{n} f' >x> g}$$

$$\text{SeqV}\ \frac{f \xrightarrow{!v} f'}{f >x> g \xrightarrow{\tau} f' >x> g \mid [v/x]\, g}$$

$$\text{PruneLeft}\ \frac{f \xrightarrow{l} f'}{f <x< g \xrightarrow{l} f' <x< g}$$

$$\text{PruneN}\ \frac{g \xrightarrow{n} g'}{f <x< g \xrightarrow{n} f <x< g'}$$

$$\text{PruneV}\ \frac{g \xrightarrow{!v} g'}{f <x< g \xrightarrow{\tau} [v/x]\, f}$$

$$\text{OtherN}\ \frac{f \xrightarrow{n} f'}{f\ ;\ g \xrightarrow{n} f'\ ;\ g}$$

$$\text{OtherV}\ \frac{f \xrightarrow{!v} f'}{f\ ;\ g \xrightarrow{!v} f'}$$

$$\text{OtherStop}\ \frac{f \xrightarrow{\perp} \textbf{stop}}{f\ ;\ g \xrightarrow{\perp} g}$$

Table 4.2: Internal Structural Operational Semantic Rules of Orc.

Orc Intermediary Language

In the OIL representation, variable names are eliminated using de Bruijn indices [Bru72]. In the lambda calculus form [Chu85], while functions are nameless, variable names are still used. The de Bruijn indices specify the number of levels to traverse from a *reference depth* to reach the binding value. An example in lambda calculus is $\lambda x.\lambda y.\ x$ becomes $\lambda.\lambda.\ 2$. Note that de Bruijn indexed reference depth from one and OIL indexes reference depth from zero.

4.3 Causality

In this section, the transformation rules for implementing causality within Orc are presented with examples.

We consider any Orc program, which has been already parsed and expanded into its Orc calculus intermediate form. In this program, we distinguish the actions, which are the site calls, the corresponding returns and the publications. An *event* is the occurring of such an action during the execution of the Orc program. The events are linked with causal dependencies, which force the events to be executed in a certain order. We can distinguish three kinds of dependencies:

- the dependencies that are imposed by the control flow of the program defined by the semantics of the Orc combinators;
- the dependencies that are imposed by the binding mechanism of Orc variables;
- the dependencies that are provided by the server executing the site calls. These external dependencies are not part of the Orc description. We will consider that the possible return of a site call is directly caused by this call. In the case where

there exist other dependencies imposed by the server (use of a channel for example), we will consider that there are exposed by the server and are part of the returned value given as a pair (v, X), where v is the return value and X a set of events that are the direct causes of the return.

As shown in Fig. 4.1, the execution of an Orc program/expression proceeds by parsing into the Orc Intermediary Language (OIL) form, that consists of only the core Orc calculus. Our methodology is to provide rewriting rules over OIL that enhances publications with their causal pasts. This enhanced OIL form, when compiled, produces both the original publication along with causal information. Similarly, this can be extended to tracking QoS increments with OIL rewriting rules.

Figure 4.1: Technique to enrich Orc outputs with Causality.

4.3.1 The algebra of causality

Orc events are of three types: *publications*, *site calls* and *site returns*. An event is a pair:

$$e = (v, X)$$

where v is the value of the Orc-event (publication, call, return) and X is, recursively, a finite set of pairs of the same kind as e. For e as above, we write $X = \downarrow e$, $v = v(e)$ by abuse of notation. $\downarrow e$ is empty for the initial events. This defines inductively the domain of events $\mathcal{E}$. $\mathcal{E}$ can be equipped with a partial order relation $\leq$ defined as a prefix relation:

$$e \leq e' \quad \text{iff} \quad \exists \{e_i \mid 1 \leq i \leq n\} \text{ s.t. } (e_1 = e) \wedge (e_n = e') \wedge \bigwedge_{1<i\leq n} e_i \in \downarrow e_{i-1}$$

and we denote by $\rightarrow$ be the transitive reduction of partial order $\leq$ on causalities:

$$e \rightarrow e' \text{ iff } e \leq e' \text{ and } \nexists e'' : \; e < e'' < e'$$

Considering two events $e, e' \in \mathcal{E}$, it is always possible to build a least upper bound (lup) of $\{e, e'\}$ by considering a new event $e'' = (v(e''), \{e, e'\})$ (as usual e'' is denoted by $e \vee e'$). It is also possible to extend an event e with an other event by creating a new event $e' = (v(e'), \{e\})$. We will note as $e \oplus e'$ this new event. $\oplus$ is thus monotonic with respect to the prefix order $\leq$. $\vee$ and $\oplus$ are just simple union of sets.

4.3.2 Transformation rules

The transformation rules, directed by the syntax of an Orc program, are presented inductively. They use a context c, which is a set of events. The expression of the left hand side, bracketed as $[\![]\!]_c$ is rewritten in the expression of the right hand side. The intention of these rules is to propagate the causes (in c) over the Orc combinators to push them toward the publications. The transformed program will publish events $(v(e), \downarrow e)$ in which $v(e)$ is the value published by the original program and $\downarrow e$ the set of events that are causes of this publication. See Figure 4.2 for corresponding rules.

$$
\begin{array}{rcl}
[\![\textbf{def } v(x_1,\ldots,x_n) = f]\!]_c & \rightarrow & \textbf{def } v(x_1,\ldots,x_n) = [\![f]\!]_c \\
[\![f \mid g]\!]_c & \rightarrow & [\![f]\!]_c \mid [\![g]\!]_c \\
[\![f >x> g]\!]_c & \rightarrow & [\![f]\!]_c >x> [\![g]\!]_{\{x\}} \\
[\![f <x< g]\!]_c & \rightarrow & [\![f]\!]_c <x< [\![g]\!]_c \\
[\![v]\!]_c & \rightarrow & (v, c) \\
[\![x]\!]_c & \rightarrow & x >(v, _)> (v, \{x\} \cup c) - v \text{ fresh} \\
\text{– function call} & & \\
[\![v(x_1,\ldots,x_n)]\!]_c & \rightarrow & (x_1,\ldots,x_n) >((v_1, X_1),\ldots,(v_n, X_n))> \\
& & v(((v_1, X_1 \cup c),\ldots,(v_n, X_n \cup c)) - v_i, X_i \text{ fresh} \\
\text{– site call} & & \\
[\![v(x_1,\ldots,x_n)]\!]_c & \rightarrow & (x_1,\ldots,x_n) >((v_1, _),\ldots,(v_n, _))> \\
& & (\text{"}v\text{"}, \bigcup_{1\leq i\leq n} x_i \cup c) >u> v(v_1,\ldots,v_n) \\
& & >(v', X)> (v', X \cup \{u\}) - v_i, v', X \text{ fresh}
\end{array}
$$

Figure 4.2: The basic rules

We now *justify* these rules with comments on each transformation:

- The transformation of an Orc program f starts by evaluating $[\![f]\!]_\emptyset$ having a null causal past. This is passed as a top level expression to any Orc program.
- During a function call, the causes are propagated by adding the context to each causal pasts of the specific parameter.
- The publications are done in three constructs: evaluation of a value v, evaluation of a variable x, or site calls.
  - Publishing a value v is just replaced by the publication of the value-causality pair (v, c) where c is the current context.
  - Publication of a variable x needs to recover the value v of the event x and then to publish a value-causality pair $(v, \{x\} \cup c)$, the causes of this new event being now the aggregation of x and c, the current context.
  - The result of the transformation of site calls is a program able to publish a set of associated events: one event for the call action, one event for the publication of the returned value (v, X) (if the site does not respond, the second event will not be produced). We decided to name the call event by the name of the site.
- For the Orc combinators, the rules are:
  - For the $\mid$ combinator, the causal past is replicated for each of the expressions.

- For the $>x>$ combinator, a dependency in the control flow is recorded. In $f >x> g$, the publications of f are the causes of the publications of g with variable x conveying these causes.
- For the $<x<$ combinator, causal pasts are recorded for each expression with the pruning operator propagating the the value-causality pair using variable x.

The transformations are derived from the syntax of any Orc program. In order for *any* Orc engine to derive the causality of an executed Orc program, this causality calculus must be followed.

Tracking causality on programs using the "otherwise" operator: the otherwise ("f ; g") Orc combinator poses a specific problem. According to its semantics, g must be executed if f halts. The problem is that f is allowed to run until it halts: during this phase, it can produce events (there is no roll-back mechanism). In term of causality, we thus have to track the events that are produced before halting and to consider that there are possible causes of events of g. Since there is no explicit variable to piggyback the causal information between f and g, we have to record the events of f in a special memory. It can be declared in Orc using the Ref() site - a rewritable reference value.

The idea is to maintain the set of maximum events (with respect to the causal order relation) and to use it to implement the otherwise operator. Now, any publishing action is replaced by a publish/record action, denoted for instance by track(u), where u is a publication event. This function can be defined in Orc by:

$$
\begin{array}{l}
\text{val trace} = \text{Ref}([]) \\
\quad \textbf{def}\ max(u,m) = \ \text{if member}(u,m)\ \text{then signal} \\
\qquad \text{else}\ trace?\ >t>\ trace := append([u],t) \gg \text{signal} \\
\textbf{def}\ record(u) = trace?\ >m>\ \max(u,m) \\
\textbf{def}\ track(u) = (u, record(u)) > (y, _) > y
\end{array}
$$

The actual implementation of this tracking can be done more efficiently in Orc by using Scala traits such as mutable sets. The complete transformation system is shown in Figure 4.3.

4.3.3 Orc with Causality: Examples

This was done by appending the OIL output with causal information of events. A program starts off with an orcCauses.EmptyCause site that produces a tuple of `(signal,Set())`. The output of the transformed program returns the Orc publication with its causal past with `Set()` representing a null causal past. Examples of transformed outputs produced by this are shown in Table 4.3.
The OIL output with causality in **bold** is given for the expression $1 >x> x$ in Fig. 4.4. The `orcCauses.WrapConstant` and `orcCauses.WrapVariable` sites implement causality for constants and variables, respectively.

4.4 Causality and QoS Tracking

In this section, we extend the rules for causality to QoS tracking in Orc. A reader is referred to Chapter 3 for further details on QoS algebra and the "weaving" procedure.

$$
\begin{array}{rcl}
[\![\textbf{def } v(x_1,\ldots,x_n) = f]\!]_c & \rightarrow & \textbf{def } v(x_1,\ldots,x_n) = [\![f]\!]_c \\
[\![f \mid g]\!]_c & \rightarrow & [\![f]\!]_c \mid [\![g]\!]_c \\
[\![f >x> g]\!]_c & \rightarrow & [\![f]\!]_c >x> [\![g]\!]_{\{x\}} \\
[\![f <x< g]\!]_c & \rightarrow & [\![f]\!]_c <x< [\![g]\!]_c \\
[\![f \textbf{ ; } g]\!]_c & \rightarrow & [\![f]\!]_c \textbf{ ; } track(("h", trace.getAll())) >x> [\![g]\!]_{\{x\}} \\
 & & -x \text{ fresh} \\
[\![v]\!]_c & \rightarrow & track((v,c)) \\
[\![x]\!]_c & \rightarrow & x >(v,_)> track((v,\{x\}\cup c)) - v \text{ fresh} \\
\text{– function call} & & \\
[\![v(x_1,\ldots,x_n)]\!]_c & \rightarrow & (x_1,\ldots,x_n) >((v_1,X_1),\ldots,(v_n,X_n))> \\
 & & track(v(((v_1,X_1\cup c),\ldots,(v_n,X_n\cup c))) \\
 & & -v_i, X_i \text{ fresh} \\
\text{– site call} & & \\
[\![v(x_1,\ldots,x_n)]\!]_c & \rightarrow & (x_1,\ldots,x_n) >((v_1,_),\ldots,(v_n,_))> \\
 & & track(("v", \bigcup_{1\le i\le n} x_i \cup c)) >u> v(v_1,\ldots,v_n) \\
 & & >(v',X)> track((v', X\cup\{u\})) \\
 & & -v_i, v', X \text{ fresh}
\end{array}
$$

Figure 4.3: The complete rules

Orc Expression	**Output**
$1 >x> x$	`(1, Set((signal, Set()), (1, Set((signal, Set()))))))`
def $f(x) = x$ $f(1)$	`(1, Set((1, Set((signal, Set()))))))`
import site (+) = "orc.lib.math.Add" 1+2	`(3, Set((1, Set((signal, Set()))),` `(2, Set((1, Set((signal, Set()))))))))`

$((2 \gg x) <x< (1 \gg 3)) \gg 4 \mid 5$	`(5, Set((signal, Set())))`
	`(4, Set((3, Set((signal, Set()),`
	`(3, Set((1, Set((signal, Set()))))))))))`

Table 4.3: Causality Enhanced Orc output

4.4.1 QoS domain

Referring to [BJK+12], the QoS domains consist of:

$$\mathbb{Q} = (\mathbb{D}_q, \leq_q, \oplus_q, \lhd_q)$$

where the dummy symbol "q" refers to a particular (possibly multi-dimensional) QoS metric. The neutral element for $\oplus_q$ is denoted 0. Instances of $\mathbb{Q}$ are listed next.

4.4.1.1 The special competition operator

The special operator $\lhd_q$ is trivial for QoS metrics that are attached to the token, regardless of any ambient metrics or attribute:

$$q \lhd_q (q(1),\ldots,q(k)) \quad = \quad q \tag{4.1}$$

where $(q(1),\ldots,q(k))$ is the tuple of other tokens for consideration in the competition (the losers). Reason is that, for this case, waiting for all competing incoming tokens has no extra cost, and so does comparing. To summarize, for a QoS domain not involving ambient metrics but only token-held metrics, operator $\lhd_q$ is trivial.

```
<oil>
  <sequence>
   <left>
    <call>
     <target><constant><site>orcCauses.EmptyCauses</site></constant></target>
     <args></args>
    </call>
   </left>
   <right>
      <sequence varname="x">
        <left>
          <call>
        <target><constant><site>orcCauses.WrapConstant</site></constant></target>
        <args>
                <constant><integer>1</integer></constant>
                <variable index="0"></variable>
        </args>
       </call>
        </left>
        <right>
          <call>
        <target><constant><site>orcCauses.WrapVariable</site></constant></target>
        <args>
                <variable varname="x" index="0"></variable>
                <variable index="0"></variable>
        </args>
       </call>
        </right>
      </sequence>
    </right>
  </sequence>
</oil>
```

Figure 4.4: OIL Rewriting to include Causal Information.

In contrast, operator $\lhd_q$ is non trivial for QoS metrics such that:

1. some ambient metrics is involved;
2. solving the competition requires collecting incoming tokens for which the ambient metrics may not be optimal.

A typical example of an ambient metrics is duration, as it relies on time. As soon as the considered ambient metrics is involved in combination with additional dimensions (security, cost, quality of data...), solving conflicts generically requires receiving tokens for which the ambient metrics may not be optimal. The $\lhd_q$ competition operator delivers for the outgoing token (which is selected according to best multi-dimensional QoS) an ambient metrics that is equal to the worst one among the tokens taking part in the comparison. Whenever the multiple dimensions are taken as equal citizens (no priority) or else some priority is applied, makes no difference.

However, for the special case of duration acting as a one-dimensional metrics, selection can occur as soon as the first incoming token is received and there is no need to wait for more tokens. Thus, operator $\lhd_q$ is still trivial for this special case of an ambient metrics.

Latency

This is a case of a one-dimensional ambient metrics. Thus, operator $\lhd_q$ is trivial.

$$\text{type } \textit{latency}: \begin{cases} \mathbb{D}_q & = & \mathbb{R} \\ \leq_q & = & \leq \\ \oplus_q & = & + \\ \lhd_q & : & \text{trivial, see (4.1)} \end{cases}$$

Data quality

$$\text{type } \textit{data quality}: \begin{cases} (\mathbb{D}_q, \leq_q) & = & \text{any finite partial order} \\ \oplus_q & = & \vee_q \text{ (associated to } \leq_q) \\ \lhd_q & : & \text{trivial, see (4.1)} \end{cases}$$

This QoS domain captures in particular the following sub-cases:

- *valid/exception*: the finite domain consists of {valid, exception$_I$}, for $I \subseteq \{1 \ldots n\}$, with valid $\leq_q$ exception$_I$ $\leq_q$ exception$_J$ whenever $I \subseteq J$; exception$_I$ means that exception$_i$ have been raised for $i \in I$.
- *security*: the simplest case is to have a security level equal to {high,low} with high $\leq_q$ low. More general partial orders are used in the area of security.
- *reliability*: This can be captured using the domain ({valid, invalid},$\leq_r$), with valid $\leq_r$ invalid. Other operators follow as for the case of data quality. A service returning "invalid" is an indication of a failure.

This category of metrics is non ambient.

Cost

We assume a set $\mathbf{Q}$ of *cost types* and a labeling function $\lambda : \mathbf{Q} \mapsto \mathbb{R}_+$.

$$\text{type } \textit{cost}: \begin{cases} \mathbb{D}_q & = & \mathbf{Q} \mapsto \mathbb{N} \text{ (multiset of cost quanta)} \\ \leq_q & = & \text{partial order of functions with values in } \mathbb{N} \\ \oplus_q & = & \vee_q \text{ (associated to } \leq_q) \\ \lhd_q & : & \text{trivial, see (4.1)} \end{cases}$$

The actual accumulated cost is then obtained by summing over all the cost quanta collected within q.

A slight variation of this QoS domain can be defined to capture stock levels. The difference is that stock can be both incremented or decremented. We must then replace $\mathbf{Q} \mapsto \mathbb{N}$ by $\mathbf{Q} \mapsto \mathbb{Z}$ and adapt the order and $\oplus$ operation accordingly.

This category of metrics is non ambient.

4.4.2 Composite QoS, no ambient metrics involved

Here we consider the case of a multi-dimensional QoS domain, where the different dimensions are handled on an equal basis, with no priority. Observe that the resulting product order cannot be total.

$$\text{type } \textit{composite}: \begin{cases} \mathbb{D}_q & = & \mathbb{D}_{q_1} \times \mathbb{D}_{q_2} \times \cdots \times \mathbb{D}_{q_n} \\ \leq_q & = & \leq_{q_1} \times \leq_{q_2} \times \cdots \times \leq_{q_n} \\ \oplus_q & = & \oplus_{q_1} \times \oplus_{q_2} \times \cdots \times \oplus_{q_n} \\ \lhd_q & : & \text{trivial, see (4.1)} \end{cases}$$

The product order may be altered by considering a *priority*, which is a partial order $\preceq_q$ over the set $\{1, \ldots, n\}$ of dimensions. Thus:

$$(q_1, \ldots, q_n) <_q (q'_1, \ldots, q'_n) \quad \text{iff} \quad \exists 1 \leq j \leq n : \begin{cases} & \forall i \prec j : q_i = q'_i \\ \wedge & q_j <_{q_j} q'_j \end{cases}$$

This yields a total order if the priority and the orders for all dimensions are total. A particular case is that of lexicographic order where the priority is just the natural order among integers.

4.4.3 Composite QoS, with ambient metrics involved

Here we consider the case of a two-dimensional QoS domain, where the second dimension only is ambient.

$$\text{type } \textit{composite/ambient}: \begin{cases} \mathbb{D}_q = \mathbb{D}_{q_1} \times \mathbb{D}_{q_2} \\ \leq_q = \leq_{q_1} \times \leq_{q_2} \\ \oplus_q = \oplus_{q_1} \times \oplus_{q_2} \\ q \lhd_q (q(1), \ldots, q(k)) = (q_1, \max_{i \in I} q_2(i)) \end{cases}$$

where $I \subseteq \{1, \ldots, k\}$ is the subset of the tokens having participated to the competition when the winner is decided.

4.4.4 Extending Orc for QoS

In its basic form, Orc does offer a way to select one publication among several candidate ones, namely by using the pruning operator. Indeed, in the Orc expression

$$f \; <x< \; (E_1 \mid E_2 \mid \cdots \mid E_n) \tag{4.2}$$

the first publication by E_1, E_2, ..., or E_n, preempts any future publication of the parallel composition $g \underline{\Delta}\; E_1 \mid E_2 \mid \cdots \mid E_n$. Since only one publication of g is picked, all possible publications of g are in mutual conflict when in the context of (4.2). One can regard (3.19) as implementing task (b) for the particular case when the conflict is resolved on the basis of the time of occurrence of the conflicting publications, seen as a QoS parameter — only the earliest one survives. We propose to lift the Orc pruning operator by resolving the conflict on the basis of an arbitrary QoS parameter q given as a parameter of the generalized pruning:

$$f \; <x<_{\boldsymbol{q}} \; (E_1, E_2, \ldots, E_n) \tag{4.3}$$

Expression (4.3) is macro-expanded in core Orc

$$\begin{aligned} & f \; <x<_{\boldsymbol{q}} \; (E_1, E_2, \ldots, E_n) \\ \underline{\Delta} \quad & f \; <x< \; \mathbf{sort}_q(E_1, E_2, \ldots, E_n) \end{aligned} \tag{4.4}$$

where expression $\mathbf{sort}_q(E_1, E_2, \ldots, E_n)$ stores all first publications of $E_1, E_2, \ldots, E_n$; upon termination of the entire expression, it then selects a best one according to the partial order defined by QoS parameter q.[1] For the special case where QoS parameter q is just the response time d, then $f \; <x<_{\boldsymbol{d}} \; (E_1, E_2, \ldots, E_n)$ boils down to $f \; <x< \; (E_1, E_2, \ldots, E_n)$, the original pruning operator.

Remark: For expression $\mathbf{sort}_q(E_1, E_2, \ldots, E_n)$, we assume the $\mathbf{sort}_q$ operator waits until the entire expression terminates. It is also possible that $\mathbf{sort}_q$ knows the "best" value that cannot be improved upon; in such cases, it can wait until a site produces this and discards responses from other sites.

4.4.5 Enhancing the algebra of causality to support QoS: first attempt

We first propose a first attempt based on a direct extension of the causality calculus developed in Section 4.3. A QoS-event is a triple

$$e = (v, q; X)$$

[1] Since QoS values may be partially ordered, this choice could be non-deterministic.

where v is an Orc-event (publication), q is its QoS-increment, and X is, recursively, a finite set of pairs of the same kind as e. For e as above, we write $X = \downarrow e$, $v = v(e)$, and $q = q(e)$ by abuse of notation. $\downarrow e$ is empty for the initial events. Special cases are of interest:

- $q(e) = 0$ (neutral contribution to the QoS by event e) but $v(e)$ non trivial corresponds to publications that are not site calls or do not contribute to the QoS.
- $v(e) = \epsilon$ (absence of publication for event e) but $q(e) \neq 0$ corresponds to hidden actions contributing to the QoS.

This defines inductively the domain of QoS-events $\mathcal{E}_q$. $\mathcal{E}_q$ can be equipped with a partial order relation $\leq$ defined as a prefix relation:

$$e \leq e' \quad \text{iff} \quad \exists \{e_i \mid 1 \leq i \leq n\} \text{ s.t. } (e_1 = e) \wedge (e_n = e') \wedge \bigwedge_{1<i\leq n} e_i \in \downarrow e_{i-1}$$

and we denote by $\rightarrow$ be the transitive reduction of partial order $\leq$ on causalities:

$$e \rightarrow e' \text{ iff } e \leq e' \text{ and } \nexists e'' : \; e < e'' < e'$$

So far $q(e)$ is *not* the cumulated QoS value when v is published; it is rather the *increment*, to the QoS, caused by publishing v. So, the natural guess for the cumulated QoS is

$$Q(e) \quad = \quad \left(\bigvee_{e' \rightarrow e} Q(e') \right) \oplus_q q(e) \tag{4.5}$$

Warning: Now, at this point we must observe that this formula does not take into account the effect of the competition function when selecting the best event in operator $\mathbf{sort}_q$. To take this into account, we must enhance the causality calculus to keep track of the events that were in immediate conflict will all events that occurred. Actually, this needs to be done only for immediate conflicts resulting from applying the $\mathbf{sort}_q$ operator. This enhancement is developed in the following section.

4.4.6 Enhancing the algebra of causality to support QoS: the right solution

A QoS-event is a tuple

$$e = (v, q; X, Y)$$

where

- v is an Orc-event (publication);
- q is its QoS-increment;
- X and Y are, recursively, finite sets of tuples of the same kind as e.

For e as above, $X = \downarrow e$ is the set of *causes* of e, $Y = \#(e)$ is the set of events that are in immediate conflict with e under operator $\mathbf{sort}_q$ (thus, $\#(e) = \emptyset$ by convention if v does not result from applying $\mathbf{sort}_q$), $v = v(e)$, and $q = q(e)$ by abuse of notation. $\downarrow e$ is empty for the initial events.

This defines inductively the domain of QoS-events $\mathcal{E}_q$. $\mathcal{E}_q$ can be equipped with a partial order relation $\leq$ defined as a prefix relation:

$$e \leq e' \quad \text{iff} \quad \exists \{e_i \mid 1 \leq i \leq n\} \text{ s.t. } (e_1 = e) \wedge (e_n = e') \wedge \bigwedge_{1<i\leq n} e_i \in \downarrow e_{i-1}$$

Orc Expression	Output
1 | 2	`(1, 8, Set((signal, 0, Set(), Set())), Set((signal, 0, Set(), Set())))` `(2, 5, Set((signal, 0, Set(), Set())), Set((signal, 0, Set(), Set())))`

$((2 \gg x) <x<$ $(1 \gg 3)) \gg 4 \mid 5$	`(5, 1, Set((signal, 0, Set(), Set())), Set((signal, 0, Set(), Set())))`
	`(4, 7, Set((3, 5, Set((signal, 0, Set(), Set()),`
	`(3, 5, Set((1, 5, Set((signal, 0, Set(), Set())),`
	`Set((signal, 0, Set(), Set))))), Set((1, 5, Set((signal, 0, Set(), Set))),`
	`Set((signal, 0, Set(), Set))))))), Set((signal, 0, Set(), Set))))),`
	`Set((3, 5, Set((signal, 0, Set(), Set)),`
	`(3, 5, Set((1, 5, Set((signal, 0, Set(), Set())),`
	`Set((signal, 0, Set(), Set))))), Set((1, 5, Set((signal, 0, Set(), Set))),`
	`Set((signal, 0, Set(), Set))))))), Set((signal, 0, Set(), Set))))))`

Table 4.4: Causality and QoS Enhanced Orc output

and we denote by $\rightarrow$ be the transitive reduction of partial order $\leq$ on causalities.

Now, we are ready to correct formula (4.5) by taking QoS-based conflicts into account. For e an event that occurred, its cumulated QoS $Q(e)$ is computed as follows:

$$Q(e) \quad = \quad \left(\left(\bigvee_{e' \rightarrow e} Q(e') \right) \oplus_q q(e) \right) \lhd \left(Q(e') \mid e' \in \#(e) \right) \tag{4.6}$$

4.4.7 The rules

The rules follow directly from Figure 4.3 and formula (4.6).

4.4.8 Orc with Causality and QoS: Examples

We once again use the OIL rewriting to include both causality and QoS in 4.4. A program starts off with an OrcCauses.EmptyCause site that produces a tuple of **`(signal, EmptyQoS, Set(), Set())`**. This follows the formulation given in Section 4.4.6 with conflicting events tracked as well. The output produces `(Functional Data, QoS Increment, Causality, Conflicting Events)`. Note that only QoS increments are displayed: these may be aggregated into the eq.(4.6) to produce the end-to-end QoS when needed (for example on the final publication).

4.5 Related work

The work by Lamport [Lam78] introduces the notion of logical clocks in distributed systems. Events in a distributed system may be only partially ordered with respect to such clocks. For example, $a \rightarrow b$ implies that a is in the causal past of b; if a and b are concurrent, neither is in the causal past of the other. By assigning a logical clock C_i for each process P_i, conditions to ensure logical time increments may be presented. For instance. if an event $<a>$ in process P_i is sent to process P_j marked with event $<b>$, then $C_i<a> \leq C_j<b>$ (implying $a \rightarrow b$). This can be provided with implementation rules: if a process receives an event with a timestamp T, it should advance its own clock to $C_j \geq T$. An extension of logical time to total ordering of events is also presented. Total ordering $a \Rightarrow b$ is ensured if and only if: $C_i<a> \leq C_j<b>$; or $C_i<a> = C_j<b>$ and processes $P_i \prec P_j$. An example presented where total ordering of events is necessary - shared resources among many distributed processes.

This has been extended in [Mis11] *virtual time* in client-server networks. In this, the concept of virtual time/timeouts (logical time with relevant magnitude) is proposed as a tool in distributed systems. The virtual time may be used to to control independent threads in distributed systems to be executed with some ordering (in virtual time). An event e in this model, if scheduled to happen at time t in the real world, is placed in

the scheduler queue with signature (e, t). Further, if an event has to proceed after k units of time (`Vwait`(t)), the event will be processed as $(e, t + k)$.

A client-service paradigm is proposed in [Mis11]. The clients all have virtual clocks with each event being a `start` or and `end` event; servers do no have a clock but can be shared memories, databases and support `get()` or `put()` operations. The `start` and `end` events on each client has a partial order relationship. Furthermore, causality (events $x \prec y$), monotonicity ($t_x \prec t_y$) and eagerness properties of the system may be studied (eagerness and duration waiting constraints are absent in [Lam78]). A time-stamping algorithm is presented with timing constraints presented for passive, active and quiescent events. The monotonicity conditions on the events can be used to impose order on causally unrelated events. An application of this algorithm is in distributed simulations, where clients cannot advance their clocks until no other messages are received from clients with lower clocks. The algorithm presented can be used to set bounds on the execution of the clients (in virtual time); this can lead to properties such as deadlock-freeness in such distributed settings.

In [JJJR94], the causal partially ordered set is used for online evaluation of properties in distributed computations. Such techniques for trace evaluation in distributed composite services are also studied in [SG03]. A partial order trace analyzer is presented for predicate detection: to determine if a trace satisfies certain properties. Offline analysis of causality in traces is used in [AMW+03] for debugging web-based distributed applications. The use of causal reasoning and resource management for project planning is proposed in [SKD01]. Transformation of concurrent Java programs has been proposed in [BT98] for testing and debugging. Such transformations enables analysis of vector clocks for detecting race conditions in multithreaded programs.

In [PH12], execution traces are used to validate QoS metrics in distributed applications, with focus on modifying traces to maintain QoS levels. In [BMRS10], a soft-constraint logic programming technique is used to solve QoS routing problems, that resembles the QoS algebra proposed in this paper. In [BM11, DNFM+05], c-semiring based algebraic operators are used for QoS composition modeling in SOA. A prioritized choice operator is introduced in this formalism to enhance constraint dependent negotiation specifications. This is also applied to specifying service level agreements in [BM07].

In our work, we extend the publications of distributed systems to publish the causal past of each event. By providing transformation rules on the concurrent programming Orc [KQCM09], the publications are enriched with causal information. This is useful for debugging and can be replicated on other Orc engines (not true of traces). The causality transformation can also be used to accumulate QoS according to developed algebra. While concepts like virtual time and logical time impose control flow constraints on the execution, the transformation rules merely provide the causal history of the events in a distributed setting. This can be extended with rules (such as with virtual time) to diagnosis and ordering of event execution, dependent on the causal past of the event.

4.6 Conclusions

The tracking of causality in distributed environments such as service oriented systems can lead to better debugging and thread level analysis. In this paper, we have demonstrated the inclusion of causal semantics into Orc, a concurrent programming language. Using the Orc calculus form, rules have been presented to publish both events and their causal past. These transformation rules have been extended to QoS algebra, with the causal transformations enabling tracking QoS metrics. The implementation has been done using rewriting rules over the Orc Intermediary language (OIL) form. Such techniques for tracking causality/QoS are generic in nature and may be applied to other

concurrent formalisms.

Chapter 5

Variability Modeling and QoS Analysis of Web Services Orchestrations

Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste
IRISA/INRIA, Campus Universitaire de Beaulieu,
Rennes-Cedex, France.

Claude Jard
ENS Cachan, IRISA, Université Européenne
de Bretagne, Bruz, France.

Abstract

The ever-growing choice in diverse services is making *service orchestration variability* an essential aspect of a composite web service. Influence of this variation on the Quality of Service (QoS) of a composite service is critical and the focus of our work. In this paper, we present a methodology to first model orchestration variability using a *feature diagram* (FD). The FD specifies a product line of orchestrations represented as *configurations* of invoked/rejected atomic services. Second, due to the potentially large set of configurations we employ combinatorial testing techniques to automatically generate configurations covering all valid *pairwise interactions* between services. Third, we analyze QoS variation for each configuration using probabilistic models of QoS. Using a *crisis management system* case study we experimentally show that pairwise generation covers all QoS outliers and eliminates analysis of $> 75\%$ of all possible configurations. The QoS analysis of the pairwise configurations reveals unsafe/ineffective configurations, helps determine realistic Service Level Agreements (SLAs), and provides valuable feedback to help remodel an orchestration.

5.1 Introduction

Inherent choice in an ever-growing world of services is making *orchestration variability* a significant aspect of a composite web service. The different ways of orchestrating atomic services can be seen as either multiple variants of a composite service created offline or an online composite service that reconfigures dynamically. In either case, we expect to observe variation in Quality of Service (QoS) across different orchestrations. This variation in QoS must not only take into account service variability but also the uncertainty/probabilistic nature of QoS itself.

It is important to consider orchestration variability and its implications on composite service behavior. For instance, not considering variability leads to misrepresentation of contractual agreements on QoS [TP05]. Contractual agreements such as service level agreements (SLAs) [PB08] are the industry standard to ensure QoS compliance between service providers and customers. Usual deviations from SLAs are a result of non-incorporation of QoS variability and in particular QoS outliers in its specification. Therefore, we need systematic analysis of variability in order to improve robustness of contractual SLAs.

Modeling variability in web service orchestrations and analyzing the consequent variation in QoS is the principal subject of this paper. We present a methodology to model orchestration variability using *feature diagrams* (FDs). Feature diagrams [KCH+90] provide a graphical constraints-based framework to specify a product-line of orchestrations. Each orchestration in the product-line is represented as an authorized configuration of invoked/rejected atomic services. In most cases the FD specifies a very large set of configurations making exhaustive sampling infeasible. Instead, we sample the set of all possible configurations by systematically analyzing configurations covering all valid pairwise service interactions [BV05]. Finally, we use probabilistic models of QoS [RBHJ08] to analyze variants of orchestrations derived from all valid configurations.

We use our methodology to investigate merits of systematically sampling the set of all configurations of web service orchestrations. Random sampling of configurations, generally employed, is both ineffective and expensive because it cannot be systematic and requires computing QoS values for a large number of configurations. Moreover, random sampling is not easy when FD constraints like mutual exclusion/requirement need to be satisfied. This work focuses on the adaptation of combinatorial interaction testing (CIT) [CDFP97] to select a sample of configurations that covers all pairwise interactions of services while satisfying all FD constraints. We use the recently proposed scalable approach in [PSK+10] for generating these configurations. CIT is based on the observation that most of the faults are triggered by interactions between a small number of variables [KW04]. For example, consider the output quality of printing web pages depending on a hypothetical combination of parameters represented in Table 5.1.

Parameters	**Options**
Operating System	Windows, Linux, Macintosh
Browser	IE, Firefox, Chrome, Opera
Printer Model	HP, Canon, Xerox, Epson
Printer Type	Ink-Jet, Laser
Orientation	Portrait, Landscape
Size	A3, A4, A5, A6
Color	B/W, Multicolor

Table 5.1: Examples of printing parameters requiring comparison.

An exhaustive generation of combinations of these parameter options would entail

1536 cases with many redundancies. Pairwise coverage of optional combinations would require just 17 tests, resulting in a reduction of close to 99%. The number of exhaustive tests will increase exponentially with addition of more parameters/options requiring an employment of efficient sampling strategies.

Pairwise coverage test generation has been used to detect faults in software systems in prior work [BV05], [CDFP97]. However, the application of these coverage-based techniques to sample configurations in service orchestrations is yet to be examined. This work performs such an examination through a series of experiments that aim at investigating several facets of the question: is pairwise service interaction sampling of orchestration configurations effective for overall QoS analysis and the consequent definition of a global SLA?

All experiments are based on a *crisis management system* (CMS) case study described comprehensively in [KGM09]. This paper reports on the following questions:

- Is it possible to automatically sample the orchestration configurations space to select configurations that cover all pairwise service interactions?
- What global QoS metrics can we infer from a pairwise sample?
- How stable is the SLA computed from a pairwise sample? This question is related to the fact that the automatic generation of pairwise configurations is not deterministic and thus the global contract might vary depending on the generated *sample*.
- Is pairwise sampling more effective and efficient compared to exhaustive sampling of the configuration space?

From our experimentation, it is shown that analysis of a family of configurations (and their corresponding QoS values) can be accurately represented by a small set of configurations satisfying pairwise interactions. Consistency of various generated pairwise solutions are also demonstrated through simulations. This comprehensive analysis of variability helps the orchestrator understand the global QoS extremities of the composite service before negotiating a SLA agreement. Deterioration in service quality or non-compliance of SLA standards during online deployment of the service is thus prevented. Improvements in the orchestration model to eliminate some deviant configurations (causing excessive deterioration of end-to-end QoS) or grouping a family of configurations with similar QoS behavior are other extensions of this technique.

This paper is organized as follows. Section 5.2 provide foundations required for our methodology. These include feature diagrams, Orc, pairwise configuration generation and formal description of QoS metrics. The methodology followed in this paper is briefly presented in Section 5.3. In Section 5.4 the crisis management system (CMS) is described. Comprehensive analysis of the CMS case study is done in Section 5.5. Emphasis was placed on the probabilistic distribution simulations and efficient pairwise generation of configurations. Evaluation of these schemes to generate families of QoS output was done in 5.5.3. Study of the robustness of pairwise interactions and its comparison with exhaustive configurations was also done in 5.5.4. Related work in literature is presented in Section 5.6 followed by conclusions and perspectives in Section 5.7.

5.2 Foundations

5.2.1 Modeling Variability in Composite Services

Variability in a composite service derives from choice in several available online services. Each of these configurations represents a set of invoked or rejected atomic services.

Selection of some services in a configuration may compulsorily link the selection of other services, while mutually excluding other services. In this paper, we model the variability in service configurations using a feature diagram (used interchangeably with feature model) often used to model Software Product Lines (SPLs).

Feature Diagrams (FD) introduced by Kang et al. [KCH+90] compactly represent all the products of a SPL (referred to as *configurations* in this paper) in terms of features which can be composed. Feature diagrams have been formalized to perform SPL analysis [SHTB07]. In [SHTB07], Schobbens et al. propose a generic formal definition of FD which subsumes many existing FD dialects. We define a FD as follows:

- A FD consists of k features $f_1, f_2, ..., f_k$
- A feature f_i may be associated with a software asset such as an atomic service.
- Features are organized in a parent-child relationship in a tree T. A feature with no further children is called a leaf.
- A parent-child relationship between features f_p and f_c are categorized as follows:
  - *Mandatory* - child feature f_c is required if f_p is selected.
  - *Optional* - child feature f_c *may* be selected if f_p is selected.
  - *OR* - at least one of the child-features f_{c1},f_{c2},..,f_{c3} of f_p must be selected.
  - *Alternative (XOR)* - one of the child-features f_{c1},f_{c2},..,f_{ck} of f_p must be selected.
- Cross tree relationships between two features f_i and f_j in the tree T are categorized as follows:
  - f_i requires f_j - The selection of f_i in a product implies the selection of f_j.
  - f_i excludes f_j - f_i and f_j cannot be part of the same product and are *mutually exclusive*.

Using the FD we create and validate configurations (i.e a selection of features in the FD) of atomic services invocations/rejections.

5.2.2 Service Orchestration using Orc

While the FD describes a set of services invoked/rejected, it is crucial to formally describe the causal link between the invoked atomic services using an orchestration. The business process execution language (BPEL) [IBM+07], an industry standard for describing orchestrations, has the disadvantages of inherent complexity of the language and restrictions in combinatorial service descriptions [KCW06]. Orc [MC07] serves as a simple yet powerful concurrent programming language to describe web services orchestrations. Though the Orc language is used for our study, the presented methodology is sufficiently general to be applied to other languages like BPEL.

The fundamental declaration used in the Orc language is a *site*. When a *site* is made available to Orc, its type is also made available to the Orc. The type of a *site* is itself treated like a service - it is passed the types of its arguments, and responds with a return type for those arguments. An Orc *expression* represents an execution and may call services to publish some number of values (possibly zero). The *parallel* combinator $F|G$, where F and G are Orc expressions, runs by executing F and G concurrently. The *sequential* combinator, written $F >x> G$ or $F \gg G$, combines the expression F, which may publish some values, with another expression G, which will use the values as they are published; x transmits the values from F to G. The execution of the *pruning*

combinator $F <x< G$ starts by executing F and G in parallel. Whenever F publishes a value, that value is published by the entire execution. When G publishes its first value, that value is bound to x in F, and then the execution of G is immediately terminated. In the *otherwise* combinator, written $F; G$ first site F is executed. If F completes and has not published any values, then G executes. If F did publish one or more values, then G is ignored. In the *fork-join* combinator (F, G), two processes F and G are invoked concurrently. The process waits until a response is obtained from both atomic services. Further examples of using these combinators can be seen in [MC07].

5.2.3 Configuration Generation from Feature Diagram

Combinatorial interaction testing (CIT) has been proposed by Cohen et al. [CDFP97] to select a subset of all combinations of variables that define the input domain of a program, while still guaranteeing a certain level of coverage. This has led to the definition of pairwise interaction testing, or 2-wise testing. This samples the set of all combinations in such a way that all possible pairs of variable values are included in the set of test data. Pairwise testing has been generalized to t-wise testing which samples the input domain to cover all t-wise combinations.

Definition. 1 ***Covering Array*** *- A covering array $CA(N; t, k, v)$ is a $N \times k$ array of data taken from an alphabet of size v, with the property that every $N \times t$ sub-array contains all ordered subsets of size t from v symbols at least once.*

In this definition, N is the number of experiments, the strength t of the array is the parameter that allows achieving 2-wise (pairwise), 3-wise or t-wise combinations. The k columns on this array correspond to all the variables in the input domain. For the generation of services configurations, k is the number of services, and v is 2 since we have only boolean variables (services may be present or absent in a configuration). The problem of generating a minimal covering array for a set of variables is a complex optimization problem that has been studied in extensive prior work for example [CDFP97]. It is important to notice that there exist very few studies that have tackled the automatic generation of CIT in the presence of constraints between variables. In order to include properties that forbid combinations of values, CIT generation techniques have to allow the introduction of constraints in the algorithms that generate covering arrays. We have developed a solution to generate t-wise configurations that satisfy all constraints modeled in a feature model [PSK+10]. This solution is based on the Alloy analyzer and SAT solving.

As the CIT removes redundant solutions, there are a myriad of sets of configurations that satisfy all the pairwise constraints. So, there are many sets of pairwise configuration solutions (referred to as *samples* from now) that exist for a particular feature diagram. The consistency of these samples of solutions must be tested to determine the accuracy and stability in selecting pairwise combinations.

5.2.4 QoS Aspects of the Orchestration

The use of hard contracts to regulate QoS parameters such as response time, availability and so on has been the norm for most SLAs. However these take into account many outliers that are the result of some rare deviations in QoS which generate pessimistic SLAs. Probabilistic analysis of QoS parameters as shown in [RBHJ08] [HWTS07] provides a more realistic study of actual web services' behavior.

The following QoS parameters have been chosen:

1. *Latency / Response Time* (T) - Denotes the overall delay due to the time taken by a web service to respond. It is a discrete value that may be modeled as a long tailed distribution incorporating some *rare deviations*.

2. *Availability* (α) - The probability that a service is active and can respond to a service call. For a well managed service, this value is generally quite high.

3. *Cost* (χ) - Refers to the monetary cost associated with each invocation of a particular atomic service.

4. *Data Quality* (ξ) - A subjective measure of trade off to high Cost and Response times of web services. It measures the "Quality" of the output of the web service and the beneficial aspects of including a new atomic service into the composite orchestration.

Extending these QoS parameters to an orchestration involves the use of Orc combinators as described previously. Taking two sites s_i and s_j, the QoS parameters may be applied as shown in Table 5.2 depending on the Orc combinators used. The cases of composing the service s_{ij} using the *sequential* and *fork-join* combinators have been considered. The latency, cost and availability metrics for the composite service s_{ij} are derived as shown in [CMSA02] with $Max(p, q)$ representing the maxima of the values p and q.

Expression	$s_{ij} \triangleq s_i \gg s_j$	$s_{ij} \triangleq (s_i, s_j)$
Latency	$T(s_{ij}) = T(s_i) + T(s_j)$	$T(s_{ij}) = Max(T(s_i), T(s_j))$
Cost	$\chi(s_{ij}) = \chi(s_i) + \chi(s_j)$	$\chi(s_{ij}) = \chi(s_i) + \chi(s_j)$
Availability	$\alpha(s_{ij}) = \alpha(s_i) \times \alpha(s_j)$	$\alpha(s_{ij}) = \alpha(s_i) \times \alpha(s_j)$

Table 5.2: QoS metrics discussed in [CMSA02] extended to Orc combinators.

5.3 Methodology

We present a methodology designed to examine: (a) A superior technique for sampling the possible configurations to ensure efficient portrayal of QoS behavior of a composite service; (b) The need for probabilistic analysis of QoS in variable service orchestrations. The following steps summarize our methodology:

1. The inputs are: (a) Variability and constraints of a set of configurations of services modeled in a FD; (b) A composite service orchestration in Orc to specify causality and service interactions. The modeling inputs may be specified as a 3-tuple (S, FD, O) where:

   - S is the set of services that can be used. In a configuration, subsets $S_1, ..., S_N$ of these services are used.
   - FD is the constraints for the services included in a particular configuration.
   - O is the set of orchestrations $O_1, ..., O_M$ in a composite service. These orchestrations invoke the services $S_1, ..., S_N$ according to the configuration constraints specified by the FD.

2. The CIT with pairwise constraints satisfied is then used to sample a set of configurations from the FD. This represents a subset of configurations that effectively cover all the exhaustive configurations in the FD.

3. For each of the sampled configurations we analyze the QoS for orchestrations invoking all atomic services in the configuration. These include a set of parameters to analyze tradeoff between atomic services' inclusion / deletion between configurations. Probabilistic models of response time are used to provide an accurate portrayal of the services' behavior along with comparison with other QoS metrics.

4. Comparisons with randomly generated configurations and consistency over multiple sample sets is included to experimentally study the robustness of the proposed pairwise analysis scheme.

For the rest of the paper, we explain in detail this methodology applied to the crisis management system case study.

5.4 Crisis Management System Case Study

Drawing from the comprehensive documentation in [KGM09], the chosen composite service models a typical crisis management system (CMS). The need for such crisis management systems has grown significantly over time with efficient collaboration of various (distributed) parties responsible for speedy assistance and recovery. These are examples of emergency situations that are unpredictable and lead to severe after-effects unless handled immediately. A CMS facilitates this process by orchestrating the communication, co-ordination and deployment between all parties involved in handling the crisis. A thorough analysis of QoS aspects of a CMS will not only ensure optimal performance of such mission critical systems, but also ensure speedy and reliable assistance to the parties in need of aid.

5.4.1 Feature Diagram of CMS

In Figure 5.1, we present the Crisis Management System (CMS) FD [KGM09]. The CMS FD contains several features that are associated with software assets represented by atomic services. For example, the *Local Operator* feature is represented by the *GSMLocalOperator* web service. Constraints such as optional, requires and mutual exclusion (XOR) are also incorporated. For example, the *LocalOperator* and *InternationalOperator* features are mutually exclusive while the *HospitalAdmit* feature requires the *Ambulance* feature.

Figure 5.1: Feature Diagram / Model of the Crisis Management System with associated real-world service assets.

5.4.2 Service Orchestrations in CMS

A host of web services used for the orchestration are described in detail in Table 5.3. These have generic input-output descriptions that can be modified according to requirements.

Web Service	Description
CrisisOrchestration	Uses the customer input to orchestrate the CMS system
CrisisManager	Selects the emergency services to include in the orchestration
CommunicationManager	Selects the communication services to include in the orchestration
GPSLocation	Sets up the GPS location of the emergency area
GSMLocalOperator	Sets up a local GSM communication link for personnel
GSMIntlOperator	Sets up an international GSM communication link for personnel
Ambulance	Contacts and waits for a response from nearby ambulance agencies
Hospital	Contacts and waits for a response from nearby hospitals
Police	Contacts and waits for a response from nearby police stations
Surveillance	Connects to surveillance tapes from the affected area
Fire	Contacts and waits for a response from fire stations

Table 5.3: Web Services in the CMS Orchestration.

The FD (Fig. 5.1) and the orchestration (Fig. 5.2) cover two dimensions that are complementary to each other. While the FD represents the variability in the configurations, the orchestration specifies the order in which the services are called. Making use of the terminology in [SHTB07], *primitive* features are "features" that are of interest and that will be incorporated in real-world services. On the contrary, *decomposable* features are just intermediate nodes used for decomposition. It is up to the modeler to determine such classification of features in the FD. We extend the semantics given in [SHTB07] to ensure compatibility of an orchestration with the feature model as follows:

- The set of available services S are the *primitive* nodes of the FD D;
- For each orchestration, the set of corresponding services invoked (denoted N);
- $N \subseteq S$ in a configuration;

```
def CrisisOrchestration(call,type) = CommunicationManager(call) >>
  CrisisManager(type)
def CommunicationManager(call) = (l,in) >>
  (LocalOperator(in),IntlOperator(in),GPSLocation())
def GPSLocation() = (x,y)
def GSMLocalOperator(l) = Let(query(l) | Timer(l))
def GSMIntlOperator(in) = Let(query(in) | Timer(in))

def CrisisManager(type) = (f,a,h,p,s) >>
  (Fire(f),Ambulance(a),Hospital(h),Police(p),Surveillance(s))
def Fire(f) = Let(query(f) | Timer(f))
def Ambulance(a) = Let(query(a) | Timer(a))
def Hospital(h) = Let(query(h) | Timer(h))
def Police(p) = Let(query(p) | Timer(p))
def Surveillance(s) = Let(query(s) | Timer(s))
```

Table 5.4: Orc representation of the CMS orchestration.

Figure 5.2: Composite Web Service Orchestration of the CMS.

- A model of D is a subset of its (*primitive* and *decomposable*) nodes;
- There must exist a model of D ($[[D]]$) such that $[[D]] \cap S = N$ (a model of a FD is a subtree that is valid w.r.t. the operators and the dependence relation).

Drawing from the real-world services and the constraints shown in Fig. 5.1, the composite service may be developed by an orchestrator. Automatic compositions of composite services from feature model constraints (with additional attributes to describe orchestration interactions), is out of the scope of this paper and will be investigated in future work.

The composite service orchestration is represented succinctly in Fig. 5.2 and the Orc representation is presented in Table 5.4. Calling the *CrisisOrchestration* service invokes the *CommunicationManager* and *CrisisManager* operations in sequence. The *CommunicationManager* service calls the *GPSLocation* and either one of the *GSMLocalOperator* and the *GSMIntlOperator* services that are mutually exclusive (*Mux*). The outputs are synchronized and merged (*Merge*) before dynamically invoking the optional services through the *CrisisManager*. The varying timer values are used to invoke / discard the *Fire*, *Ambulance*, *Hospital*, *Police* and *Surveillance* services. The outputs of these services are merged and synchronized. In the Orc model presented in Table 5.4, the generic service *query()* is used to represent the invocation of a particular web service. The setting of timer values (*Timer()*) results in the various associated configurations in the system and is an example of defining orchestration parameters. Another level of control is the global timeout value associated with the composite service. This has to be associated with the overall SLA of the composite service to provide optimal durations for response. A more verbose BPEL representation of the orchestration is presented, re-emphasizing the clarity and elegance of the Orc representation.

```
<sequence>
  <invoke CommunicationManager    inputVariable="Call"/>
  <flow>
```

```
    <invoke GPSLocation/>
    <pick>
      <onMessage><invoke GSMLocalOperator    inputVariable="l"/></onMessage>
<onAlarm "Timer(l)"><empty /></onAlarm>
    </pick>
    <pick>
      <onMessage><invoke GSMIntlOperator    inputVariable="in"/></onMessage>
      <onAlarm "Timer(in)"><empty /></onAlarm>
    </pick>
  </flow>
  <invoke CrisisManager    inputVariable="type"/>
  <flow>
    <pick>
      <onMessage><invoke Fire    inputVariable="f"/></onMessage>
<onAlarm "Timer(f)"><empty /></onAlarm>
    </pick>
    <pick>
      <onMessage><invoke Ambulance    inputVariable="a"/></onMessage>
      <onAlarm "Timer(a)"><empty /></onAlarm>
    </pick>
    <pick>
      <onMessage><invoke Hospital    inputVariable="h"/></onMessage>
      <onAlarm "Timer(h)"><empty /></onAlarm>
    </pick>
    <pick>
      <onMessage><invoke Police    inputVariable="p"/></onMessage>
      <onAlarm "Timer(p)"><empty /></onAlarm>
    </pick>
    <pick>
      <onMessage><invoke Surveillance    inputVariable="s"/></onMessage>
      <onAlarm "Timer(s)"><empty /></onAlarm>
    </pick>
  </flow>
<sequence>
```

5.5 Experiments

We perform experiments using the methodology described in Section 5.3 for the CMS case study. This involved simulating probabilistic QoS of atomic services, pairwise generation of configurations and finally, analysis of composite services' probabilistic QoS behavior for the variable configurations.

5.5.1 Simulation of QoS Distributions

The first step is simulating the probabilistic response time distributions of each atomic web service as done in [RBHJ08]. For this, we make use of the *t-location distribution* fitting feature in MATLAB as shown in Fig. 5.3. By varying the degrees of freedom ν and non-centrality parameter δ in the *dfittool* of MATLAB, it is possible to generate various heavy tailed distributions that mimic the response times of web services. These are used to simulate the response times of actually invoked atomic services. This t-distribution fitting was used to generate various distributions of services' response times with varying parameters.

5.5.2 Generating a sample of configurations for CMS

We transform the CMS FD to constraint satisfaction problem model in the language Alloy as described in [PSK+10]. All pairwise interactions between features are transformed to Alloy *predicates*. The goal of solving the Alloy model is to find the minimal set of configurations that cover conjunctions of all valid pairwise predicates. The first step involves *detection* of all valid pairs that conform to the FD. In the second step, we construct conjunctions of pairwise predicates and solve them via incrementally increasing the scope of the solution size. The result is a minimal set of configurations

Figure 5.3: Distribution fitting of actual response times of a web service invocation.

that cover conjunctions of all valid pairs.

A set of 15 configurations, **C1** to **C15**, were deemed sufficient by the pairwise generation methodology to represent the configuration sample space. These are shown in Table 5.5 with a × representing service invocation. Guidelines for setting experimental parameters in order to efficiently generate solutions may be found in [PSK+10].

Web Service	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	**C11**	**C12**	**C13**	**C14**	**C15**
CrisisOrchestration	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
CommunicationManager	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
CrisisManager	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
GPSLocation	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
GSMLocalOperator	×	×	×				×	×				×			×
GSMIntlOperator				×	×	×			×	×	×		×	×	
Fire	×			×			×		×			×	×	×	
Ambulance		×	×	×		×		×		×		×	×	×	
Hospital		×	×	×		×		×		×		×	×	×	
Police	×	×		×				×	×	×	×	×	×		
Surveillance	×	×		×				×			×	×			

Table 5.5: Web services in the Orchestration and the variable Configurations (**C1** to **C15**) with × representing a service invocation.

In relation to the configurations in Table 5.5 examples of two generated cases are shown in Fig. 5.4. These configurations cover tuples specified in nos. **C1** and **C15**. While the basic configuration **C15** has none of the optional services, **C1** has three of the optional services invoked. Such variability in orchestration can produce radically different QoS values.

While this view makes use of static invocation of an orchestration (based on the FD configurations), another view is also possible: dynamic invocation of the configurations in a FD by a self-reconfiguring composite service. This would create orchestrations dynamically and link them to a particular FD configuration. However, due to the added control of systematic configuration generation from FDs, we resort to static invocation of orchestrations.

5.5.3 Evaluating QoS of a Composite Service

The efficacy of the QoS analysis procedure was tested experimentally. The web services of the CMS were assigned random response times from a range of heterogeneous t-distributions. The range of parameter values for these distributions in MATLAB included degrees of freedom (ν) varying from 3 to 6 and non-centrality (δ) varying from 5 to 10 seconds.

For an invoked service, the individual timeout value was set sufficiently high (95 percentile of the response time distribution). The global timeout value was also set

Figure 5.4: Varying configurations of the atomic services (a) Configuration **C1** (b) Configuration **C15**.

sufficiently high (300 seconds) to allow capture of outliers in the distribution. For each chosen configuration, $\mathbf{10,000}$ Monte-Carlo runs on the chosen services in the orchestration (representing a partial order of the composite service) was performed. The response time of the orchestration was collected during each run to generate an associated distribution.

As seen in Fig. 5.5, the pairwise generated configurations cover a range of response time distributions. The three worst performing configurations (**C4**, **C8**, **C12**) are compared as an example. The median and 90 percentile changes between these configurations are shown. This demonstrates the use of a few configurations to test significant changes in QoS parameters in a composite service.

In Fig. 5.5, the three worst performing configurations have a significant contribution to the percentile deviations of the response time distribution. This is further seen in the *box-plot* representation in Fig. 5.6. On each box, the *red* central mark is the median, the horizontal edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme data points (not considered outliers) and outliers plotted individually. The boxplot captures the minima, 25, 50, 75 and 95 percentile values of a configuration's response time distribution. The three worst performing configurations (**C4**, **C8**, **C12**), in terms of response times' values, are once again compared in the box-plot (horizontal dotted lines passing through the medians).

Additional parameters such as availability of a service, the cost entailed in calling atomic services and output data quality is also studied in tandem. Using the combinators described in Table 5.2, the QoS parameters were analyzed for each configuration generated by the pairwise interactions. Setting atomic service availability to $\mathbf{0.95}$ (representing service availability in 95% of invocations) the composite availability each configuration is shown in Table 5.7. The output data quality ξ is related to the cost χ by the constant κ given by $\xi = \chi/\kappa$ (assuming linear increase in data quality with each atomic service invocation). For example, setting the $\chi = 5$ units for each invoked atomic service, the cost of each configuration is shown in Table 5.7. Furthermore, setting $\kappa = 20$, the output data quality of the configurations may also be derived. A higher availability and data quality with lower costs and response times are desirable. For example, comparing **C3** and **C4**, calling additional services entails lower availabil-

Figure 5.5: Response times of the pairwise configurations with emphasis on comparing the three configurations with highest response times.

ity and higher costs to the orchestrator, albeit with additional output data quality. Though simplistic in outlook (due to subjectivity of cost and data quality of atomic services), this trade-off of parameters must be taken into account. These myriad of QoS parameters accurately quantify run-time behavior of the composite service.

From these results, the orchestrator can have a global overview of the performance of the composite service. The possibilities include:

1. Setting the SLA keeping into account the worst performing configuration. This will prevent contract deviation during actual deployment of the service.

2. Setting a family of SLAs for a set of configurations taking into account trade-offs between QoS metrics and the output quality of configurations. This leads to a product line of composite services with extensively analyzed SLAs. For example, the configurations **C2**, **C8** and **C13** with very similar characteristics can be grouped as a separate line of services.

3. Eliminating certain deviating configurations to improve the overall performance. This may be done by adding further constraints in the orchestration/feature models. For example, consider the services **C4** and **C12**. Eliminating these configurations (by addition of constraints) reduces the output data quality by 0.25 units as seen Table 5.7. However, it improves the 90, 50, 75 and 25 percentiles of the overall response time distributions by **11.53**, **10.3**, **9.3** and **8.87** seconds respectively. These are significant durations if the orchestrator of a composite service is vying to compete with other companies offering lower response time durations for similar quality services.

Using the pairwise analysis scheme, these imperative qualitative results are obtained with quantitative efficiency even when the number of services are considerably large.

5.5.4 Evaluating the Pairwise Sampling Technique

To experimentally test the efficacy of combinatorial testing the **15** pairwise configurations (Table 5.5) were compared with all the **64** exhaustive independent configurations

Figure 5.6: Box-plot representation of the pairwise configurations with the median values marked for the extreme cases.

of the CMS orchestration. As shown in Fig. 5.7, the comparison is made using the 25, 50, 75 and 90 percentiles of response time distributions for 10,000 Monte-Carlo runs in MATLAB. These families of exhaustive configurations (with few millisecond redundant deviations) are represented by one pairwise configuration. The pairwise configurations are able to capture the extreme values representing greater than **55** seconds of quantile deviation. This represents greater than 75% decrease in the number of exhaustive tests, which will increase in an exponential fashion with introduction of new services.

The accuracy of the pairwise sampling scheme is further demonstrated in Table 5.6 where the *mean* and *maximum* deviations of the pairwise values from the nearest exhaustive values are provided. These are expressed as a percentage of the mean inter-family response time difference. The inter-family response time difference is the average difference between percentile values of two adjacent pairwise samples (8.96 seconds). Compared to this difference, the deviation in accuracy between the pairwise and exhaustive samples can be ignored for practical purposes. Thus, for such orchestrations with numerous configurations, using pairwise interactions is a sufficient choice in order to examine the entire sample space.

Percentile values	25	50	75	90
Mean	1.1326%	1.3471%	1.3438%	1.5471%
Maximum	9.0075%	7.2147%	7.1243%	5.2030%

Table 5.6: Deviations of the pairwise and exhaustive analysis values.

Given one orchestration, there can be many different sets of configurations that cover all pairwise services interactions. To evaluate the efficacy of each *sample* solution, the QoS behavior was computed on the various generated configurations present in a sample. This was done in order to evaluate the stability of pairwise interaction coverage as a sampling heuristic to estimate the global QoS for an orchestration. A collection of **40** samples that satisfy the pairwise interaction testing were generated for the CMS. The statistics of the worst performing configuration (with highest response time) in each sample was collected through 10,000 Monte-Carlo runs and is shown in Fig. 5.8. For example, the highest response time in Fig. 5.7 has the 25, 50, 75 and 90 percentile

Figure 5.7: Comparison of pairwise and exhaustive generation of configurations with 25, 50, 75 and 90 percentile values of response time distributions.

values as 73, 81, 91.5 and 104 seconds, respectively. The objective of studying this variance is to check whether the entire range of QoS values: minima (representing no optional services) to maxima (representing all or most optional services) are present in each pairwise sample. In Fig. 5.8, the percentile values show only a few milli-seconds of deviance. The highest variance of 0.8 seconds seen in the 90 percentile value may be attributed to outliers included in the extreme configurations. An example showing the percentile deviation of two pairwise samples is also shown in Fig. 5.9.

Variance study over a range of samples display the need to analyze many percentiles to accurately estimate the deviation of particular configurations. Use of more than one sample should improve robustness of the offline analysis framework as certain extreme configurations may not occur always. Use of domain specific information may also be required to further ensure robustness of samples. As QoS metrics are modeled as random variables, performing more than one analysis study (combination of more than one sample and various percentile levels) should yield more robust results for SLA computation.

5.6 Related Work

The combinatorial testing framework described by Cohen et al. [CDFP97] has been applied extensively to efficient testing for fault detection. In the work of Cohen et al. [CDS08], this technique is extended to software product lines with highly configurable systems. Modeling variability in SPLs using feature models is the work of Jaring and Boschet [JB02] where they show that the robustness of a SPL architecture is related to the type of variability. To ensure that constraints in the FD are incorporated in the efficient sampling of t-wise tests, the solver proposed by Perrouin et al. [PSK+10] is used. In [MMLP09], variability in software as a service applications are modeled using

Figure 5.8: Percentile values of most deviant scenarios generated by pairwise interactions for the CMS orchestration.

the orthogonal variability model to study the customization choices in such workflows.

Pre-deployment testing of SLAs has been studied by Di Penta et al. [PCE07], where they make use of genetic algorithms to generate test data causing SLA violations. Analysis of white and black box approaches are provided in the paper. In [BCP+05], Bruno et al. make use of regression testing to ensure that an evolving service maintains the functional and QoS assumptions. The service consistency verification due to evolution is done by executing test suites contained in a XML encoded facet attached to the service.

The use of probabilistic QoS and soft contracts was introduced by Rosario et. al [RBHJ08] and Bistarelli et al. [BS09b]. Instead of using fixed hard bound values for parameters such as response time, the authors proposed a soft contract monitoring approach to model the QoS measurement. The composite service QoS was modeled using probabilistic processes by Hwang et al. [HWTS07] where the authors combine orchestration constructs to derive global probability distributions.

In our paper, we extend these two notions to analyze the QoS of a composite orchestration under various configurations. The hard contract notions of end-to-end QoS are replaced by the probability quantile based approach. This provides the service provider the technique for estimating composite service QoS distributions and estimating the global soft contract SLA. Though formal analysis of end-to-end QoS has been studied in Cardoso et al. [CMSA02], there are no practical testing tools available for the service provider. The pairwise testing procedure has been shown to outperform other testing techniques in [CDFP97]. We extend this testing tool to develop a generic testing methodology to query end-to-end QoS of a web service.

Related empirical studies of optimal QoS compositions make use of genetic programming in Canfora et al. [CPEV05a] and linear programming in Zeng et al. [ZBN+04]. These are dynamic techniques to choose the best possible atomic services and configurations keeping QoS in mind. This differs from our work as they assume that there are choices in the best possible atomic web services. The goal in our paper is to analyze the variable configurations that may result due to invocation or non-invocation of particular web services.

Figure 5.9: Comparison of two pairwise samples with 25, 50, 75 and 90 percentile values of response time distributions.

5.7 Conclusion and Perspectives

Accurate offline analysis of a composite web service before its deployment is essential to ensure non-repudiation of a SLA contract. This is necessary to maintain optimal QoS behavior of mission-critical services such as crisis management. In order to do this, the service provider must keep in mind the probabilistic aspect of QoS parameters and the variable configurations in a composite service. In this paper, we study an analysis framework to test the QoS of an orchestration before deployment. Further, the notion of systematic pairwise sampling procedure has also been demonstrated, which provides a more efficient sampling of the configuration space than exhaustive trails while still maintaining sufficient coverage. Larger FD and orchestration models can be analyzed using the divide-and-compose approaches [PSK+10] to handle this scalability issue. This should provide a simple, systematic and stochastically correct methodology for pre-deployment QoS analysis of a composite service.

While this paper concentrates on a particular composition of fixed atomic services, a future area of interest would be optimal compositions. The use of configurations and scenarios modeled by a FD leads to a family of composite services. These, in turn, may be used to generate many versions of the orchestrations. This will prove useful for both obtaining realistic QoS bounds and product generation of families of services.

Metric	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**	**C11**	**C12**	**C13**	**C14**	**C15**
Availability (α)	0.6634	0.6302	0.6983	0.5987	0.7738	0.6983	0.7351	0.6302	0.6983	0.6634	0.6983	0.5987	0.6302	0.6634	0.7738
Cost (χ)	40	45	35	50	25	35	30	45	35	40	35	50	45	40	25
Data Quality (ξ)	2.0000	2.2500	1.7500	2.5000	1.2500	1.7500	1.5000	2.2500	1.7500	2.0000	1.7500	2.5000	2.2500	2.0000	1.2500

Table 5.7: Availability, Data Quality and Cost of the pairwise configurations.

Chapter 6

Pairwise Testing of Dynamic Composite Services

Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste
IRISA/INRIA, Campus Universitaire de Beaulieu,
Rennes-Cedex, France.

Claude Jard
ENS Cachan, IRISA, Université Européenne
de Bretagne, Bruz, France.

Abstract

A composite service orchestration soliciting multiple atomic services is plagued by a number of sources of variation. For instance, availability of an atomic service and its response time are two important sources of variation. Moreover, the number of possible variations in a composite service increases exponentially with increase in the number of atomic services. Testing such a composite service presents a crucial challenge as its often very expensive to exhaustively examine the variation space. Can we effectively test the dynamic behavior of a composite service using only a subset of these variations? This is the question that intrigues us. In this paper, we first model composite service variability as a feature diagram (FD) that captures all valid configurations of its orchestration. Second, we apply *pairwise testing* to sample the set of all possible configurations to obtain a concise subset. Finally, we test the composite service for selected pairwise configurations for a variety of QoS metrics such as response time, data quality, and availability. Using two case studies, *Car crash crisis management* and *eHealth management*, we demonstrate that pairwise generation effectively samples the full range of QoS variations in a dynamic orchestration. The pairwise sampling technique eliminates over 99% redundancy in configurations, while still calling all atomic services at least once. We rigorously evaluate pairwise testing for the criteria such as: a) ability to sample the extreme QoS metrics of the service b) stable behavior of the extracted configurations c) compact set of configurations that can help evaluate QoS tradeoffs and d) comparison with random sampling.

6.1 Introduction

In a service-oriented world actors such as data sources, knowledge bases, people, processes, businesses, hardware sensors/actuators and software systems are all seen as services. In such a world, a *composite* service orchestrates a number of self-contained *atomic* services to perform complex tasks. The unpredictable and dynamic nature of each of these atomic services ultimately renders the functional and non-functional behavior of a composite service unpredictable and dynamic. For instance, the crisis management system in a large city orchestrates a number of atomic services such as the ambulance service, police service, GPS service, and phone service. The variable nature of each of these services renders the overall behavior of the crisis management system variable and dynamic.

Untested dynamic behavior of a composite service can have several critical consequences. For instance, a crisis management system dealing with an earthquake must mobilize a multitude of services within a predictable time frame and seldom deviate from it. An untested composite service may exhibit unreliable deviations from contractual agreements on Quality of Service (QoS) [TP05]. Service level agreements (SLAs) [PB08] are the industry standard to specify constraints on QoS for both service providers and consumers. Habitual deviations from SLAs are a result of ignoring QoS outliers and dynamic behavior of a composite service.

A key challenge in testing a composite service emerges from its inherent variability. We enlist three important dimensions to *composite service variability* (a) The variation in selection/non-selection of equivalent atomic services used in a composite service (b) The variation in QoS of each of these atomic services leads to variations in composite service QoS. For instance, in [RBHJ08] we develop probabilistic models of QoS variability in atomic services (c) The variation in the way atomic services are called in a composite service such as in sequence or in parallel. In this paper, we are primarily concerned with the first two sources of variability. Changes in the orchestration can be triggered by these sources, enabling self-* composite services. We use the general term dynamic composite service to encompass self-* service-oriented systems.

With an increase in number of equivalent atomic services there is an exponential increase in the invocations of a composite service. It is impractical and computationally expensive to test a composite service for all its possible variations. Therefore we ask, can we effectively test the dynamic behavior of a composite service using only a *subset of these variations*? Answering this question is the subject of this paper.

We present a methodology for combinatorial interaction testing (CIT) dynamic composite services. In particular, we perform *pairwise testing* of composite services. The methodology consists of three main phases: (1) Modeling variability in a composite service (2) Generation of composite service configurations satisfying pairwise interactions (3) Analyzing these composite service configurations to test composite service QoS. In our approach, we model the variability of a composite service as a feature diagram where each feature represents an atomic service. Inter-feature constraints represent dependencies between atomic services. Feature Diagrams (FD) [KCH+90] provide a formal framework to specify authorized variations in the configuration of a composite service. We transform the feature diagram and pairwise interactions between features (or atomic services) to a single constraint satisfaction problem in the formal specification language Alloy [Jac08]. We solve the Alloy model to generate valid configurations of the composite service. The generation methodology is an extension of our previous work[PSK+10] to dynamic composite services. We empirically investigate the QoS of the resulting configurations. We demonstrate that combinatorial interaction testing (CIT) [CDFP97] to select a subset of configurations that covers *all valid pairwise interactions* of services is an efficient technique to sample configurations of an orches-

tration. Our premise is based on the observation that most software faults are triggered by interactions between a small number of variables [KW04]. For example, consider the *car crash crisis management system* case study [KGM09] that we will examine in this paper. With 25 optional features that may / may not be invoked in a specific orchestration, the total number of exhaustive tests required will be $33,554,432$. This is an extremely large number of tests that would considerable time and effort for QoS analysis. The number of tests satisfying pairwise interaction is just 185 reducing the number of required tests by **99.99**%.

Pairwise testing has been used to detect faults in software systems in extensive prior research [CDFP97]. Our main contribution is the application of pairwise testing to sample configurations in dynamic composite services: one form of efficient self-monitoring of variable behavior. This is based on the hypothesis that composite services' QoS behavior uncover faults in a service-oriented systems where choice of atomic services and the orchestration between them are primary artifacts. The extensive empirical studies, based on two case studies which are the car crash crisis management system (C^3MS) [KGM09] and a eHealth administration system, support our **claims about pairwise testing** dynamic composite services:

1. **C1:** Pairwise testing is an sufficient coverage strategy for dynamic composite service orchestrations
2. **C2:** Pairwise testing covers a wide range of QoS in dynamic composite services
3. **C3:** Pairwise testing is better than random testing
4. **C4:** Pairwise testing is a stable strategy to define global SLA for a dynamic composite service
5. **C5:** Pairwise testing is useful to generate families of orchestrations with differing SLAs

The paper is organized as follows. Section 6.2 provide foundational material to understand our paper. This includes feature diagrams in 6.2.1, the Orc language for specifying orchestrations in 6.2.2, pairwise configuration generation in 6.2.4, and formal description of QoS metrics in 6.2.5.The methodology followed in this paper is discussed in Section 6.3. The case studies for experiments are put forth in Section 6.4. The experiments related to QoS analysis are presented in 6.5. Comparison with respect to random generation and the stability of pairwise analysis are shown in 6.5.3 and 6.5.4, respectively. Further deliberation and perspectives of our analysis scheme are presented in Section 6.5.5. Threats to the validity of the empirical studies are discussed in Section 6.5.6. Related work in literature is put forth in Section 6.6. We conclude in Section 6.7.

6.2 Foundations

In this section we present background or foundational ideas required to understand the rest of the paper. We present these concepts to make the paper as self-contained as possible.

6.2.1 Feature Diagrams

Feature Diagrams (FD) introduced by Kang et al. [KCH+90] compactly represent all the products (referred to as *configurations* in this paper) of a software product line (SPL) in terms of features which can be composed. Feature diagrams have been

formalized to perform SPL analysis [SHTB07]. In [SHTB07], Schobbens et al. propose a generic formal definition of FD which subsumes many existing FD dialects. We define a FD as follows:

- A FD consists of k features $f_1, f_2, ..., f_k$
- Each feature f_i may be associated with a software asset such as an atomic service.
- Features are organized in a parent-child relationship in a tree T. A feature with no children is called a leaf.
- A parent-child relationship between features f_p and f_c are categorized as follows:
  - *Mandatory* - child feature f_c is required if f_p is selected.
  - *Optional* - child feature f_c *may* be selected if f_p is selected.
  - *OR* - at least one of the child-features f_{c1},f_{c2},..,f_{c3} of f_p must be selected.
  - *XOR* - one of the child-features f_{c1},f_{c2},..,f_{ck} of f_p must be selected.
- Cross tree relationships between two features f_i and f_j in the tree T are categorized as follows:
  - f_i requires f_j - The selection of f_i in a product implies the selection of f_j.
  - f_i excludes f_j - f_i and f_j cannot be part of the same product and are *mutually exclusive*.

6.2.2 Service Orchestrations using Orc

A dynamic composite service is an orchestration of atomic services. We express the orchestration of atomic services available in an FD using the Orc language. Orc [MC07] serves as a simple yet powerful concurrent programming language to describe and execute service orchestrations.

The fundamental declaration used in the Orc language is a *site*. The type of a *site* is itself treated like a service - it is passed the types of its arguments, and responds with a return type for those arguments. An Orc *expression* represents an execution and may call external services to publish some number of values (possibly zero).

Orc has the following combinators that are used on various examples as seen in [MC07]. The *Parallel* combinator $F|G$, where F and G are Orc expressions, runs by executing F and G concurrently. Whenever F or G communicates with a service or publishes a value, $F|G$ does so as well. The execution of the *Sequential* combinator $F >x> G$ starts by executing F. Sequential operators may also be written compactly as $F \gg G$. Values published by copies of G are published by the whole expression, but the values published by F are not published by the whole expression; they are consumed by the variable binding. If there is no response from either of the sites, the expression does not terminate. While the above two composition operators are for creating threads, Orc uses the following construct to prune operations. The *Pruning* combinator, written $F <x< G$, allows us to block a computation waiting for a result, or terminate a computation. The execution of $F <x< G$ starts by executing F and G in parallel. Whenever F publishes a value, that value is published by the entire execution. When G publishes its first value, that value is bound to x in F, and then the execution of G is immediately terminated. The *Otherwise* combinator, written $F;G$ has the following execution. First, F is executed. If F completes, and has not published any values, then G executes. If F did publish one or more values, then G is ignored. The publications of $F;G$ are those of F if F publishes, or those of G otherwise. In the *Fork-Join* combinator, two processes are invoked and run concurrently. The process

waits until a response is obtained from both. This may be represented as (F, G) where the process waits for responses from both atomic services F and G.

6.2.3 Feature Diagrams with Orchestrations

The FD and the orchestration cover two dimensions that are complementary to each other. While the FD represents the variability in the configurations, the orchestration specifies the order in which the services are called. Making use of the terminology in [SHTB07], *primitive* features are "features" that are of interest and that will be incorporated in real-world services. On the contrary, *decomposable* features are just intermediate nodes used for decomposition. It is up to the modeler to determine such classification of features in the FD. We extend the semantics given in [SHTB07] to ensure compatibility of an orchestration with the feature model as follows:

- The set of available services S are the *primitive* nodes of the FD D;
- For each orchestration, the set of corresponding services invoked (denoted N);
- $N \subseteq S$ in a configuration;
- A model of D is a subset of its (*primitive* and *decomposable*) nodes;
- There must exist a model of D ($[[D]]$) such that $[[D]] \cap S = N$ (a model of a FD is a subtree that is valid w.r.t. the operators and the dependence relation).

Drawing from the real-world services and the constraints shown in a FD, the composite service may be developed by an orchestrator.

6.2.4 Combinatorial Interaction Testing

We use combinatorial interaction testing (CIT) to synthesize a subset of configurations represented by the FD of a dynamic composite service. Originally, CIT was proposed by Cohen et al. [CDFP97] to select a subset of all combinations of variables that define the input domain of a program, while still guaranteeing a certain level of coverage. This has led to the definition of pairwise interaction testing, or 2-wise testing. This samples the set of all combinations in such a way that all possible pairs of variable values are included in the set of test data. Pairwise testing has been generalized to t-wise testing which samples the input domain to cover all t-wise combinations. In this paper, a set of test data is often represented in the form of a *covering array* that contains all t-wise interaction of features in a FD.

Definition. 2 ***Covering Array*** - *A covering array CA $(N; t, k, v)$ is a $N \times k$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size t from v symbols at least once.*

From the definition of a covering array, the strength t of the array is the parameter that allows achieving 2-wise (pairwise), 3-wise or t-wise combinations. The k columns on this array correspond to all the variables in the input domain which in our case are the features in a FD. For the generation of dynamic composite service configurations, k is the number of services, and v is 2 since we have only boolean variables (services may be present or absent in a configuration). The covering array is a set of configurations of features.

We demonstrate the concept of a minimal covering array using an example. Consider the set of four atomic services (A, B, C, D) with varying response times. The atomic services can be composed in 2^4 exhaustive combinations. However, if we consider the

service combinations in pairs, we require fewer configurations. These can be subsumed by 6 sets of configurations that cover these pairs of interactions resulting in removal of 62.5% of redundancies. This is shown in Table 6.1 where, for example, interaction (A, B) refers to calling both service A and B while (A, ¬B) refers to calling only A with B explicitly not invoked.

Pairwise Interaction	**Configurations**
(A, B); (A, C); (A, D); (B, C); (C,D)	(A, B, C, D)
(A, ¬B); (A, ¬C); (A, ¬D)	(A)
(B, D); (B, ¬A); (B, ¬C); (D, ¬A)	(B, D)
(C, ¬A); (C, ¬B); (C, ¬D)	(C)
(D, ¬B); (D, ¬C)	(A, D)
(B, ¬D)	(A, B, C)

Table 6.1: Subsuming pairwise interactions in configurations

Essentially, the use of pairwise sampling reduces the number of cases needed to generate a range of outputs, a few of which that may be considered faulty. Consider a system S having a set of inputs p and a set of outputs q. With random testing, in which input vectors satisfying p are randomly generated, and the output of each execution is compared with the postcondition q as a set of tests. As structural features of system S are hidden, the efficacy of using manually designed test cases can be seen mainly through their cost effectiveness. In our case, we view this as the decrease in the number of samples needed to generate extreme output values (faults).

Let $\omega \in p$ be a set of tests for the system S. This produces a set of specifications $\omega \xrightarrow{S} q'$, where $q' \in q$. A successful set of tests is one that has a minimal cardinality of cases $|\omega|$ and maximal variance in the set of outputs q'. This generates a range of values as the system output. Empirical studies have shown pairwise sampling to be superior for precisely such a case - efficiently generating a minimal set of tests to generate all dual combinations of input values. This in turn produces a range of outputs q' that have higher variance than other comparative techniques of similar cardinality $|\omega|$.

The problem of generating a minimal covering array for a set of variables is a complex optimization problem that has been studied in extensive prior work for example [CDFP97]. It is important to notice that there exist very few studies that have tackled the automatic generation for CIT in the presence of constraints between variables [CDS08]. In order to include properties that forbid combinations of values, CIT generation techniques have to allow the introduction of constraints in the algorithms that generate covering arrays. In recent work [PSK+10], we present a solution to generate t-wise configurations that satisfy all simultaneously constraints modeled in a feature diagram.

We transform the feature diagram to constraint satisfaction problem model in the language Alloy as described in [PSK+10]. The features in the FD are transformed to concepts in Alloy called *signatures*. Inter-feature constraints in the FD are transformed to Alloy *facts*. All pair-wise interactions between features are transformed to Alloy *predicates*. The goal of solving the Alloy model is to find the minimal set of configurations that cover conjunctions of all valid pair-wise predicates. The first step involves *detection* of all valid pairs that conform to the FD. In the second step, we construct conjunctions of pair-wise predicates and solve them via incrementally increasing the scope of the solution size. The result is a minimal set of configurations that cover conjunctions of all valid pairs. At times the SAT solver in Alloy is not scalable for a large FD. We apply divide-and-compose approaches as described in [PSK+10] to handle this scalability issue.

6.2.5 QoS Aspects of the Orchestration

In this paper, we test dynamic composite services for their probabilistic QoS behavior. In this section we summarize our work in [RBHJ08], that presents the derivation of composite service QoS behavior from individual atomic service behaviors. Probabilistic analysis of QoS parameters as described in [RBHJ08] [HWTS07] provide a more realistic study of actual services' behavior. The following QoS parameters have been chosen for experiments in this paper:

1. *Latency / Response Time* (T) - Denotes the overall delay due to the time taken by a service to respond. It is a discrete value that may be modeled as a long tailed distribution incorporating some *rare deviations*.

2. *Availability* (α) - The probability that a service is active and can respond to a service call. For a well managed service, this value is generally quite high.

3. *Cost* (χ) - Refers to the monetary cost associated with each invocation of a particular atomic service.

4. *Data Quality* (ξ) - A subjective measure of trade off to high Cost and Response times of services. It measures the "Quality" of the output of the service and the beneficial aspects of including a new atomic service into the composite orchestration.

These QoS metrics are normally defined for an atomic service. We derive these QoS metrics for a dynamic composite service by analyzing its orchestration. This analysis involves giving a semantic to a composite service QoS based on individual atomic service QoS and the Orc combinators (see Section 5.2.2) associating them. Taking two sites s_i and s_j, the QoS metrics may be computed as shown in Table 6.2 based on the Orc combinators in use. The cases of composing the service s_{ij} using the *sequential* and *fork-join* combinators have been considered. The latency, cost and availability metrics for the composite service s_{ij} are derived as shown in [CMSA02] with $Max(p, q)$ representing the maxima of the values p and q. For the sequential case, the latency and cost of the composite service is a sum of the atomic services' parameters while the availability is a product of such parameters. Similarly, the maxima of the atomic services' response times contributes to the global response time under parallel invocation.

Orc Code	$s_{ij} \triangleq s_i \gg s_j$	$s_{ij} \triangleq (s_i, s_j)$
Latency	$T(s_{ij}) = T(s_i) + T(s_j)$	$T(s_{ij}) = Max(T(s_i), T(s_j))$
Cost	$\chi(s_{ij}) = \chi(s_i) + \chi(s_j)$	$\chi(s_{ij}) = \chi(s_i) + \chi(s_j)$
Availability	$\alpha(s_{ij}) = \alpha(s_i) \times \alpha(s_j)$	$\alpha(s_{ij}) = \alpha(s_i) \times \alpha(s_j)$

Table 6.2: QoS metrics extended to Orc combinators.

Some QoS metrics of an atomic service may be modeled as a random variable conforming to a probability distribution. We need to simulate the QoS metric by sampling from a probability distribution. For instance, we need to simulate the probabilistic response time distributions of each atomic service as done in [RBHJ08]. By varying the degrees of freedom ν and non-centrality parameter δ in the t-distribution *dfittool* of MATLAB, it is possible to generate various heavy tailed distributions that mimic the response times of services. We sample these distributions to simulate the response times of actually invoked atomic services. In this paper, the t-distribution fitting was used to generate various distributions of services' response times.

6.3 Methodology

We present the methodology for pairwise testing and QoS analysis of dynamic composite services.

1. **Inputs:** The inputs to our methodology is tuple (S, FD, O, $Strategy$):

   (a) S is the set of all atomic services that can be used in a dynamic composite service.

   (b) FD is a feature diagram that specifies various features in a dynamic composite service and the constraints between them. Primitive features in an FD are each associated with an atomic service S_i. A valid configuration C_k of a FD is the set of m features $f_1, f_2, ..., f_M$ that conform to the constraints in the FD. The features in valid configurations represents sets of atomic services $S_1, S_2, ..., S_N$. The sets are subsets of S. See Section 5.2.1 for formal definition of a FD.

   (c) O is the overall orchestration of the dynamic composite service. The orchestration is reconfigured based on valid configurations of the FD. The orchestration O may be reconfigured to orchestrations $O_1, O_2, .., O_N$ for all valid configurations $C_1, C_2, .., C_N$ of the FD. An orchestration only invokes the set of atomic services present in a valid configuration of the FD. In our paper, O is an Orc orchestration. See Section 5.2.2 for brief description of Orc.

   (d) $Strategy$ is the strategy used to generate configurations. In this paper, we consider two strategies to guide generation of valid FD configurations:

      i. Random Generation : We randomly select configurations conforming to FD by solving the Alloy model representing only the FD.

      ii. Pairwise Generation : We generate a set of configurations that satisfy all pairwise interactions between features in FD. These configurations also satisfy the constraints in the FD.

2. **Configuration Generation:** We generate the configurations using the technique described in [PSK+10] and briefly outlined in Section 5.2.3. The process involves transformation of the FD to a constraint satisfaction problem in Alloy. A chosen $Strategy$ to generate configurations is also transformed in conjunction with the Alloy model. Solving the Alloy model gives valid configurations. Let the set of output configurations be $C_1, C_2, ...C_N$ for a chosen strategy $Strategy$.

3. **Empirical Analysis of QoS:** The output configurations from the previous step $C_1, C_2..., C_N$ reconfigures O to orchestrations $O_1, O_2, ..O_N$ by selecting only the atomic services that are present in each of the configurations. We compute QoS for each of the orchestrations invoking all atomic services in the configuration using the semantics described in Section 5.2.4. We use the experiments to address the questions motivated in Section 6.1.

6.4 Case Studies

We consider two case studies for our experiments as described in Sections 6.4.1 and 6.4.2.

6.4.1 Car Crash Crisis Management System

The need for crisis management systems has grown significantly over time [KGM09]. Crisis management involves identifying, assessing, and handling the crisis situation. A crisis management system facilitates this process by orchestrating the communication between all (distributed) parties involved in handling the crisis. The car crash crisis management system (C^3MS) [KGM09] includes all the functionalities of a general crisis management systems, and some additional features specific to car crashes such as facilitating the rescuing of victims at the crisis scene and the use of tow trucks to remove damaged vehicles. As described in [KGM09], the main goals of this system include: a) Facilitating the rescue mission carried out by the police / firemen and providing them with detailed information on the location of the crash. b) Managing the dispatch of ambulances or other alternate emergency vehicles to transport victims from the crisis scene to hospitals. c) Coordinating the first-aid missions by providing relevant medical history of identified victims by querying data bases of local hospitals. d) Ushering the medical treatment process of victims by providing important information about the crash to the concerned workers. e) Managing the use of tow trucks to remove obstacles and damaged vehicles from the crisis scene.

In Figure 6.1, we present the Car Crash Crisis Management System (C^3MS) FD [KGM09]. The C^3MS FD contains several features that are associated with software assets represented by atomic services. For example, the *Paramedic* feature is represented by the *Paramedic* service. Some sets of features like *Police* and *PoliceMan* are subsumed by a single service *Police*. Constraints such as optional, requires and mutual exclusion (XOR) are also incorporated. For example, the *GPS* and *GSM* features are mutually exclusive while the *Doctor* feature requires the *PublicHospital* feature.

Figure 6.1: C^3MS Feature Diagram.

```
--C3MS Orchestration--

def CrisisManagementSystem() =
  CrisisType() >> (HumanVictims(h), Witness(w)) >> Area()
  >>  CommunicationManager(gs,gp) >> InternalResource(o) >>
  (Mission(), ITOption(IT),ExternalServices(), Medical(md))

def CommunicationManager(gs,gp) = (GSM(gs),GPS(gp))
```

```
def InternalResource(o) =
  (AidMaterial(),Coordinator(),Paramedic(),Observer(o))
def Mission() = (RemoveObstacle(robs),Rescue(re),Observe(ob),Transport(tr),
def Investigation(in),NurseWounded(nw),SortWounded(sw))
def ITOption(IT) = (su,au,db) >>
  (SurveillanceSystem(su),AuthenticationSystem(au), DatabaseSystem(db))
def ExternalServices() = (ExternalCompany(ec),GovernmentServices())
def Medical(md) = (pac,ph) >> (PrivateAmbulance(pac),PublicHospital(ph))

def ExternalCompany(ec) = tt >> TowTruck(tt)
def GovernmentServices() = (Police(pm),Fire(fm))
def PublicHospital(ph) = (hw,amb,doc) >>
  (HospitalWorker(hw),Ambulance(amb),Doctor(doc))
```

Figure 6.2: Composite Service Orchestration of the C^3MS.

The composite service orchestration is represented succinctly in Fig. 6.2 and the Orc pseudo code is also presented. They represent a family of configurations that may be invoked. Calling the *CrisisManagementSystem()* service invokes other services like *CrisisType* and *InternalResource()* operations in sequence. These services, in turn, may call other services in parallel or services, passing some parameters in the process. For instance, the *CommunicationManager()* services calls the *GPS()* and *GSM()* services in parallel, while passing some parameter values for invocation of these services. The setting of these parameter results in the various associated configurations in the system. Operations such as mutual exclusion (*MUX*) and synchronization (*Merge*) may be performed using Orc constructs. Another level of control is the global timeout value associated with the composite service. This has to be associated with the overall SLA of the composite service to provide optimal durations for response.

6.4.2 eHealth Management System

The need for efficient hospital management stems has been discussed in [SAP06]. A hospital administration system is devised to remove some of the inefficiency plaguing

current protocols such as cumbersome admission time, duplicate data entry, redundant lab tests, ineffective treatment coordination, and billing processes. Drawing inspiration from [SAP06] composite health care applications are required to connect various parties and locations. The information flows seamlessly across organizational and system boundaries emitting from the use of such a centralized orchestration. This enhanced visibility gives everyone involved a unified view of relevant information and gives process owners the ability to improve existing methods and procedures. The eHealth system can be viewed as an extension of the C^3MS medical services to transport injured victims for speedy treatment of injuries. Examples of the utility of healthcare applications include: a) Healthcare providers can access the medical information of a prospective patient and use ambulance services to transfer the client to relevant healthcare facilities. b) Physicians can review a patient's medical history even though this data resides in several systems managed by diverse providers. c) Insurance claims and financial options can be updated and handled in a speedy way. d) Doctors can use a composite application to determine the appropriate medication for a patient, order the drug, check the status of pharmacy approval, and monitor how the drug is dispensed. e) Special needs of the patient such as catering specific food items and lab tests can be coordinated in an effective way.

Fig. 6.3, presents the eHealth management system FD. Similar to the C^3MS FD, it contains several features that are associated with software assets represented by atomic services. Constraints such as optional, requires and mutual exclusion (XOR) are also incorporated. Two versions of the similar service $Ambulance_f$ and $Ambulance_s$ are in mutual exclusion. These atomic features or services can be set to varying QoS values resulting in interesting combinations of services.

Figure 6.3: eHealth Feature Diagram.

The web services used for the orchestration of the eHealth system are shown in Fig. 6.4. The operations are generic with services such as *HealthForms* and *InsuranceForms* used to request relevant medical history and insurance status of the patient, respectively.

The Orc pseudo code for the eHealth system is presented with the distinguishing feature being the choice of services that can be used to perform similar goals. For instance, either one of the mutually exclusive (MUX) services $Testing_f()$ or $Testing_s()$ services can be used to request for lab tests. However, the QoS associated with each of

Figure 6.4: Composite Web Service Orchestration of the eHealth system.

these services is different resulting in varying overall composite service QoS.

```
--eHealth Orchestration--

def HealthEmergencySystem() = Transport() >> HospitalAdmit() >> Billing()
  >> Discharge()

def Transport() = a >> Ambulance(a)
def HospitalAdmit() = Documents() >(hf,in)>
 (HealthRecords(hf),InsuranceRecords(in)) >> AdmitRoom() >> Treatment()
def AdmitRoom() = (sr,w) >> (SpecialRoom(sr),Ward(w))
def Treatment() = (d,t,c,p) >>
 ((Doctor(d),Testing(t),Catering(c),Pharmacy(p))

def Ambulance(a) = Let(Ambulance_f() | Ambulance_s())
def HealthRecords(hf) = Let(HealthRecords_f() | HealthRecords_s())
def InsuranceCompany(in) = Let(InsuranceCompany_f() | InsuranceCompany_s())
def SpecialRoom(sr) = Let(SpecialRoom_f() | SpecialRoom_s())
def Ward(w) = Let(Ward_f() | Ward_s())
def Doctor(d) = Let(Doctor_f() | Doctor_s())
def Testing(t) = Let(Testing_f() | Testing_s())
def Catering(c) = Let(Catering_f() | Catering_s())
def Pharmacy(p) = Let(Pharmacy_f() | Pharmacy_s())
```

6.5 Experiments

Based on the methodology in Section 6.3 we perform experiments involving pairwise generation of configurations followed by simulations to obtain probabilistic QoS of dynamic composite services. We consider both case studies for these experiments.

6.5.1 Evaluating QoS of the Car Crash Crisis Management System

Configuration Generation: We first use the approach presented in [PSK+10] to generate a minimal set (given the resource constraints) of configurations that satisfy all valid pairwise interactions in the C^3MS case study. The input settings to the configuration generator are (a) Maximum scope for Alloy solver (b) Maximum time to solve (c) Divide-and-compose strategy for scalable generation. The maximum scope is set to 8 and maximum time to 2000 milli-seconds with use of *incremental growth* strategy. Through this technique, **185** configurations for the C^3MS case study were generated. The 185 configurations satisfy all valid pairwise interactions between services in the C^3MS FD that originally specify 2^{25} configurations. All invalid pairs that do not conform to the FD are rejected by the approach. For instance, the not including the *Mission* feature in a configuration is invalid as it is a mandatory feature.
Computing Response Time: Second, we compute response times for these 185 configurations. We assign each atomic service in the dynamic composite service a t-distribution to model response time. The random settings for the atomic service t-distributions were degrees of freedom ν from 3 to 8 and non-centrality parameter δ from 5 to 15 seconds, respectively. We choose these values to provide diversity in atomic response times. For a chosen atomic service (in the current configuration), the individual timeout value was set to 95 percentile of the response time distribution. This largely ensures that the composite service obtains the result of the atomic service and not a timeout. For each of the 185 configuration, we obtain $\mathbf{10,000}$ Monte-Carlo samples of response times from all atomic services in a configuration. We compute the composite service response time from these atomic service response times. We collect the response times for the composite service for each configuration to create a t-distribution for the composite service. We set the global timeout of the composite service to a sufficiently high value (300 seconds) to allow capture of outliers in the distribution.

As seen in Fig. 6.5, the pairwise generated configurations cover a range of response time distributions. The distributions were sorted in increasing order of response time and are shown. The slowest and the fastest composite services are marked. Their median values are shown to be **113** and **201** seconds, respectively. This demonstrates the use of a few configurations to test significant changes of about **88** seconds response time in a composite service. These results support the claims **C1** and **C2** in Section 6.1, that pertain to the effectiveness of pairwise sampling to generate a wide range of orchestrations and output QoS values.
Computing other QoS metrics: We compute additional QoS metrics such as availability of a service, the cost entailed in calling atomic services and output data quality for the 185 configurations. We compute QoS for a composite service based on rules given in Table 5.2 for different Orc combinators in an orchestration. For example, when we set atomic service availability to **0.99** (representing service availability in 99% of invocations) the composite availability of each configuration is shown in Fig. 6.6. The output data quality ξ is related to the cost χ by the constant κ given by $\xi = \chi/\kappa$ (assuming linear increase in data quality with each atomic service invocation). The output data quality ξ is can also be derived exponentially from the cost χ by $\xi = e^{\chi/\kappa}$. For example, setting the $\chi = 5$ units for each invoked atomic service, the cost of each configuration is shown in Fig. 6.6. Furthermore, setting $\kappa = 20$, the linear and expo-

Figure 6.5: Response time distributions of the 185 pairwise configurations for C^3MS.

nential output data quality of the configurations may also be derived. These variations in data-quality, response time and cost help analyze trade-offs between QoS parameters. These variations in QoS parameters substantiate the claim **C5** about pairwise testing in Section 6.1 referring to its use in generating families of composite services.

Figure 6.6: Availability, Data Quality and Cost of the pairwise configurations of C^3MS.

6.5.2 Evaluating QoS of the eHealth System

Configuration Generation: For the eHealth system, we generate 188 configurations that satisfy all valid pairwise interactions from a total set of 2^{12} configurations. The initial settings for configuration generation were exactly the same as in the C^3MS case study.

Computing Response Time: For each of the 188 configurations, we model atomic

service QoS as t-distributions. The parameters of these distribution are chosen in random in certain bounds to ensure diversity. The parameter degrees of freedom ν was from 3 to 8 and non-centrality parameter δ from 5 to 15 seconds, respectively. For the faster services (marked with the subscript $_f$), the δ parameter was set between 3 to 5 seconds, representing a faster response to a service call.

We obtain **10,000** Monte-Carlo samples of response times for each of the atomic services and compute the composite service response time distribution. As seen in Fig. 6.7, the pairwise generated configurations cover a wide range of response time distributions. The distributions are sorted in increasing order of response time. The slowest and the fastest composite services are marked with median values. In the case of eHealth, the 30 seconds range in response time values is due to the added diversity of *choice* in choosing a *fast* or *slow* atomic service.

Figure 6.7: Response time distributions of the 188 pairwise configurations for eHealth.

Computing other QoS metrics: We use the rules for combinators described in Table 5.2 to compute QoS of composite service orchestrations. Setting atomic service availability to 0.99 the composite availability each configuration is shown in Fig. 6.8. We observe that the cost of the composite service varies with the choice of fast or slow services. A faster service (with subscript $_f$) is set *double* the cost of its slower (with subscript $_s$) counterpart. This changes the range of cost and data quality available for different configurations as seen in Fig. 6.8.

6.5.3 Comparison with Random Sampling

It has been shown in [CDFP97] that pairwise interaction testing of such configurations is advantageous over random testing since its systematic and provides a better coverage. With random runs, it is impossible to determine if all the atomic services have been invoked at least once. The configurations leading to extreme test case values need not be necessarily generated during random runs and there may be many redundant configurations invoked repeatedly. Setting SLAs based on random runs is both non-robust and can lead to habitual deviance. Generating families of configuration with accurately fixed bounds on QoS is also not possible. For these reasons, pairwise generation has comparative advantages over random runs.

Three sets of random configurations were generated as shown in Fig. 6.9, each with

Figure 6.8: Availability, Data Quality and Cost of the pairwise configurations of eHealth.

original configuration size 185. In each case, the number of valid configurations was found to be 17, 21 and 24 resulting in a maximum efficient generation percentage of 12.97%. Not only are there deviations in the number of valid configurations for each run (17, 21, 24), but also in the QoS metrics output in each run. SLA deviations are a result of resorting to such insufficient random runs of a composite service, which might generate invalid and redundant scenarios. To test the effectiveness of combinatorial

Figure 6.9: Three runs of random generation of configurations for C^3MS.

testing the **185** pairwise configurations were compared with random samples for the C^3MS. All the mandatory features were set to be invoked with the constrained and optional features randomized in invocation for the random case. This random sampling was performed by a Markov decision process of traversing features in the FD, which will always lead to generation of valid configurations (based on constraints). The com-

parison with pairwise is shown in Fig. 6.10 and it is seen that random generation can cover a large range of QoS values if sufficient number of configurations are generated. To determine that number, however, requires analysis of pairwise interactions. The random configurations are deficient as they cannot guarantee a) invocation of every possible service at least once; b) generating the extreme configurations for a particular composite service in every sample. When compared to the pairwise generation scheme that covered all pairs of services, the random generation covered only **8.8**% of the service pairs. This shows that the same set of services are redundantly invoked in many configurations during random generation. Thus, for such orchestrations with numerous

Figure 6.10: Comparison of pairwise and random response time (arranged in increasing order) of percentile values for 185 configurations of C^3MS.

configurations, using pairwise interactions is a sufficient choice in order to examine the entire sample space. These results support our claim **C3** in Section 6.1, referring to the comparison between pairwise and random sampling.

6.5.4 Consistency of Pairwise Samples

Given one orchestration, there can be many different sets (or solutions) of configurations that cover pairwise services interactions. Thus, we compute QoS behavior over different samples of configurations. This aims at evaluating the *stability of pairwise testing* as a sampling technique to estimate the global QoS for a dynamic composite service. A collection of **10** samples that satisfy the pairwise interaction testing were generated for the eHealth case. The percentile statistics of the configurations in each sample was collected through 10,000 Monte-Carlo runs and is shown in Fig. 6.11. The lowest and highest percentile values of the configurations in each sample were collected. The mean inter-sample difference for the random case is **12.94** seconds compared to **6.44** seconds for the pairwise case. Further, these were compared with **10** samples of randomly generated configurations (with 300 configurations in each sample) in Fig. 6.11. Again, all the mandatory features were set to be invoked with the constrained and optional features randomized in invocation for the random configurations. The number of valid configurations for each sample ranged between 3.5% to 9% of the 300 configurations. Comparing the two cases, the stability of the pairwise generation is demonstrated through its consistently low standard deviation values in Table 6.3 when

compared to random samples. Once again, the lowest and the highest percentile values of all the configurations in a particular sample are compared. These results support claim **C4** in Section 6.1, referring to the stability of pairwise sampling.

Figure 6.11: Comparing stability of pairwise and random samples for eHealth.

Percentile	25(min.)	25(max.)	50(min.)	50(max.)	75(min.)	75(max.)	90(min.)	90(max.)
Pairwise Std. Dev. (secs.)	2.18	1.52	2.59	1.73	2.90	1.82	3.19	1.83
Random Std. Dev. (secs.)	4.14	4.17	4.21	4.51	4.43	4.76	4.63	5.07

Table 6.3: Standard Deviation values for pairwise and random samples.

6.5.5 Perspectives due to Analysis

The methodology evaluated for the C^3MS and the eHealth orchestrations can lead to many possibilities for improving QoS metrics for composite services. This includes setting the SLA keeping into account the worst performing configuration. This will prevent contract deviation during actual deployment of the service.

A family of SLAs for a set of configurations taking into account trade-offs between QoS metrics and the output quality of configurations may be proposed. This leads to families of composite services with extensively analyzed SLAs. Configurations may be grouped along with their QoS behavior to develop an extended product line of composite services. For example, categories of services may be constructed for the C^3MS orchestration (based on Figs. 6.5 and 6.6) as shown in Table 6.4. Similarly, the two categories of service families for the eHealth case (Figs. 6.7 and 6.8) is shown in Table 6.5. In both cases, the family of services with higher data quality is traded-off by a slightly higher response time.

While the diversity in QoS families for the C^3MS is due to optional services that may / may not be included, the variability in the eHealth case is mainly due to other factors. An inherent choice in replacing a *slow* atomic service with a *fast* counterpart can lead to a range of QoS values. Generated configuration families can use of combination of these options of optimally compose atomic services to specific QoS bounds. These service families can have associated contracts (albeit in the soft-sense as in [RBHJ08]) to monitor deviations from specifications. These instances support our claim **C5** in Section 6.1, that pertains to developing families of composite service orchestration with

Configuration Families	Bronze	Silver	Gold
90 percentile Response Time (T)	< 183 s	< 216 s	≥ 216 s
Median Response Time (T)	< 150 s	< 179 s	≥ 179 s
Availability (α)	> 0.75	> 0.71	≥ 0.71
Cost (χ)	< 60	< 70	≥ 70
Linear Data Quality (ξ)	< 3	< 3.5	≥ 3.5
Exponential Data Quality (ξ)	< 20	< 30	≥ 30

Table 6.4: Configuration families for C^3MS.

Configuration Families	Standard	Premium
90 percentile Response Time (T)	< 171 s	≥ 171 s
Median Response Time (T)	< 139 s	≥ 139 s
Availability (α)	> 0.85	≤ 0.85
Cost (χ)	≤ 40	> 40
Linear Data Quality (ξ)	≤ 2	> 2
Exponential Data Quality (ξ)	≤ 8	> 8

Table 6.5: Configuration families for eHealth.

significantly different QoS behavior. With numerous possible combinations of atomic services, such a dedicated families of services with significantly different QoS outputs enable accurate monitoring of services provided. The pairwise scheme is both a robust and compact representation of the behavior space of the set of orchestrations. This provides an effective pre-SLA technique to enunciate the QoS metrics and threshold levels.

6.5.6 Threats to Validity

This section considers the threats to the validity of the experimental results. These may be *internal* (whether there is a bias/error in the experimental design which could affect the causal relationship) or *external* (ability to generalize the results of the experiment to industrial practice).

The hypothesis studied in this paper concerns the use of pairwise sampling to evaluate QoS of large orchestrations. Sources of internal error can be a result of the MiniSAT solver used to generate the pairwise configurations or the MATLAB statistical tools used for QoS evaluation. These tools have not been compared with available alternatives for consistency of results. Furthermore, the assumption is that for each sample of configurations, the pairwise analysis scheme can provide consistently large range of QoS values. Systematic bias in QoS may be introduced in samples when extreme cases are not generated.

To ensure scalability to large industry level FDs, the pairwise generation in [PSK+10] makes use of incremental growth / binary splitting schemes. Redundancies in the number of configurations can be seen due to these schemes. For generating more than one sample of solutions, the symmetry breaking scheme in Alloy was used. This introduces more constraints with each proceeding sample, which increases the time required to generate such samples.

6.6 Related Work

The combinatorial testing framework described by Cohen et al. [CDFP97] has been applied extensively to efficient testing for fault detection. In the work of Cohen et al. [CDS08], this technique is extended to software product lines with highly configurable systems. Modeling variability in SPLs using feature models is the work of Jaring and Boschet [JB02] where they show that the robustness of a SPL architecture is related to the type of variability. To ensure that constraints in the FD are incorporated in the efficient sampling of t-wise tests, the scalable solver proposed by Perrouin et al. [PSK+10]

is used. In [MMLP09], variability in software as a service applications are modeled using the orthogonal variability model to study the customization choices in such workflows. In the recent work by Arcuri and Briand [AB11], random testing has been formally shown to perform arbitrarily worse than pairwise testing, when constraints are present among features.

Pre-deployment testing of SLAs has been studied by Di Penta et al. [PCE07], where they make use of genetic algorithms to generate test data causing SLA violations. Analysis of white and black box approaches are provided in the paper. In [BCP+05], Bruno et al. make use of regression testing to ensure that an evolving service maintains the functional and QoS assumptions. The service consistency verification due to evolution is done by executing test suites contained in a XML encoded facet attached to the service.

The use of probabilistic QoS and soft contracts was introduced by Rosario et. al [RBHJ08] and Bistarelli et al. [BS09b]. Instead of using fixed hard bound values for parameters such as response time, the authors proposed a soft contract monitoring approach to model the QoS measurement. The composite service QoS was modeled using probabilistic processes by Hwang et al. [HWTS07] where the authors combine orchestration constructs to derive global probability distributions.

In our paper, we extend these two notions to analyze the QoS of a composite orchestration under various configurations. Effective sampling of orchestrations is necessary specially in conjunction with exceedingly flexible and large configuration spaces. When combined with the probabilistic behavior QoS behavior of services, this provides an accurate portrayal of the composite service's end-to-end QoS. In a recent submission [KSB+10], similar methodology is used to compare pairwise and exhaustive analysis of configuration spaces in smaller orchestrations. In this paper, that notion is extended to comparison with random runs of larger configuration spaces (where exhaustive analysis is impossible). This entails a scalable approach for robust pairwise interaction generation that is not required for the smaller examples. The case studies and corresponding experiments are much larger in this paper and study the effect of not only orchestration variability, but also choice in compatible atomic service counterparts. Correspondingly, this requires further experiments on the sampling robustness and comparison with random generation, which is not included in [KSB+10].

Though formal analysis of end-to-end QoS has been studied in Cardoso et al. [CMSA02], there are no practical testing tools available for the composite service provider. The pairwise testing procedure has been shown to outperform other testing techniques in [CDFP97]. We extend this testing tool to develop a generic testing methodology to query end-to-end QoS of a web service. Related empirical studies of optimal QoS compositions make use of genetic programming in Canfora et al. [CPEV05a] and linear programming in Zeng et al. [ZBN+04]. These are dynamic techniques to choose the best possible atomic services and configurations for SLAs. The goal in our paper is to analyze the dynamic configurations that may result due to invocation/non-invocation of particular web services when atomic SLAs have already been established.

6.7 Conclusion

We demonstrate that combinatorial interaction testing and in particular pairwise testing effectively portrays the overall behavior of a dynamic composite service. Pairwise testing drastically reduces the number of composite service configurations while successfully analyzing a wide range of QoS values. It provides good coverage for two large case studies (C^3MS and eHealth). We also observe that the analysis remains stable over multiple solutions for the same case study. Pairwise testing is superior to random generation of configurations in terms of coverage and stability of results. Pairwise test-

ing helps specify SLAs based on a deterministic and systematic sampling scheme rather than random sampling. We use our approach to create many families of composite services which can be seen as products with varying costs and SLAs. We largely augment the predictability of a dynamic composite service by performing offline pairwise testing in advance.

Chapter 7

Optimizing Decisions in Web Services Orchestrations

Ajay Kattepur, Albert Benveniste
IRISA/INRIA, Campus Universitaire de Beaulieu,
Rennes-Cedex, France.

Claude Jard
ENS Cachan, IRISA, Université Européenne
de Bretagne, Bruz, France.

Abstract

Web services orchestrations conventionally employ exhaustive comparison of runtime quality of service (QoS) metrics for decision making. The ability to incorporate more complex mathematical packages are needed, especially in case of workflows for resource allocation and queuing systems. By modeling such optimization routines as service calls within orchestration specifications, techniques such as linear programming can be conveniently invoked by non-specialist workflow designers. Leveraging on previously developed QoS theory, we propose the use of a high-level flexible query procedure for embedding optimizations in languages such as Orc. The *Optima* site provides an extension to the sorting and pruning operations currently employed in Orc. Further, the lack of an objective technique for consolidating QoS metrics is a problem in identifying suitable cost functions. We employ the *analytical hierarchy process (AHP)* to generate a total ordering of QoS metrics across various domains. With constructs for ensuring *consistency* over subjective judgements, the AHP provides a suitable technique for producing objective cost functions. Using the *Dell Supply Chain* example, we demonstrate the feasibility of decision making through optimization routines, specially when the control flow is QoS dependent.

7.1 Introduction

A *composite* web service is an application whose implementation calls other self-contained *atomic* services. A composite web service *orchestration* specifies the interaction, management and coordination between these atomic services. Such a composite service can take decisions to invoke or pass parameters to atomic services depending on returned data and quality of service (QoS) metrics. Traditional orchestrations make use of simple comparisons of returned values from atomic services for decision making purposes. While such comparisons are plausible in small orchestrations, involved operations such as multi-criteria decisions from a directory of hundreds of distributed services would require optimizations strategies. With QoS metrics modeled as random variables [HWTS07], the use of probabilistic contracts for service level agreements (SLAs) [RBHJ08] becomes mandatory. Optimizing these random variables for decision making is a natural extension of the probabilistic nature of both composition as well as contracts.

As switching between technologies while developing workflows is detrimental, integration of optimization techniques as part of the specifications of a service orchestration or choreography is required. We show that optimization of QoS metrics can be formulated within concurrent programming languages like Orc [KQCM09]. Employing specialized *sites* that perform optimization routines, alternatives to conventional sorting and searching techniques may be incorporated within workflow specifications.

As the designers of such workflows are assumed to be non-specialists in optimization modeling, we propose techniques for formulating complex queries through simple user judgements / constraints. This will relieve the dependency on domain-specific and involved concepts such as queuing and process management theory in order to generate realistic cost functions. Weighing parameters effectively is done by employing the analytic hierarchy process (AHP) [Saa80]. It provides a simple approach for retaining consistency of subjective evaluations of QoS metrics across different domains.

To prevent deadlock in an orchestration where there are intricate links between parameters, it is essential that optimal settings are employed. This is demonstrated in the QoS dependent choreography of the *Dell* supply chain example [KZC+04]. By modeling this choreography as a *linear programming* problem, we demonstrate the efficacy of our technique to ensure contractual obligations with shared resources. Due to the tractable nature of AHP, cost functions can be generated to set suitable resupply batch sizes for varying demand rates. This exemplifies clearly a situation where the control flow is dependent on optimal setting of parameters.

The paper is organized as follows: Section 7.2 presents background material required for understanding the rest of the paper. This includes optimization models, Orc language for orchestrations, the analytic hierarchy process and QoS aspects of web services. The methodology proposed in this paper is outlined in Section 7.3 with emphasis on formulating optimizations in web services. Section 7.4 elucidates the Dell logistics example as an optimization of QoS metrics. Extending this notion to general orchestration problems, in Section 7.5, we formulate a general site that provides such optimization routines in the Orc context. Results for optimization runs of both examples are presented in Section 7.6. This is followed by related work and conclusions in Sections 7.7 and 7.8, respectively.

7.2 Fundamentals

7.2.1 Optimization models

Optimization problems may be formulated as [BV09]:

$$\begin{aligned} &\textbf{min} \quad f_0(a,x) \\ &s.t. \quad f_i(a,x) \leq 0, \quad i = 1,\ldots,m \end{aligned} \tag{7.1}$$

where f_0 is the objective function, f_i are the set of constraint functions dependent on the input vector $x = (x_1, x_2, \ldots, x_N)^T$ and model parameters $a = (a_1, a_2, \ldots, a_M)^T$. This can be solved in a variety of linear, non-linear, stochastic and exhaustive search techniques. Approximate bounds to reduce stochastic uncertainty can also be used. This can lead to three categories of minimization problems.

- Minimization of primary expected costs subject to secondary cost constraints.

$$\begin{aligned} &\textbf{min} \quad F_0(a,x) \\ &s.t. \quad F_i(a,x) \leq F_i^{max}, \quad i = 1,\ldots,m \end{aligned} \tag{7.2}$$

  where $F_0(a,x)$ is the primary goal, $F_i(a,x)$ are secondary constraints with worst-case bounds represented by F_i^{max}.

- Minimization of the cost function with positive weights $k_0, k_1, \ldots, k_m$.

$$\textbf{min} \quad \sum_{i=0}^{m} k_i F_i(a,x) \tag{7.3}$$

- Minimization of the maximum weighted expected costs.

$$\textbf{min} \quad \max_{0 \leq i \leq m} k_i F_i(a,x) \tag{7.4}$$

Such formulations of cost functions with constraints can be applied to a variety of decisions within the web services framework. Further analysis of aspects such as Pareto-Optimality and multi-objective decision making may be explored in [MA04].

7.2.2 QoS in Web Services

Available literature on industry standards in QoS [Men02] provide a family of QoS metrics that are needed to specify SLAs. These can be subsumed into the following four general QoS observations [1]:

1. $\delta \in \mathbb{R}_+$ is the service latency. When represented as a distribution, this can subsume other metrics such as availability and reliability of the service.

2. $\$ \in \mathbb{R}_+$ is the per invocation service cost.

3. $\zeta \in \mathbb{D}_\zeta$ is the output data quality. This can represent other metrics such as data security level and non-repudiation of private data over a scale of values.

4. $\lambda \in \mathbb{R}_+$ is the inter-query interval, equivalent to considering the query rate for a service. Performance of the service will depend on negotiations with the amount of queries that can be made in a given time interval.

[1] Aspects such as scalability, interoperability and robustness are not dealt with as they are specific to the supplier side operation (not necessarily part of SLAs).

Along with QoS, the web service performs its task and returns some functional data $\rho \in \mathbb{D}_\rho$ as the output. The tuple of *(Data value, QoS value)* is used for the decision process within orchestrations. The implementation of Orc allows such typing to be specified for input and output parameters, which can be extended to QoS typing for orchestrations.

For comparing metrics with differing scales and units of measurement, a normalization or scaling technique is needed. As developed in [LB10] [ZBN+04], the normalization of QoS values $\mathbf{q}_i$ in a domain $\mathbb{D}_Q$ can be performed using a scaling function, prior to optimization. The scaling function $S(\mathbf{q}_i)$ in eq. (7.5) ensures that the range of QoS values falls within $[0, 1]$ for equivalent comparison. Essentially, this prevents larger scale values in domains (eg. latency) nullifying optimal selection in smaller valued domains (eg. boolean valued availability).

$$S(\mathbf{q}_i) = \frac{\mathbf{q}_i - \mathbf{q}_{min}}{\mathbf{q}_{max} - \mathbf{q}_{min}} \tag{7.5}$$

where $\mathbf{q}_{min}$ and $\mathbf{q}_{max}$ are the minima and maxima of the (available) distributions of these QoS domains. A generic range of values for metrics such as data quality or service invocation costs may be reduced to a comparable scales via this method. An example of scaling measured values is shown in Table 7.1. The measured values are scaled to the range $[0, 1]$ with the scaling invariant to changes in measurement units of, for instance, the response time δ.

Metric	**Measurement** $\mathbf{q}_i$	**Scaled Value** $S(\mathbf{q}_i)$
$\delta(hours)$	(0.017, 0.001, 0.0095, 0.01)	(1, 0, 0.53125, 0.5625)
$\delta(seconds)$	(61.2, 3.6, 34.2, 36)	(1, 0, 0.53125, 0.5625)
$\$(Euros)$	(9.5, 3.4, 6.8, 12)	(0.7093, 0, 0.3953, 1)
$\zeta([1, 10])$	(6, 1, 3, 8)	(0.7143, 0, 0.2857, 1)

Table 7.1: Scaling QoS metrics across domains to the range [0,1].

7.2.3 Analytic Hierarchy Process

Multiple dimensions in web services' QoS are only partially ordered, with comparisons between domains not possible. In order to use optimization routines, a total ordering of these domains is mandatory. To reconcile this, the analytic hierarchy process (AHP) can be used. Introduced by [Saa80], AHP can be used to objectify subjective evaluations of multi-criteria decisions, which essentially develops tradeoffs between domains. In order to briefly explain the AHP, we make use of an example.

Consider the pairwise assignment of relative ranks for QoS metrics as defined by a user. It is a matrix that defines the relative change between dependent QoS metrics δ, \$, ζ, λ and ρ. For simplicity, all parameters are classified as the same hierarchical level with values assigned using the relative comparison shown in Fig. 7.1. This in turn will produce a matrix $\mathbf{W} = (\mathbf{w}_{ij})$ as shown in eq. 7.6 with the subjective pairwise comparison of criterion.

$$\mathbf{W} = \begin{array}{c} \\ \delta \\ \$ \\ \zeta \\ \lambda \\ \rho \end{array} \begin{array}{c} \begin{array}{ccccc} \delta & \$ & \zeta & \lambda & \rho \end{array} \\ \begin{pmatrix} 1 & 1 & 5 & 3 & 5 \\ 1 & 1 & 5 & 3 & 5 \\ 1/5 & 1/5 & 1 & 1 & 2 \\ 1/3 & 1/3 & 1 & 1 & 3 \\ 1/5 & 1/5 & 1/2 & 1/3 & 1 \end{pmatrix} \end{array} \tag{7.6}$$

The principal eigenvector of the *positive reciprocal matrix* $\mathbf{W}$ provides the relative rankings of the parameters. As the principal diagonal of the matrix $\mathbf{W}$ consists of real

The Fundamental Scale for Pairwise Comparisons		
Intensity of Importance	Definition	Explanation
1	Equal importance	Two elements contribute equally to the objective
3	Moderate importance	Experience and judgment slightly favor one element over another
5	Strong importance	Experience and judgment strongly favor one element over another
7	Very strong importance	One element is favored very strongly over another; its dominance is demonstrated in practice
9	Extreme importance	The evidence favoring one element over another is of the highest possible order of affirmation
Intensities of 2, 4, 6, and 8 can be used to express intermediate values. Intensities 1.1, 1.2, 1.3, etc. can be used for elements that are very close in importance.		

Figure 7.1: Comparison Scale for AHP [Saa80].

values, the principal eigenvector (and corresponding highest eigenvalue) are also real valued.

Definition. 3 ***Perron Frobenius Theorem:*** *For a given positive matrix* $\mathbf{W}$*, the only positive vector* υ *and only positive constant* c *that satisfy* $\mathbf{W}\upsilon = c\upsilon$*, is a vector* υ *that is a positive multiple of the principle eigenvector of* $\mathbf{W}$ *and the only such* c *is the principal eigenvalue of* $\mathbf{W}$*.*

This eigenvector may be normalized to provide the *priority vector* for the QoS metrics. This will generate a weighted cost function for minimization, which is superior to cost function weights obtained by least squares [Saa03]. For the example above, the linear cost function after generating the normalized weight vector is shown in eq. (7.7) with scaling of values done previously according to eq. (7.5).

$$\mathbf{Z} = 0.3625\delta + 0.3625\$ + 0.0935\zeta + 0.1237\lambda + 0.0579\rho \tag{7.7}$$

A unique feature of the AHP is its ability to estimate consistency in the subjective evaluation of criteria.

Definition 1 *A* $n \times n$ *positive reciprocal matrix* $\mathbf{W} = (\mathbf{w}_{ij})$ *is a* ***Consistent Matrix****, if the highest eigenvalue* c_{max} *equals* n*. This is equivalent to* $\mathbf{w}_{ij} = \upsilon_i/\upsilon_j$*, where the eigenvector* υ *corresponds to eigenvalue* c_{max}*. Since small changes in* $\mathbf{w}_{ij}$ *imply changes in* c_{max}*, the deviation from* n *is a deviation from consistency given by* $(c_{max} - n)/(n-1)$ *which is called the* ***consistency index (CI)****.*

This technique evaluates the perturbation in the highest eigenvalue due to changes in subjective evaluation of metrics in $\mathbf{W}$. The values of the consistency index are used to generate a consistency ratio (CR), that is used to determine the consistency of the comparison. The consistency ratio must be ≤ 0.1, indicating deviations from subjective evaluations are less than an order of magnitude [Saa80]. For the example above, the highest eigenvalue has the value 5.122, producing a CI = 0.0280 and a CR = 0.0252, which is within the specified limits. Techniques outlined in [Saa03] provide steps and tools to improve consistency in the weight matrix.

7.3 Methodology

The following steps are used to solve optimization problems in web services:

1. **Scaling Inputs**: Obtain the pair of QoS domains and vector of values $(\mathbb{D}_Q, \mathbf{q})$ required for evaluation of the orchestration. For each domain $\mathbb{D}_Q$, scale the values $\mathbf{q}$ to the range $[0, 1]$ as specified in eq. (7.5).

2. **Consistent Judgements**: Extract the comparative judgement matrix $\mathbf{W} = (\mathbf{w}_{ij})$ from the user. From this, obtain the maximum eigenvalue c_{max} and the corresponding normalized eigenvector υ. If this judgement matrix is not *consistent*, examine the judgment for an entry $\mathbf{w}_{ij}$ for which $\mathbf{w}_{ij}\upsilon_j/\upsilon_i$ is the largest, and see if this entry can reasonably be made smaller. Such a change of $\mathbf{w}_{ij}$ also produces a new comparison matrix with a smaller eigenvalue, resulting in a possibly consistent matrix [Saa03]. This process may be performed either manually or automatically through iterative perturbations of $\mathbf{W}$ until consistency is achieved. Once a consistent matrix is obtained, the objective function $\mathbf{Z}$ to be minimized with linear weights υ and $(\mathbb{D}_Q, \mathbf{q})$ values may be generated.

3. **Constraints**: The scaled optimization constraints $\mathbf{C}$ in the form $(\mathbb{D}_Q, \preceq, K_Q)$, where $\mathbb{D}_Q$ is a QoS domain, $\preceq$ is a specified partial order and K_Q is the threshold value (constant or distribution quantiles), may also be set by the user.

4. **Optimization**: With a selected constraint satisfying solver with inputs $(\mathbf{Z}, \mathbf{C})$, optimization is performed. If constraints $\sum_{i=1}^{N} x_i = 1, x_i \in \{0, 1\}$ for model variables x exists in $\mathbf{C}$, it implies an integer programming problem (eg. selecting a single site). In the absence of such a constraint, the solver employs a conventional linear programming approach (eg. finding an optimal setting from a continuous distribution).

The only inputs required from the user are the judgement matrix and constraints over QoS domains. This methodology is intended to enhance previous theory [RBJ09b] with optimization routines to compare returned QoS token values.

7.4 Formulating Optimization Problems

In this section, we investigate the Dell supply chain, a choreography of *Dell Plant* and *Supply* orchestrations with a shared *Revolver* resource. This exemplifies the optimization of setting inventory levels to ensure efficient control flow and preventing contractual deviations.

The working of the Dell supply chain is taken from [KZC+04]. After a customer places an order, either by phone or through the Internet on `www.dell.com`, Dell processes the order through financial evaluation (credit checking) and configuration evaluations (checking the feasibility of a specific technical configuration), which takes two to three days, after which it sends the order to one of its manufacturing plants in Austin, Texas. These plants can build, test, and package the product in about eight hours. The general rule for production is first in, first out, and Dell typically plans to ship all orders no later than five days after receipt. In most cases, Dell has significantly less time to respond to customers than it takes to transport components from its suppliers to its assembly plants. To compensate for long lead times and buffer against demand variability, Dell requires its suppliers to keep inventory on hand in the Austin revolvers (for "revolving" inventory). Revolvers are small warehouses located within a few miles of Dell's assembly plants. Each revolver is shared by several suppliers. Inventory in revolvers is owned and managed by suppliers and charged to Dell indirectly through component pricing. To help suppliers make good management decisions, Dell shares its forecasts with them once per month.

The overall architecture of the Dell supply chain is shown in Fig. 7.2. Boxes denote peers (actors of the system), and multiple boxes or icons indicate that there exist several

Figure 7.2: Architecture of the Dell example.

instances of the considered peer. As the "Dell supervisor" involves monitoring (a topic in itself) and a lot of algorithmic inventory management, we leave aside this part of the application.

In the *Dell* supply chain [KZC+04], QoS metrics are functional in nature, with slight changes in optimal settings sending the supply chain to a dead state. The *Dell* application is a system that processes orders from customers interacting with the Dell webstore. According to [KZC+04], this consists of the following prominent entities:

- *Dell Plant* - Receives the orders from the Dell webstore and is responsible for the assembly of the components. For this they interact with the *Revolvers* to procure the required items.
- *Revolvers* - Warehouses belonging to Dell which are stocked by the suppliers. Though Dell owns the revolvers, the inventory is owned and managed by the *Suppliers* to meet the demands of the *Dell Plant*.
- *Suppliers* - They produce the components that are sent to the revolvers at Dell. Periodic polling of the *Revolvers* ensures estimates of inventory levels and their decrements.

The interaction between the *Dell Plant*, *Revolvers* and the *Suppliers* may be summarized in Fig. 7.3. The requests made by the plant for certain items will be favorably replied to if the revolvers have enough stock. This stocking of the revolvers is done independently by the suppliers. The suppliers periodically poll (withdraw inventory levels) from the revolvers to estimate the stock level. In such a case, a contract can be made on the levels of stock that must be maintained in the revolver. The customer side agreement limits the throughput rate. The supplier side agreement ensures constant refueling of inventory levels, which in turn ensures that the delay time for the customer is minimized. Thus, it represents a *choreography* comprising two plant-side and supplier-side orchestrations interacting via the revolver as a shared resource.

The critical aspect in the Dell choreography is efficient management of revolver levels. As discussed in [KZC+04], for the efficient working of the supply chain, the interaction between the Dell Plant and the Supply-side workflows should be taken into account. This will involve optimizing critical QoS metrics listed in Table 7.2. They are also presented informally in Fig. 7.3.

For the plant-side behavior, the demand λ_t reduces the current revolver level ($\mu_t = \mu_{t-1} - \lambda_t$). Constant polling at a rate ρ ensures the re-fueling of revolver inventory within a supply delay δ_{sup}. When the value of the revolver token drops below a critical level μ_c, the supplier begins the process of refueling the inventory. The refueling batch size β is governed by the maximal capacity of the revolver μ_{max}. Optimal setting of

t	Unit of time with $t \in 1, 2, ...T$ hours
λ_t	Number of queries per unit time that the plant requests the revolver
δ_{cust}	Waiting time for the plant
μ_t	Stock level for an item in the revolver at time t
μ_c	Critical stock levels of the item in the revolver
μ_{max}	Maximum stock level allowed in the revolver
ρ	Inventory polling period of the supplier
β	Size of the refueling batch from the supplier
δ_{sup}	Delay period for refueling the revolver
$\upsilon_{\mu_c}, ...\upsilon_\beta$	Normalized eigenvector from the consistent AHP matrix

Table 7.2: QoS Metrics for the Dell Supply Chain.

Figure 7.3: QoS interactions in the Dell supply chain.

these parameters minimizes the customer waiting time δ_{cust}. If the supplier does not refuel on time, the choreography sets into deadlock with the plant waiting for (possible) restocking. A deadlock occurs when a choreography reaches a state that (1) is not final and (2) can not be left without violating the message ordering of the choreography.

Considering estimated distributions of customer demand and refueling delays, effective settings for supplies may be set. Using AHP weights, the optimization procedure is given as a linear programming problem (without any integer constraints). More classical logistics cost functions [Rar98] can also be applied to similar problems.

$$\textbf{minimize} \;\; \mathbf{Z} \;=\; \upsilon_{\mu_c}\mu_c + \upsilon_{\mu_{max}}\mu_{max} + \upsilon_\beta\beta \tag{7.8}$$

Subject to the following user-specified constraints:

$$0 \leq \mu_c \leq \mu_{max} \tag{7.9}$$

$$0 \leq \beta \leq \mu_{max} \tag{7.10}$$

$$\mu_{max} - \mu_c + (\lambda_t \times \delta_{sup}) = \beta \tag{7.11}$$

$$0 \leq \mu_{max} \leq K \times \lambda_t \tag{7.12}$$

Constraints in eqs. (7.9) and (7.10) limit the revolver critical level μ_c and the supplier batch size β to be less than the maximal revolver capacity μ_{max}. The constraint in eq. (7.11) essentially controls the revolver batch size, dependent on the critical / maximum level in the revolver and the plant query rate λ_t. Estimates of λ_t are provided to the supplier during the polling period through measured decrements in the revolver

levels. The supplied batch β also incorporates the decrement in inventory since the critical level was detected, and the delay in restocking δ_{sup}. Finally, the constraint in eq. (7.12) prevents overstocking of items in the revolver by limiting the capacity to be proportional to the demand. Optimal setting of these parameters is tested by the constant demand for products λ_t which must be delivered while minimizing the customer delay δ_{cust}. Essentially, these constraints ensure the revolver level does not fall to zero, which would mean rejection or long delays in orders (deadlock in the choreography).

7.5 Optimization Routines in Orc

While the previous sections demonstrate the utility of optimization techniques when applied to decisions in workflows, it is imperative to provide a convenient technique to embed such mathematical packages within orchestrations. Extending Orc [MC07] with a suitable interface will enable smooth integration of optimization libraries for the utility of workflow designers. In this section, we provide a high-level specification of optimizing QoS metrics within Orc.

Orc [MC07] serves as a simple yet powerful concurrent programming language to describe web services orchestrations. The fundamental declaration used in the Orc language is a *site*. The type of a *site* is itself treated like a service - it is passed the types of its arguments, and responds with a return type for those arguments. An Orc *expression* represents an execution and may call external services to publish some number of values (possibly zero).

Orc has the following combinators that are used on various examples as seen in [MC07]. The *Parallel* combinator $X|Y$, where X and Y are Orc expressions, runs by executing X and Y concurrently; returns from X and Y are interleaved. Whenever X or Y communicates with a service or publishes a value, $X|Y$ does so as well. The execution of the *Sequential* combinator $X >t> Y$ starts by executing X. Sequential operators may also be written compactly as $X \gg Y$. Values published by copies of Y are published by the whole expression, but the values published by X are not published by the whole expression; they are consumed by the variable binding. If there is no response from either of the sites, the expression does not terminate. The *Pruning* combinator, written $X <t< Y$, allows us to block a computation waiting for a result, or terminate a computation. The execution of $X <t< Y$ starts by executing X and Y in parallel. Whenever X publishes a value, that value is published by the entire execution. When Y publishes its first value, that value is bound to t in X, with the execution of Y immediately terminated. The *Otherwise* combinator, written $X;Y$ has the following execution. First, X is executed. If X completes, and has not published any values, then Y executes. If X did publish one or more values, then Y is ignored. The publications of $X;Y$ are those of X if X publishes, or those of Y otherwise.

Consider the following two Orc expressions - one of which chooses the fastest responding service; another produces the lowest costing service value:

```
def minLatencySite() = s <s< (Site_1 |...| Site_N)
def minCostSite() = (Site_1 ,..., Site_N) >(c_1 ,..., c_N)> minimum([c_1
    ,..., c_N])
```

Combining these expressions in Orc can currently be done with priorities, that is, choosing a site with lower cost over one with lower latency, or vice versa. This can be detrimental in typical situations involving more than one QoS metric. Finding an optimal service that provides a "middle path" solution from various domains can be beneficial. Such an expression in Orc with weights `w`:

```
def optimalSite() = (Site_1 ,..., Site_N) >((d_1,c_1) ,...,(d_N,c_N))>
minimum([ w*d_1 + (1-w)*c_1 ,..., w*d_N + (1-w)*c_N ])
```

A drawback of the above formulation is that exhaustive comparison of metrics are still used. In order to overcome this, the selection of services can be formulated as an optimization problem. Such a formulation is useful in a variety of orchestrations where the control flow is dependent on optimal resolution of competition between services. A point to note here is that the fastest service cannot be given priority as the orchestration waits for responses from all services (until timeout).

In [RBJ09b], the "best" operator provides a general function for comparison of a variety of metrics. We propose an extension of this to satisfy more complex queries, when "enumerate and evaluate" is both ineffective and slow. Moreover, there are no standard sets of QoS parameters that are declared in general for all orchestrations - which draws the need for a framework for totally ordered metrics.

7.5.1 QOrc: Upgrading Orc for QoS management

A proposal is making use of a QoS enhanced orchestration declaration called *QOrc*. Every invoked service responds with not only the desired output data but also with a set of QoS values. So, selection of a service can entail complex queries dependent on a variety of parameters for optimization. Consider a *site* `Optima` that may be invoked during an orchestration run. This site has input tuple `(QoS, AHPWeight, Constraint, Routine)` where `QoS` is the set of QoS domains with a list of corresponding values, `AHPWeight` is a set of (normalized) weights dependent on AHP criterion, `Constraint` are the (normalized) constraint functions and `Routine` is the optimization protocol to be employed. The user can specify the routine to be either binary integer or linear programming depending on the problem. A typical implementation in Orc is:

```
def class Optima()=
  type Latency = Number
  type Cost = Number

  val Latency = Ref()
  val Cost = Ref()
  val QoS = (Latency,Cost)
  val AHPWeight = (0.3,0.7)
  val Constraint = ((Latency,("<:"),0.5), (Cost,("<:"),0.8))
  val Routine = "bin"
  def Optimization(QoS, AHPWeight, Constraint, Routine) = lpsolve
  stop
```

For example, the following orchestration describes optimal selection from three generic services, while using the `Optima` site.

```
signal >> (Site1(), Site2(), Site3()) >(s1,s2,s3)>
Optima().Optimization(
([s1.latency?,s1.cost?],[s2.latency?,s2.cost?],[s3.latency?,s3.cost?]),
AHPWeight, Constraint, Routine)
```

A library of optimization routines available as services allow complex decision making in orchestrations, even to non-specialized users of such tools. As described in the COIN-OR (COmputational INfrastructure for Operations Research) project [FMM08] [FG01], a host of solvers and APIs are provided for integrating optimization. A variety of input formats such as AMPL (A Modeling Language for Mathematical Programming), MPS (Mathematical Programming System) and GAMS (General Algebraic Modeling System) may be used to specify the problems.

7.5.2 Interfacing QOrc to Optimization Services

We use the example of the LP file format used for the open-source *lpsolve*[2] solver to demonstrate the compatibility of an input from Orc. The input syntax of the LP format uses an *Objective Function* with associated *Constraints* and variable *Declarations*. With the inputs provided from the Orc `Optima` site, the optimization problem can be conveniently formulated to the binary integer problem. Formulation of linear or more complex quadratic problems can follow this procedure to conceal intricacies of mathematical packages from non-specialist users. The transformation of these inputs, through an interface, into a LP optimization routine is represented below:

- Generate variables `x1, x2 ... xN`, where `N` equals the number of participating services. These are the variables that will be the valued as 1 or 0 during optimization and represent the selection / rejection of a particular `SiteN()`.
- The `AHPWeight` values `(w1, w2)`, and corresponding `QoS` values `[l1,...,lN], [c1,...,cN]` are combined with the variables to generate a linearly weighted cost function `(w1 l1 + w2 c1)x1 +...+ (w1 lN + w2 cN)xN`.
- The `Constraint` values provide the specified domains, partial orders and corresponding thresholds `(K1, K2)`, which are transformed into (`l1 x1 +...+ lN xN <= K1`; `c1 x1 +... + cN xN <= K2`).
- As the `Routine` "binary integer" is set, values `x1, x2,..., xN` are further constrained to be binary valued. A further constraint automatically specified is the `x1 + x2 +...+ xN = 1`, restricting only a single site is selected by the optimization procedure.

The results of such a transformation produces a LP format of the problem, that can be solved by the lpsolve optimization solver. Due to the elegant nature of Orc, this is equivalent to calling another (possibly external) *Site* with input `Optima` format and output LP format.

```
/* Objective function */
min: (0.3 l1 + 0.7 c1) x1 + (0.3 l2 + 0.7 c2) x2 + (0.3 l3 + 0.7 c3) x3;
/* Variable bounds */
l1 x1 + l2 x2 + l3 x3 <= 0.5;
c1 x1 + c2 x2 + c3 x3 <= 0.8;
x1 + x2 + x3 = 1;
bin x1, x2, x3;
```

This can be enhanced in future with direct calls to optimization packages (local or external) from within Orc as described. This would prevent switching between technologies while developing workflows in Orc and associated management of QoS dependent decisions. An example of encoding the optimization routine in Orc below with the output of the resulting optimization plugged into the LPSolve IDE in Fig. 7.4.

```
type Latency = Number
type Cost = Number
val Latency = Ref()
val Cost = Ref()
val QoS = (Latency,Cost)
val AHPWeight = (0.3,0.7)
val Constraint = ((Latency,("<:"),0.5), (Cost,("<:"),0.8))
val Routine = "bin"
def compareorder(p) =
  if (p="<:") then "<=" else if (p=":>") then "=>" else "="
```

[2] http://lpsolve.sourceforge.net/5.5/

```
--/* Objective function */
--min: (0.3 l1 + 0.7 c1) x1 + (0.3 l2 + 0.7 c2) x2 + (0.3 l3 + 0.7 c3) x3;
--/* Variable bounds */
--l1 x1 + l2 x2 + l3 x3 <= 0.5;
--c1 x1 + c2 x2 + c3 x3 <= 0.8;
--x1 + x2 + x3 = 1;
--bin x1, x2, x3;

def Optima(QoS,AHPWeight,Constraint,Routine) =
 QoS>(q1,q2,q3)> AHPWeight >(A1,A2)> Println("/* Objective function */") >>
 Println("min: "+( (A1*head(q1)) + ((A2)*last(q1)) )+" x1 +
 "+( (A1*head(q2)) + ((A2)*last(q2)) )+" x2 +
 "+( (A1*head(q3)) + ((A2)*last(q3)) )+" x3;")
 >> Println("/* Variable bounds */") >> Constraint
 >((D1,p1,t1),(D2,p2,t2))> Println(head(q1)+" x1 + "+head(q2)+" x2 +
 "+head(q3)+" x3 "+compareorder(p1)+" "+t1+" ;") >>
 Println(last(q1)+" x1 + "+last(q2)+" x2 + "+last(q3)+"
 x3 "+compareorder(p2)+" "+t2+";")
 >> Println("x1 +  x2 +  x3 "+compareorder("=")+" 1;")
 >> Println(Routine+ " x1, x2, x3;")

-- Sites return random latency, cost values
def Site1() = Dictionary() >s1> s1.latency:=URandom()
 >> s1.cost:=URandom() >> s1
def Site2() = Dictionary() >s2> s2.latency:=URandom()
 >> s2.cost:=URandom() >> s2
def Site3() = Dictionary() >s3> s3.latency:=URandom()
 >> s3.cost:=URandom() >> s3

signal >> (Site1(), Site2(), Site3()) >(s1,s2,s3)>
Optima(([s1.latency?,s1.cost?], [s2.latency?,s2.cost?],
 [s3.latency?,s3.cost?]), AHPWeight, Constraint, Routine)
```

7.6 Optimal Decision Results

The results of the optimization procedure are described in this section. Rather than concentrating on optimization aspects (primal-dual feasibility, relative error, number of iterations, etc.), we focus on the implications of using optimization as a tool in orchestrations.

Consider a plant-side site in Orc, `order(orderQnty, dellStock)`, that decrements ordered quantities `orderQnty` from the revolver counter `dellStock`. Similarly, the supply-side site `supply(polling, criticalLvl, batch, dellStock)`, polls with interval `polling` and refuels with `batch` quantity when the `dellStock` counter falls below `criticalLvl`. Over a period of time, with varying frequencies of orders, it is essential that the supplier varies parameters accordingly to ensure consistent revolver levels. Guarantees for plant and supply chain behavior may be proposed with revolver levels as a common resource for monitoring. A portion of the Orc representation for the plant-side and the supplier-side behavior is shown:

```
val dellStock = Counter(K)

--Plant--
def order(orderQnty,dellStock) = dellStock.value() >curLvl>
  (if(orderQnty<curLvl) then (decrement(dellStock,orderQnty) >> orderQnty)
  else (order(orderQnty,dellStock)))

--Supplier--
def supply(polling,criticalLvl,batch,dellStock) = dellStock.value() >curLvl>
  (if(curLvl<=criticalLvl) then (increment(dellStock,criticalLvl)
  >> Rtimer(polling) >> supply(polling,criticalLvl,batch,dellStock))
```

Figure 7.4: Output of the Orc program plugged into the LPSolve IDE.

```
else (Rtimer(polling) >> supply(polling,criticalLvl,batch,dellStock)))
```

While the plant decrements the stock level counter for the order of size `K_1`, the supplier refuels inventory with a batch size `K_2` when the stock level falls below `l_c`. Orc sites such as `Counter` and `Rtimer` are used in this program. The `increment` and `decrement` sites are used to represent the change in the counter values with ordering and refueling, respectively.

The AHP weight matrix shown in Table 7.3 is used for the optimization of the problem described in Section 7.4 using the `linprog` function in MATLAB. These are the judgement criteria that can be fixed by the user / service orchestrator as the inputs to the optimization solver. The polling period ρ is set to a constant of 1 hour to limit model parameters. By setting the customer demand and supplier delay distributions, the

	$\mathbf{w}_{\mu_c}$	$\mathbf{w}_{\mu_{max}}$	$\mathbf{w}_{\beta}$	**Normalized Vector** v
$\mathbf{w}_{\mu_c}$	1	1/3	1/5	0.1047
$\mathbf{w}_{\mu_{max}}$	3	1	1/3	0.2583
$\mathbf{w}_{\beta}$	5	3	1	0.6370
$c_{max} = 3.0385$, CI $= 0.0193$, CR $= 0.0370$				

Table 7.3: Parameters for Dell supply chain optimization.

optimization produces the distributions of the refuel batch size, critical and maximum stock levels as shown in Fig. 7.5. These settings, when applied to the Dell system provides the system performance as shown in Fig. 7.6. The cumulative distribution of the revolver inventory remains stochastically above the critical distribution for 10000

runs, being refueled periodically by the supplier. As a consequence, the revolver stock level μ_t does not drop to zero throughout the simulation period. This demonstrates that the optimization formulation through AHP is robust to changes in inputs of demand and delay distributions. Though scaling as in eq. (7.5) has been employed, the normalized values are omitted from figures (to demonstrate realistic outputs).

Figure 7.5: Optimal setting of parameters in the Dell Supply Chain.

Figure 7.6: Distributions of the inventory levels in the Dell system.

As further seen in one particular setting of the Dell example in Fig. 7.7, the linear programming method converges within a few iterations to the optimal value. This is true for well formulated linear programming problems with optimal outputs produced (relative errors of the order of $\leq 10^{-6}$) for most input settings.

Parameters for optimal evaluations such as relative error, maximum number of iterations and so on can be set conveniently with most generic optimization solvers. Such a precise setting of parameters are needed for orchestrations like the Dell supply chain, to prevent unwarranted delay in production and supply of parts (choreography deadlock). This example highlights the crucial use of optimization and associated packages for managing QoS in complex workflows.

Such optimization of decisions within web services serve two purposes :

```
Command Window

  Residuals:   Primal      Dual      Duality     Total
               Infeas     Infeas      Gap         Rel
               A*x-b    A'*y+z-f    x'*z        Error
  ---------------------------------------------------
  Iter    0:  1.76e+002 5.59e+000 1.35e+003 1.00e+002
  Iter    1:  2.07e-014 3.43e-002 8.13e+001 6.73e-001
  Iter    2:  1.37e-012 1.25e-016 5.39e+000 1.31e-001
  Iter    3:  2.75e-012 1.67e-016 4.76e-002 1.22e-003
  Iter    4:  1.48e-012 1.39e-017 2.47e-006 6.36e-008
  Iter    5:  6.47e-015 5.46e-014 2.47e-013 7.18e-014
Optimization terminated.
```

Figure 7.7: Optimization output for a single setting of the Dell example in MATLAB.

1. Optimization of a variety of parameters should produce efficient and beneficial choices in service orchestrations. Such optimizations can be applied to intricate workflows in logistics and queuing systems to optimize resource allocations. Expecting comparative judgement between metrics and simple constraints reduces the onus of a orchestration designer to understand intricacies of more complex optimization modeling in workflows. Incorporating this technique within languages such as Orc should provide a useful tool for designers of complex workflows.

2. Efficient, multi-dimensional tradeoff dependent decision making. This means a lower priority metric should not prevent a composite service run. Conventional enumerate options would wait until timeout to find the "best" return from a particular metric. Constraint dependent optimization can prevent this with all metrics crossing the constraint threshold available immediately for selection.

7.7 Related Work

Analysis of QoS in web services orchestrations has received considerable attention. In [HWTS07], Hwang et al. use QoS parameters as random variables for composition. Rosario et al. [RBHJ08] provide a framework for probabilistic contracts modeling QoS parameters as random variables. Instead of using fixed hard bound values for parameters such as response time, the authors proposed a soft contract monitoring approach to model the QoS bounds. This is further developed with a theory for QoS modeling within the Orc framework in [RBJ09b]. We extend the "best" operator from this theory to accommodate alternatives to exhaustive search.

Though there are many techniques available for optimizing functions [BV09] routines needed to incorporate them into orchestrations is still a developing area. In the paper by Alrifai and Risse [AR09a] the use of mixed integer programming is proposed to find the optimal decomposition of global QoS constraints into local constraints. Optimal QoS compositions make use of genetic programming in Canfora et al. [CPEV05a] and linear programming in Zeng et al. [ZBN+04]. The use of a reputation guided selection and feedback dependent policy for web services is outlined in [LB10]. In [YZL07], the optimization of dynamic service compositions are modeled as a multidimension-multichoice knapsack problem (MMKP). MMKP of medium sizes can be solved by most commercial integer-linear programming solvers, as employed in this paper. A framework for specifying optimizations within Orc workflows would aid in deploying real-world applications. This can then be combined with a host of optimization solvers [FMM08] [FG01] applied to most QoS dependent decisions in service orchestrations.

In this paper, we extend the concepts of optimizing cost function defined via AHP to complex queries in workflows. Extending such a framework to orchestrations can provide more complex queries to be incorporated with flexibility in comparing domains.

The Dell optimization example from [KZC+04] provide realistic case studies within the web service framework where optimal QoS values affect functioning of the orchestration.

Analytical hierarchy process developed by Saaty [Saa80] has been shown to be applied to diverse fields including manufacturing, logistics, finance and management. Work by Ho [HLH10] reviews the combination of AHP to mathematical models including linear programming, integer linear programming, mixed integer linear programming, and goal programming. An application of AHP for automated negotiation of SLAs are studied in [CCP07]. In [YSS05], another multi-criteria decision making approach (PROMETHEE) is used to extend the decision making for exhaustive comparison of web services' QoS.

7.8 Conclusion

With increasing need for decision making capabilities in services orchestrations, the use of mathematical packages like optimization should be employed for leveraging QoS dependent choices. Embedding optimization routines as part of orchestration specifying languages like Orc provides the capability to use these tools for runtime decision making in a variety of workflows. A simple extension of user defined criterion and constraints is proposed to specify such optimization problems for non-specialist workflow designers. By applying the AHP, we show that a consistent minimizing cost function can be developed for total ordering QoS metrics. Demonstrating this methodology for the Dell supply chain example, it is shown to be effective in solving realistic problems in resource allocation and logistics. Such techniques are required to estimate optimal decisions on runtime, dependent on variations in associated QoS parameters.

Chapter 8

Importance Sampling/Splitting of Probabilistic Contracts in Web Services

Ajay Kattepur IRISA/INRIA, Campus Universitaire de Beaulieu,
Rennes-Cedex, France.

Abstract

With web services quality of service (QoS) modeled as random variables, the accuracy of sampled values for precise service level agreements (SLAs) come into question. Samples with lower spread are more accurate for calculating contractual obligations of the population, which is typically not the case for web services QoS. Moreover, the extreme values in case of heavy-tailed distributions (eg. 99.99 percentile) are seldom observed through limited sampling schemes. To improve the accuracy of contracts, we propose the use of variance reduction techniques such as importance sampling and importance splitting. A consequence of this is a formulation of a more precise SLA definition for orchestrations and choreographies. We demonstrate this for contracts involving *demand* and *refuel* operations within the *Dell* supply chain example. Using measured values, efficient forecasting of future deviation of contracts may also be performed. A consequence of this is a more precise definition of sampling, measurement and variance tolerance in SLA declarations.

8.1 Introduction

Web services continue to attract applications in many areas [ACKM04]. With increasing efforts to standardize performance of web services, focus has shifted to Quality of Service (QoS) levels. This is important to consider in case of orchestrations that specify the control flow for multiple services. To this end, contractual guarantees and service level agreements (SLAs) [BSC01] are critical to ensure adequate QoS performance.

QoS metrics being random variables, the treatment of contractual obligations tends toward probabilistic criterion [RBHJ08]. Contractual obligations may be specified as varying percentile values of such distributions rather than "hard" values. In [RBJ09b], composition and monitoring such contracts with stochastic dominance have been examined.

As metrics such as response time and throughput rates can have heavy tails, estimating extreme values becomes difficult with few observations. The *availability* of a web service might need contracts for extreme percentiles in the response time profile (99.99 percentile). For instance, an ambulance or disaster management web service must be available 24×7, indicating a high availability requirement. These values are dependent on sampled random values and can lead to high variance in contractual guarantees.

The use of *importance sampling* and *importance splitting* [Buc04] is proposed as a solution to these problems. Disadvantages of conventional Monte-Carlo techniques such as high variance of percentile values may be eliminated. In case of heavy tailed distributions, unobserved extreme percentiles can be quantified with higher accuracy. These are stochastically "important" observations to estimate contractual deviations. These issues are demonstrated with the *Dell* example [KZC+04], a choreography involving *Dell Plant* and *Supplier* orchestrations. We study more accurate bounds for supplier contracts with varying plant demand rates. Further, we show how QoS metrics such as stock level deviations (specially long delays) can be estimated with low variance.

Using a precise declaration of distribution structures, exact quantiles and variance in measurements can be derived. This form of extended specifications will reduce ambiguities in SLA specifications, for example in standards such as WSLA [LKD+03]. Monitoring deviations in contracts can also be simplified with stochastic dominance relations replaced by probability of exceeding specific quantile values.

The rest of the paper is organized as follows: Section 8.2 contains foundations for the paper including QoS in web services 8.2.1, probabilistic contracts 8.2.2 and Orc 8.2.3. Importance sampling and Importance Splitting are briefly introduced in Section 8.3. The Dell application is introduced in Section 8.4 with the "Plant" and "Supplier" workflows interacting in a choreography. The two application of importance sampling with respect to the Dell supply chain are described in Sections 8.4.1 and 8.4.2. Upgrading current WSLA specifications to include such quantile values are studied in 8.5. Related work and conclusions of the paper are included in Sections 8.6 and 8.7, respectively.

8.2 Foundations

This section introduces concepts required to understand the rest of the paper.

8.2.1 QoS in Web Services

Available literature on industry standards in QoS [W3c03] provide a family of QoS metrics that are needed to specify SLAs. These can be subsumed into the following four general QoS observations:

1. *Service latency or Response Time* $\in \mathbb{R}_+$ - When represented as a distribution, this can subsume other metrics such as availability and reliability of the service.

2. *Per invocation service cost* $\in \mathbb{R}_+$ - Can be modeled as a number extracted from a uniform distribution of cost ranges.

3. *Output Data Quality* $\in \mathbb{N}$ - This can represent other metrics such as data security level and non-repudiation of private data over a scale of values.

4. *Inter-query interval* $\in \mathbb{R}_+$ - equivalent to considering the query rate for a service. Performance of the service will depend on negotiations with the amount of queries that can be made in a given time interval.

Along with QoS, a functional web service returns some data value. The tuple of *(Data value, QoS value)* is used for the decision process within orchestrations. To handle such diverse domains, metrics and algebra for QoS, a framework is proposed in [RBJ09b]. Using such an algebra, QoS metrics may be defined explicitly with domains, increments and comparisons within service orchestrations.

8.2.2 Probabilistic Contracts

To handle such diverse domains, metrics and algebra for QoS, a framework is proposed in [RBJ09b]. Using such an algebra, QoS metrics may be defined explicitly with domains, increments and comparisons within service orchestrations.

For a domain $\mathbb{D}_Q$ of a QoS parameter Q, behavior can be represented by its distribution F_Q:

$$F_Q(x) = \mathbb{P}(Q \leq x) \tag{8.1}$$

Making use of stochastic ordering [BD03], this is refined for probability distributions F and G over a totally ordered domain $\mathbb{D}$:

$$G_Q \preceq F_Q \quad \Longleftrightarrow \quad \forall x \in \mathbb{D}_Q, \quad G_Q(x) \geq F_Q(x) \tag{8.2}$$

That is, there are more chances of being less than x (partial order $\preceq$) if the random variable is drawn according to G than according to F. A QoS contract must specify the obligations of the two parties:

- The obligations that the orchestration has regarding the service are seen as *assumptions* by the service - the orchestration is supposed to meet them.

- The obligations that the service has regarding the orchestration are seen as *guarantees* by the service - the service commits to meeting them as long as assumptions are met.

Definition. 4 *A probabilistic contract is a pair (Assumptions, Guarantees), which both are lists of tuples* $(Q, \mathbb{D}_Q, F_Q)$*, where* Q *is a QoS parameter with QoS domain* $\mathbb{D}_Q$ *and distribution* F_Q*.*

Once contracts have been agreed, they must be monitored by the orchestration for possible violation as described in [RBHJ08].

8.2.3 Orc

Orc [MC07] serves as a simple yet powerful concurrent programming language to describe web services orchestrations. The fundamental declaration used in the Orc language is a *site*. The type of a *site* is itself treated like a service - it is passed the types of its arguments, and responds with a return type for those arguments. An Orc *expression* represents an execution and may call external services to publish some number of values (possibly zero).

Orc has the following combinators that are used on various examples as seen in [MC07]. The *Parallel* combinator $X|Y$, where X and Y are Orc expressions, runs by executing X and Y concurrently; returns from X and Y are interleaved. Whenever X or Y communicates with a service or publishes a value, $X|Y$ does so as well. The execution of the *Sequential* combinator $X >t> Y$ starts by executing X. Sequential operators may also be written compactly as $X \gg Y$. Values published by copies of Y are published by the whole expression, but the values published by X are not published by the whole expression; they are consumed by the variable binding. If there is no response from either of the sites, the expression does not terminate. The *Pruning* combinator, written $X <t< Y$, allows us to block a computation waiting for a result, or terminate a computation. The execution of $X <t< Y$ starts by executing X and Y in parallel. Whenever X publishes a value, that value is published by the entire execution. When Y publishes its first value, that value is bound to t in X, with the execution of Y immediately terminated. The *Otherwise* combinator, written $X;Y$ has the following execution. First, X is executed. If X completes, and has not published any values, then Y executes. If X did publish one or more values, then Y is ignored. The publications of $X;Y$ are those of X if X publishes, or those of Y otherwise. An interested reader is referred to [1] for further documentation, examples and idioms of Orc.

8.3 Rare Event Simulation Techniques

In case of web services' SLAs, these rare event simulations can be used to determine the occurrence of failure or deviation from contracts. Traditional Monte-Carlo (MC) methods waste a lot of time in a region of the state space which is "far" from the rare set of interest. Modifying the underlying distributions to move "near" the states of interest provides a more efficient means of analysis. With typical Monte-Carlo (MC), if the mean $\mu = 10^{-5}$ and if we want the expected number of occurrences of this event to be at least 100, we must take approximately $N = 10^7$ runs. For lower values of N, not even a single occurrence of this event may be seen - leading to the faulty conclusion that the event does not occur.

The use of rare event simulations in web services:

1. *High Availability Contracts* - High availability refers to the ability of a system to perform its function continuously (without interruption) for a significantly longer period of time than the reliabilities of its individual components would suggest. The degree of availability can be characterized by orders of magnitude. In order to compose services with pre-defined contracts, such high availability requirements need precise definitions. Once contractual guarantees are agreed upon, using importance sampling, the rate of deviation from extreme percentiles may be extracted. This can then be composed to obtain end-to-end availability of composite services.

[1] http://orc.csres.utexas.edu/index.shtml

2. *Forecasting* - In case of large web services, monitoring is dependent on samples of observed values obtained from a QoS distribution. For such cases, non-deviation of contracts may not necessarily imply adequate performance levels. If the customer of an atomic service desires to query the probability of the service failing over an interval, conventional sampling may not suffice. In such cases, the observed values can be combined with importance sampling to provide better estimates for contractual deviation. With more precise estimates, contingency plans can be made to replace certain services with alternatives in case of outage scenarios.

As in the case of most statistical techniques, the monitoring of contracts is also based on *samples* of the *population* of QoS. If the variance in values of the sample set is large then the mean is not as representative of the data as if the spread of data is small. If only a sample is given and we wish to make a statement about the population standard deviation(from which the sample is drawn), then we need to use the sample standard deviation. If $Q_1, Q_2, ..., Q_N$ is a sample of N observations, the sample variance is given by:

$$s^2 = \frac{\sum_{i=1}^{N}(Q_i - \bar{Q}^2)}{N-1} \tag{8.3}$$

with $\bar{Q}$ as the sample mean. This sample standard deviation can be used to represent the deviation in the population QoS output and is used in this paper.

8.3.1 Importance Sampling

Importance sampling (IS) [Buc04] increases the probability of the rare event while multiplying the estimator by an appropriate likelihood ratio so that it remains unbiased. Consider the case of a random variable Q with probability density function (PDF) F_Q for which the probability of a rare event $\mathbb{P}(H(Q) > \Phi)$ is to be estimated. Here $H(Q)$ is a continuous scalar function and Φ is the threshold. Using Monte-Carlo, one generates independent and identically distributed samples $Q_1, Q_2, ...Q_N$ from the PDF F_Q and then estimates the probability:

$$\mathbb{P}_{MC} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}_{H(Q_i)>\Phi} \tag{8.4}$$

where $\mathbf{1}_{H(Q)>\Phi}$ is 1 if $H(Q) > \Phi$ and 0 otherwise. For a rare event, such a technique needs many runs for low variance estimates.

With Importance Sampling (IS) [Buc04], variance can be reduced without increasing the number of samples. The idea is to generate samples $Q_1, Q_2, ...Q_N$ from an auxiliary PDF G_Q and then estimate probability:

$$\mathbb{P}_{IS} = \frac{1}{N}\sum_{i=1}^{N} H(Q_i)\mathbf{1}_{H(Q_i)>\Phi}\frac{F_Q(Q_i)}{G_Q(Q_i)} \tag{8.5}$$

It is evident that G_Q should be chosen such that it has a thicker tail than F_Q. If F_Q is large over a set but G_Q is small, then $\left(\frac{F_Q}{G_Q}\right)$ would be large and it would result in a large variance. It is useful if we can choose G_Q to be similar to F_Q in terms of shape. Analytically, we can show that the best G_Q is the one that would result in a variance that is minimized [Buc04]. In order to perform this selection, some sort of knowledge about the distribution is assumed, either through theory or pre-collected statistical data.

Instead of a pure Monte-Carlo run for determining the assumption/guarantee contracts as in [RBHJ08], the importance sampling algorithm may be used. Such a procedure requires generation of accurate quantiles and threshold from probabilistic distributions. Rather than concentrating on median / mean values that may be estimated with high confidence, it is critical for precise SLAs to determine > 90 percentiles of such distributions with accuracy. With inputs of a sampling distribution G, observed distribution F and the function H to select the occurrence of the "rare" event:

Algorithm 3: Importance Sampling

1 Choose G such that $sup(G) \supset sup(F \cdot H)$
2 **for** $i = 1...n$ **do**
3 Generate $X_i \sim G$
4 Set $W(X_i) = \frac{F(X_i)}{G(X_i)}$
5 Return $\mathbb{E}[H(X)] = \frac{\sum_{i=1}^{n} W(X_i)H(X_i)}{n}$

Choosing the right distribution to sample from will involve using the cross entropy method [RK04], which can provide near optimal distributions for variance reduction. A generic cross entropy algorithm is given as follows with $\mathbf{v}$ representing tuning parameter for distribution families $f(X_i; \mathbf{v})$.

Algorithm 4: Cross Entropy Algorithm

1 Choose initial parameter vectors $\mathbf{v}_0$; set $t = 1$
2 Generate $X_1, \ldots, X_N$ from distribution $f(.; \mathbf{v}_{t-1})$
3 **while** $\mathbf{v}_t \neq \mathbf{v}_{t-1}$ **do**
4 Solve $\mathbf{v}_t = \underbrace{\arg\max}_{\mathbf{v}} \frac{1}{n} \sum_{i=1}^{N} H(X_i) \frac{F(X_i)}{f(X_i; \mathbf{v}_{t-1})} \; log \; f(X_i; \mathbf{v})$
5 Set $t = t + 1$
6 Return distribution $G(.) = f(.; \mathbf{v}_t)$

8.3.2 Importance Splitting

Another alternative to traditional Monte Carlo that does not depend on the prior knowledge of distributions. This makes use of multiple quantiles α to determine thresholds that must be crossed to draw closer to possibility of obtaining rare events. Importance splitting (ISP) is based on the assumption that rarity comes from the occurrence of a low probability transition that can be decomposed in several higher-probability transitions.

Using a stochastic process $X(\mathbf{S})$ in a space $\mathbf{S}$, the state may be partitioned into two subsets $\mathbf{S} = \mathbf{C} \cup \mathbf{R}$. Here, $\mathbf{R}$ is the set of states that are rare and of interest (eg. 99 percentile), while $\mathbf{C}$ are the more commonly observed states. As the steady state is reached ($X(\mathbf{S}) \Rightarrow X_\infty$), a measure that is needed is $\mathbb{E}(f(X_\infty))$, where $f(x) = 1_{\{x \in \mathbf{R}\}}$. Traditional importance splitting [LDT07] algorithms divide the sample space into fixed quantiles α and produce new values from samples $\phi(X)$ crossing these threshold quantiles (using Metripolis-Hastings or Gibbs sampling algorithms). Essentially, this process increases the likelihood of observing quantiles that are "far" away from general observations.

Improvements proposed in [CDMFG11] intend to remove fixing these quantiles when the system is a black box, without any knowledge of the quantile ranges. In this algorithm, multiple levels are sequentially generated starting from an initial level $\Phi(X)_{(1-p_0)}$

Algorithm 5: Importance Splitting

1 Set $k = 0$
2 Generate N_k samples $X_1^k, ..., X_{N_k}^k$ from $f_k(X)$ and their α quantiles q_α^k
3 Set threshold S to determine the rare event $\mathbb{P}(\phi(X) > S)$
4 **while** $q_\alpha^k < S$ **do**
5 Determine subset $\phi(X) > q_\alpha^k$ and its conditional density f_k
6 $k = k + 1$
7 Return $\mathbb{P} = (1-\alpha)^k \times \frac{1}{N_k} \sum_{i=1}^{N_k} \mathbf{1}_{\phi_{(X_i^k)} > S}$

with p_0 typically between $0.75 \leq p_0 \leq 0.8$. Once the final threshold level $\mathbf{L}$ is crossed, the probability of the rare event is estimated. As this is closer to the web services' distributions case (where quantiles may be unknown), we make use of this algorithm as the ISP case for the rest of the paper.

Algorithm 6: Importance Splitting with adaptive levels [CDMFG11]

1 Set $k = 0$
2 Generate N samples $X_1, ..., X_N$ from $f_k(X)$ and their $(1-p_0)$ quantile $L_1 = \Phi(X)_{(1-p_0)}$
3 Set threshold $\mathbf{L}$ of the quantile to be crossed
4 **while** $L_k < \mathbf{L}$ **do**
5 Generate N samples from $X|\Phi(X)_{(1-p_0)} > L_k$ Compute the quantile $L_{k+1} = \Phi(X)_{(1-p_0)}$
6 $k = k + 1$
7 Return $\mathbb{P} = (p_0)^k \times \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}_{(1-p_0) > L}$

8.4 Dell Supply Chain

To demonstrate the variation in the QoS domains in real-world services, we study the *Dell* example [KZC+04]. The *Dell* application is a system that processes orders from customers interacting with the Dell webstore. After a customer places an order, either by phone or through the Internet on `www.dell.com`, Dell processes the order through financial evaluation (credit checking) and configuration evaluations (checking the feasibility of a specific technical configuration), which takes two to three days, after which it sends the order to one of its manufacturing plants in Austin, Texas. These plants can build, test, and package the product in about eight hours. The general rule for production is first in, first out, and Dell typically plans to ship all orders no later than five days after receipt. In most cases, Dell has significantly less time to respond to customers than it takes to transport components from its suppliers to its assembly plants. To compensate for long lead times and buffer against demand variability, Dell requires its suppliers to keep inventory on hand in the Austin revolvers (for "revolving" inventory). Revolvers are small warehouses located within a few miles of Dell's assembly plants. Each revolver is shared by several suppliers. Inventory in revolvers is owned and managed by suppliers and charged to Dell indirectly through component pricing. To help suppliers make good management decisions, Dell shares its forecasts with them once per month.

The overall architecture of the Dell supply chain is shown in Fig. 8.1. Boxes denote peers (actors of the system), and multiple boxes or icons indicate that there exist several

Figure 8.1: Architecture of the Dell example.

instances of the considered peer. As the "Dell supervisor" involves monitoring (a topic in itself) and a lot of algorithmic inventory management, we leave aside this part of the application.

According to [KZC+04], this consists of the following prominent entities:

- *Dell Plant* - Receive the orders from the Dell webstore and are responsible for the assembly of the components. For this they interact with the *Revolvers* to procure the required items.
- *Revolvers* - Warehouses belonging to Dell which are stocked by the suppliers. Though Dell owns the revolvers, the inventory is owned and managed by the *Suppliers* to meet the demands of the *Dell Plant*.
- *Suppliers* - They produce the components that are sent to the revolvers at Dell. Periodic polling of the *Revolvers* ensure estimates of inventory levels and their decrements.

Essentially, there are a *Dell Plant* and *Supplier* orchestrations that are choreographed through common *Revolvers*. The critical aspect in the Dell choreography is efficient management of revolver levels. It is a shared *buffer* resource that is accessed by both the Dell Plant and the Suppliers. As discussed in [KZC+04], for the efficient working of the supply chain, the interaction between the Dell Plant and the Supply-side workflows should be taken into account. The QoS metrics are functional in nature, with slight changes in optimal settings sending the supply chain to a dead state.

Consider a plant-side site in Orc, `order(orderQnty, dellStock)`, that decrements ordered quantities `orderQnty` from the revolver counter `dellStock`. Similarly, the supply-side site `supply(polling, criticalLvl, batch, dellStock)`, polls with interval `polling` and refuels with `batch` quantity when the `dellStock` counter falls below `cricticalLvl`. Over a period of time, with varying frequencies of orders, it is essential that the supplier varies parameters accordingly to ensure consistent revolver levels. Guarantees for plant and supply chain behavior may be proposed with revolver levels as a common resource for monitoring. A portion of the Orc representation for the plant-side and the supplier-side behavior is shown below.

```
val dellStock = Counter(K)

--Plant--
def order(orderQnty,dellStock) = dellStock.value() >curLvl>
   (if(orderQnty<curLvl) then (decrement(dellStock,orderQnty) >> orderQnty)
   else (order(orderQnty,dellStock)))

--Supplier--
```

```
def supply(polling,criticalLvl,batch,dellStock) = dellStock.value() >curLvl>
   (if(curLvl<=criticalLvl) then (increment(dellStock,criticalLvl)
   >> Rtimer(polling) >> supply(polling,criticalLvl,batch,dellStock))
   else (Rtimer(polling) >> supply(polling,criticalLvl,batch,dellStock)))
```

While the plant decrements the stock level counter for the order of size `orderQnty`, the supplier refuels inventory with a batch size `batch` when the stock level falls below `criticalLvl`. Orc sites such as `Counter` and `Rtimer` are used in this program. The `increment` and `decrement` sites are used to represent the change in the counter values with ordering and refueling, respectively.

The requests made by the plant for certain items will be favorably replied to if the revolvers have enough stock. This stocking of the revolvers is done independently by the suppliers. The suppliers periodically poll (withdraw inventory levels) from the revolvers to estimate the stock level. In such a case, a contract can be made on the levels of stock that must be maintained in the revolver. The customer side agreement limits the throughput rate. The supplier side agreement ensures constant refueling of inventory levels, which in turn ensures that the delay time for the customer is minimized. Thus, it represents a *choreography* comprising two plant-side and supplier-side orchestrations interacting via the revolver as a shared resource.

8.4.1 Contract Composition

For the *Dell* example, as QoS metrics are inherent to the functionality of the choreography, specifying explicitly probabilities of outage is necessary. Proposed are the following two concrete metrics that qualitatively evaluate these workflows:

- **Assumption**: The *demand* (number of orders/hour) distributions from the Dell plant made to a particular revolver. It is the prerogative of the plant to maintain demand within acceptable range of the contracts.
- **Guarantee**: The *delay* (hours) distribution in obtaining products from revolvers. This, in turn, is dependent on the availability of products in the revolver. The suppliers ensure efficient and timely refueling to maintain acceptable delays in the supply chain.

Consider the *assumption* on the query rate of the customer shown as an exponential distribution as in Fig. 8.2. Repeatedly pinging the service in order to receive boundary values of the distribution is expensive and not reflective of run-time performance. This is demonstrated for three values in Table 1 with 10000 runs. Conventional Monte-Carlo does not detect the probability of inter-query periods being less than 100, 50 or 20 minutes (which can be fallaciously interpreted as the rare event never occurring).

Using an importance sampling distribution, accurate mean and sampling variance values are produced for the probability of crossing these thresholds. Such a level of accuracy is needed specially for critical web services (crisis management such as ambulance or fire stations). For conventional web services contracts as well, such precise contractual obligations can reduce the need for extended monitoring of services contracts.

A corresponding *guarantee* from the service provider regarding the response time may be estimated as a long tailed distribution (Fig. 8.3, Table 2). Once again we concentrate on the outlying percentile values. The outputs for the traditional Monte-Carlo runs produce higher sample variance compared to the importance sampling scheme.

We repeat the same experiment using importance splitting in a similar setting for response time distributions that cannot be fit into a particular distribution. For example, consider the distribution shown in Fig. 8.4 that is a typical web service response time measured online. When such a service intends to provide a *guarantee*, extreme

Figure 8.2: Inter-query period distributions and fitting.

Figure 8.3: Response time distributions and fitting.

quantiles are not observed by conventional Monte-Carlo. This can be remedied by using importance splitting as shown in Table 8.3. We make use of algorithm 6 to measure the probability of percentile values crossing the specified threshold.

However, in most general settings, it may be assumed that a distribution fit is possible for latency or inter-query intervals. In such cases, importance sampling is quicker and computationally less expensive. When this is not possible, a combination of either of these (IS/ISP) techniques may be used.

Inter-query period (mins.)	mean MC	variance MC	mean IS	variance IS
100	0	0	0.0086	0.0094
50	0	0	0.0018	7.36×10^{-5}
20	0	0	7.99×10^{-5}	7.37×10^{-6}

Table 8.1: Inter-query periods by Monte-Carlo (MC) and Importance Sampling (IS).

Percentile	Latency (mins.)	mean MC	variance MC	mean IS	variance IS
99.9	44.61	0.0022	2.456×10^{-6}	0.0018	3.5548×10^{-7}
99.99	69.58	5.2×10^{-4}	5.65×10^{-7}	3.04×10^{-4}	3.82×10^{-8}
99.999	125.70	1.1×10^{-4}	1.19×10^{-7}	3.47×10^{-7}	3.12×10^{-9}

Table 8.2: Latency by Monte-Carlo (MC) and Importance Sampling (IS) schemes.

Figure 8.4: Measured response time and thresholds (without distribution fitting).

Percentile	Latency Threshold(mins.)	mean ISP	variance ISP
99	20.0	0.0048	0.0025
99.9	40.0	1.269×10^{-4}	1.1081×10^{-4}
99.99	50.0	9.985×10^{-5}	1.4979×10^{-7}

Table 8.3: Latency by Importance Splitting (ISP) schemes.

8.4.2 Forecasting

Traditional forecasting models like autoregressive moving averages [MW77] rely heavily on accurate mathematical modeling of workflow processes. In this section, we propose using pre-identified contracts / observations to provide an easier method of forecasting outages in web services orchestrations. Consider a Dell revolver with critical stock of 10 items, refueling batch 50 items and a polling period of 10 hours. With an *assumption* distribution of orders/hour shown in Fig. 8.5, the response time distribution obtained over a period of 1 week is shown in Fig. 8.6. If an item is available, it is procured immediately. Else, it is refueled with a supplier delay when polling detects sub-critical revolver levels.

Figure 8.5: Assumption: *Plant* side demand distributions.

In order to develop a *guarantee* distribution, the Dell plant must estimate the probability that delays over 72, 96 or 120 hours are experienced (leading to cancellation in orders). Through importance sampling, these values can be better estimated as in

Figure 8.6: Guarantee: *Supplier* side procurement delays.

Table 3. Notice that the variance through importance sampling is several orders of magnitude lower than conventional Monte-Carlo. The Dell plant can provision more stringent supplier obligations to reduce the delays. For instance, changing the critical stock to 50 items, refueling batch 200 items produces a new set of values, with lower probabilities of crossing outlying values as shown in Table 4.

Such changes produced by improved supplier performance is barely observed through traditional Monte-Carlo sampling, thus proving the efficacy of Importance Sampling. Application of forecasting through pre-negotiated contracts emphasize the need for precise contractual obligations needed in web services.

Delay (hours)	**mean MC**	**variance MC**	**mean IS**	**variance IS**
72	0.002	3.2×10^{-3}	0.0016	1.72×10^{-7}
96	0	0	3.88×10^{-4}	3.71×10^{-8}
120	0	0	1.02×10^{-4}	6.25×10^{-9}

Table 8.4: Original contract estimates.

Delay (hours)	**mean MC**	**variance MC**	**mean IS**	**variance IS**
72	0	0	4.08×10^{-4}	1.35×10^{-8}
96	0	0	9.91×10^{-5}	1.89×10^{-9}
120	0	0	2.734×10^{-5}	6.74×10^{-10}

Table 8.5: Reformulated contract estimates providing lower probabilities of delay.

8.5 Upgrading WSLA Specifications

The WSLA framework has components to specify and monitor contracts within the web services. The currently employed WSLA framework is of the form:

```
<Obligations>

<Schedule name="MainSchedule">
   <Period>
   <Start>2011-04-01 T12:00</Start>  <End>2011-05-01 T12:00</End> </Period>
   <Interval>
   <Hours>1</Hours>  <Minutes>0</Minutes>  <Seconds>0</Seconds> </Interval>
</Schedule>
```

```
<SLAParameter name="AverageResponseTime" type="float" unit="seconds">
   <Metric>AverageResponseTime</Metric> </SLAParameter>
<Predicate xsi:type="wsla:Less">
   <SLAParameter>AverageResponseTime</SLAParameter>
      <Value>5</Value>
</Predicate>

</Obligations>
```

In such a case, the contract is formulated as a `AverageResponseTime` value that must not cross a threshold. The monitoring procedure is also specified with an `<Interval>` period of 1 hour. Such a monitoring strategy can lead to outages (when not monitored) or very pessimistic contracts. Based on rich specifications of languages such as QML [FK98], this can be refined with precise probabilistic percentile values of QoS distributions.

```
<Obligations>

<Assumptions>
<SLAParameter name="InterQueryPeriod" type="float" unit="seconds">
   <Metric>InterQueryPeriod</Metric> </SLAParameter>
<Predicate xsi:type="wsla:Greater">
   <SLAParameter>InterQueryPeriod</SLAParameter>
      <Percentile>99</Percentile>  <Value>30</Value>
      <Percentile>50</Percentile>  <Value>70</Value>
      <MeasurementVariance>10^-3</MeasurementVariance> </Predicate>
</Assumptions>

<Guarantees>
<SLAParameter name="ResponseTime" type="float" unit="seconds">
   <Metric>ResponseTime</Metric> </SLAParameter>
<Predicate xsi:type="wsla:Less">
   <SLAParameter>ResponseTime</SLAParameter>
      <Percentile>99</Percentile>  <Value>15</Value>
      <Percentile>50</Percentile>  <Value>6.50</Value>
      <MeasurementVariance>10^-3</MeasurementVariance> </Predicate>
</Guarantees>

</Obligations>
```

The contract now specifies the contract from the *assumption-guarantee* viewpoint. For any measurement period, the `<Percentile>` values of the `ResponseTime` should be less than the specified bounds. On the other hand, the `InterQueryPeriod` should be greater than the threshold values. In both cases, the *sample variance* is taken into account.

Such a framework allows for distributions to be used for both contractual specification and monitoring deviations. Essentially, it provides a "low variance bound" for probabilistic contracts (with some deviations allowed). To generate contractual obligations with low variance, importance sampling is needed, specially for outlying values in the distributions. In languages such as Orc [MC07], these interfaces can be provided as prelude to orchestration descriptions. An example of a simple SLA declaration in Orc would be of the form:

```
def class ResponseTimeSLA()=
   val percentileVal = {. per95 = 80000, per75 = 60000, per50 = 45000 .}
   def prctile(h:t,k) = 10000*URandom()
   def ResponseTime([d1,d2,d3]) =
       prctile([d1,d2,d3], 95) <= percentileVal.per95 &&
       prctile([d1,d2,d3], 75) <= percentileVal.per75 &&
       prctile([d1,d2,d3], 50) <= percentileVal.per50
```

```
stop

signal >> ResponseTimeSLA().ResponseTime([1,2,3])
```

Here, the function `prctile` is similar to MATLAB with the operation performed on the list of monitored response time values `[d1,...dN]`. For orchestrations written in Orc, this can provide a suitable interface for QoS declarations before proceeding with the functional descriptions of the workflows.

8.6 Related Work

In [W3c03], a general overview of QoS aspects to consider in web services are discussed. This is formulated as contractual obligations in the WSLA framework [LKD+03]. Further work on precise contracts for web services is done in [SLE04]. The proposed SLAng protocol supports different tiers of the web services' architecture including CPU load, security and backup of data. The general purpose QML [FK98] is equipped with fine grained types, aspects, attributes, constraints and contractual specification at the QoS interface level.

The use of probabilistic QoS and contracts was introduced by Rosario et al. [RBHJ08] and Bistarelli et al. [BS09b]. Instead of using hard bound values for parameters such as response time, the authors proposed a probabilistic contract monitoring approach to model the QoS bounds. The composite service QoS was modeled using probabilistic processes by Hwang et al [HWTS07] where the authors combine orchestration constructs to derive global probability distributions.

In [GGMT08], Gallotti et al propose using a probabilistic model checker to assess non-functional quality attributes of workflows such as performance and reliability. Validating SLA conformance is studied by Boschi et al [BDZ06]. A series of experiments to evaluate different sampling techniques in an online environment is studied. Models such as SALSA [BHS+10] (Simulated Annealing Load Spreading Algorithm) use queuing theory to autonomously meet SLAs, without a priori over-dimensioning resources.

The use of importance sampling to change probability of occurrence of events in well known [Buc04]. An associated work in this area is importance splitting [MPLG10]. Importance splitting considers the estimation of a rare event by deploying several conditional probabilities during simulation runs, reducing the need to identify importance distributions as used in this case.

8.7 Conclusion

QoS aspects are critical to the functioning of most web service orchestrations and choreographies, needing more precise specifications of SLAs. This is difficult as distributions of QoS values have high variance when sampled with inefficient Monte-Carlo techniques. In most cases, the tails of QoS distributions are either neglected or averaged out in contractual specifications. Applying importance sampling to such distributions can provide better estimates of outlying values with relatively low variance. As demonstrated in this paper on the *Dell* supply chain application, importance sampling can have significant imperatives for both contract composition as well as forecasting deviations for critical services. The extension of this approach in case of WSLA specifications are also provided with a precise definition of sample variance.

Chapter 9

Negotiation Strategies for Probabilistic Contracts in Web Services Orchestrations

Ajay Kattepur, Albert Benveniste
IRISA/INRIA, Campus Universitaire de Beaulieu,
Rennes-Cedex, France.

Claude Jard
ENS Cachan, IRISA, Université Européenne
de Bretagne, Bruz, France.

Abstract

Service Level Agreements (SLAs) have been proposed in the context of web services to maintain acceptable quality of service (QoS) performance. This is specially crucial for composite service orchestrations that can invoke many atomic services to render functionality. A consequence of SLA management entails efficient negotiation protocols among orchestrations and invoked services. In composite services where data and QoS (modeled in a probabilistic setting) interact, it is difficult to select an individual atomic service to negotiate with in order to improve end-to-end QoS performance. A superior improvement in one negotiated domain (eg. latency) might mean deterioration in another domain (eg. cost); improvement in one of the invoked services may be annulled by another due to the control flow specified in the orchestration. In this paper, we propose a integer programming formulation based on first order stochastic dominance as a strategy for re-negotiation over multiple services. A consequence of this is better end-to-end performance of the orchestration compared to random selection of services for re-negotiation. We also demonstrate this optimal strategy can be applied to negotiation protocols specified in languages such as *Orc*. Such strategies are necessary for composite services where QoS contributions from individual atomic services vary significantly.

Web services continue to attract applications in many areas [ACKM04]. With increasing efforts to standardize performance of web services, focus has shifted to Quality of Service (QoS) levels. Maintaining efficient QoS levels of invoked services is a major prerogative of composite web service orchestrations, in order to maintain end-to-end QoS requirements. Contractual guarantees and service level agreements (SLAs) [BSC01] are critical to ensure adequate QoS performance of such composite services.

An important aspect of such service level agreements is negotiation among service providers [CCP07] [YKL+07]. The orchestration considers the end-to-end QoS against individual SLAs agreed with service providers. Negotiation ensures an acceptable level of QoS is maintained in composite service orchestrations, where the deterioration of individual services result in deteriorating overall performance. End-to-end performance is generally estimated through Monte-Carlo runs for composite services, having complex data and QoS interactions.

QoS metrics being random variables, the treatment of such contractual obligations tends toward probabilistic criterion [RBJ09b]. SLAs (used synonymously with contracts) may be specified as varying percentile values of such distributions rather than "hard" values. In [RBJ09b], composition and monitoring such contracts with stochastic dominance have been examined.

In this paper, we examine negotiation of such probabilistic contracts having assumptions and guarantees. If the assumptions on certain metrics (such as *throughput*) is maintained by an orchestration, the sub-contractors guarantee a certain level or performance, for example *latency*. Considering this setting, the negotiation involves improvement in guarantees of the sub contractors such that overall improvement in end-to-end QoS is observed.

The problem here is to select the necessary service to re-negotiate with. In case of large orchestrations having both returned *data* and *QoS* values interacting, improving one service might not necessarily improve the end-to-end QoS. By the term *improvement*, we refer to *first order dominance* [BD03], that has been used to compare probability distributions (in the sense, drawing from one distribution is more likely to produce lower values). In composite service orchestration where individual sites may be invoked using a number of constructs (parallel, in sequence, fork-join, using timeouts/halting), identifying a particular service that may improve end-to-end QoS is difficult.

In order to overcome this difficulty, we formulate the problem as an optimization over minimizing *cost* of re-negotiation with respect to improvements in *latency* distribution (in the first order dominance sense): this is referred to as a *optimization strategy*. Using the notion of monotonicity [BRBH08], an improvement in the QoS performance of an individual service contributes positively to the overall improvement in the QoS (though with varying data-dependent contributions). As we are dealing with distributions and uncertainties, the use of stochastic dominance constraints is necessary. Stochastic dominance [BD03] has been used extensively in econometrics and related areas to perform decisions based on uncertainties. We make use of linear relaxations of these stochastic constraints [NRR06] to formulate it as in integer programming problem. This provides a straightforward optimization problem that can be solved to obtain the most efficient re-negotiation strategy considering end-to-end QoS.

We evaluate our approach on a generic `GarageOnline` example that has both *data* and *QoS* values interacting. From our evaluation, we demonstrate that optimizing over constraints relating to stochastic dominance will produce better end-to-end contracts over multiple rounds of negotiation. This is compared against random selection of services (referred to as *random strategy*) for re-negotiation of composite services. As it is difficult to estimate the contribution of individual services to overall improvement, we believe such an optimization strategy is the best possible approach for transaction based orchestrations.

As specifying such negotiations is not possible in conventional languages like WSLA [LKD+03] (multiple rounds, percentile calculations), we specify both the functional specification and the negotiation language in Orc [KQCM09]. Making use of optimization specifications in [KBJ11], the integer programming formulation as a negotiation strategy can be specified in Orc. An advantage of this is that runtime deterioration can be monitored to enter re-negotiation directly with participating services.

The rest of the paper is organized as follows: Section 9.1 provides foundation material for our paper including QoS in web services in Section 9.1.1, probabilistic contracts in Section 9.1.2, contract negotiations in Section 9.1.3 and an overview of Orc in Section 9.1.4. The optimization formulation based on first order stochastic dominance constraints is presented in Section 9.2. In Section 9.3 we introduce the `GarageOnline` example. The problem of re-negotiating with individual services from a composite orchestration context is presented Section 9.4. The methodology used to overcome these problems are discussed in Section 9.4.3. Negotiation specifications as an extension of Orc is presented in 9.5. Discussion of results from the negotiation strategy is presented in Section 9.6. Related literature and conclusions are finally presented in Sections 9.7 and 9.8.

9.1 Foundations

This section provides a broad overview of topics relevant to our work.

9.1.1 Web services' QoS

Available literature on industry standards in QoS [TF06] provide a family of QoS metrics that are needed to specify SLAs. These can be subsumed into the following four general QoS observations [1]:

1. $\delta \in \mathbb{R}_+$ is the service latency. When represented as a distribution, this can subsume other metrics such as availability and reliability of the service.

2. $\$ \in \mathbb{R}_+$ is the per invocation service cost.

3. $\zeta \in \mathbb{D}_\zeta$ is the output data quality. This can represent other metrics such as data security level and non-repudiation of private data over a scale of values.

4. $\lambda \in \mathbb{R}_+$ is the inter-query interval, equivalent to considering the query rate for a service. Performance of the service will depend on negotiations with the amount of queries that can be made in a given time interval.

Along with QoS, the web service performs its task and returns some functional data $\rho \in \mathbb{D}_\rho$ as the output.

9.1.2 Probabilistic Contracts

For a domain $\mathbb{D}_Q$ of a QoS parameter Q, behavior can be represented by its distribution F_Q:

$$F_Q(x) = \mathbb{P}(Q \leq x) \tag{9.1}$$

Making use of stochastic ordering [BD03], this is refined for probability distributions F and G over a totally ordered domain $\mathbb{D}$:

$$G_Q \preceq F_Q \quad \Longleftrightarrow \quad \forall x \in \mathbb{D}_Q, \quad G_Q(x) \geq F_Q(x) \tag{9.2}$$

[1] Aspects such as scalability, interoperability and robustness are not dealt with as they are specific to the supplier side operation (not necessarily part of SLAs).

That is, there are more chances of being less than x (partial order $\leq$) if the random variable is drawn according to G than according to F.

Following the established approach of WSLA [LKD+03], a contract must specify the obligations of the two parties.

- The obligations that the orchestration has regarding the service are seen as *assumptions* by the service - the orchestration is supposed to meet them.
- The obligations that the service has regarding the orchestration are seen as *guarantees* by the service - the service commits to meeting them as long as assumptions are met.

Definition. 5 *A probabilistic contract is a pair (Assumptions, Guarantees), which both are lists of tuples* $(Q, \mathbb{D}_Q, F_Q)$*, where* Q *is a QoS parameter with QoS domain* $\mathbb{D}_Q$ *and corresponding distribution* F_Q*.*

Once contracts have been agreed, they must be monitored by the orchestration for possible violation as described in [RBJ09b]. Monitoring applies to each contracted distribution F individually, where F is the distribution associated to some QoS parameter Q having partially ordered domain $\mathbb{D}_Q$. By monitoring the considered service, the orchestration can get an estimate of the actual distribution of Q.

The problem is, for the orchestration, to decide whether or not G complies with F, where compliance is defined according to:

$$\sup_{x \in \mathbb{D}_Q} \{F_Q(x) - G_Q(x)\} \leq \epsilon \tag{9.3}$$

where ϵ is the level of deviation allowed from the contractual distribution.

Monotonicity - It is important to make note of monotonicity in orchestrations as specified in [BRBH08]. This implies that a superior performance of a particular service invoked in the orchestration contributes positively to the overall performance of the orchestration. Such an assumption is crucial in negotiation based framework where contracts are composed.

9.1.3 Contract Negotiation

Contract negotiations relates to the procedure of parties (clients and providers) agreeing on the terms of an SLA. The typical negotiation steps are the following:

1. The provider publishes a template describing the service and its possible terms, including the QoS and possible compensations in case of violation.
2. The client fetches the template, and fills it in with values which describe the planned resource usage.
3. This new document, which engages neither party, is sent to the provider. Receiving this, the provider, based on the current resource availability and customer policies, sends back to the client a quote. This quote corresponds to values on which the provider would probably agree (but this is by no means binding), based on the clients needs.
4. The client, if satisfied with the quote, applies his/her signature to the document, and sends it back to the provider as a SLA proposal.
5. The provider, receiving the proposal, is free to reject or accept it. In the latter case, the proposal becomes an SLA officially signed by both parties, and starts to be a valid legal document.

The quotes exchange (steps 2 and 3) can be repeated any number of times. The parties may freely modify the different terms: lower fees, lower QoS, longer time slots, fewer resource needs, lower compensations and so on. Once a contract has been signed and agreed, the necessity of changing it could be envisaged (re-negotiation).

9.1.4 Orc

Orc [KQCM09] serves as a simple yet powerful concurrent programming language to describe web services orchestrations. The fundamental declaration used in the Orc language is a *site*. The type of a *site* is itself treated like a service - it is passed the types of its arguments, and responds with a return type for those arguments. An Orc *expression* represents an execution and may call external services to publish some number of values (possibly zero).

Orc has the following combinators that are used on various examples as seen in [KQCM09]. The *Parallel* combinator $F \mid G$, where F and G are Orc expressions, runs by executing F and G concurrently; returns from F and G are interleaved. Whenever F or G communicates with a service or publishes a value, $F \mid G$ does so as well. The execution of the *Sequential* combinator $F >x> G$ starts by executing F. Values published by copies of G are published by the whole expression, but the values published by F are not published by the whole expression; they are consumed by the variable binding. If there is no response from either of the sites, the expression does not terminate. Sequential operators may also be written compactly as $F \gg G$. The *Pruning* combinator, written $F <x< G$, allows us to block a computation waiting for a result, or terminate a computation. The execution of $F <x< G$ starts by executing F and G in parallel. Whenever F publishes a value, that value is published by the entire execution. When G publishes its first value, that value is bound to x in F, with the execution of G immediately terminated. The *Otherwise* combinator, written F **;** G has the following execution. First, F is executed. If F completes, and has not published any values, then G executes. If F did publish one or more values, then G is ignored. The publications of F **;** G are those of F if F publishes, or those of G otherwise.

Declarations are used to bind values to be used in Orc expression. The declaration `val` $x = G$, followed by expression F, executes G, and binds its first publication to x, to be used in F. The declaration `def` $E(x) = F$ defines a function named E whose formal parameter list is x and body is expression F. A call $E(p)$ is evaluated by replacing the formal parameters x by the actual parameters p in the body F. Orc supports three types of data structures: tuples, such as (f, g), finite lists, such as $[f, g]$ and records, such as $\{.f, g.\}$. Further details on sites representing semaphores, channels and class declarations may be found in the Orc documentation [2].

9.2 Optimization Formulation

In this section, we formulate the re-negotiation strategy as an optimization problem. We start from a general stochastic optimization setting and move on to the web services' re-negotiation strategy. As stochastic optimization involve comparison of random variables, this can be suitably applied to distributions of QoS values. To reduce the search space and complexity of comparison, approximations to convert the problem to integer programming proposed by [NRR06] is used.

[2] http://orc.csres.utexas.edu/documentation.shtml

9.2.1 Stochastic Optimization

For formulating the problem with first order stochastic dominance, we define a triple $(\Omega, \mathbf{F}, \mathbb{P})$ as a probability space, where Ω is the entire space, $\mathbf{F}$ is a subset of this space where probability measure $\mathbb{P}$ is defined. For the space of all random variables $\mathbf{X}$ defined on the $(\Omega, \mathbf{F})$, the right continuous cumulative distribution function (CDF) is defined as $F_X(\eta) = \mathbb{P}(X \leq \eta), \quad \eta \in \mathbb{R}, \; X \in \mathbf{X}$. For variable $X, Y \; \in \mathbf{X}$, X dominates Y in the first order sense $X \succeq Y$, if $F_X(\eta) \leq F_Y(\eta), \forall \eta \in \mathbb{R}$. The stochastic optimization problem with first order dominance constraints may be written as:

$$\begin{array}{ll} \min & f(X) \\ \text{subject to:} & X \succeq Y, \quad X, Y \in \mathbf{X} \end{array} \tag{9.4}$$

Let $(\Omega, 2^{\Omega}, \mathbb{P})$ be a probability space with finite events $\Omega = (\omega_1, .., \omega_W)$ and corresponding probabilities $p_1, ..., p_W$. Consider a discrete random variable $Y \in \mathbf{X}$ with finite support and realizations y_i with probabilities q_i $(i = 1, 2, .., m)$. Assuming a right continuous step function F_Y with $y_1 < y_2 ... < y_m$, the first order stochastic dominance constrains in eq. (9.4) may be written as:

$$\mathbb{P}(X \leq y_i) \leq \mathbb{P}(Y \leq y_i), \quad i = 1, .., m \tag{9.5}$$

We have $\mathbb{P}(Y \leq y_i) = \sum_{k=1}^{i-1} q_k$ and the $\mathbb{P}(X \leq y_i) = \sum_{w=1}^{W} p_w z_{iw}; \quad i = 1, ..., m; \quad w = 1, ..., W$ such that binary decision variables:

$$z_{iw} = \begin{cases} 1 & \text{if } y_i - X(\omega_w) > 0 \\ 0 & \text{otherwise} \end{cases}$$

If we define a big number $M \in \mathbb{R}$ satisfying $M \geq \max_w \{y_m - X(\omega_w)\}$, the first order stochastic constraint problem may be formulated as a binary integer programming problem:

$$\begin{array}{ll} \min & f(X) \\ \text{Subject to:} & y_i - X(\omega_w) \leq M z_{iw} \\ & \displaystyle\sum_{w=1}^{W} p_w z_{iw} \leq \sum_{k=1}^{i-1} q_k \\ & z_{iw} \in \{0, 1\} \\ & X \in \mathbf{X} \\ & i = 1, ..., m; \quad w = 1, ..., W \end{array} \tag{9.6}$$

Such a formulation has been proposed in [NRR06] as a relaxation to first order dominance constraints.

9.2.2 Web Services' Negotiation

Assume there are $\mathcal{S}_1, ...\mathcal{S}_N$ services characterized by the triple $\mathcal{S}_j := (c_j, F'_j(\delta), F_j(\delta))$ representing cost of the service c_j along with reformulated F'_j and original latency F_j distribution. As we are dealing with re-negotiating a single service from this set, we also use the inverse proportionality of cost c_j and the mean/median of the reformulated distribution $\widehat{F}'_j$ (a bigger reformulated mean/median would mean higher cost). While we focus on the tradeoff between latency and cost, other metrics can be formulated

similarly using weighted or lexicographic ordering.

$$\begin{aligned} \min \quad & \sum_{j=1}^{N} s_j c_j \\ \text{Subject to:} \quad & F'_j(\delta) s_j \preceq F_j(\delta) s_j \\ & c_j = \frac{K_j}{\widehat{F}'_j} \\ & \sum_{j=1}^{N} s_j = 1, \quad s_j \in \{0,1\} \end{aligned} \tag{9.7}$$

Continuing with the binary integer formulation, we assign cumulative distributions F_j with realizations f_{wj} and corresponding probabilities $p_{wj}, w = 1, ..., W$ with index j referring to a particular service. Similarly, assign cumulative distributions F'_j with realizations f'_{ij} and corresponding probabilities $q_{ij}, i = 1, ..., m$. With $F'_j \approx \sum_{k=1}^{i-1} q_{kj}$ and the $F_j \approx \sum_{w=1}^{W} p_{wj} z_{iwj}$; $i = 1, ..., m$; $w = 1, ..., W$ and binary decision variables:

$$z_{iwj} = \begin{cases} 1 & \text{if } f'_{ij} - f_{wj} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Also define big numbers $M_j \in \mathbb{R}$ satisfying $M_j \geq \max_w \{f'_{ij} - f_{wj}\}$ (with f_{wj} characterized by p_{wj}), the first order stochastic constraint problem may be formulated as a binary integer programming problem. The optimization formulation in eq. (9.8) essentially selects the best service $\mathcal{S}_j$ to re-negotiate with in terms of cost and latency. It makes use of the selection procedure outlined in eq. (9.7) with the relaxation of first order dominance on latency as in eq. (9.6). In this formulation, we select a single service $\mathcal{S}_j$ (indexed by s_j) with the lowest corresponding negotiation cost c_j that provides at least first order dominance with respect to previous latency distributions.

$$\begin{aligned} \min \quad & \sum_{j=1}^{N} s_j c_j \\ \text{Subject to:} \quad & (f'_{ij} - f_{wj}) s_j \leq M_j z_{iwj} s_j \\ & s_j \sum_{w=1}^{W} p_{wj} z_{iwj} \leq s_j \sum_{k=1}^{i-1} q_{kj} \\ & c_j = K_j / \tfrac{1}{m} \sum_{i=1}^{m} f'_{ij} \\ & z_{iwj} \in \{0,1\}; \; f'_{ij}, f_{wj} \in \mathbf{X} \\ & \sum_{j=1}^{N} s_j = 1, \quad s_j \in \{0,1\} \\ & i = 1, ..., m; \; w = 1, ..., W; \; j = 1, ..., N \end{aligned} \tag{9.8}$$

Note that higher order stochastic dominance tests exist [BD03]; however, we limit our formulation to first order dominance for contract compliance. This is done so that pointwise comparison of QoS values can be made as equivalent to comparing distributions (see Theorem 5 in [RBJ09b]).

9.3 GarageOnline Example

We consider the `GarageOnline` demonstrative example presented informally in Fig. 9.1. It describes a web services orchestration to hire/order cars from garages with associated credit and insurance companies. When an order with a preference of *Gold/-Standard* insurance is placed, the fastest responding `Garage` is chosen. The services

```
def GarageOnline(Order,Preference) =
  val GarageList = ["Garage A", "Garage B"]
  val CreditList = ["Credit A", "Credit B"]
  val InsureList = ["Insure A", "Insure B"]
  val InsureGoldList = ["InsureGold"]
  def bestQ(comparer,publisher) =  head(sortBy(comparer,collect(publisher)))
  def comparePrice(x, y) = x.price? <= y.price?
  def compareTime(x, y) = x.time? <= y.time?
  def inquireTime(List) = each(List) >sup> Dictionary() >ProductDetails>
    ProductDetails.Company := sup >> ProductDetails.time :=
    (Rclock().time()) >> ProductDetails
  def inquirePrice(List) = each(List) >sup> Dictionary() >ProductDetails>
    ProductDetails.Company := sup >> ProductDetails.price := c
    >> ProductDetails

  def GenerateInvoce(Order,Preference) = Dictionary() >Invoice>
    Invoice.SubmitOrder := Invoice.ordernumber? >>
    Invoice.acceptedTime := Rclock().time() >> (Invoice,Preference)
  def Garage(Invoice) = bestQ(compareTime, defer(inquireTime,GarageList))
    >q> Invoice.GarageQuote := q
  def Credit(Invoice) = bestQ(comparePrice, defer(inquirePrice,CreditList))
    >q> Invoice.CreditQuote := q
  def Insure(Invoice,Preference) = if Preference = "Gold" then
    defer(inquirePrice,InsureGoldList) else
    bestQ(comparePrice, defer(inquirePrice,InsureList))
    >q> Invoice.InsureQuote := q

GenerateInvoce(Order,Preference) >(Invoice,Preference)> Garage(Invoice) >>
 (Credit(Invoice),Insure(Invoice,Preference)) >x> Invoice
```

Table 9.1: The `GarageOnline` Orc specification.

for `Credit` and `Insurance` are then chosen depending on the lowest price returned. Notice that if the preference is set to "Gold", the orchestration chooses the `InsureGold` service. The Orc specification of the `GarageOnline` orchestration is presented in Table 9.1. The `Dictionary()` site is used as a mutable map from field names to values which are obtained using the . access. Values held by references are obtained using `x?` and set using `x:=y`. Operations on lists proposed in Orc are used to efficiently deal with multiple sites offering similar functionalities. In this specification we make use of the `bestQ` site to select among multiple domains (pruning with respect to latency or other QoS domains). In this orchestration multiple functionalities (eg. returned *best* price) and QoS values (eg. *best* latency) interact. Due to such subtle interactions contributing to the end-to-end contractual guarantees, re-negotiation with a particular service may not necessarily improve overall performance considerably (as discussed in Section 9.4). Hence, an optimization formulation is required in case of such orchestrations to choose a feasible negotiation plan.

9.4 Composite Contract Re-Negotiation

This section starts from the runtime negotiation of sites making use of the competition operator proposed in [RBJ09a]. The difficulty in choosing an optimal strategy for re-negotiation for end-to-end contractual obligations is then analyzed. Finally, a methodology for re-negotiation is provided keeping in mind monotonic conditions.

Figure 9.1: The `GarageOnline` orchestration.

9.4.1 Runtime Negotiation

Making use of the QoS calculus proposed in [RBJ09a], the QoS domain is a tuple $\mathbb{Q} = (\mathbb{D}, \leq, \oplus, \triangleleft)$ defined as:

1. $(\mathbb{D}, \leq)$ defines a QoS domain along with associated partial order. For domains such as latency and cost, the partial ordering $\leq$ is preferred while for domains such as data quality, partial ordering$\geq$ is preferred.

2. $\oplus : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$ defines the increments in QoS values (which can be zero). Note that the operator $\oplus$ is monotonic with $\delta'_1 \leq \delta_1$ and $\delta'_2 \leq \delta_2$ implying $\delta'_1 \oplus \delta'_2 \leq \delta_1 \oplus \delta_2$.

3. $\triangleleft : \mathbb{D} \times \mathbb{D}* \rightarrow \mathbb{D}$ is the competition operator that may be applied as choosing the "best" choice in many ways: pareto optimal, lexicographic or weighted choice. In case of synchronization across domains, for example $(c_1, \delta_1) \triangleleft (c_2, \delta_2)$ when ordered lexicographically would mean if $c_1 \leq c_2$ then $(c_1, \max(\delta_1, \delta_2))$ else $(c_2, \max(\delta_1, \delta_2))$. Note that the competition operator is monotonic with $\delta'_1 \leq \delta_1$, $\delta'_2 \leq \delta_2$ implying $\delta'_1 \triangleleft \delta'_2 \leq \delta_1 \triangleleft \delta_2$.

The runtime specification of the "best" operator specified in Section 9.2 makes use of domains *(cost,latency)* for competing services for negotiation. The $\triangleleft$ operator is monotonic as demonstrated below. Consider two services $\mathcal{S}_1, \mathcal{S}_2$ with QoS increments to the orchestration δ_1, δ_2. They are both candidates for negotiation with the tuple of $(c_1, \delta'_1), (c_2, \delta'_2)$. The operator $\triangleleft$ works by:

$$\begin{aligned} \mathcal{S}_1 \triangleleft \mathcal{S}_2 = \quad & \min(s_1 c_1 + s_2 c_2) \\ & \text{subject to: } \delta'_1 \leq \delta_1, \delta'_2 \leq \delta_2, \\ & c_i = K_i/\delta'_i, \textstyle\sum_i s_i = 1, s_i \in \{0, 1\} \end{aligned} \tag{9.9}$$

As we have chosen the minimum from cost domain c with a partial order condition $\leq$ on latency, the operation is indeed monotonic. The chosen (re-negotiated) service will not deteriorate the latency $\delta' \leq \delta$, which in turn will not deteriorate the end-to-end QoS for a monotonic orchestration. Similar competition policies are shown to be monotonic using branching cells and unfolding of Petri nets in [RBJ09a].

9.4.2 End-to-end QoS

We examine the difficulty of re-negotiating with individual service when data and QoS interact in a composite service orchestration. A set of services called with Orc combinators sequential ($\gg$), fastest response (pruning $\ll_\delta$ where δ refers to latency) or "best" response (fork-join $\ll_q$ where q refers to other metrics such as price, returned data, security level etc.) can have significant differences in overall contribution to the contract. We summarize the effect of a change ϵ in contractual obligations for latency (can be seen as a shift in the median relative to the ordering). Assume here monotonic orchestrations with an improved contract contributing positively to overall behavior. The Orc combinators determine the effect on overall behavior:

- *Sequential* - The original contract would be $A \overset{\delta_A}{\gg} B$. If a reformulated contract decreases the value, it will be seen by the whole orchestration as: $A \overset{\delta_A - \widehat{\epsilon}_A}{\gg} B$, where $\widehat{\epsilon}$ is the contribution to the end-to-end QoS.

- *Fastest Response* - The expression $\ll_\delta (A|B)$ (also written as $\texttt{let}(A|B)$) refers to choosing the "best" service according to δ. The original contract would choose the fastest responding service $\min(\delta_A, \delta_B)$. Now we consider improving the contract such that we have to choose from $\min(\delta_A - \epsilon_A, \delta_B - \epsilon_B)$. However, the end-to-end QoS will decrease only by $\widehat{\epsilon}$ that is dependent on the "best" latency response.

- *Best value* - The expression $\ll_q (A, B)$ refers to choosing the "best" service according to q (as in fork-join $(A, B) >(x, y)> (x, y)$). The original contract would entail the worst responding service $\max(\delta_A, \delta_B)$. Now we consider improving the contract such that we have to choose from $\max(\delta_A - \epsilon_A, \delta_B - \epsilon_B)$. However, the end-to-end QoS will decrease only by $\widehat{\epsilon}$ that is dependent on the "best" value response.

- *Orchestration* - Emphasis must be placed here on the difficulty in selecting a particular service in case of a composite orchestration. For example, consider the Orc expression $((\texttt{let}(A \gg B \gg C) \mid X), Y)$ with the orchestration aware of the causal history of individual sites. Latency improvements ϵ in say site B (called sequentially) can improve the overall latency of $\texttt{let}(A \gg B \gg C)$ and still be nullified by the performance of site Y ($\widehat{\epsilon} \approx 0$). Thus, choosing a sequential or "best" cost/QoS service that can provide optimal improvement in the overall performance is difficult.

As discussed, the *level* of improvement may not lead to first order dominance over previously observed contracts. However, the end-to-end QoS improvement $\widehat{\epsilon}$ is indeed positive due to our assumption of monotonicity. Our methodology provides local improvements to contracts, which in turn improves end-to-end QoS (or does not, at least, deteriorate it).

9.4.3 Re-negotiation methodology

In order to produce an optimal strategy for re-negotiation in orchestrations, we present the following methodology:

1. Using subcontracts F_j proposed by individual services $\mathcal{S}_j$, generate an end-to-end contract for the orchestration. Note that some services have no sub-contracts and must be accepted as-is for performance. Denote this overall contract of the orchestration as F_{orch}.

2. If the end-to-end contract is acceptable, stop and accept all sub-contracts. Else, proceed to step 3.

3. Re-negotiate with one of the sub-contractors offering lowest costing improvements for an improved contract (in the first order sense). This choice is performed using Eq. 9.8. Once a new contract F'_a for service $\mathcal{S}_a, a \in j$ is selected, repeat step 1 and 2. Increment the number of *rounds* of re-negotiation.

In the rest of the paper, this methodology is also referred to as an *optimal strategy* for re-negotiation. Unlike [RBJ09a], we use only the guarantees for re-negotiation and ignore the assumptions of the orchestration (throughput of service calls). However, formulating the assumptions as a tradeoff can be done using a similar methodology.

As we are considering improvements in only the sub-contractors performance, we intend to study the effect on end-to-end QoS. As discussed in Section 9.4.2, combinators used in languages such as Orc/BPEL cannot be incorporated into the formulation without introducing some bias (sequentially invoked services given larger weights) to selecting a particular service - hence, it is ignored and the notion of monotonicity is used. For a monotonic orchestration, the re-negotiation formulation will always improve the overall contract F_{orch}. This follows from the objective function (c_j, F'_j, F_j) used. As the new service performs with a new contract: $F'_j \preceq F_j$ (partial ordering defines the dominance relation), the monotonic orchestration cannot deteriorate due to the re-negotiated contract. This might incidentally be due to the inverse relation between cost and latency, which seems a plausible model for service behavior (higher costs produce better service). Note that we make use of the property of first order stochastic dominance that allows ordinary comparison of QoS values for monotonic orchestrations.

9.5 Negotiation Specification

While WSLA [LKD+03] as been proposed for specifying SLA contracts, standard languages for negotiation are limited. We have introduced a percentile-based approach to improve WSLAs in [Kat11]. However, the WSLA language is not sufficient to deal with negotiation specifications and multiple rounds of proposals. Further, WSLA does not support percentile based handling or optimization among metrics, which limits its applicability to our methodology.

We believe languages such as Orc, that can provide access to external sites and inherently support recursion, to be a better alternative to specify multiple negotiation rounds. This can be extended to specify optimization in Section 9.2 making use of optimization sites proposed in [KBJ11]. Thus, the procedure can be directly applied to specifications of orchestrations, used to re-configure runtime aspects of the specification. Essentially, a `Negotiation` service specified in Orc can use historical runtime metrics to re-negotiate with services.

To fulfill the negotiation process described in Section 9.4.3, the `Negotiation` service has the following operations:

- The `getQoS` service, when passed the inputs `sitex` and `QoS` returns the QoS values. This may be a distribution (latency) or a constant value (security level) and are stored in a `Channel()` site. When a list of values from a distribution is obtained, it is converted into quantile values and associated probabilities (`l` $\rightarrow$ `(f, p)`).

- The `getOffer` site, when passed the inputs `sitex` and `QoS` returns a tuple of `(QoSnew, cost)`. This represents the increments provided by the individual sites (`sitex`) for the QoS domain queried. It must be noted that a `null` value may be returned by sites that do not wish to re-negotiate or that have an *as-is* acceptance policy.

- The **Optima** site [KBJ11] can be used to specify the integer programming formulation when presented with a set of distributions. It can return the optimal site for a new contract according to eq. (9.8). We use online optimization services such as *lp-solve* provided by [FG01] for performing the optimization procedure. Note that the `Cost` weight is kept as 1 in the optimization specification.

We provide this specification for a simple version of `GarageOnline` having only `GarageA` and `GarageB` as part of the negotiation protocol. This is specified in Orc in Table 9.2.

```
def Negotiation(GarageOnline) =
  type Latency = Number
  type Cost = Number
  def getQoS(site, QoSDom) =
    val chsite = Channel()
    chsite.put(QoS) >> stop; chsite.getAll()
  def getOffer(site, QoSDom) =
    val chsite = Channel()
    chsite.put(QoS) >> stop; (chsite.getAll(), Cost)
  def Optima(Objective, Weight, Constraint, Solver) = Output

getQoS(GarageA(),latency) >(l_GarageA)> (f_GarageA,p_GarageA)
 >> getQoS(GarageB(),latency) >(l_GarageB)> (f_GarageB,p_GarageB)
 >> getOffer(GarageA(),latency) >(l_GarageA,c_GarageA)>
   ((f'_GarageA,q_GarageA),c_GarageA)
 >> getOffer(GarageB(),latency) >(l_GarageB,c_GarageB)>
   ((f'_GarageB,q_GarageB),c_GarageA)
 >> Optima([c_GarageA,c_GarageB], 1, (f_GarageA - f'_GarageA,(:>),K1),
    (sum(q_GarageA) - sum(p_GarageA),(:>),K2),
    (f_GarageB - f'_GarageB,(:>),K1),
    (sum(q_GarageB) - sum(p_GarageB),(:>),K2), ''binary integer'')
```

Table 9.2: The `GarageOnline` Negotiation specification in Orc using FIFO channels (`Channel()`) and pattern matching.

Once this procedure is completed, a Monte-Carlo run of the overall orchestration `GarageOnline` may be performed to estimate the improvement in QoS. In case the improvement is not satisfactory, a renewed round of negotiation may be necessary. An advantage of this technique is that runtime evaluation of the orchestrations can trigger re-negotiation protocols to be entered. This is advantageous for systems where QoS deterioration may mean substantial loss of revenue.

9.6 Negotiation Results

For the `GarageOnline` example, it is difficult to estimate which service to re-negotiate with in case of probabilistic SLAs. We use the optimization strategy developed in Section 9.4.3 to re-negotiate with individual atomic services. The re-negotiation is done using Eq. 9.8 with 10,000 values generated for both distributions. The values of f_j and f'_j are set as the $0.1, 0.2, ...0.9$ quantile in each case with associate probability p values. Here, f_j refers to the quantiles from the original latency distribution and f'_j refers to the new distribution associated with a cost c_j for a site $\mathcal{S}_j$.

Fig. 9.2 compares the improvements with respect to latency (t-location distribution with varying medians) and Fig. 9.2(c) for cost (generated from a normal distribution). As noticed, random choices made at runtime for 10 rounds of negotiation produces neither superior latency nor lower costs. By a round, we refer to one negotiation policy accepted with a particular service. Hence, an efficient technique for selecting the service

to be negotiated with (Eq. 9.8) is needed. As demonstrated, the improvements in the end-end-end contract for both latency and cost produce significant improvements using an optimal strategy. The negotiation strategy required 2.53 minutes on average per round on MATLAB 2011a running on a 4GB RAM Windows 7 machine.

Figure 9.2: Latency improvements with 10 rounds of negotiation for `GarageOnline` (a) Using optimization strategy (b) Random strategy. The Fig. (c) shows Cost incurred with 10 rounds of negotiation.

Further, we demonstrate the improvements seen from individual services in Fig. 9.3 after 10 rounds of negotiation. Some services (eg. `GarageA`) might improve with every round while others (eg. `CreditB`) are not selected for negotiation owing to higher costs in the optimal strategy. This differs from those services that are selected randomly for re-negotiation, hence producing higher costs in Fig. 9.2(c). The experiments were conducted in MATLAB using the `bintprog` output.

As we see in Fig. 9.3, the services do not contribute equally to overall performance of the orchestration. An optimization strategy is thus imperative when a number of rounds of negotiations are taking place. The advantage of using such a strategy is monotonic improvement in end-to-end QoS, with re-negotiated sites guaranteeing latency improvements from a first order dominance point of view.

An example of the diagnosis of the optimization is presented in Fig. 9.4 with the value X representing the vector for service re-negotiation (1 represents the index of the selected service). Other commercial solvers such as CPLEX or LPSOLVE should work similarly when invoked as a web service.

9.7 Related Work

The use of probabilistic QoS and soft contracts was introduced by Rosario et al. [RBJ09a] and Bistarelli et al. [BS09b]. Instead of using fixed hard bound values for parameters such as latency, the authors proposed a soft contract monitoring approach to model the QoS measurement. The composite service QoS was modeled using probabilistic processes by Hwang et al. [HWSP04] where the authors combine orchestration constructs to derive global probability distributions.

Service level agreements (SLAs) have been specified in a number of papers using WSLA [LKD+03]. In [DDK+04], the framework needed for handling SLAs are described

Figure 9.3: Improvements in performance of individual services after 10 rounds of optimization strategy.

in detail. In [Kat11], we have presented methodology for introducing percentile based constraints to WSLA framework, that is essential for probabilistic models. This is in line with high-level and rich specification languages such as QML [FK98] that provide a variety of constructs for QoS management. Related studies of optimal QoS compositions make use of genetic programming in Canfora et al. [CPEV05a] and linear programming in Zeng et al. [ZBN+04]. These make use of optimization strategies for composition but do not deal with negotiations.

Probabilistic models for on the fly negotiation of service agreements are presented in [CCP07]. Multiple QoS dimensions are considered with tradeoffs included for negotiation between a single service provider and customer/orchestration. The automatic negotiation process in [YKL+07] aims at identifying the maximum quality level admissible with respect to the user budget. In [BS09b], negotiation is said to proceed using relaxed constraints with algebraic operators modeled in a semi-ring.

While these papers examine algorithms for negotiation with individual services, the problem of optimizing negotiation and end-to-end QoS has not been studied. This is an important problem to consider, specially in the case of data and QoS dependent orchestrations. We make use of relaxations in stochastic constraints proposed in [NRR06] to develop a generic strategy for re-negotiation. Such a strategy is critical when there are a number of services with different contributions to the end-to-end QoS.

The use of dynamic/stochastic programming [SDR09] in service selection has been proposed in [GNZ+06]. However, incorporating such stochastic constraints entails heavy computational costs that may not be necessary. Alternatives to using first order dominance constraints include lighter higher order constraints [LL01]. In this paper, we make use of linear approximations of stochastic constraints [NRR06] to improve computational efficiency. Using the optimization specifications provided in [KBJ11], a number of online solvers [FG01] may be invoked within Orc. This specification allows for runtime re-negotiation of contractual obligations with the optimal service (dependent on cost, latency) selected.

```
Command Window
>> [X,FVAL] = bintprog(f,A,b,Ae,be,[],optimset('Display','iter','Diagnostics','on'))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   Diagnostic Information

Number of variables: 6


 Number of 0-1 binary integer variables:     6
 Number of linear inequality constraints:    1
 Number of linear equality constraints:      1

Algorithm selected
   LP-based branch-and-bound


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 End diagnostic information

Explored   Obj of LP   Obj of best   Unexplored   Best lower    Relative gap
 nodes    relaxation  integer point     nodes    bound on obj  between bounds
*     1        3.519        3.519          0           3.519           0%
Optimization terminated.

X =

     0
     0
     1
     0
     0
     0


FVAL =

    3.5189

>> |
Start
```

Figure 9.4: An example of an optimization run in MATLAB.

9.8 Conclusions

In this paper, we study the negotiation of SLAs in composite service orchestrations. While available literature deals with one-to-one negotiation between customers and clients, little work has been done in optimizing negotiation to improve end-to-end performance of an orchestration. This procedure is difficult to perform when data and QoS interact, when varied control flow combinators are applied in orchestrations: thus leading to uncertainty in an individual service's contribution to overall performance. We demonstrate that using an integer programming formulation, constraints may be specified to select the service that provides the best re-negotiation strategy. This, in turn, would perform much better than random re-negotiation strategies. These procedures may be specified in languages such as Orc to aid in runtime re-negotiation. As a consequence, it provides an optimal strategy for re-negotiation for composite web services orchestrations.

Chapter 10

Implementation Overview

In this chapter, we present the implementation of some of the QoS management concepts over Orc. We first review the Orc implementation in Scala; then we provide an overview of the QoS transformation in Orc/OIL; the effect of monotonicity on various QoS management tasks are then discussed; finally, we provide a platform for QoS management with contribution from various chapters in this thesis.

10.1 Orc

Orc ver 2.0 has been changed considerably from its early implementation in [CjM05], with the new codebase in Scala [OSV08]. This has reduced the number of lines of code significantly compared to the previous Java implementation. As Scala provides better *type* inheritance procedures, it supports the dynamically typed syntax of Orc. While the type checker checks for allowed types of values, it is now possible to provide type information to function declarations. The updated compiler in Orc now provides support for such dynamic typing. This can lead to better compiler and parser error messages to aid in debugging.

A standard format for the Orc Intermediary Language (OIL) form has also been included `http://orc.csres.utexas.edu/oil.xsd`. This provides an abstract syntax tree form with the original program represented in the Orc calculus. In the OIL representation, variable names are eliminated using de Bruijn indices [Bru72]. In the lambda calculus form, while functions are nameless, variable names are still used. The de Bruijn indices specify the number of levels to traverse from a *reference depth* to reach the binding value.

The Orc 2.0 implementation allows a user to view this OIL representation that can be appended using XML rewriting tools. An OIL version of the Orc expression x <x< $((1 \gg 4) \mid (2 \gg 5) \mid 3)$ is shown below. The pruning operation is the parent of the tree with variable in the left hand side (of the pruning) and the expression $((1 \gg 4) \mid (2 \gg 5) \mid 3)$ evaluated on the right.

```
<oil>
   <left>
     <sequence>
       <left>
         <prune varname="x">
           <left>
             <sequence>
               <left><constant><integer>2</integer></constant></left>
               <right><variable varname="x"></variable></right>
             </sequence>
           </left>
           <right>
             <sequence>
               <left><constant><integer>1</integer></constant></left>
```

```
                <right><constant><integer>3</integer></constant></right>
              </sequence>
            </right>
          </prune>
        </left>
        <right>
          <constant><integer>4</integer></constant>
        </right>
      </sequence>
    </left>
    <right>
      <constant><integer>5</integer></constant>
    </right>
  </parallel>
</oil>
```

The standard library in Orc 2.0 has also been updated providing new data structures, timers and XML manipulation tools.

10.2 Weaving QoS algebraic rules

Figure 10.1: QoS enhanced Orc output.

As seen in chapter 3, we provide a model to return a tuple of (data,QoS value) during the value returned from each site. This is shown in Fig. 10.1, with the "weaving" performed over the input Orc program. Not that a central role is performed by the QoS algebraic rules $(\preceq, \oplus, \triangleleft, \vee)$ in generating the QoS composed values. Built in sites such as clock, channels and references may be used to calculate and store the QoS increments. The automatic generation of a QoS "weaved" site can be performed in two ways: using Orc declarations or through OIL transforms.

10.2.1 Updating OIL

As every Orc program generates an OIL version encoded in the Orc calculus, the XML code can be used for pattern matching places where the site can be called. For example, the goal expression of

$$\textbf{def } f(x) = x$$
$$f(1)$$

can be invoked in parallel with a `QoS.wrapRT` site that weaves in the QoS values with the rules `QoS.RTAlgebra`. Note that this modification may be done either externally or using the Scala language in which Orc has been coded.

```
<oil>
  <parallel>
   <left>
      <unclosedvars></unclosedvars>
      <declaredefs>
        <defs>
          <definition varname="f" arity="1" typearity="0">
            <body><variable index="0"></variable></body>
          </definition>
        </defs>
        <body>
          <call>
            <target><variable varname="f" index="0"></variable></target>
            <args><constant><integer>1</integer></constant></args>
          </call>
        </body>
      </declaredefs>
    </left>
   <right>
    <call>
      <target><constant><site>QoS.WrapRT</site></constant></target>
      <args><variable varname="f" index="2"></variable></args>
    </call>
   </right>
  </parallel>
</oil>
```

An example output of the goal expression with tuple of (functional output, QoS increment, Causality) is:

```
(1, [], Set(1, [], Set(signal, [], Set())))
```

10.2.2 *Q-Orc*: Updating Orc with QoS

These are the implementation steps to enhance functional Orc specifications with QoS:

1. *SLA / QoS Declaration*: A Library of pre-defined QoS classes that specify the types, domains, operations and units for various metrics. This makes use of the `class` construct in Orc that provides the capability to implement sites within Orc. This can include a variety of QoS domains (Latency, Cost, Security, Reliability, Throughput) that can be instantiated and re-used when required. Note that multiple units may be specified: eg. seconds and milliseconds for latency/throughput, cost in items/currencies. As the classes are declared within Orc, the classes/definitions may be updated when required. An example is shown for the domain of Latency below:

```
--Latency.inc

def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))
val curTime  = Rclock()
def LatencyIncrement(sitex)  = (sitex,curTime.time()) >(sitex,d)>
  (sitex,curTime.time()-d)

type Millisecond  = Number
def Millisecond()  = signal
type Second  = Number
def Second()  = signal

-- Types and Definitions for Response Time
```

```
type ResponseTime  = Number
def class ResponseTime(unit)  =
  def QoS(String)  :: Number
  def QoS(sitex) = signal >> LatencyIncrement(sitex)
   >(_,q)> (Ift(unit = Millisecond) >> q | Iff(unit = Millisecond) >> q/1000)

  def QoSOplus(Number,Number) :: Number
  def QoSOplus(rt1,rt2) = rt1+rt2

  def QoSCompare(Number,Number) :: Number
  def QoSCompare(rt1,rt2) = rt1 <= rt2

  def QoSCompete(Number,Number) :: Number
  def QoSCompete(rt1,rt2) = bestQoS(QoSCompare,[rt1,rt2])

  def QoSVee(Number,Number) :: Number
  def QoSVee(rt1,rt2) = max(rt1,rt2)
  stop
```

Once these classes are declared, the SLA type checker checks for Ambient/Non-ambient QoS types before passing the control flow. If the correct typing is not found, the SLA site blocks further evaluation. A non-ambient metric (eg. cost) has competition operator defined *trivially*. However, for ambient metrics (eg. latency), the competition operator must be specified explicitly when combined with non-ambient metrics. This would specify whether the lexicographic ordering implies a infemum/supremum for the ambient metrics.

```
-- SLA.inc

include "Latency.inc"
include "CPUutilization.inc"
include "InterQuery.inc"
include "Cost.inc"
include "Security.inc"
include "Reliability.inc"

def class NonAmbient(QoStype) =
  def QoSCompete(Number,Number) :: Number
  def QoSCompete(q1,q2) = head(sortBy((<:), [q1,q2]))
  signal

def class Ambient(QoStype,competition) =
  def QoSCompete(Number,Number) :: Number
  def QoSCompete(q1,q2) = head(sortBy(competition,  [q1,q2]))
  signal
```

2. *QoS Registry*: This registers services with relevant QoS classes, QoS metric units and specific handles for accessing them. The registry is defined using the `records` data structure that can match keys to a record pattern. By providing multiple QoS units, it is possible to re-use the same class of QoS metrics multiple times by the same set of sites. Note that handles are also specified – additional information that must be returned by the service in order to satisfy the QoS requirements. Instances of handles include cost increments (items/currency), latency increments (milliseconds/seconds) and security levels that need to be specified. As the orchestration requires these increments to generate the end-to-end QoS increment for each domain, the sites must imperatively provide these handles. A site can also be neutral (zero increment) to certain domains.

   An example is provided with three sites `f`, `g` and `h` specified with various QoS domains, units and handles. An additional `QoSMatch` site matches the site identifier with these values when invoked.

```
-- QoSRegistry.inc

val QoSRegistry =
 [
  {. name = "f", QoSDom = ResponseTime, QoSUnit = Millisecond,
     Handle = LatencyIncrement .},
  {. name = "f", QoSDom = Cost, QoSUnit = CurrencyDollars,
     Handle = CostValue .},

  {. name = "g", QoSDom = ResponseTime, QoSUnit = Second,
     Handle = LatencyIncrement .},
  {. name = "g", QoSDom = Cost, QoSUnit = CurrencyDollars,
     Handle = CostValue .},

  {. name = "h", QoSDom = ResponseTime, QoSUnit = Second,
     Handle = LatencyIncrement .},
  {. name = "h", QoSDom = InterQueryTime, QoSUnit = Second,
     Handle = []  .},
  {. name = "h", QoSDom = Cost, QoSUnit = CurrencyDollars,
     Handle = CostValue .},
  {. name = "h", QoSDom = SecurityLevel, QoSUnit = Level,
     Handle = SecurityValue .}
 ]

def QoSMatch(siteID) = each(QoSRegistry) >M> Ift(M.name = siteID)
  >> (M.QoSDom,M.QoSUnit,M.Handle)
```

3. *Validating Registry Entries*: As the registry contains many services with possibly conflicting QoS domains, accurate mapping of service and domains may be needed. This can be done either *permissively* (not specified domains produce zero increments) or *strictly* (restricting QoS domains according to the mapping).

   For a Orc expression **def** `f() = (g1(),...gN())` (service `f` invoking `gi`), an injective partial function is defined as $\text{QoS}(f) \mapsto \text{QoS}(g_i)$. Note that this definition allows for multiple instances of QoS classes to be defined for each of these services. A `QoSValidate` site is implemented that checks for strict conformance of QoS domains between the caller/callee sites.

```
def QoSValidate(callersiteID,caleesiteID) =
 (collect(defer(QoSMatch,callersiteID)),
 collect(defer(QoSMatch,caleesiteID)))
 >(A,B)> ( Ift(A.QoSDom = B.QoSDom) >> signal
 | Iff(A.QoSDom = B.QoSDom) >> Println("Registry Entries Missing")
 >> stop)
```

4. *QoS Weaving*: The QoS Weaver site weaves the values generated by the sites (with appropriate domains and handles) and generates the tuple of `Data, QoS`. Note that the check for the domains and handles are strict with computation stopped otherwise. For Ambient metrics (eg. `Latency`, `CPUutlization`) the competition operator must be specified. While the supremum for latency is treated as `max`, for CPU utilization, it is treated as `min` (implying inefficient usage).

```
-- QoSWeaver.inc

def QoSWeaver(site,(lookup,unit,handle)) =

   def ResponseTimeCheck(competition) =
    Ift(lookup = ResponseTime && handle = LatencyIncrement) >>
    (Ambient(ResponseTime,competition)
    >> (ResponseTime(unit).QoS(site)))
    ; stop
```

```
   def CPUutlizationCheck(competition) =
    Ift(lookup = CPUutilization && handle = CPUutilizationValue) >>
    (Ambient(CPUutilization,competition) >>
    (CPUutilization(unit).QoS(site)))
    ; stop

   def CostCheck() =
    Ift(lookup = Cost && handle = CostValue) >>
    (NonAmbient(Cost) >> Cost(unit).QoS(site,CostValue()))
    ; stop

   def SecurityCheck() =
    Ift(lookup= SecurityLevel && handle = SecurityValue) >>
    (NonAmbient(SecurityLevel) >> SecurityLevel(unit).QoS(
        site,SecurityValue()))
    ; stop

   def InterQueryCheck() =
    Ift(lookup= InterQueryTime) >>
    (NonAmbient(InterQueryTime) >> InterQueryTime(unit).QoS(site))
    ; stop

    def ReliabilityCheck() =
    Ift(lookup= Reliability && handle = ReliabilityValue) >>
    (NonAmbient(Reliability) >>
    Reliability(unit).QoS(site,ReliabilityValue()))
    ; stop

signal >> v<v<(ResponseTimeCheck(max)| CostCheck() | SecurityCheck()
 | InterQueryCheck()) |ReliabilityCheck() | CPUutlizationCheck(min)
```

The weaver also incorporates a `QoS` site that wraps a specific site with the QoS increment produced by it.

```
def QoS(site,identifier) =
   val Data = Ref()
   def QoSCollect(v) = collect(defer2(QoSWeaver,Data?,v))
site >d> Data:=d >> collect(defer(QoSMatch,identifier)) >v>
 (Data?, map(QoSCollect,v))
```

5. *Functional Declaration*: Once these steps are completed, the Orc expression may be written after including the necessary sites (QoS Declarations, Registries). The site declarations must specify the site identifiers that would be invoked from the registry. An example of a site `composite` is also shown that invokes two of these sites after performing the `QoSValidate` step, to check for correct QoS Domain declaration.

```
-- Site Definitions

-- def f(x) = x
def class f(x) =
  def function() = x
  def QoSID() = "f"
  stop

-- def g(y) = y*y
def class g(y) =
  def function() = y*y
        def QoSID() = "g"
  stop

-- def h(k) = k + k
```

```
def class h(k) =
  def function() = k+k
        def QoSID() = "h"
  stop

-- def composite(m) = f(m) + g(m)
def composite(m) =
  def function() = QoSValidate("f","g") >> ((d1+d2,append(q1,q2))
  <((d1,q1),(d2,q2))< (f(m),g(m)))
  def QoSID() = "composite"
  stop
```

Once the sites have been declared, the goal expression may be written with the relevant classes included. The QoS weaving automatically equips relevant sites with their QoS increments. Note that the functional declarations can make use of the QoS outputs as well: eg. `append` combines two lists of QoS increments.

```
-- functional.orc

include "SLA.inc"
include "QoSRegistry.inc"
include "QoSWeaver.inc"
include "SiteDefs.inc"

def QoSsite(sitex) = QoS(sitex.function(),sitex.QoSID())

-- Example function that is weaved
1 >> 2 >> ((m,n) <(m,n)< QoSsite(f(7)) | QoSsite(f(8)))
|
QoSsite(g(12)) >(_,q1)> QoSsite(g(13)) >(v,q2)> (v,append(q1,q2))
|
QoSsite(h(25))
|
QoSsite(composite(3))
```

The output of the above functional declaration is provided below. Every publication is followed by a list of associated QoS increments. This may be used either to make control flow decisions within the orchestrations or aggregated for SLA monitoring.

```
-- Data and QoS weaved outputs

(7, [[1], [[6, 4, 9]]])
(50, [[0], [333], ["High"], [[5, 1, 7]]])
(169, [[1], [[7, 9, 8]], [2], [[9, 3, 2]]])
(12, [[6], [[3, 6, 6]], [0], [[4, 4, 5]]])
```

Complete code of this implementation with a larger example may be seen in Appendix 11.6. It can also be checked out as a read-only copy from [1].

10.2.3 Monotonicity

The effect of monotonicity in orchestrations has been well discussed in Chapter 3. In Fig. 10.1, monotonicity plays a crucial role in the design of the orchestration. For data-independent orchestrations, this conditions are met by default. In case of data-dependent monotonic orchestrations, locally optimal selection is enough to ensure global optimality. In order to ensure monotonicity in data dependent workflows, a sufficient condition is that, each time branching has occurred in net, a join occurs right after.

When this condition cannot be met, techniques specified in Section 3.4.5 may be used for enforcing monotonicity. One of the techniques is to aggregate two successive

[1] http://qorc.googlecode.com/svn/trunk/

transitions in each contributing branch (in the non-monotonic orchestration) and regard the result as a single transition. An alternative is to change the QoS evaluation procedure by isolating the subnet that leads to monotonicity. Pessimistic bounds on QoS are taken for this subnet to ensure monotonicity. Though these techniques provide different QoS composition outputs, both enforce that monotonicity conditions are met.

10.3 Platform for QoS Management

Once a QoS enhanced output of an orchestration is obtained, management of contractual obligations can be done as outlined in some of the chapters of this thesis. Fig. 10.2 demonstrates some of these tasks and are elaborated.

10.3.1 Management Tasks

1. *Monotonic Orchestration Design*: The functional specification of the orchestration must be done in languages such as Orc keeping in mind monotonicity as specified in Section 10.2.3. This assumption is critical for contract based management, with an improvement in one of the subcontracted services not deteriorating end-to-end contracts.

2. *Optimal Late Binding*: Collected data on the behavior of invoked services can lead to interesting QoS management issues. One of these is QoS dependent decision making in chapter 7. Setting up optimization problems that are dependent on the performance of some of the invoked services can lead to better end-to-end outputs. Properties such as load balancing such that assumptions of the orchestration are not violated can also be done. Optimization also improves the role performed by the "best" operator $\triangleleft$: instead of conventional enumeration and evaluation, the operation can be converted to optimizing over certain QoS domains with constraints.

3. *Contract Composition*: The *end-to-end SLA* contractual distribution may be obtained using either conventional Monte-Carlo or importance sampling techniques. Alternatives using analytic techniques (with known probabilities of choice) or Markov Chains may also be used. The contractual obligations will follow the assumption-guarantee procedure outlined in chapter 8. This would generate two distributions, the obligations of the orchestration with respect to invoked services and the guarantees in QoS performance provided by those services.

4. *Negotiation*: The end-to-end SLA also leads to negotiation of contractual obligations as seen in chapter 9. Rather than modifying the design of the orchestration, negotiating with one of the sub-contractors for improved QoS performance may be performed. This can involve choosing one of the sub-contracted services to produce a significant change (in the first order stochastic dominance sense) in the end-to-end contractual obligations.

5. *Monitoring*: The end-to-end SLA contract may then be monitored for deviations in real-time. This can be done through soft contractual techniques by monitoring historical data / forecasting (chapter 8). The notion of monotonicity introduced in chapter 3 is crucial here, as an improved performance by an invoked service will improve and not deteriorate the end-to-end performance.

Other aspects such as reconfigurations dependent on QoS outputs are not examined in our work.

10.3.2 Experimentation

As many of these procedures involve the use of statistical toolboxes, MATLAB or R mathematical environments are used. The QoS (eg. latency) distributions may be used to produce distributions (Monte-Carlo/Sampling) with an example of distribution fitting shown in Fig. 10.3. The link between statistical libraries such as those in MATLAB/R requires linking MATLAB with Orc. While web services can be directly invoked in MATLAB using wrapper functions (such as `createClassFromWsdl`), it may be advantageous to capture streams of data published by an orchestration. Though there are many ways of doing this (eg. write to file, client-server interaction), the *MatClipse* plugin (`http://code.google.com/a/eclipselabs.org/p/matclipse/`) is one of the easiest options. Like Orc, this is a plugin that allows invoking MATLAB within Eclipse.

For QoS Management, we focus on the following tasks:

1. *Contract Composition and Negotiation* - Once the QoS weaved outputs have been provided, the end-to-end QoS may be computed. Repeated calls to the orchestration provides a stream of values that may be generated as probabilistic contracts within MATLAB. The obtained QoS values may be fit to particular classes of distributions to generate these contracts. Sub-contracts that are not satisfactory may be re-negotiated with to result in a reformulated end-to-end contract.

2. *Monitoring* - As monitoring involves real-time data to be evaluated, the QoS is read through Orc. For generating distributions and comparing first order dominance, MATLAB may be used, either through the plugin or client/server option. Any reconfiguration necessary is performed within Orc, depending on the contractual deviations reported.

The code involved in the implementation of some of these techniques has been provided in Appendix 11. However, extensions to Orc for optimization (chapter 7) and negotiation (chapter 9) may be used to specify these techniques.

Figure 10.2: Platform for contract based management of monotonic orchestrations.

Figure 10.3: Distribution fitting in MATLAB.

Figure 10.4: Web service invocation via SOAP.

In order to have a realistic notion of QoS, it may be necessary to deploy services on a network and perform some measurements. For this, we make use of the NetBeans IDE that allows to deploy web service on the GlassFish 3.1 server. An example web service that returns a random cost with the SOAP invocation/response requests is shown in Fig. 10.4. Such web services may be invoked over a LAN/WAN network to provide realistic end-to-end QoS: taking into account network delays, routing and service processing times.

Chapter 11

Appendices

This section collects some additional supplementary material for various chapters presented previously.

11.1 Proofs from Chapter 3

11.1.1 Proof of 2

Throughout the proof, we fix an arbitrary value ω for the daemon. We first prove the sufficiency of condition (3.14). Let $\mathcal{N}'$ be such that $\mathcal{N}' \geq \mathcal{N}$. Since operators $\oplus$ and $\lhd$ are both monotonic, see 3.1, we have, by 2 and formulas (3.10) and (3.11):

$$E_\omega(\kappa(\mathcal{N}', \omega), \mathcal{N}') \geq E_\omega(\kappa(\mathcal{N}', \omega), \mathcal{N})$$

By (3.14) applied with $\kappa = \kappa(\mathcal{N}', \omega)$, we get that

$$E_\omega(\kappa(\mathcal{N}', \omega), \mathcal{N}) \geq E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N})$$

holds. This proves the sufficiency of condition (3.14). We prove necessity by contradiction. Let $(\mathcal{N}, \omega, \kappa^\dagger)$ be a triple violating condition (3.14), in that

$\kappa^\dagger$ cannot occur, but
$E_\omega(\kappa^\dagger, \mathcal{N}) \geq E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N})$ does not hold.

Now consider the $\mathcal{N}$ net $\mathcal{N}' = (N, D, Q', Q_{\text{init}})$ where the family Q' is such that, $\forall t \in \kappa^\dagger$, $\xi'_t(\omega) = \xi_t(\omega)$ holds, and $\forall t \notin \kappa^\dagger$, using conditions 1 and 3 for operator $\oplus$ in 3.1 together with the assumption that $(\mathbb{D}, \leq)$ is an upper lattice, we can inductively select $\xi'_t(\omega)$ such that the following two inequalities hold:

$$\bigvee_{t \in \kappa^\dagger} q_t \leq (\bigvee_{p' \in {}^\bullet t} q_{p'}) \oplus \xi'_t(\omega) \tag{11.1}$$

$$\xi_t(\omega) \leq \xi'_t(\omega) \tag{11.2}$$

Condition (11.2) expresses that $\mathcal{N}' \geq \mathcal{N}$. By 1 defining QoS composition policy, (11.1) implies that configuration $\kappa^\dagger$ can win all competitions arising in step 3 of QoS composition policy, $\kappa(\mathcal{N}', \omega) = \kappa^\dagger$ holds, and thus

$$E_\omega(\kappa(\mathcal{N}', \omega), \mathcal{N}') = E_\omega(\kappa^\dagger, \mathcal{N}') = E_\omega(\kappa^\dagger, \mathcal{N})$$

However, $E_\omega(\kappa^\dagger, \mathcal{N}) \geq E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N})$ does not hold, which violates monotonicity.

11.1.2 Proof of 3, Sufficiency

Let φ_N be the net morphism mapping U_N onto N and let $\mathcal{N}$ be any OrchNet built on U_N. We prove that condition (3.14) of 2 holds for $\mathcal{N}$ by induction on the number of transitions in the maximal configuration $\kappa(\mathcal{N},\omega)$ that actually occurs. The base case is when it has only one transition. Clearly this transition has minimal QoS increment and any other maximal configuration has a greater end-to-end QoS value.

Induction Hypothesis Condition (3.14) of 2 holds for any maximal occurring configuration with $m-1$ transitions ($m>1$). Formally, for an OrchNet $\mathcal{N}, \forall\omega\in\Omega, \forall\kappa\in\mathcal{V}(N)$,

$$E_\omega(\kappa,\mathcal{N}) \geq E_\omega(\kappa(\mathcal{N},\omega),\mathcal{N}) \tag{11.3}$$

must hold if $|\{t\in\kappa(\mathcal{N},\omega)\}| \leq m-1$.

Induction Argument Consider the OrchNet $\mathcal{N}$, where the actually occurring configuration $\kappa(\mathcal{N},\omega)$ has m transitions and let

$$\emptyset = \kappa_0(\omega) \prec \kappa_1(\omega)(=\kappa) \prec \cdots \prec \kappa_{M(\omega)}(\omega) = \kappa(\mathcal{N},\omega)$$

be the increasing chain of configurations leading to $\kappa(\mathcal{N},\omega)$ under QoS composition policy, see (1) — to shorten the notations, we write simply κ instead of $\kappa_1(\omega)$ in the sequel of the proof. We assume that $M(\omega)\leq m$. Let t be the unique transition such that $t\in\kappa_1(\omega)$ and set $\widehat{t}=\{t\}\cup t^\bullet$. Let κ' be any other maximal configuration of $\mathcal{N}$. Then two cases can occur.

- $t\in\kappa'$: In this case, comparing the end-to-end QoS of $\kappa(\mathcal{N},\omega)$ and κ' reduces to comparing
$$E_\omega(\kappa(\mathcal{N},\omega)\setminus\widehat{t},\mathcal{N}^\kappa) \text{ and } E_\omega(\kappa'\setminus\widehat{t},\mathcal{N}^\kappa)$$
where $\mathcal{N}^\kappa$ is the *future* of κ in $\mathcal{N}=(N,V,A,Q,Q_{\text{init}})$, obtained by replacing N by N^κ, restricting V, A, and Q to N^κ, and replacing Q_{init} by $E_\omega(\kappa,\mathcal{N})$, the QoS cost of executing configuration κ.

  Since $\kappa(\mathcal{N},\omega)\setminus\widehat{t}$ is the actually occurring configuration in the future $\mathcal{N}^\kappa$ of transition t, using our induction hypothesis, then
$$E_\omega(\kappa'\setminus\widehat{t},\mathcal{N}^\kappa) \geq E_\omega(\kappa(\mathcal{N},\omega)\setminus\widehat{t},\mathcal{N}^\kappa)$$
holds, which implies
$$E_\omega(\kappa',\mathcal{N}) \;\geq\; E_\omega(\kappa(\mathcal{N},\omega),\mathcal{N})$$

- $t\notin\kappa'$: Then there must exist a transition $t'\in\kappa'$ such that t and t' belong to the same branching cell B. Hence, $\varphi_N(t)^\bullet=\varphi_N(t')^\bullet$ follows from the structural condition of 3. The futures $\mathcal{N}^\kappa$ and $\mathcal{N}^{\kappa'}$ thus are isomorphic: they only differ in the initial colors of their places. If Q_{init} and Q'_{init} are the initial QoS values for the futures $\mathcal{N}^\kappa$ and $\mathcal{N}^{\kappa'}$, then $Q_{\text{init}}\leq Q'_{\text{init}}$ holds (since $\xi_t\leq\xi_{t'}$, $t^\bullet$ has QoS lesser than $t'^\bullet$ by monotonicity of $\oplus$). On the other hand,
$$E_\omega(\kappa(\mathcal{N},\omega),\mathcal{N}) \;=\; E_\omega(\kappa(\mathcal{N},\omega)\setminus\widehat{t},\mathcal{N}^\kappa) \tag{11.4}$$
and
$$E_\omega(\kappa',\mathcal{N}) \;= E_\omega(\kappa'\setminus\widehat{t'},\mathcal{N}^{\kappa'})$$

Now, since $\mathcal{N}^{\kappa'}$ and $\mathcal{N}^{\kappa}$ possess identical underlying nets and $\mathcal{N}^{\kappa'} \geq \mathcal{N}^{\kappa}$, then we get

$$E_\omega(\kappa' \setminus \widehat{t'}, \mathcal{N}^{\kappa'}) \quad \geq \quad E_\omega(\kappa' \setminus \widehat{t'}, \mathcal{N}^{\kappa}) \tag{11.5}$$

Finally, applying the induction hypothesis to (11.4) and using (11.5) yields $E_\omega(\kappa', \mathcal{N}) \geq E_\omega(\kappa(\mathcal{N}, \omega), \mathcal{N})$.

This proves that condition (3.14) of 2 holds and finishes the proof of the theorem.

11.1.3 Proof of 3, Necessity

We will show that when the structural condition of 3 is not satisfied by N, Orchnet $\mathcal{N}_N$ can violate condition (3.14) of 2, the necessary condition for monotonicity.

Let B be any branching cell in U_N that violates the structural condition of 3. Since N is sound, all transitions in B are reachable from the initial place and so there are transitions $t_1, t_2 \in B$ such that ${}^\bullet t_1 \cap {}^{\bullet}_{-} t_2 \neq \emptyset$, ${}^\bullet\varphi(t_1) \cap {}^{\bullet}_{-}\varphi(t_2) \neq \emptyset$ and $\varphi(t_1)^\bullet \neq \varphi(t_2)^\bullet$.

Define $[t] = \lceil t \rceil \setminus \widehat{t}$ and $\kappa = [t_1] \cup [t_2]$. κ is a configuration. Since $t_1^\bullet \neq t_2^\bullet$, without loss of generality, we assume that there is a place $p \in t_1^\bullet$ such that $p \notin t_2^\bullet$. Let t^* be a transition in $\mathcal{N}^\kappa$ such that $t^* \in p^\bullet$. Such a transition must exist since p can not be a maximal place: $\varphi(p)$ can not be a maximal place in N which has a unique maximal place. Now, consider the Orchnet $\mathcal{N}' > \mathcal{N}$ obtained as follows: using repeatedly condition 3 for operator $\oplus$ in 3.1, $\xi'_{t_1}(\omega) = \xi_{t_1}(\omega), \xi'_{t_2}(\omega) \geq \xi_{t_1}(\omega)$, and, for all other $t \in B, \xi'_t(\omega) \geq \xi'_{t_2}(\omega)$. For all remaining transitions of $\mathcal{N}'$, with the exception of t^*, the QoS increments are the same as that in $\mathcal{N}$ and thus are finite for ω. Finally, select $\xi'_{t^*}(\omega)$ such that

$$\xi_{t_1}(\omega) \oplus \xi'_{t^*}(\omega) > \mathbf{Q}^*(\omega) \tag{11.6}$$

where $\mathbf{Q}^*(\omega) \in \mathbb{D}$ will be chosen later—here we used the additional condition of 3 regarding $\mathbb{D}$, together with condition 3 for operator $\oplus$ in 3.1. Transition t_1 has a minimal QoS increment among all transitions in the branching cell B. It can therefore win the competition, thus giving raise to an actually occurring configuration $\kappa(\mathcal{N}', \omega)$. Select $\mathbf{Q}^*(\omega)$ equal to the maximal value of the end-to-end QoS of the set K of all maximal configurations κ that do not include t_1 (e.g., when t_2 fires instead of t_1). By (11.6), since t^* is in the future of t_1, we thus have $E_\omega(\kappa(\mathcal{N}', \omega), \mathcal{N}') \geq \xi_{t_1}(\omega) \oplus \xi'_{t^*}(\omega) > \mathbf{Q}^*(\omega) \geq E_\omega(\kappa, \mathcal{N}')$ for any configuration κ and, therefore, $\mathcal{N}'$ violates the condition (3.14) of 2.

11.2 Pairwise Products in Chapters 5 and 6

The Pairwise configurations from one *sample* for the Crisis Management System (CMS) is presented in Table 11.2. The configurations are generated using the constraint satisfaction technique presented in [PSK+10].

11.3 Dell Example in Chapters 7 and 8

11.3.1 Orc code

Following is the complete Orc code for the Dell choreography.

```
-- -- -- --
-- Orc additional operators for QoS
-- -- -- --

-- bestQoS simulates the QoS-based prune operator -- both arguments are lambdas
```

```
product name: Configuration 0_0 Features: CrisisManagementSystem LongDistanceCall
GPS InternalResource FirstAidMaterial HumanResource SystemAdmin Worker Coordinator
FAWorker Observer RemoveObstacle Rescue Mission NurseTheWounded SortTheWounded Area
Small ExternalServicesUsed GovernmentalServices Police Policeman AuthenticationSystem
CrisisType SuddenCrisis MajorAccident CarCrash ExternalCompany GarageTowTruck
MedicalServices Fireman PublicHospital HospitalWorker Ambulance FirstAidWorker
ITOption

product name: Configuration 1_0 Features: CrisisManagementSystem LongDistanceCall
GPS InternalResource FirstAidMaterial HumanResource SystemAdmin Worker Coordinator
FAWorker Observer Transport Mission NurseTheWounded SortTheWounded Area Small
ExternalServicesUsed GovernmentalServices Policeman Witness CrisisType SuddenCrisis
MajorAccident CarCrash MedicalServices PublicHospital Ambulance FirstAidWorker

product name: Configuration 2_0 Features: CrisisManagementSystem LongDistanceCall
GPS InternalResource FirstAidMaterial HumanResource SystemAdmin Worker Coordinator
FAWorker Observer RemoveObstacle Rescue Mission Area Small ExternalServicesUsed
GovernmentalServices CrisisType SuddenCrisis MajorAccident CarCrash MedicalServices
PublicHospital HospitalWorker FirstAidWorker

product name: Configuration 3_0 Features: CrisisManagementSystem LongDistanceCall
GSM GPS InternalResource FirstAidMaterial HumanResource SystemAdmin Worker
Coordinator FAWorker Rescue Observe Transport Mission Area Small ExternalServicesUsed
GovernmentalServices FireDepartment Policeman HumanVictims CrisisType SuddenCrisis
MajorAccident CarCrash ExternalCompany Fireman ITOption

product name: Configuration 4_0 Features: CrisisManagementSystem LongDistanceCall
InternalResource FirstAidMaterial HumanResource SystemAdmin Worker Coordinator
FAWorker RemoveObstacle Transport Mission Investigation NurseTheWounded Area
Small ExternalServicesUsed GovernmentalServices Policeman DatabaseSystem
AuthenticationSystem CrisisType SuddenCrisis MajorAccident CarCrash ExternalCompany
GarageTowTruck MedicalServices PrivateAmbulanceCompany PublicHospital HospitalWorker
Ambulance FirstAidWorker ITOption
```

Table 11.1: Configurations from a single sample for the CMS.

```
def bestQoS(publisher, comparer) = head(sortBy(comparer, publisher))

-- -- -- --
-- Simulation Utilities
-- -- -- --

val SimElaspedTime = Rclock().time()
def randomElement(xs) = index(xs, Random(length(xs)))

-- -- -- --
-- Suppliers and Stock;
-- StockLevels assigns a counter to each hardware supplier
-- QueryLevels assigns the query rate to each harware supplier
-- Stock Price assigns a higher cost to AMD (faster polling, smaller refuel quantities)
--   compared to NVIDIA
-- -- -- --

val HwSupplierList = ["AMD", "NVIDIA"]
val SwSupplierList = ["Symantec", "McAfee"]


import class HashMap = "java.util.HashMap"
val StockLevels = HashMap() >hm> (each(HwSupplierList) >v> hm.put(v, Counter()) >> stop ; hm)
val QueryLevels = HashMap() >hm> (each(HwSupplierList) >v> hm.put(v, Counter()) >> stop ; hm)
val StockPrice = HashMap() >hm> (head(HwSupplierList) >v> hm.put(v, 1500) >> last(HwSupplierList)
 >v> hm.put(v, 1000) >> stop ; hm)

-- -- -- --
-- SLA declaration at choreography level
-- -- -- --

-- Increases the query count for a supply revolver
type QoSQueryRate = Number
def QoSQueryRate(Supplier) = QueryLevels.get(Supplier).value()
```

```
def QoSQueryRateOplus(Supplier) = QueryLevels.get(Supplier).inc()

-- Compares cost of products
type QoSCost = Number
def QoSCostCompare(QoSCost, QoSCost) :: QoSCost
def QoSCostCompare(q1, q2) = q1.price? <: q2.price?

-- Compares availability of products
type QoSAvail = Boolean
def QoSAvailCompare(QoSAvail, QoSAvail) :: QoSAvail
def QoSAvailCompare(q1, q2) = if ~q1.available? then false else if ~q2.available? then true
 else QoSCostCompare(q1, q2)

-- Stock Level operations (increment and decrement)
type QoSStock = Number
def QoSStock(Supplier) = StockLevels.get(Supplier).value()
def QoSStockIncrement(Supplier) = StockLevels.get(Supplier).inc()
def QoSStockDecrement(Supplier) = StockLevels.get(Supplier).dec()

-- declare which QoS parameter has priority
-- Calls one of the above QoS comparers according to priority
def QoSPriority(x, y) =  randomElement(["Cost", "Availability"]) >priority>
 (if priority = "Cost" then QoSCostCompare(x,y) else QoSAvailCompare(x, y))


-- -- -- --
-- Dell Orchestration
-- -- -- --

-- external site, returns random result
def ValidateOrder(SalesOrder) = if URandom() <: 0.95 then signal else stop
def CheckCustomerCredit(SalesOrder) = if URandom() <: 0.95 then signal else stop

def ValidateSalesOrder(SalesOrder) = (ValidateOrder(SalesOrder), CheckCustomerCredit(SalesOrder))
def NotifyFail(SalesOrder, reason) = Println("Order "+SalesOrder.id?+" failed: "+reason) >> stop

def GenerateProductionOrder(SalesOrder) = Dictionary() >ProductionOrder>
 ProductionOrder.SalesOrder := SalesOrder >> ProductionOrder.AcceptedTime := SalesOrder.WaitTime?
 >> ProductionOrder

def InquireAvail(ProductionOrder, SupplierList) = each(SupplierList) >supp> Dictionary()
 >ProductDetails> ProductDetails.Supplier := supp >>
 ProductDetails.price := StockPrice.get(supp) >>
 ProductDetails.available := (QoSStock(supp) >= 1) >> ProductDetails

def InquirePrice(ProductionOrder, SupplierList) = each(SupplierList)
 >supp> Dictionary() >ProductDetails> ProductDetails.Supplier := supp
 >> ProductDetails.price := Random(100) >> ProductDetails

def OrderHw(ProductionOrder, supp) = QoSStockDecrement(supp) >> Rwait(Random(1000)) >>
 ProductionOrder.HwSupplier := supp

def OrderSw(ProductionOrder, supp) = Rclock().wait(Random(1000))
 >> ProductionOrder.SwSupplier := supp

-- Note race on availability between "inquire" and "order" steps -- possible timeout

def ProcureHardwareComponents(ProductionOrder) =
 bestQoS(collect(defer2(InquireAvail, ProductionOrder,
 HwSupplierList)),  ProductionOrder.SalesOrder?.QoSCompare?) >ProductDetails>
 ProductionOrder.HwQuote := ProductDetails
 >> OrderHw(ProductionOrder, ProductDetails.Supplier?)

def ProcureSoftwareComponents(ProductionOrder) =
 bestQoS(collect(defer2(InquirePrice, ProductionOrder, SwSupplierList)),
 QoSCostCompare) >ProductDetails> ProductionOrder.SwQuote := ProductDetails >>
 OrderSw(ProductionOrder, ProductDetails.Supplier?)

def PackAndShipProduct(ProductionOrder) = ProductionOrder.shipTime :=
 SimElaspedTime - ProductionOrder.SalesOrder?.WaitTime?


def Dell(SalesOrder) =
 (ValidateSalesOrder(SalesOrder) ; NotifyFail(SalesOrder, "Sales order rejected")) >>
```

```
 GenerateProductionOrder(SalesOrder) >ProductionOrder>
 ((x <x< Some(ProcureHardwareComponents(ProductionOrder)) | Rwait(2000) >> None()) >Some(x)> x ;
 NotifyFail(SalesOrder, "Hardware not received on time")) >>
 ProcureSoftwareComponents(ProductionOrder) >>
 PackAndShipProduct(ProductionOrder) >>
 ProductionOrder

-- -- -- --
-- Supplier Orchestration
-- -- -- --

def AwaitReorderStockLevel(Supplier, PollPeriod, ReorderLevel) =
 if StockLevels.get(Supplier).value() <= ReorderLevel then signal
 else Rclock().wait(PollPeriod) >> AwaitReorderStockLevel(Supplier, PollPeriod, ReorderLevel)

def ReplenishStock(Supplier, ReorderQuantity) = (signals(ReorderQuantity) >>
 QoSStockIncrement(Supplier) >> stop) ; signal

def ManageSupplierInventory(Supplier, PollPeriod, ReorderLevel, ReorderQuantity) =
 AwaitReorderStockLevel(Supplier, PollPeriod, ReorderLevel) >>
 ReplenishStock(Supplier, ReorderQuantity) >>
 Println(Supplier+" replenishing stock, Revolver Level "+QoSStock(Supplier)) >>
 ManageSupplierInventory(Supplier, PollPeriod, ReorderLevel, ReorderQuantity)

-- -- -- --
-- Run Simulation
-- -- -- --

-- input QoS tokens: (order id, priority(cost/availability), cost, wait time)
def SimulateOrders(n) = Dictionary() >SalesOrder> SalesOrder.id := n >>
 SalesOrder.QoSCompare := QoSPriority >> SalesOrder.Cost := 0 >>
 SalesOrder.WaitTime := SimElaspedTime >> Println("Order "+SalesOrder.id?+" created")
 >> SalesOrder  | Rwait(Random(500)) >> SimulateOrders(n+1)


-- output QoS tokens: (revolver query rate, revolver level, cost, wait time)
ManageSupplierInventory("AMD", 5000, 5, 10)
| ManageSupplierInventory("NVIDIA", 10000, 10, 30)
| SimulateOrders(1) >SalesOrder> Dell(SalesOrder) >ProductionOrder>
 QoSQueryRateOplus(ProductionOrder.HwSupplier?) >>
 Println("Order "+ProductionOrder.SalesOrder?.id?+" shipped with
 "+ProductionOrder.HwSupplier?+" and "+ProductionOrder.SwSupplier?) >> Println("QoS Metrics:
 "+ProductionOrder.HwSupplier?+" Query Count: "+QoSQueryRate(ProductionOrder.HwSupplier?)+"
 Revolver Level "+QoSStock(ProductionOrder.HwSupplier?)+" Customer Wait Time "
 +ProductionOrder.shipTime?+" Cost "+ProductionOrder.HwQuote?.price?) >>  stop
```

11.3.2 MATLAB code

The MATLAB implementation of the Dell example discussed in Chapter 7 with a sample optimization output.

```
%Set up distributions for delay/throughput
for uu=1:1:20000
v1 = 3;
delta1 = 5;
tdist(uu) = (randn+delta1) ./ sqrt(2.*randg(v1./2) ./ v1);
expdist(uu) = 4 -2 .* log(rand);
end

%Set up AHP weight and the optimization
f = [0.1047; 0.2583; 0.6370;];
A = [1 -1 0 ; 0 -1 1];
B = [-expdist(ceil(1+19999*rand))*tdist(ceil(1+19999*rand)); 0];
Ae = [-1 1 -1];
Be = [-expdist(ceil(1+19999*rand))*tdist(ceil(1+19999*rand))];
[X,fval,output] = linprog(f, A,B, Ae, Be, [0 0 0],[],[],optimset('Diagnostics','on','Display','iter'));

%Assign optimization values
cust_delay = tdist(ceil(1+19999*rand));
cust_through = expdist(ceil(1+19999*rand));
critical_stock = X(1);
```

```
max_level = X(2);
batch = X(3);
supply_delay = tdist(ceil(1+19999*rand));
t_refuel =  max(tdist);

%Run the Dell Simulation
critical_lvl = critical_stock;
batch_size = batch;
t = 0; mm = 1;
current_lvl = max_level;
current_lvl = max_level -  expdist(ceil(1+19999*rand))*mm;
if current_lvl < critical_lvl
   t = t+1;
   if t >= t_refuel
       current_lvl = current_lvl + batch_size;
       t=0; mm=0;
   end
end
current_stockvalue = current_lvl
mm = mm + 1;

MATLAB OUTPUT:
-------------------------------------------------------------
Number of variables: 3

 Number of linear inequality constraints:    2
 Number of linear equality constraints:      1
 Number of lower bound constraints:          3
 Number of upper bound constraints:          0

Algorithm selected
   large-scale: interior point

  Residuals:   Primal     Dual     Duality    Total
               Infeas    Infeas      Gap       Rel
               A*x-b    A'*y+z-f    x'*z      Error
  ---------------------------------------------------
  Iter    0:  1.90e+002 5.59e+000 1.35e+003 1.00e+002
  Iter    1:  7.94e-014 6.35e-001 1.70e+002 6.35e-001
  Iter    2:  1.28e-010 1.22e-016 3.65e+001 3.64e-001
  Iter    3:  6.70e-011 2.01e-016 3.09e+000 4.32e-002
  Iter    4:  5.88e-013 2.08e-016 4.17e-003 5.90e-005
  Iter    5:  1.02e-013 2.56e-016 2.08e-007 2.95e-009
Optimization terminated.

current_stockvalue =

   67.8043
```

11.4 Importance Sampling/Splitting codes for Chapter 8.

```
% Importance Sampling
for ii=1:1:I
  for jj=1:1:J
    % Original Distribution
    response(jj) =  nctrnd(3,5);
  end
% Mean Monte-Carlo with threhsold tt
MeanMC(ii) = sum(response >=tt)/J;
end

for ii=1:1:I
  % New distribution for sampling
  x = nctrnd(3,10,1,K);
  for kk=1:1:K
    h = x(kk)>=tt;
    b = nctrnd(3,5)/x(kk);
    w(kk) = h*b;
  end
```

```
% Mean Importance Sampling
MeanIS(ii) = sum(w)/K;
end

% Importance Splitting
X = nctrnd(5.9,6,N,1);
% Estimate the alpha quantile
q_alpha = quantile(X,alpha);
i=0;
while (q_alpha < S)
   w = (X>q_alpha);
   Y = randsample(X,N,'true',w);
   % Using Metropilos hastings Algorithm
   p = MH(quantile(Y,alpha), N, @(x)( nctpdf(x,5.9,6) ) );
   X = (p>=q_alpha).*p + (p<q_alpha).*Y;
   % Estimate the new alpha quantile
   q_alpha = quantile(X,alpha);
   i=i+1;
end

% Metropolis-Hastings function
function [ epsilon ] = MH( epsilon_0, num_iterations, fpdf )
% pre-generate the random variables
normal_randoms = randn(num_iterations,1);
uniform_randoms = rand(num_iterations,1);
epsilon = zeros(num_iterations,1);
previous_epsilon = epsilon_0;
% for each random element
for i = 1:num_iterations
    % add a normally distributed noise
    epsilon_tilde = previous_epsilon + normal_randoms(i);
    if fpdf( epsilon_tilde ) > fpdf( previous_epsilon )
        epsilon(i) = epsilon_tilde;
    else
        if uniform_randoms(i) <= fpdf( epsilon_tilde ) / fpdf( previous_epsilon )
            epsilon(i) = epsilon_tilde;
        else
            epsilon(i) = previous_epsilon;
        end
    end
    % update the previous draw
    previous_epsilon = epsilon(i);
end
end
```

11.5 Stochastic Dominance for Chapter 9

```
function[Result] = S_StochasticDominance_trials1(mu_A,sig_A,mu_B,sig_B)

% order of dominance
Q=1;

% Discretized pdfs
N=2^16;
J=10^6;
dd=normrnd(mu_A,sig_A,1,J);
uu=normrnd(mu_B,sig_B,1,J);
Hi=max([dd uu]);
Lo=min([dd uu]);
h=(Hi-Lo)/(N-1);
X=[Lo+h : h : Hi]';

Iq_A = 1/h*(nctcdf(X+h/2,mu_A,sig_A)-nctcdf(X-h/2,mu_A,sig_A));
Iq_B = 1/h*(nctcdf(X+h/2,mu_B,sig_B)-nctcdf(X-h/2,mu_B,sig_B));

% check dominance
for q=2:Q+1
    Iq_A=h*cumsum(Iq_A);
    Iq_B=h*cumsum(Iq_B);
end
```

```
Result=0; %'No dominance up to order ' num2str(Q)
Condition_AdomB=prod(0+(Iq_A<=Iq_B));
if Condition_AdomB
    Result=1; %'A order-' num2str(Q) ' dominates B'
end
Condition_BdomA=prod(0+(Iq_B<=Iq_A));
if Condition_BdomA
    Result=2; %['B order-' num2str(Q) ' dominates A' ]
end
```

11.6 Q-Orc Implementation Outline from Chapter 10.

Presented here is the weaving procedure for the `TravelAgent` example with some special features:

- *Metrics*: Multiple domains of QoS metrics are presented. These include ambient metrics of latency and CPU utilization that are presented with competition operators defined.
- *travelAgent*: The example presented makes use of the `QoSRegistry` and `QoSWeaving` sites to specify the QoS domains called by the sites. Examples: `TravelAgent(SalesOrder, Budget, Cost, Latency, Reliability, InterQuery, Security, CPUutlization)`, `AirlineCompany(GenerateInvoice,Cost)`.
- *Function + QoS*: The functional specification can also make use of QoS values. An example is appending the final weaved output with the sum of `[Cost(Items). QoSOplus(Hotel, Airline)]`.

```
{- Latency.inc.orc -- Orc program Latency.inc
 -
 - Created by akattepu on 07-Aug-2012 17:05:11
 -}

def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))
val curTime = Rclock()
def LatencyIncrement(sitex) = (sitex,curTime.time()) >(sitex,d)> (sitex,curTime.time()-d)

type Millisecond = Number
def Millisecond() = signal
type Second = Number
def Second() = signal

-- Types and Definitions for Response Time
type ResponseTime = Number
def class ResponseTime(unit) =
  def QoS(String) :: Number
  def QoS(sitex) =
   signal >> LatencyIncrement(sitex) >(_,q)> (Ift(unit = Millisecond) >> q
    | Ift(unit = Second) >> q/1000)

  def QoSOplus(Number,Number) :: Number
  def QoSOplus(rt1,rt2) = rt1+rt2

  def QoSCompare(Number,Number) :: Number
  def QoSCompare(rt1,rt2) = rt1 <= rt2

  def QoSCompete(Number,Number) :: Number
  def QoSCompete(rt1,rt2) = bestQoS(QoSCompare,[rt1,rt2])

  def QoSVee(Number,Number) :: Number
  def QoSVee(rt1,rt2) = max(rt1,rt2)
  stop
```

```
{- CPU.inc.orc -- Orc program CPU.inc
 -
 - Created by akattepu on Sep 5, 2012 4:26:28 PM
 -}

def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))
val curTime = Rclock()

type Percentage = Number
def Percentage() = signal
def CPUutilizationValue(sitex) = (sitex,curTime.time()) >(sitex,d)> (sitex,curTime.time()-d)
 >(_,delta)> (100 /(1+ delta))

-- Types and Definitions for Response Time
type CPUutilization = Number
def class CPUutilization(unit) =
  def QoS(String) :: Number
  def QoS(sitex) = signal >> CPUutilizationValue(sitex)

  def QoSOplus(Number,Number) :: Number
  def QoSOplus(rt1,rt2) = rt1+rt2

  def QoSCompare(Number,Number) :: Number
  def QoSCompare(rt1,rt2) = rt1 >= rt2

  def QoSCompete(Number,Number) :: Number
  def QoSCompete(rt1,rt2) = bestQoS(QoSCompare,[rt1,rt2])

  def QoSVee(Number,Number) :: Number
  def QoSVee(rt1,rt2) = min(rt1,rt2)
  stop

{- Cost.inc.orc -- Orc program Cost.inc
 -
 - Created by akattepu on 07-Aug-2012 17:06:40
 -}

def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))
val curTime = Rclock()
def CostValue() = [Random(10),Random(10),Random(10)]

type Items = Number
def Items() = signal
type CurrencyEuros = Number
def CurrencyEuros(y) = y*100
type CurrencyDollars = Number
def CurrencyDollars(y) = y*80

-- Types and Definitions for Cost
type Cost = Number
def class Cost(unit) =
  def QoS(String,List[Number]) :: List[Number]
  def UnitConverter(X,unit) = Ift(unit=Items) >> X | Ift(unit=CurrencyEuros) >> map(CurrencyEuros,X)
   | Ift(unit=CurrencyDollars) >> map(CurrencyDollars,X)
  def QoS(sitex,c)=
  val s = Ref([])
  signal >> (s? >q> (Ift(q=[]) >> s:=c >> s? | Iff(q=[]) >> QoSOplus(s?,c)  >v> s:=v
   >> UnitConverter(s?,unit) ))

  def QoSOplus(List[Number],List[Number]) :: List[Number]
  def QoSOplus(c1,c2) =
    def Oplus([],[]) = []
    def Oplus(x:xs,y:ys) = (x+y):Oplus(xs,ys)
    Oplus(c1,c2)

  def QoSCompare(List[Number],List[Number]) :: List[Number]
  def QoSCompare(c1,c2) =
    def Compare([],[]) = true
    def Compare(x:xs,y:ys) = (x <= y) && Compare(xs,ys)
    Compare(c1,c2)

  def QoSCompete(List[Number],List[Number]) :: List[Number]
  def QoSCompete(c1,c2) = bestQoS(QoSCompare,[c1,c2])
```

```
  def QoSVee(List[Number],List[Number]) :: List[Number]
  def QoSVee(c1,c2) =
    def Vee([],[]) = []
    def Vee(x:xs,y:ys) = max(x,y):Vee(xs,ys)
    Vee(c1,c2)
  stop

{- InterQuery.inc.orc -- Orc program InterQuery.inc
 -
 - Created by akattepu on 07-Aug-2012 17:05:47
 -}

def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))
val curTime = Rclock()
type Millisecond = Number
def Millisecond() = signal
type Second = Number
def Second() = signal

-- Types and Definitions for Inter Query Time
type InterQueryTime = Number
def class InterQueryTime(unit)=
  def QoS(String) :: Number
  def QoS(sitex) =
    val s = {. r = Ref(0), c = Channel() .}
    signal >>(sitex,
    s.r? >p> (s.c.put(curTime.time()-p) | s.r:=(curTime.time())) >> Dictionary()
     >sitex> sitex.InterQueryTime := s) >> (Ift(unit = Millisecond) >> s.r?
     | Ift(unit = Second) >> s.r?/1000)

  def QoSCompare(Number,Number) :: Number
  def QoSCompare(it1,it2) = it1 >= it2

  def QoSCompete(Number,Number) :: Number
  def QoSCompete(it1,it2) = bestQoS(QoSCompare,[it1,it2])
  stop

{- Security.inc.orc -- Orc program Security.inc
 -
 - Created by akattepu on 07-Aug-2012 17:07:23
 -}

def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))
val curTime = Rclock()
type Level= String
def Level() = signal
def SecurityValue() = if (URandom() :> 0.5) then "High" else "Low"

-- Types and Definitions for Security
type SecurityLevel = String
def class SecurityLevel(unit) =
  def QoS(String,String) :: String
  def QoS(sitex,security) =
    val k = Ref(0)
    signal >>(k? >> Dictionary() >s> s.sitename := sitex >> s.sec := security >> s.sec?)

  def QoSOplus(String,String) :: String
  def QoSOplus(sl1,sl2) = Ift(sl1="High" && sl2="High") >> "High"
  |  Iff(sl1="High" && sl2="High") >> "Low"

  def QoSCompare(String,String) :: String
  def QoSCompare(sl1,sl2) = (sl1 = "High" && sl2="Low")

  def QoSCompete(String,String) :: String
  def QoSCompete(sl1,sl2) = bestQoS(QoSCompare,[sl1,sl2])

  stop

{- Reliability.orc -- Orc program Reliability
 -
 - Created by akattepu on Sep 5, 2012 2:24:36 PM
 -}
```

```
def bestQoS(comparer, publisher) = head(sortBy(comparer, publisher))

type Level= String
def Level() = signal
def ReliabilityValue() = if (URandom() :> 0.5) then "in_operation" else "failure"

-- Types and Definitions for Security
type Reliability = String
def class Reliability(unit) =
  def QoS(String,String) :: String
  def QoS(sitex,reliability) =
    val k = Ref(0)
    signal >>(k? >> Dictionary() >s> s.sitename := sitex >> s.sec := reliability >> s.sec?)

  def QoSOplus(String,String) :: String
  def QoSOplus(sl1,sl2) = Ift(sl1="in_operation" && sl2="in_operation") >> "in_operation"
                          |  Iff(sl1="in_operation" && sl2="in_operation") >> "failure"

  def QoSCompare(String,String) :: String
  def QoSCompare(sl1,sl2) = (sl1 = "in_operation" && sl2="failure")

  def QoSCompete(String,String) :: String
  def QoSCompete(sl1,sl2) = bestQoS(QoSCompare,[sl1,sl2])

  stop

-------------------------------------------------------------------------------------

{- SLA.inc.orc -- Orc program SLA.inc
 -
 - Created by akattepu on 02-Aug-2012 10:31:29
 -}

include "Latency.inc"
include "CPUutilization.inc"
include "InterQuery.inc"
include "Cost.inc"
include "Security.inc"
include "Reliability.inc"


def class NonAmbient(QoStype) =

 def QoSCompete(Number,Number) :: Number
 def QoSCompete(q1,q2) = head(sortBy((<:), [q1,q2]))

signal

def class Ambient(QoStype,competition) =

 def QoSCompete(Number,Number) :: Number
 def QoSCompete(q1,q2) = head(sortBy(competition,  [q1,q2]))

signal

-------------------------------------------------------------------------------------
{- QoSRegistry.inc.orc -- Orc program QoSManager.inc
 -
 - $Id$
 -
 - Created by akattepu on 02-Aug-2012 10:59:39
 -}

--include "QoSWeaver.inc"

val QoSRegistry =
 [
  {. name = "TAgent", QoSDom = ResponseTime, QoSUnit = Millisecond, Handle = LatencyIncrement .},
  {. name = "TAgent", QoSDom = CPUutilization, QoSUnit = Percentage, Handle = CPUutilizationValue .},
  {. name = "TAgent", QoSDom = InterQueryTime, QoSUnit = Second, Handle = []  .},
  {. name = "TAgent", QoSDom = Cost, QoSUnit = CurrencyEuros, Handle = CostValue .},
  {. name = "TAgent", QoSDom = SecurityLevel, QoSUnit = Level, Handle = SecurityValue .},
  {. name = "TAgent", QoSDom = Reliability, QoSUnit = Level, Handle = ReliabilityValue .},
```

```
  {. name = "Airline", QoSDom = Cost, QoSUnit = Items, Handle = CostValue .},

  {. name = "Hotel", QoSDom = Cost, QoSUnit = Items, Handle = CostValue  .}
 ]

def QoSMatch(siteID) = each(QoSRegistry) >M> Ift(M.name = siteID) >> (M.QoSDom,M.QoSUnit,M.Handle)

def QoSValidate(callersiteID,caleesiteID) =  (collect(defer(QoSMatch,callersiteID)),
 collect(defer(QoSMatch,caleesiteID)))  >(A,B)> ( Ift(A.QoSDom = B.QoSDom) >> signal
 | Iff(A.QoSDom = B.QoSDom) >> Println("Registry Entries Missing") >> stop)

-----------------------------------------------------------------------------------------
{- QoSWeaver.orc -- Orc program QoSWeaver
 -
 - Created by akattepu on 08-Aug-2012 14:50:38
 -}

def QoSWeaver(site,(lookup,unit,handle)) =

   def ResponseTimeCheck(competition) =
    Ift(lookup = ResponseTime && handle = LatencyIncrement) >>
    Some(Ambient(ResponseTime,competition))
    >> (ResponseTime(unit).QoS(site))
    ; stop

   def CPUutlizationCheck(competition) =
    Ift(lookup = CPUutilization && handle = CPUutilizationValue) >>
    Some(Ambient(CPUutilization,competition))
    >> (CPUutilization(unit).QoS(site))
    ; stop

   def CostCheck() =
    Ift(lookup = Cost && handle = CostValue) >>
    Some(NonAmbient(Cost)) >> Cost(unit).QoS(site,CostValue())
    ; stop

   def SecurityCheck() =
    Ift(lookup= SecurityLevel && handle = SecurityValue) >>
    (Some(NonAmbient(SecurityLevel))
    >> SecurityLevel(unit).QoS(site,SecurityValue()))
    ; stop

   def InterQueryCheck() =
    Ift(lookup= InterQueryTime) >>  (Some(NonAmbient(InterQueryTime))
    >> InterQueryTime(unit).QoS(site))
    ; stop

    def ReliabilityCheck() =
    Ift(lookup= Reliability && handle = ReliabilityValue)
    >>  (Some(NonAmbient(Reliability))
    >> Reliability(unit).QoS(site,ReliabilityValue()))
    ; stop

signal >> v<v<(ResponseTimeCheck(max)| CostCheck() | SecurityCheck() | InterQueryCheck())
 |ReliabilityCheck() | CPUutlizationCheck(min)

def QoS(site,identifier) =
   val Data = Ref()
   def QoSCollect(v) = collect(defer2(QoSWeaver,Data?,v))
site >d> Data:=d >> collect(defer(QoSMatch,identifier)) >v> (Data?, map(QoSCollect,v))
-----------------------------------------------------------------------------------------

{- TAgent.orc -- Orc program TAgent
 -
 - Created by akattepu on 03-Aug-2012 17:01:05
 -}
include "SLA.inc"
include "QoSRegistry.inc"
include "QoSWeaver.inc"

val AirlineList = ["Airline 1", "Airline 2"]
val HotelList = ["Hotel A", "Hotel B"]

--BestQoS and simulation utilities
```

```
def bestQ(comparer,publisher) =  head(sortBy(comparer,collect(publisher)))
def cat() = if (Random(1)=1) then "Economy" else "Premium"
val simElaspedTime = Rclock()

--Sites declared
def TravelAgent(SalesOrder,Budget,Cost,Latency,Reliability,InterQuery,Security,CPUutlization) =
  def GenerateOrder(SalesOrder,Budget) = Dictionary() >GenerateInvoice>
    GenerateInvoice.TravelAgent := SalesOrder.ordernumber?
    >> GenerateInvoice.acceptedTime := simElaspedTime.time()
    >> Println("Order "+GenerateInvoice.TravelAgent?+"
    accepted at time "+GenerateInvoice.acceptedTime?) >> (GenerateInvoice,Budget)
  def inquireCost(List) = each(List) >sup> Dictionary() >ProductDetails>
   ProductDetails.Company := sup >> ProductDetails.cost := Random(50)
   >> ProductDetails
  def inquireCategory(List) = each(List) >sup> Dictionary()
   >ProductDetails> ProductDetails.Company := sup
   >> ProductDetails.cost := Random(50) >> ProductDetails.category := cat()
   >> ProductDetails
  def compareCost(x, y) = x.cost? <= y.cost?
  def compareCategory(x, y) = if x.category?="Economy" then false
   else if y.category?="Economy" then true else compareCost(x, y)

  def AirlineCompany(GenerateInvoice,Cost) =
   QoS(bestQ(compareCost, defer(inquireCost,AirlineList))
   >q> GenerateInvoice >> GenerateInvoice.AirQuote := q, "Airline") >(_,AirQoS)>
   GenerateInvoice.AirQoS := AirQoS >> GenerateInvoice.AirQoS? >aq> Println(aq)
  def HotelBooking(GenerateInvoice,Cost) =
 QoS(bestQ(compareCategory, defer(inquireCategory,HotelList))
   >q> GenerateInvoice >> GenerateInvoice.HotelQuote := q, "Hotel") >(_,HotelQoS)>
   GenerateInvoice.HotelQoS := HotelQoS >> GenerateInvoice.HotelQoS? >hq> Println(hq)

  def CheckBudget(GenerateInvoice,Budget) = if (GenerateInvoice.AirQuote?.cost? +
   GenerateInvoice.HotelQuote?.cost? <: Budget) then GenerateInvoice
 else (Println("Resubmit Order "+GenerateInvoice.TravelAgent?) >> Dictionary()
 >SalesOrder> SalesOrder.ordernumber:= GenerateInvoice.TravelAgent?
 >> (SalesOrder,GenerateOrder(SalesOrder,Budget)))
  def timeout(x, t, SalesOrder) = Let(Some(x)
 | (Rwait(t) >> notifyFail(SalesOrder, "Timeout") >> None()))
  def notifyFail(SalesOrder, reaSalesOrdern) = Println("Order "+SalesOrder.id?+" failed:
   "+reaSalesOrdern) >> stop

  signal>> QoS((timeout((GenerateOrder(SalesOrder,Budget) >(GenerateInvoice,Budget)>
   AirlineCompany(GenerateInvoice,Cost) >> HotelBooking(GenerateInvoice,Cost) >>
   CheckBudget(GenerateInvoice,Budget)) , 10000, SalesOrder) >Some(GenerateInvoice)>
   GenerateInvoice),"TAgent")

--Simulation
def simulateOrders(2) = stop
def simulateOrders(n) = Dictionary() >SalesOrder> SalesOrder.ordernumber:= n >> SalesOrder
   | Rwait(Random(100)) >> simulateOrders(n+1)

simulateOrders(1) >SalesOrder> TravelAgent(SalesOrder,150,Cost,ResponseTime)
 >(GenerateInvoice,QoS)> ((GenerateInvoice.TravelAgent?,QoS) |
 (GenerateInvoice.AirQoS?,GenerateInvoice.HotelQoS?) >([[aq]],[[hq]])> (aq,hq) >>
 (GenerateInvoice.TravelAgent?,append(QoS,[Cost(Items).QoSOplus(aq,hq)])))

-----------------------------------------------------------------------------------------
Output:

Order 1 accepted at time 18
[[[1, 6, 2]]]
[[[2, 4, 9]]]
(1, [[3], [14], [230], [[8, 8, 1]], ["High"], ["failure"]])
(1, [[3], [14], [230], [[8, 8, 1]], ["High"], ["failure"], [3, 10, 11]])
```

Bibliography

[AB06a] Samy Abbes and Albert Benveniste. True-concurrency probabilistic models: Branching cells and distributed probabilities for event structures. *Inf. Comput.*, 204(2):231–274, 2006.

[AB06b] Mustafa Adacal and Ayse B. Bener. Mobile web services: A new agent-based framework. *IEEE Internet Computing*, pages 58–65, 2006.

[AB11] Andrea Arcuri and Lionel Briand. Formal analysis of the probability of interaction fault detection using random testing. *IEEE Transactions on Software Engineering*, 99, 2011.

[ACH98] Cristina Aurrecoechea, Andrew T. Campbell, and Linda Hauw. A survey of qos architectures. *Multimedia Systems*, 6:138–151, 1998.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[ACP11] Marco Abundo, Valeria Cardellini, and Francesco Lo Presti. Optimal admission control for a qos-aware service-oriented system. In *ServiceWave*, pages 179–190, 2011.

[AD05] Rainer Anzbock and Schahram Dustdar. Modeling and implementing medical web services. *Data & Knowledge Engineering*, 55:203–236, 2005.

[AGI+06] Danilo Ardagna, Gabriele Giunta, Nunzio Ingraffia, Raffaela Mirandola, and Barbara Pernici. QoS-Driven Web Services Selection in Autonomic Grid Environments. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (2)*, volume 4276 of *Lecture Notes in Computer Science*, pages 1273–1289. Springer, 2006.

[AGM08] Danilo Ardagna, Carlo Ghezzi, and Raffaela Mirandola. Model driven qos analyses of composed web services. In *ServiceWave*, 2008.

[AGP08] Assel Akzhalova, Mahbub Gani, and Iman Poernomo. Model driven qos management via dynamic programming. In *12th Enterprise Distributed Object Computing Conference Workshops*, pages 87–95, 2008.

[Ala92] Alain Bensoussan. *Stochastic Control of Partially Observable Systems*. Cambridge University Press, 1992.

[ALZH04] T. Abdelzaher, Ying Lu, Ronghua Zhang, and D. Henriksson. Practical application of control theory to web services. In *American Control Conference*, volume 3, 2004.

[AMW+03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *SIGOPS Oper. Syst. Rev.*, 37(5):74–89, October 2003.

[And96] Gordon Anderson. Nonparametric tests of stochastic dominance in income distributions. *Econometrica*, 64:1183–1193, 1996.

[AP05] Danilo Ardagna and Barbara Pernici. Global and Local QoS Guarantee in Web Service Selection. In Christoph Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812, pages 32–46, 2005.

[AR09a] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *World Wide Web Conference, Madrid, Spain*, 2009.

[AR09b] Mohammad Alrifai and Thomas Risse. Efficient qos-aware service composition. *Whitestein Series in Software Agent Technologies and Autonomic Computing*, pages 75–87, 2009.

[Ark02] A. Arkin. Busness process modeling language 1.0. Technical report, http://www.bpmi.org, 2002.

[BAM+08] Antonia Bertolino, Guglielmo De Angelis, Antinisca Di Marco, Paola Inverardi, Antonino Sabetta, and Massimo Tivoli. A framework for analyzing and testing the performance of software services. In *ISoLA*, 2008.

[BCG+03] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. *Proc. 1st Intl. Conf. on Service Oriented Computing (ICSOC)*, pages 43–58, 2003.

[BCG+05] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of web services in colombo. In *Proc. of 13th Itallian Symp. on Advanced Database Systems*, 2005.

[BCP+05] M. Bruno, G. Canfora, M. Di Penta, G. Esposito, and V. Mazza. Using test cases as contract to ensure service compliance across releases. In *Proc. of the 3rd Intl. Conf. in Service-Oriented Computing, Amsterdam, The Netherlands*, 2005.

[BD03] Garry F. Barrett and Stephen G. Donald. Consistent tests for stochastic dominance. *Econometrica*, 71:71–104, 2003.

[BDZ06] E. Boschi, S. Denazis, and T. Zseby. A measurement framework for inter-domain sla validation. *Elsevier Computer Communications*, 29:7.3–716, 2006.

[Ber07] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control: Vols. 1 and 2*. Athena Scientific, 2007.

[BFG05] Karthikeyan Bhargavan, Cedric Fournet, and Andrew D. Gordon. A semantics for web services authentication. *Theoretical Computer Science*, 340:102–153, 2005.

[BFHS03] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specifiation: A new approach to design and analysis of e-service composition. In *World Wide Web Conf. (WWW)*, 2003.

[BGG04] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *ICSOC'04, November 15-19, 2004,*, 2004.

[BGHM04] D. Berardi, M. Gruninger, R. Hull, and S. McIlraith. Towards a first-order ontology for web services. In *W3C Workshop on Constraints and Capabilities for Web Services*, 2004.

[BHS+10] Bas Boone, Sofie Van Hoecke, Gregory Van Seghbroeck, Niels Joncheere, Viviane Jonckers, Filip De Turck, Chris Develder, and Bart Dhoedt. Salsa: Qos-aware load balancing for autonomous service brokering. *Journal of Systems and Software*, 83:446–456, 2010.

[BJK+12] Albert Benveniste, Claude Jard, Ajay Kattepur, Sidney Rosario, and John Thywissen. Qos-aware management of monotonic service orchestrations. *Formal Methods in System Design (under review), Springer*, 2012.

[BK02] Falko Bause and Pieter S. Kritzinger. *Stochastic Petri Nets - An Introduction to the Theory*. Vieweg Verlag, 2002.

[BKM07] Manfred Broy, Ingolf H. Kruger, and Michael Meisinger. A formal model of services. *ACM Transactions Software Engineering and Methodology*, 16:1–39, 2007.

[BM07] Maria Grazia Buscemi and Ugo Montanari. Cc-pi: a constraint-based language for specifying service level agreements. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 18–32. Springer-Verlag, 2007.

[BM11] Maria Grazia Buscemi and Ugo Montanari. Qos negotiation in service composition. *The Journal of Logic and Algebraic Programming*, 80:13–24, 2011.

[BMR08] Azzedine Benameur, Fabio Massacci, and Nataliya Rassadko. Security views for outsourced business processes. In *SWS'08, Fairfax, Virginia*, 2008.

[BMRS10] Stefano Bistarelli, Ugo Montanari, Francesca Rossi, and Francesco Santini. Unicast and multicast qos routing with soft-constraint logic programming. *ACM Trans. Comput. Logic*, 12(1):5:1–5:48, November 2010.

[Bon11] Bonitasoft. Open solution: Open source bpm. Technical report, http://www.bonitasoft.com/products/features, 2011.

[BP06] Antonio Brogi and Razvan Popescu. From bpel processes to yawl workflows. In *3rd International Workshop on Web Services and Formal Methods (WS-FM)*, 2006.

[BPE07] Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Standard, Available from: http://docs.oasisopen.org/wsbpel/2.0/wsbpel-v2.0.pdf, April, 2007.

[BRBH08] Anne Bouillard, Sidney Rosario, Albert Benveniste, and Stefan Haar. Monotony in service orchestrations. Technical report, INRIA Research Report 6528, 2008.

[BRBH09] Anne Bouillard, Sidney Rosario, Albert Benveniste, and Stefan Haar. Monotonicity in Service Orchestrations. In Giuliana Franceschinis and Karsten Wolf, editors, *Petri Nets*, volume 5606 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2009.

[BRMO01] Athman Bouguettaya, Abdelmounaam Rezgui, Brahim Medjahed, and Mourad Ouzzani. Internet computing support for digital government. *Practical Handbook of Internet Computing*, pages 1–14, 2001.

[Bru72] Nocolaas Govert De Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Elsevier)*, 34:381–392, 1972.

[BS09a] Stefano Bistarelli and Francesco Santini. A nonmonotonic soft concurrent constraint language for sla negotiation. *Electr. Notes Theor. Comput. Sci.*, 236:147–162, 2009.

[BS09b] Stefano Bistarelli and Francesco Santini. Soft constraints for quality aspects in service oriented architectures. In *Young Researchers Workshop on Service-Oriented Computing*, 2009.

[BSC01] Preeti Bhoj, Sharad Singhal, and Sailesh Chutani. Sla management in federated environments. *Computer Networks*, 35(1):5–24, 2001.

[BT98] A. Bechini and K.-C. Tai. Design of a toolset for dynamic analysis of concurrent java programs. In *Proc. of the 6th Intl. Workshop on Program Comprehension*, pages 190–197, 1998.

[Buc04] James A. Bucklew. *Introduction to Rare Event Simulation*. Springer Series in Statistics, 2004.

[BV05] K. Z. Bell and M. A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. In *3rd International Conference on Information and Communications Technology*, 2005.

[BV09] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2009.

[BZ07] Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *SC 2007, LNCS 4829*, 2007.

[CCGM06] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, and Raffaela Mirandola. A framework for optimal service selection in broker-based architectures with multiple qos classes. In *IEEE Services Computing Workshops*, 2006.

[CCGP10] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, and Francesco Lo Presti. Adaptive management of composite services under percentile-based service level agreements. In *ICSOC 2010, LNCS 6470*, pages 381–395, 2010.

[CCL05] Lei Cao, Jian Cao, and Minglu Li. Genetic algorithm utilized in cost-reduction driven web service selection. In *Proc. Intl. Conf. Computational Intelligence and Security*, pages 679–686, 2005.

[CCMN04] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *World Wide Web Conference*, 2004.

[CCP07] Cinzia Cappiello, Marco Comuzzi, and Pierluigi Plebani. On automated generation of web service level agreements. *CAiSE, LNCS, Springer-Verlag Berlin Heidelberg*, 4495:264–278, 2007.

[CD99] Luca Cardelli and Rowan Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25:309–316, 1999.

[CDFP97] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Trans. on Software Engineering*, 23:437–444, 1997.

[CDMFG11] F. Cérou, P. Del Moral, T. Furon, and A. Guyader. Sequential monte carlo for rare event estimation. *Statistics and Computing, Springer Netherlands*, 2011.

[CDS08] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. on Software Engineering*, 34, 5:633–650, 2008.

[CGK+11] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *Software Engineering, IEEE Transactions on*, 37(3):387 –409, may-june 2011.

[CH09] David Cohn and Richard Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, pages 3–9, 2009.

[Chu85] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1985.

[CjM05] William Cook and jayadev Misra. Implementation outline for orc, http://orc.csres.utexas.edu/papers/orcimpdraft.pdf. Technical report, UT Austin, 2005.

[CK07] Tom Chothia and Jetty Kleijn. Q-automata: Modelling the resource usage of concurrent components. *Electronic Notes in Theoretical Computer Science*, 175:153–167, 2007.

[CM] William R. Cook and Jayadev Misra. Orchestration in orc: A deterministic distributed programming model. Department of Computer Sciences, University of Texas at Austin.

[CMP+04] Cinzia Cappiello, Paolo Missier, Barbara Pernici, Pierluigi Plebani, and Carlo Batini. Qos in multichannel is: the mais approach. In *Workshops in Connection with the 4th International Conference on Web Engineering*, 2004.

[CMSA02] Jorge Cardoso, John Miller, Amit Sheth, and Jonathan Arnold. Modeling quality of service for workflows and web service processes. Technical report, LSDIS Lab, Computer Science, University of Georgia, 2002.

[CP06] Gerardo Canfora and Massimiliano Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IEEE IT Pro*, 6:10–17, 2006.

[CPE+06] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, Francesco Perfetto, and Maria Luisa Villani. Service Composition (re)Binding Driven by Application-Specific QoS. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 141–152. Springer, 2006.

[CPEV05a] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *GECCO*, 2005.

[CPEV05b] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. QoS-Aware Replanning of Composite Web Services. In *ICWS*, pages 121–129. IEEE Computer Society, 2005.

[CPEV08] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. A framework for QoS-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, 2008.

[CPM06a] William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in orc. In *Intl. Conf. on Coordination Models and Languages (COORDINATION)*, 2006.

[CPM06b] William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow Patterns in Orc. In *Coordination*, pages 82–96, 2006.

[CSM02] Jorge Cardoso, Amit P. Sheth, and John A. Miller. Workflow Quality of Service. In Kurt Kosanke, Roland Jochem, James G. Nell, and Angel Ortiz Bas, editors, *ICEIMT*, volume 236 of *IFIP Conference Proceedings*, pages 303–311. Kluwer, 2002.

[CSM+04] Jorge Cardoso, Amit P. Sheth, John A. Miller, Jonathan Arnold, and Krys Kochut. Quality of Service for workflows and Web service processes. *J. Web Sem.*, 1(3):281–308, 2004.

[CTB98] Mark E. Crovella, Murad S. Taqqu, and Azer Bestavros. Heavy-tailed probability distributions in the world wide web. In *In A Practical Guide To Heavy Tails, chapter 1*, pages 3–26. Chapman & Hall, 1998.

[D'A06] Andrea D'Ambrogio. A model-driven wsdl extension for describing the qos ofweb services. In *IEEE International Conference on Web Services*, 2006.

[DB78] Peter J. Denning and Jeffrey P. Buzen. The operational analysis of queueing network models. *Computing Surveys, Vol. 10, No. 3, September 1978*, 10:225–261, 1978.

[DDK+04] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM Systems Journal*, 43, 2004.

[DH01] M. Dumas and A.t. Hofstede. Uml activity diagrams as a workflow specification language. In *Proc. of the Intl Conf. Unified Modeling Language (UML)*, 2001.

[DNFM+05] Rocco De Nicola, Gianluigi Ferrari, Ugo Montanari, Rosario Pugliese, Emilio Tuosto, and Jean-Marie Jacquet. *Coordination Models and Languages*, chapter A Process Calculus for QoS-Aware Applications, pages 246–252. Springer Berlin / Heidelberg, 2005.

[DOW08] Gero Decker, Hagen Overdick, and Mathias Weske. Oryx: An open modeling platform for the bpm community. Hasso-Plattner-Institute, University of Potsdam, Germany, 2008.

[DS05] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *Int. J. Web and Grid Services*, 1:1–30, 2005.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[ERV02] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.

[Fah05] Dirk Fahland. Complete abstract operational semantics for the web service business process execution language. Technical report, Humboldt-Universitat zu Berlin, 2005.

[FBH05] Viktoria Firus, Steffen Becker, and Jens Happe. Parametric performance contracts for qml-specified software components. *Electronic Notes in Theoretical Computer Science*, 141:73–90, 2005.

[FG01] Robert Fourer and Jean-Pierre Goux. Optimization as an internet resource. *Interfaces: OR/MS and E-Business*, 31:130–150, 2001.

[FH05] Martin Fenton and Ciara Heavin. Closing the loop: Providing web service solutions enabling e-logistics integration. In *18th Bled eConference eIntegration in Action*, 2005.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FK98] Sven Frolund and Jari Koistinen. Qml: A language for quality of service specification. Technical report, Software Technology Laboratory, Hewlett Packard, 1998.

[FMM08] R. Fourer, J. Ma, and K. Martin. Optimization services: A framework for distributed optimization. Technical report, COIN-OR, 2008.

[FN08] Alessandro Fantechi and Elie Najm. Session types for orchestration charts. In *COORDINATION*, 2008.

[GGK+11] Stephen Gilmore, László Gönczy, Nora Koch, Philip Mayer, Mirco Tribastone, and Dániel Varró. Non-functional properties in the model-driven development of service-oriented systems. *Springer Software Systems Model*, 10:287–311, 2011.

[GGMT08] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In *Quality of Software Architectures*, pages 119 – 134. LNCS vol. 5281, 2008.

[GHS95] Dimirios Georgakopolis, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[GNZ+06] Yan Gao, Jun Na, Bin Zhang, Lei Yang, and Qiang Gong. Optimal web services selection using dynamic programming. In *11th IEEE Symposium on Computers and Communications*, 2006.

[GP07] Donna Griffin and Dirk Pesch. A survey on web services in telecommunications. *IEEE Communications Magazine*, 7:28–35, 2007.

[Har87] David Harel. Statecharts:a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HB09] Loic Helouet and Albert Benveniste. Distributed active xml and service interfaces. Technical report, INRIA Research Report no. 7082, 2009.

[HB10] Loic Helouet and Albert Benveniste. Document based modeling of web services choreographies using active xml. In *IEEE International Conference on Web Services*, 2010.

[HDHA09] Robert Harmon, Haluk Demirkan, Bill Hefley, and Nora Auseklis. Pricing strategies for information technology services: A value-based approach. In *42nd Hawaii International Conference on System Sciences*, 2009.

[HLH10] William Ho, Carman K.M. Lee, and George To Sum Ho. Multiple criteria optimization of contemporary logistics distribution network problems. *OR Insight*, 23:27–43, 2010.

[HM09] Toan Huynh and James Miller. Empirical observations on the session timeout threshold. *Information Processing and Management*, 45:513–528, 2009.

[HMM04] Tony Hoare, Galen Menzel, and Jayadev Misra. A tree semantics of an orchestration language. In *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems, NATO ASI Series*, 2004.

[HMR10] Joyce El Haddad, Maude Manouvrier, and Marta Rukoz. Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE Transaction on Services Computing*, 3:73–85, 2010.

[Hoa04] C. A. R Hoare. *Communicating Sequential Processes*. Prentice Hall International, 2004.

[HS05] Richard Hull and Jianwen Su. Tools for composite web services: A short overview. *SIGMOD Record*, 34:1–10, 2005.

[HSS05] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *3rd International Conference on Business Process Management (BPM 2005)*, 2005.

[HT03] David Harel and P.S. Thiagarajan. Message sequence charts. *UML for Real: Design of Embedded Real-time Systems, Kluwer Academic Publishers*, pages 1–29, 2003.

[HWSP04] San-Yih Hwang, Haojun Wang, Jaideep Srivastava, and Raymond A. Paul. A probabilistic qos model and computation framework for web services-based workflows. In *ER 2004, LNCS 3288, Springer-Verlag Berlin Heidelberg*, 2004.

[HWTS07] San-Yih Hwang, Haojun Wang, Jian Tang, and Jaideep Srivastava. A probabilistic approach to modeling and estimating the qos of web-services-based workflows. *Information Sciences*, 177:5484–5503, 2007.

[IBM+07] IBM, BEA, Microsoft, SAP, and Siebel. Business process execution language for web services ver 1.1. Technical report, 2007.

[IBM11] IBM. Ilog jviews enterprise. Technical report, http://www-01.ibm.com/software/integration/visualization/jviews/enterprise/, 2011.

[ICH10] Dragan Ivanovic, Manuel Carro, and Manuel Hermenegildo. Towards data-aware qos-driven adaptation for service orchestrations. In *Proceedings of the 2010 IEEE International Conference on Web Services*, ICWS '10, pages 107–114, Washington, DC, USA, 2010. IEEE Computer Society.

[IM02] IBM and Microsoft. Security in a web services world: A proposed architecture and roadmap, 2002.

[INPJ09] Paul Istoan, Gregory Nain, Gilles Perrouin, and Jean-Marc Jezequel. Dynamic software product lines for service-based systems. In *Ninth IEEE International Conference on Computer and Information Technology*, 2009.

[Jac08] Daniel Jackson. http://alloy.mit.edu. 2008.

[JB02] M. Jaring and J. Bosch. Representing variability in software product lines: A case study. *Proc. of the Second Intl. Conf. on Software Product Lines, London, UK*, pages 15–36, 2002.

[JJJR94] C. Jard, G. V. Jourdan, T. Jeron, and J. X. Rampon. A general approach to trace-checking in distributed computing systems. *Proc. of the 14th Intl. Conf. on Distributed Computing Systems*, 1994.

[jJMS02] Li jie Jin, Vijay Machiraju, and Akhil Sahai. Analysis on service level agreement of web services. Technical report, HP Laboratories Palo Alto, 2002.

[JKTB12] Claude Jard, Ajay Kattepur, John Thywissen, and Albert Benveniste. Leveraging causality for qos tracking in service oriented systems. *9th International Workshop on Web Services and Formal Methods (submitted)*, 2012.

[JMW07] Kurt Jensen, Lars Michael, and Kristensen Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. In *International Journal on Software Tools for Technology Transfer*, 2007.

[Kat11] Ajay Kattepur. Importance sampling of probabilistic contracts in web services. In *International Conference on Service Orinted Computing*, 2011.

[KBJ11] Ajay Kattepur, Albert Benveniste, and Claude Jard. Optimizing decisions in web services orchestrations. In *International Conference on Service Oriented Computing*, 2011.

[KBJ12] Ajay Kattepur, Albert Benveniste, and Claude Jard. Negotiation strategies for probabilistic contracts in web services orchestrations. In *International Conference on Web Services (ICWS)*, 2012.

[KBT+09] Babak Khosravifar, Jamal Bentahar, Philippe Thiran, Ahmad Moazin, and Adrien Guiot. An approach to incentive-based reputation for communities of web services. In *ICWS*, 2009.

[KCE+04] P. Kearney, J. Chapman, N. Edwards, M. Gifford, and L. He. An overview of web services security. *BT Technology Journal*, 22:27–42, 2004.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. *Software Engineering Institute*, 1990.

[KCJ98] Lars M. Kristensen, Soren Christensen, and Kurt Jensen. The practitioner's guide to coloured petri nets. *Int. J. STTT*, 2:98–132, 1998.

[KCM06] David Kitchin, William R. Cook, and Jayadev Misra. A Language for Task Orchestration and its Semantic Properties. In *Proc. of the Intl. Conf. on Concurrency Theory (CONCUR)*, 2006.

[KCW06] P. M. Kelly, P. D. Coddington, and A. L. Wendelborn. A simplified approach to web service development. In *Australasian workshops on Grid computing and e-research*, volume 54, pages 79–88, 2006.

[KDS+08] Marcel Karam, Sergiu Dascalu, Haidar Safa, Rami Santina, and Zeina Koteich. A product-line architecture for web service-based visual composition of web applications. *The Journal of Systems and Software*, 81:855–867, 2008.

[KGM09] J. Kienzle, N. Guelfi, and S. Mustafiz. Crisis management systems: A case study for aspect-oriented modeling. Technical report, McGill Univ., 2009.

[Kis02] Ivan Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams, Indianapolis, IN, USA, 2002.

[KKK08] Jong Myoung Ko, Chang Ouk Kim, and Ick-Hyun Kwon. Quality-of-service oriented web service composition algorithm and planning architecture. *The Journal of Systems and Software*, 81:2079–2090, 2008.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[KM03] Hristo Koshutanski and Fabio Massacci. An access control framework for business processes for web services. In *ACM Workshop on XML Security*, 2003.

[KP09] Kyriakos Kritikos and Dimitris Plexousakis. Mixed-integer programming for qos-based web service matchmaking. *IEEE Transaction on Services Computing*, 2:122–139, 2009.

[KQCM09] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The orc programming language. In *Proceedings of FMOODS/FORTE 2009*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009.

[KSB+10] Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste, and Claude Jard. Variability modeling and qos analysis of web services orchestrations. In *Proceedings of the 2010 IEEE International Conference on Web Services*, ICWS '10, pages 99–106, 2010.

[KSB+11] Ajay Kattepur, Sagar Sen, Benoit Baudry, Albert Benveniste, and Claude Jard. Pairwise testing of dynamic composite services. In *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems*, SEAMS '11, pages 138–147, 2011.

[KW04] D. R. Kuhn and D. D. Wallace. Software fault interactions and implications for software testing. *IEEE Trans. on Software Engineering*, 30:418–421, 2004.

[KZC+04] Roman Kapuscinski, Rachel Q. Zhang, Paul Carbonneau, Robert Moore, and Bill Reeves. Inventory decisions in dell's supply chain. *Interfaces*, 34:191–205, 2004.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978.

[LB10] Noura Limam and Raouf Boutaba. Assessing software service quality and trustworthiness at selection time. *IEEE Transactions on Software Engineering*, 36:559–574, 2010.

[LDT07] Pierre L'Ecuyer, Valerie Demers, and Bruno Tuffin. Rare events, splitting, and quasi-monte carlo. *ACM Transactions on Modeling and Computer Simulation, Vol. 17, No. 2, Article 9, Publication date: April 2007.*, 17:1–45, 2007.

[Ley01] F. Leymann. Web services flow language ver 1.0. Technical report, IBM, 2001.

[LKD+03] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck. Web service level agreement (wsla) language specification. Technical report, IBM Corporation, 2003.

[LL01] Moshe Levy and Haim Levy. Testing for risk aversion: a stochastic dominance approach. *Economics Letters*, 71:233–240, 2001.

[LM05] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 2005.

[LNJV01] Zhen Liu, Nicolas Niclausse, and César Jalpa-Villanueva. Traffic model and performance evaluation of web servers. *Performance Evaluation*, 46:77–100, 2001.

[LNS06] Joan Lu, Tahir Naeem, and John B. Stav. A distributed information system for healthcare web services. In *AP Web Workshops*, 2006.

[LPM+09] Kelly Lyons, Corrie Playford, Paul R. Messinger, Run H. Niu, and Eleni Stroulia. Business models in emerging online services. In *Value Creation in e-Business Management, LNBIP 36*, 2009.

[LPT07] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[LZZX10] Kwei-Jay Lin, Jing Zhang, Yanlong Zhai, and Bin Xu. The design and implementation of service process reconfiguration with end-to-end QoS constraints in SOA. *Service Oriented Computing and Applications*, 4(3):157–168, 2010.

[MA04] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26:369–395, 2004.

[Mar89] M. Ajmone Marsan. Stochastic petri nets: An elementary introduction. pages 1–29, 1989.

[MBB+89] Marco Ajmone Marsan, Gianfranco Balbo, Andrea Bobbio, Giovanni Chiola, Gianni Conte, and Aldo Cumani. The effect of execution policies on the semantics and analysis of stochastic petri nets. *IEEE Trans. Software Eng.*, 15(7):832–846, 1989.

[MC06] J. Misra and W.R. Cook. Computation Orchestration: A Basis for Wide-Area Computing. *Journal of Software and Systems Modeling*, May, 2006. Available for download at `http://dx.doi.org/10.1007/s10270-006-0012-1`.

[MC07] J. Misra and W. R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling, Springer*, 6(1):83–110, 2007.

[MCD08] Daniel A. Menascé, Emiliano Casalicchio, and Vinod K. Dubey. A heuristic approach to optimal service selection in Service Oriented Architectures. In Alberto Avritzer, Elaine J. Weyuker, and C. Murray Woodside, editors, *WOSP*, pages 13–24. ACM, 2008.

[Men02] Daniel A. Menascé. Qos issues in web services. *IEEE Internet Computing*, pages 72–75, 2002.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mis10] Jayadev Misra. Orc tutorial lectures. Technical report, Department of Computer Science, University of Texas at Austin, 2010.

[Mis11] Jayadev Misra. Virtual time and timeout in client-server networks. Technical report, University of Texas at Austin, 2011.

[MM07] Moreno Marzolla and Raffaela Mirandola. Performance prediction of web service workflows. In Sven Overhage, Clemens A. Szyperski, Ralf Reussner, and Judith A. Stafford, editors, *QoSA*, volume 4880 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2007.

[MM10] Moreno Marzolla and Raffaela Mirandola. Qos analysis for web service applications: a survey of performance-oriented approaches from an architectural viewpoint. Technical report, Department of Computer Science, University of Bologna, 2010.

[MMLP09] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18-25*, 2009.

[Mos94] Moshe Shaked and J. George Shanthikumar. *Stochastic Orders and their Applications*. Academic Press, 1994.

[Mos07] Moshe Shaked and J. George Shanthikumar. *Stochastic Orders*. Springer, 2007.

[MPLG10] Jerome Morio, Rudy Pastel, and Francois Le Gland. An overview of importance splitting for rare event simulation. *European Journal of Physics, IOP*, 31:1295–1303, 2010.

[MPR+10] Michele Mancioppi, Mikhail Perepletchikov, Caspar Ryan, Willem-Jan van den Heuvel, and Mike P. Papazoglou. Towards a quality model for choreography. In *ICSOC/ServiceWave 2009*, 2010.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–40, 1992.

[MRLD09] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Comprehensive qos monitoring of web services and event-based sla violation detection. In *MW4SOC, Urbana Champaign, Illinois*, 2009.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *IEEE*, 77:541–580, 1989.

[Mus04] Paul Muschamp. An introduction to web services. *BT Technology*, 22:9–18, 2004.

[MW77] S. Makridakis and S. Wheelwright. Adaptive filtering: An integrated autoregressive/moving average filter for time series forecasting. *Operational Research Quarterly*, 28(2):425–437, 1977.

[NRR06] Nilay Noyan, Gabor Rudolf, and Anrej Ruszczynski. Relaxations of linear programming problems with first order stochastic dominance constraints. *Operations Research Letters*, 34:653–659, 2006.

[Oas04] Oasis. Uddi version 3.0.2. Technical report, http://www.uddi.org/pubs/uddi_v3.htm, 2004.

[OAS07] OASIS. Web services business process execution language version 2.0. Technical report, OASIS Web Services Business Process Execution Language, 2007.

[OMG11] OMG. Business process model and notation (bpmn) standard ver 2.0. Technical report, Business Process Management Initiative, 2011.

[Ora11] Oracle. Business process management suite. Technical report, http://www.oracle.com/us/technologies/bpm/bpm-suite-078529.html, 2011.

[Orc11] Orc. Orc reference manual v2.0.2. Technical report, The University of Texas at Austin, 2011.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[Pau09] Cesare Pautasso. Restful web service composition with bpel for rest. *Data & Knowledge Engineering*, 69:851–866, 2009.

[PB08] Adrian Paschke and Martin Bichler. Knowledge representation concepts for automated sla management. *Decision Support Systems*, 46:187–205, 2008.

[PCE07] M. Di Penta, G. Canfora, and G. Esposito. Search-based testing of service level agreements. In *Proc. of the 9th Conf. on Genetic and evolutionary computation, London, England*, 2007.

[PD06] Yash Patel and John Darlington. A novel stochastic algorithm for scheduling qos-constrained workflows in a web service-oriented grid. In *International Conference Web Intelligence and Intelligent Agent Technology, on, pp.*, 2006.

[Pel03] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 3:46–52, 2003.

[PH12] T. Manjula Peiris and James H. Hill. Adapting system execution traces for validation of distributed system qos properties. In *15th IEEE Intl. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2012.

[PSK+10] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Automatic and scalable t-wise test case generation strategies for software product lines. In *Proc. of Intl. Conf. on Software Testing*, 2010.

[QTDC10] Lianyong Qi, Ying Tang, Wanchun Dou, and Jinjun Chen. Combining local optimization and enumeration for qos-aware web service composition. In *IEEE International Conference on Web Services*, 2010.

[Ran03] S. Ran. A model for web services discovery with qos. *ACM SIGecom Exch.*, 1:1–10, 2003.

[Rar98] R. L. Rardin. *Optimization in Operations Research*. Prentice Hall, 1998.

[RBHJ07] Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Probabilistic qos and soft contracts for transaction based web services. In *ICWS*, pages 126–133. IEEE Computer Society, 2007.

[RBHJ08] S. Rosario, A. Benveniste, S. Haar, and C. Jard. Probabilistic qos and soft contracts for transaction-based web services orchestrations. *IEEE Trans. on Services Computing*, 1(4):187–200, 2008.

[RBJ09a] Sidney Rosario, Albert Benveniste, and Claude Jard. Flexible probabilistic qos management of transaction based web services orchestrations. In *IEEE International Conference on Web Services*, pages 107 –114, 2009.

[RBJ09b] Sidney Rosario, Albert Benveniste, and Claude Jard. A theory of qos for web service orchestrations. Technical report, INRIA Research Report 6951, 2009.

[Rei92] Wolfgang Reisig. *A Primer in Petri Net Design*. Springer-Verlag, 1992.

[Rei08] Wolfgang Reisig. Towards a theory of services. In *Information Systems and e-Business Technologies*, volume 5 of *Lecture Notes in Business Information Processing*, pages 271–281. Springer Berlin Heidelberg, 2008.

[RK04] Reuven Y. Rubinstein and Dirk P. Kroese. *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Springer, 2004.

[RKB+07] Sidney Rosario, David Kitchin, Albert Benveniste, William R. Cook, Stefan Haar, and Claude Jard. Event structure semantics of orc. In Marlon Dumas and Reiko Heckel, editors, *WS-FM*, volume 4937 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2007.

[RKB+08] Sidney Rosario, David Kitchin, Albert Benveniste, William Cook, Stefan Haar, and Claude Jard. Event structure semantics of orc. In *WS-FM*, 2008.

[RLM+09] Florian Rosenberg, Philipp Leitner, Anton Michlmayr, Predrag Celikovic, and Schahram Dustdar. Towards composition as a service - a quality of service driven approach. In *IEEE International Conference on Data Engineering*, 2009.

[Ros09] Sidney Rosario. *Quality of Service issues in compositions of Web Services*. PhD thesis, Universite de Rennes 1, 2009.

[Saa80] T.L. Saaty. *The Analytic Hierarchy Process*. McGraw-Hill, New york, 1980.

[Saa03] Thomas L. Saaty. Decision-making with the ahp: Why is the principal eigenvector necessary. *European Journal of Operational Research*, 145:85–91, 2003.

[SAP06] SAP. Enterprise services architecture for healthcare - a prescription for innovation. *Solution Brief, Germany*, 2006.

[SDR09] Alexander Shapiro, Darinka Dentcheva, and Andrzej Ruszczyski. *Lectures on Stochastic Programming: Modeling and Theory*. Society for Industrial Mathematics, 2009.

[SEH02] Justin O' Sullivan, David Edmond, and Arthur Ter Hofstede. What's in a service? towards accurate description of non-functional service properties. *Distributed and Parallel Databases*, 12:117–133, 2002.

[SG03] Alper Sen and Vijay K. Garg. Partial order trace analyzer (pota) for distributed programs. *electronic Notes in Theoretical Computer Science*, 70:22–43, 2003.

[SHTB07] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks, Elsevier*, 51:456–479, 2007.

[Sig11] Signavio. Process editor - software as a service. Technical report, http://www.signavio.com/en/products/process-editor-as-a-service.html, 2011.

[SKD01] Biplav Srivastava, Subbarao Kambhampati, and Minh B. Do. Planning the project management way: Efficient planning by effective integration of causal and resource reasoning in realplan. *Artificial Intelligence*, 131:73–134, 2001.

[SLE04] James Skene, D. Davide Lamanna, and Wolfgang Emmerich. Precise service level agreements. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 179–188, 2004.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7:149–174, 1994.

[SMS+01] Akhil Sahai, Vijay Machiraju, Mehmet Sayal, Aad van Moorsel, and Fabio Casati. Automated sla monitoring for web services. In *DSOM*, 2001.

[SRE10] James Skene, Franco Raimondi, and Wolfgang Emmerich. Service-level agreements for electronic services. *IEEE Transactions on Software Engineering*, 36:288–304, 2010.

[ST07] N. Sato and Kishor S. Trivedi. Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2007.

[Tet77] Teturo Kamae and Ulrich Krengel and George L. O'Brien. Stochastic inequalities on partially ordered spaces. *The Annals of Probability*, 5(6):899–912, 1977.

[TF06] Ioan Toma and Douglas Foxvog. Non-functional properties in web services. Technical report, Web Services Modeling Ontology (WSMO) Final Draft, 2006.

[Tho90] Thomas L. Saaty. How to make a decision: the Analytic Hierarchy Process. *European Jounral of Operational Research*, 48(2):9–26, 1990.

[tHvdAAR10] A.H.M. ter Hofstede, W.M.P. van der Aalst, M. Adams, and N. Russell, editors. *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2010.

[Thy09] John A. Thywissen. Secure information flow in the orc concurrent programming language. Technical report, The University of Texas at Austin, 2009.

[TKO77] U. Krengel T. Kamae and G. L. O'Brien. Stochastic inequalities on partially ordered spaces. *Annals of Probabability*, 5:899–912, 1977.

[TLY+04] Zhangxi Tan, Chuang Lin, Hao Yin, Ye Hong, and Guangxi Zhu. Approximate performance analysis of web services flow using stochastic petri net. In *Grid and Cooperative Computing - GCC 2004*, volume 3251 of *Lecture Notes in Computer Science*, pages 193–200. Springer Berlin / Heidelberg, 2004.

[TP02] Aphrodite Tsalgatidou and Thomi Pilioura. An overview of standards and related technology in web services. *Distributed and Parallel Databases*, 12:135–162, 2002.

[TP05] Vladimir Tosic and Bernard Pagurek. On comprehensive contractual descriptions of web services. In *Proceedings of the IEEE e-Technology, e-Commerce, and e-Service*, pages 444–449, 2005.

[TZCB08] W.T. Tsai, Xinyu Zhou, Yinong Chen, and Xiaoying Bai. On testing and evaluating service-oriented software. *IEEE Computer*, 8:40–46, 2008.

[USM06] K. Munagala U. Srivastava, J. Widom and R. Motwani. Query optimization over web services. In *Intl. Conf. on Very Large Databases*, 2006.

[vdA96] Wil M. P van der Aalst. Three good reasons for using a petri-net-based workflow management system. In Proc. of the Intl. Working Conf. on Information and Process Integration in Enterprises, 1996.

[vdA97] Wil M. P. van der Aalst. Verification of workflow nets. In *ICATPN*, pages 407–426, 1997.

[vdA98] Wil M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[vdABL08] Wil M. P. van der Aalst and Kristian Bisgaard Lassen. Translating unstructured workflow processes to readable bpel: Theory and implementation. *Inf. Softw. Technol.*, 50(3):131–159, 2008.

[vdAtHKB02] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workfow patterns. Technical report, Eindhoven University of Technology, Netherlands, 2002.

[vdAtHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[vdAvH02] Wil M. P. van der Aalst and Kees M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[W3c00] W3c. Simple object access protocol (soap) 1.1. Technical report, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, 2000.

[W3c01] W3c. Web services description language (wsdl) 1.1. Technical report, http://www.w3.org/TR/wsdl, 2001.

[W3c02] W3c. Web service choreography interface (wsci) 1.0. Technical report, http://www.w3.org/TR/wsci/, 2002.

[W3c03] W3c. Qos for web services: Requirements and possible approaches. Technical report, W3C Working Group Note, Nov. 2003.

[W3c04a] W3c. Owl-s: Semantic markup for web services. Technical report, http://www.w3.org/Submission/OWL-S/, 2004.

[W3c04b] W3c. Web services architecture. Technical report, W3C Working Group, 2004.

[Wan98] Jiacun Wang. *Timed Petri Nets*. Springer International Series on Discrete Event Dynamic Systems, 1998.

[WHK08] Wolfram Wiesemann, Ronald Hochreiter, and Daniel Kuhn. A stochastic programming approach for qos-aware service composition. In *Eighth IEEE International Symposium on Cluster Computing and the Grid*, 2008.

[Win86] Glynn Winskel. Event structures. *Advances in Petri Nets*, pages 325–392, 1986.

[WKCM08] Ian Wehrman, David Kitchin, William R. Cook, and Jayadev Misra. A timed semantics of orc. *Theoretical Computer Science*, 402:234–248, 2008.

[WVKT06] Xia Wang, Tomas Vitvar, Mick Kerrigan, and Ioan Toma. A qos-aware selection model for semantic web services. In *ICSOC*, 2006.

[WWW+07] Changzhou Wang, Guijun Wang, Haiqin Wang, Alice Chen, and Rodolfo Santiago. Quality of service contract specification, establishment, and monitoring for service level management. *Journal of Object Technology: Special Issue on Advances in Quality of Service Management*, 6(11):25–44, 2007.

[WZZF09] Kaibo Wang, Xingshe Zhou, Shandan Zhou, and Ning Fu. Simplify stochastic qos admission test for composite services through lower bound approximation. In *IEEE International Conference on Services Computing*, 2009.

[XFZ08] Peng Cheng Xiong, Yu Shun Fan, and Meng Chu Zhou. Qos-aware web service configuration. *IEEE Trans. on Systems, Man and Cybernetics*, 38:888–895, 2008.

[XSCT02] Wei Xie, Hairong Sun, Yonghuan Cao, and Kishor S. Trivedi. Optimal webserver session timeout settings for web users. In *Computer Measurement Group Conference*, pages 799–820, 2002.

[YB07] Qi Yu and Atman Bouguettaya. Framework for web service query algebra and optimization. *ACM Transactions on the Web*, 5:1–34, 2007.

[YB08] Qi Yu and Athman Bouguettaya. Framework for Web service query algebra and optimization. *TWEB*, 2(1), 2008.

[YKL+07] Jun Yan, Ryszard Kowalczyk, Jian Lin, Mohan B. Chhetri, Suk Keong Goh, and Jianying Zhang. Autonomous service level agreement negotiation for service composition provision. *Future Gener. Comput. Syst.*, 23:748–759, July 2007.

[YL05] Tao Yu and Kwei-Jay Lin. Service Selection Algorithms for Composing Complex Services with Multiple QoS Constraints. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *ICSOC*, volume 3826 of *Lecture Notes in Computer Science*, pages 130–143. Springer, 2005.

[YRB+10] Qi Yu, Manjeet Rege, Athman Bouguettaya, Brahim Medjahed, and Mourad Ouzzani. A two-phase framework for quality-aware web service selection. *SOCA*, 4:63–79, 2010.

[YSS05] H. Jeong Y. Seo and Y. Song. Best web service selection based on the decision making between qos criteria of service. In *Intl. Conf. on Embedded Soft. and Sys.*, 2005.

[YZL07] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web*, 1, 2007.

[ZBD+03] Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. Quality driven web services composition. In *World Wide Web Conference*, 2003.

[ZBN+04] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30:311–326, 2004.

[ZLC07] Liangzhao Zeng, Hui Lei, and Henry Chang. Monitoring the qos for web services. In *ICSOC*, 2007.

[ZNB+08] Liangzhao Zeng, Anne H.H. Ngu, Boualem Benatallah, Rodion Podorozhny, and Hui Lei. Dynamic composition and optimization of web services. *Distrib Parallel Databases*, 24:45–72, 2008.

[Zsc10] Steffen Zschaler. Formal specification of non-functional properties of component-based software systems. *Softw Syst Model*, 9:161–201, 2010.

[ZYZB11] Huiyuan Zheng, Jian Yang, Weiliang Zhao, and Athman Bouguettaya. Qos analysis for web service compositions based on probabilistic qos. In Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari-Nezhad, editors, *ICSOC*, volume 7084 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2011.

[ZZ04] Tanja Zseby and Sebastian Zander. Sampling schemes for validating service level agreements. Technical report, Centre for Advanced Internet Architectures, 2004.

List of Figures

1.1 Un ensemble de service composite orchestration démontrant des composants. 19
1.2 Une *non-monotone* orchestration. 20

2.1 Clients invoke web services be exchanging SOAP messages. WSDL specifications are compiled into stubs and skeletons. Providers advertise their services on a UDDI registry that can be queried by the clients [ACKM04]. 40
2.2 An *Orchestration* refers to an executable process while a *Choreography* tracks the message sequences between parties and sources [Pel03]. 45
2.3 A BPEL process flow [Pel03]. 49
2.4 Aggregating QoS metrics for various domains and operations [CPEV05a]. 56
2.5 A composite service orchestration demonstrating assembly of components. 57
2.6 A *non-monotonic* orchestration. 58
2.7 Response time distribution of an API. 60
2.8 Role of a Web Service Level Agreement [LKD+03]. 62

3.1 TravelAgent1: Simple travel agent; (a) informal diagram, and (b) Petri net form, where rectangles figure transitions and rounded rectangles figure places. This orchestration has a data-independent workflow. 81
3.2 TravelAgent2: A variation of TravelAgent1 having a data-dependent workflow. 82
3.3 A simple orchestration where monotonicity does not hold; the two alternatives are selected on the basis of best cost of air fare. 83
3.4 TravelAgent3: A variation of TravelAgent2 lacking monotonicity. 83
3.5 A simple example. Only QoS values are mentioned — with no data. Each place comes labeled with a QoS value q which is the q-color of the token if it reaches that place. 90
3.6 Enforcing monotonicity through service aggregation, mid diagram, with $\delta q'_{12} = \delta q'_1 \oplus \delta q'_2$ and $\delta q''_{12} = \delta q''_1 \oplus \delta q''_2$. Pessimistic QoS evaluation, right diagram, with $\delta q_{12} = \delta q'_{12} \vee \delta q''_{12}$. 100
3.7 Separation of concerns in QoS-aware specification. The functional specification is depicted last in **boldface**, whereas the QoS part is shown in *italics* on top in the form of a rich SLA specification. 103
3.8 Rewriting rule for weaving response time. 104
3.9 Rewriting rule for weaving cost. 104
3.10 Declaration of the SLA for the TravelAgent2 orchestration. 108
3.11 Orc functional specification. 109
3.12 Orc QoS-weaved specification 110
3.13 We show results from two experiments. For each experiment we display cumulative densities of: (a) Measured latency of invoked services (b) End-to-end latency for TravelAgent 2/3 orchestrations through two evaluation schemes (c) Measured cost of invoked services (d) Returned cost invoice of TravelAgent 2/3 orchestrations. 111

4.1 Technique to enrich Orc outputs with Causality. 119
4.2 The basic rules 120
4.3 The complete rules 122
4.4 OIL Rewriting to include Causal Information. 123

5.1 Feature Diagram / Model of the Crisis Management System with associated real-world service assets. 137
5.2 Composite Web Service Orchestration of the CMS. 139
5.3 Distribution fitting of actual response times of a web service invocation. 141
5.4 Varying configurations of the atomic services (a) Configuration **C1** (b) Configuration **C15**. 142
5.5 Response times of the pairwise configurations with emphasis on comparing the three configurations with highest response times. 143
5.6 Box-plot representation of the pairwise configurations with the median values marked for the extreme cases. 144
5.7 Comparison of pairwise and exhaustive generation of configurations with 25, 50, 75 and 90 percentile values of response time distributions. 145
5.8 Percentile values of most deviant scenarios generated by pairwise interactions for the CMS orchestration. 146
5.9 Comparison of two pairwise samples with 25, 50, 75 and 90 percentile values of response time distributions. 147

6.1 C^3MS Feature Diagram. 157
6.2 Composite Service Orchestration of the C^3MS. 158
6.3 eHealth Feature Diagram. 159
6.4 Composite Web Service Orchestration of the eHealth system. 160
6.5 Response time distributions of the 185 pairwise configurations for C^3MS. 162
6.6 Availability, Data Quality and Cost of the pairwise configurations of C^3MS. 162
6.7 Response time distributions of the 188 pairwise configurations for eHealth. 163
6.8 Availability, Data Quality and Cost of the pairwise configurations of eHealth. 164
6.9 Three runs of random generation of configurations for C^3MS. 164
6.10 Comparison of pairwise and random response time (arranged in increasing order) of percentile values for 185 configurations of C^3MS. 165
6.11 Comparing stability of pairwise and random samples for eHealth. 166

7.1 Comparison Scale for AHP [Saa80]. 175
7.2 Architecture of the Dell example. 177
7.3 QoS interactions in the Dell supply chain. 178
7.4 Output of the Orc program plugged into the LPSolve IDE. 183
7.5 Optimal setting of parameters in the Dell Supply Chain. 184
7.6 Distributions of the inventory levels in the Dell system. 184
7.7 Optimization output for a single setting of the Dell example in MATLAB. 185

8.1 Architecture of the Dell example. 194
8.2 Inter-query period distributions and fitting. 196
8.3 Response time distributions and fitting. 196
8.4 Measured response time and thresholds (without distribution fitting). . 197
8.5 Assumption: *Plant* side demand distributions. 197
8.6 Guarantee: *Supplier* side procurement delays. 198

9.1 The `GarageOnline` orchestration. 209

9.2 Latency improvements with 10 rounds of negotiation for `GarageOnline` (a) Using optimization strategy (b) Random strategy. The Fig. (c) shows Cost incurred with 10 rounds of negotiation. 213
9.3 Improvements in performance of individual services after 10 rounds of optimization strategy. 214
9.4 An example of an optimization run in MATLAB. 215

10.1 QoS enhanced Orc output. 218
10.2 Platform for contract based management of monotonic orchestrations. . 225
10.3 Distribution fitting in MATLAB. 226
10.4 Web service invocation via SOAP. 226

Résumé

Les services Web sont des applications logicielles avec des implémentations hétérogènes, dont les interfaces et les incarnations peuvent être définis, décrits et découverts sur un réseau. Une orchestration de tels services Web fournit un flux de contrôle centralisé pour les services composites, qui peuvent invoquer d'autres services en utilisant une série de constructions (séquentielle, parallèle, avec des timeouts par exemple). L'objectif de cette thèse est d'étudier l'effet des paramètres de Qualité de Service (QoS) dans la performance et les obligations contractuelles de ces orchestrations. Tout d'abord, nous générons un modèle précis pour étudier la QoS probabiliste multi-dimensionnelle dans les services Web. Lorsque la dépendance des données est présente dans les orchestrations, les conditions pour assurer la monotonie sont nécessaires et sont intégrées. Nous présentons une algèbre riche pouvant gérer plusieurs dimensions de la QoS et fournir une composition de contrat probabiliste. Une conséquence de cela est l'«entrelacement» des paramètres de QoS en spécifications fonctionnelles des orchestrations, qui peut fournir d'autres fonctionnalités intéressantes comme l'ordonnancement causal d'un flux de contrôle d'orchestration. Ensuite, nous étudions les effets de ces modèles de QoS sur la gestion améliorée des SLAs (Service Level Agreement). Les applications de cette architecture de gestion de la QoS sont diverses et comprennent la prise en compte de la variabilité au sein de la gamme de produits, des progiciels mathématiques pour la prise de décision, des techniques de simulation avancées pour quantifier les contrats et des protocoles de négociation améliorés. Certaines de ces techniques sont implémentées au dessus d'Orc, un langage de programmation concurrente ayant des constructions pour gérer plusieurs aspects de spécifications d'orchestration.

Mots Clés: QoS, Orchestrations de Services Web, SLA, Orc.

Abstract

Web Services are software applications that have heterogeneous implementations, whose interfaces and bindings are capable of being defined, described and discovered over a network. An orchestration of such web services provides a centralized control flow for composite services, that can invoke other services using a series of constructs (sequential, parallel, with timeouts for instance). The focus of this thesis is to study the effect of Quality of Service (QoS) metrics in the performance and contractual obligations of such orchestrations. Firstly, we generate an accurate model to study probabilistic multi-dimensional QoS in web services. When data dependency is involved in orchestrations, conditions to ensure monotonicity are necessary and are incorporated. A rich algebra is presented that can handle multiple dimensions of QoS and provide probabilistic contract composition. A consequence of this is "weaving" QoS metrics into functional specifications of orchestrations, that can provide other interesting features such as the causal history of an orchestration control flow. Secondly, we study the effects of such models of QoS on improved Service Level Agreement (SLA) management. From incorporating product line variability and mathematical packages for decision making, to superior simulation techniques to quantify contracts and improved negotiation protocols: these are the applications of the QoS management framework. Some of these techniques are implemented over Orc, a concurrent programming language with constructs to handle multiple features of orchestration specifications.

Key Words: QoS, Web Services Orchestrations, SLA, Orc.

VU :

Le Directeur de Thèse
Albert BENVENISTE

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
Jean-Marc JEZEQUEL