

Université HASSAN 1^{er}

Faculté des Sciences et techniques

Thèse de Doctorat

Spécialité Informatique

Présentée par

Adil MAAROUF

Quality of Service and Service Level agreement in the Cloud Computing

Soutenue le 28 /09 /2017

Membre du jury :

HAQIQ Abdelkrim	Professeur à la FST Settat	Président
MARZOUK Abderrahim	Professeur à la FST Settat	Directeur de thèse
SAFI Said	Professeur à la Faculté poly. Beni Mellal	Rapporteur
BENI-HSSANE Abderrahim	Professeur Habilité à la Faculté des sciences El Jadida	Rapporteur
LAACHFOUBI Nabil	Professeur Habilité à la FST Settat	Rapporteur
EZZATI Abdellah	Professeur Habilité à la FST Settat	Examineur
JAMALI Abdellah	Professeur Habilité à l'EST de Berrchid	Examineur

2017

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

Acknowledgements

In the name of Allah, the Most Gracious and the Most Merciful. Alhamdulillah, all praises go to Allah on whom ultimately we depend for sustenance and guidance. Alhamdulillah for all the strengths and blessing He gave me in completing this thesis.

I would like to thank all the people who have provided me with valuable advice and support throughout my doctoral study.

The research work of this doctoral thesis has been carried out in Laboratory of Computer, Networks, Mobility and Modeling Laboratory, Department of Mathematics and Computer, FST, Hassan 1st University, Settat, Morocco; During the years 2013-2017.

First and foremost, I would like to express my sincere and deepest gratitude, appreciation, and special acknowledgments to my supervisor Pr. Abderrahim MARZOUK for always being supportive and encouraging throughout my thesis, and for his assistance in the preparation of this manuscript. He was very kind in granting me from his time and knowledge.

I especially thank the members of my thesis jury chaired by Pr. Abdelkrim HAQIQ from FST Settat, Hassan 1st University, who agreed to judge this work and to participate in the jury.

I would like to thank Pr. SAFI Said from Faculty Polytechnic de Beni Mellal, Pr. BENI-HSSANE Abderrahim from Faculty of Sciences El Jadida and Pr. LAACHFOUBI Nabil from FST Settat, Hassan 1st University, who agreed to report this work.

I also thank Pr. EZZATI Abdellah from FST Settat, Hassan 1st University and Pr. JAMALI Abdellah from EST Berrchid who agreed to examine this thesis.

My deepest thanks go to my loving parents for always being there for me and constantly providing me with love and encouragement. Without their continuous emotional support and encouragement, I would never have been able to complete my works. They have done so much that a simple thanks to them will not suffice. Last but not least, special thanks to my wife, Bouchra, for all her unconditional support and understanding during the long period of this research; to my wonderful son, Ayman, for his patience who always succeed to bring the

smile to my face and to my brothers for their continuous encouragement and support. You have all been my source of joy. To them I dedicate this thesis.

Abstract

The revolutionary technology of Cloud Computing offers a scalable and flexible paradigm where infrastructure, platform, and software are offered to users in the form of services. The management of such environment is inherently complex due to the large-scale number and heterogeneity of resources and the increasing number and types of services the cloud must deliver over the Internet. However, due to the highly dynamic nature of this open environment, it is important to have a binding agreement between all the service parties for ensuring trust while fulfilling the expected Quality of Service (QoS). The provisioning of these computing services by cloud providers are regulated by Service Level Agreements, which present an important element that provides some degree of assurance. In this context, SLA contract forms an appropriate solution to specify these QoS guarantees. It specifies one or more service level objectives (SLO), to guarantee that the delivered QoS satisfies the consumer expectations.

To properly operate and manage such complex situations, in this thesis, our main objectives are to propose an efficient method for automation of SLA contract establishment process, and to provide an autonomic SLA monitoring mechanism managed by a trusted third party for real time ensuring accountability and QoS in cloud services.

Firstly, for automating SLAs contract establishment, we investigate this issue and propose our conception by using Model Driven Engineering (MDE) to express the SLA contract requirements, as a mean for establishing service level agreements between a cloud provider and cloud customer in the context of a particular service provision. The cloud SLA contract language we present is a meta-model for defining cloud SLAs specifically defined for the cloud computing domain, that supports most of the characteristics of cloud. We evaluate the completeness and clarity of our SLA contract establishment for Conceptual Modeling with Bunge-Wand-Weber representational theory. We also validate our meta-model to examine the ability of the proposed SLA establishment approach to generate the SLA document in less time.

Secondly, we propose a Novel Penalty Model for calculating the penalty cost of the violations, that can efficiently help for managing and applying penalties in the cloud computing.

Lastly, we propose an autonomic SLA monitoring framework managed by trusted third party composed of two modules: the first one SLA establishment module, which aims at providing support for automated SLA generation and management. The second one, a service monitoring module to dynamically monitor QoS metrics by detecting SLA violations at runtime to verify compliances for the respective SLAs, and to propose a mechanism for an adaptive remedy rectification. The trusted third party framework equipped with such reputation module can provide real-time assessment for consumers informed decision making to continue using a service or to migrate to another service provider in the case of service degradation. This creates a fair competitiveness between providers and hence improves service performance and the reliability in the cloud.

keywords: Cloud Computing; Service Level Agreements; Quality of Service; Model Driven Engineering; Autonomic SLA Monitoring; Trusted Third Party; Autonomic MAPE-K Cycle; Violation; Penalty Cost Calculation; Reputation.

Résumé

La technologie révolutionnaire du Cloud Computing offre un paradigme évolutif et flexible où l'infrastructure, la plate-forme et les logiciels sont offerts aux utilisateurs Cloud sous forme de services. La gestion de cet environnement est intrinsèquement complexe en raison du nombre et de l'hétérogénéité des ressources et du nombre croissant et des types de services que le cloud doit fournir sur Internet. Cependant, en raison de la nature très dynamique de cet environnement ouvert, il est important d'avoir un accord contraignant entre toutes les parties du service pour assurer la confiance tout en remplissant la qualité de service attendue. La fourniture de ces services informatiques par des fournisseurs de cloud sont réglementées par des accords de niveau de service, qui présentent un élément important qui offre un certain degré d'assurance. Dans ce contexte, le contrat SLA constitue une solution appropriée pour spécifier ces garanties qualité de service (QoS). Il spécifie un ou plusieurs objectifs de niveau de service (SLO), afin de garantir que la QoS fournie répond aux attentes des consommateurs. Associée à ce nombre croissant de services basés sur le cloud, cette complexité a généré une demande urgente pour automatiser l'établissement de contrat SLA et des solutions de surveillance efficaces. Cependant, malgré son apport et intérêt capital, les SLA pour les services cloud manquent de normalisation et constituent toujours un défi pour les chercheurs, ce qui motive l'automatisation de leur traitement. Pour opérer correctement et gérer une telle situation complexe, une approche efficace pour l'automatisation du processus d'établissement de contrat SLA et un mécanisme de surveillance autonome du SLA géré par un tiers de confiance en temps réel garantissant la responsabilité et la QoS dans les services cloud est crucial.

L'objectif de cette thèse est de proposer un langage qui permet l'établissement de contrat SLA qui supporte l'instabilité de la QoS dans un environnement hautement dynamique comme le Cloud en gérant finement les violations, plus un système de surveillance autonome géré par un tiers de confiance afin de résoudre les conflits entre les signataires potentiels, et pour surveiller les violations SLA en temps réel dans le but d'assurer la surveillance en ligne des services en nuage.

A cet effet, les contributions de cette thèse se focalisent sur l'établissement et le contrôle autonome de contrat SLA en temps réel qui est devenu un problème critique pour le Cloud Computing.

Ces contributions peuvent se résumer comme suit :

Premièrement, pour automatiser l'établissement de contrat SLA, nous proposons un modèle pour exprimer les exigences de contrat de niveau de service en utilisant l'ingénierie Dirigée par les Modèles (IDM), comme un moyen pour établir des accords de niveau de service entre un fournisseur et un client de nuage dans le contexte d'une disposition de service cloud particulière. Ensuite, nous abordons le processus de construction et description du méta-modèle de contrat SLA utilisé dans notre support d'outils de preuve de concept. Pour valider notre méta-modèle, nous fournissons une description de la théorie de représentation Bunge-Wand-Weber. Nous évaluons également l'exhaustivité et la clarté de contrat SLA pour la modélisation conceptuelle. Nous présentons l'évaluation de l'expressivité du langage de modélisation proposé et de l'efficacité de l'approche d'établissement de SLA proposée. Les résultats indiquent que notre méta-modèle est capable d'obtenir une exhaustivité et une clarté satisfaisantes, ce qui est très prometteur pour décrire avec précision la structure et le comportement du monde réel. Enfin, nous validons notre méta-modèle pour examiner la capacité de l'approche d'établissement de contrat SLA proposée à générer le document SLA en moins de temps.

Deuxièmement, pour ce qui est l'application de sanctions qu'est devenue une question cruciale pour l'informatique de nuages, nous proposons un nouveau modèle de pénalité pour calculer le coût de pénalité des violations. Nous étudions et nous présentons les définitions les plus fréquemment utilisées des fonctions de pénalités SLA. Nous identifions les caractéristiques de ces fonctions tout en mettant en évidence leurs forces et leurs faiblesses. Nous enquêtons et analysons diverses méthodes de calculs de pénalités et de disponibilité des fournisseurs de cloud. Nous proposons également un modèle d'affaires pour les fournisseurs de cloud pour gérer leur profit. Un exemple d'application sera présenté pour montrer l'efficacité de l'approche proposée.

Enfin, pour le contrôle et surveillance autonome du contrat de niveau de service, nous présentons deux contributions :

Nous proposons un framework autonome de surveillance du SLA géré par un tiers de confiance composé de deux modules: le premier module d'établissement de contrat SLA, vise à fournir un support pour la génération et la gestion automatiques de SLA. Deuxièmement, un module de surveillance de service cloud pour contrôler dynamiquement les métriques QoS en détectant les violations de SLA au moment de l'exécution pour vérifier les concordances des SLA respectifs et pour proposer un mécanisme pour une rectification de remède adaptatif,

en tant que contribution au troisième niveau de maturité du paradigme de l'informatique autonome tel que défini par IBM.

Cette thèse est organisée en six chapitres. Le **premier chapitre** présente une vue générale sur l'importance de la recherche dans le cloud computing, le contrôle et l'établissement du contrat de niveau de service et l'ingénierie Dirigée par les Modèles, qui sont utilisés pour mener à bien la recherche dans cette thèse et pour identifier les lacunes de recherche dans ce domaine. Dans ce chapitre, nous définissons la problématique et la motivation de notre travail ainsi que nos contributions au domaine.

Le **chapitre 2** donne une vue d'ensemble du cloud computing comme base pour développer des modèles de SLA en nuage, y compris ses caractéristiques et ses classifications. Nous proposons un aperçu des concepts généraux et des caractéristiques de cloud, puis nous décrivons les différents mécanismes de déploiement et la catégorisation de services cloud. Plus, nous discutons les terminologies des acteurs de la communication en nuage. Ensuite, nous donnons une description globale du contrat de niveau de service représentés par des langages spécialisés de spécifications SLA. Les SLAs sont au coeur de cette thèse, servant de point de départ de toutes les solutions proposées dans divers domaines que cette thèse aborde. Dans le même cadre, nous soulignons l'importance de l'ingénierie Dirigée par les Modèles et ses concepts liés à nos objectifs. Nous enchaînons avec une présentation du concept de tiers de confiance dans le cloud computing. Nous mettons en avant la nécessité de montrer l'importance de l'informatique autonome pour le cloud, expliquant le rôle essentiel du SLA dans le contexte des systèmes autonomes et dans la gestion des nuages. Finalement, nous fournissons une étude comparative des simulateurs pour le cloud computing en identifiant les caractéristiques des outils mettant en évidence leurs forces et leurs faiblesses. L'objectif de cette revue est de mieux comprendre les approches et méthodes utilisées et ainsi pouvoir identifier les plus appropriées pour notre thèse

Le **chapitre 3** s'intéresse à l'automatisation du processus d'établissement de contrat SLA. Dans un premier lieu, nous proposons un modèle pour exprimer les exigences de contrat de niveau de service en utilisant l'ingénierie Dirigée par les Modèles (IDM), comme un moyen pour établir les accords de niveau de service entre un fournisseur et un client de nuage dans le contexte d'une disposition de service cloud particulière. Ce chapitre présente non seulement le méta-modèle pour décrire le contrat SLA en nuage, mais aussi le processus par lequel il a été développé, y compris les différentes étapes du méta-modèle qui ont été élaborées. Nous commençons par une section introductive. Après, nous présentons certains travaux connexes sur l'établissement du SLA, qui énonce nos motivations et décrit notre approche. Ensuite, nous abordons le processus de construction et description du méta-modèle contrat SLA utilisé dans notre support d'outils de preuve de concept. Pour valider notre méta-

modèle, nous fournissons une description de la théorie de représentation Bunge-Wand-Weber. Nous évaluons également l'exhaustivité et la clarté de contrat SLA pour la modélisation conceptuelle. Les résultats indiquent que notre méta-modèle est capable d'obtenir une exhaustivité et une clarté satisfaisantes, ce qui est très prometteur pour décrire avec précision la structure et le comportement du monde réel. Enfin, nous validons notre méta-modèle pour examiner la capacité de l'approche d'établissement de contrat SLA proposée à générer le document SLA en moins de temps. Enfin, le chapitre est conclu avec un résumé de nos contributions et de leurs futures extensions potentielles.

Le **chapitre 4** introduit un nouveau modèle de pénalité pour calculer le coût de pénalité des violations. Ce modèle est basé sur une fonction avancée de calcul de pénalité prenant en considération les caractéristiques du Cloud. Nous commençons par un aperçu du contexte dans le domaine concernant cette thématique. Ensuite, nous présentons un aperçu de certaines approches pour la définition des fonctions de pénalité SLA. Ces différentes approches, cependant, ne satisfont pas à toutes les exigences pour formuler des expressions de pénalités complexes dans un seul modèle non ambigu. Nous étudions et nous présentons les définitions les plus fréquemment utilisées des fonctions de pénalités SLA. Nous identifions les caractéristiques de ces fonctions tout en mettant en évidence leurs forces et leurs faiblesses. Nous enquêtons et analysons diverses méthodes de calculs de pénalités et de disponibilité des fournisseurs de cloud. Nous proposons également un modèle économique pour les fournisseurs de cloud pour gérer leur profit. Un exemple d'application sera présenté pour montrer l'efficacité de l'approche proposée. Finalement, le travail relève quelques défis et orientations futures de la recherche.

Le **chapitre 5** s'adresse au contrôle et la surveillance autonome du contrat de niveau de service. Pour révéler notre motivation et notre contribution, il est utile de revoir certains travaux de recherche sur les systèmes de surveillance de cloud, les frameworks et les architectures. Après, nous donnons une vision globale sur la recherche lancée dans les systèmes de Réputation. Par la suite, nous abordons certaines approches concernant le tiers de confiance pour surveiller les accords de niveau de service en temps réel. Sur ce, nous proposons un framework autonome de surveillance du SLA géré par un tiers de confiance composé de deux modules: le premier module d'établissement de contrat SLA, vise à fournir un support pour la génération et la gestion automatique de SLA. Deuxièmement, un module de surveillance de service cloud pour contrôler dynamiquement les métriques QoS en détectant les violations de SLA au moment de l'exécution pour vérifier les concordances des SLA respectifs et pour proposer un mécanisme pour une rectification de remède adaptatif, en tant que contribution au troisième niveau de maturité du paradigme de l'informatique autonome tel que défini par IBM. Le framework est validé avec des scénarios sur le temps de réponse et

la disponibilité, les résultats obtenus sont prometteurs. Ils confirment que notre framework gère les SLAs de manière efficace, car il détecte toutes les violations à communiquer aux parties concernées et identifie les clauses de pénalités particulières qui peuvent être utilisées pour modifier la réputation d'un fournisseur au fil du temps. Le framework équipé d'un tel module de réputation peut fournir une évaluation en temps réel pour aider les consommateurs à la prise de décision éclairée pour continuer à utiliser un service ou de migrer vers un autre fournisseur de services en cas de dégradation du service. Cela crée une compétitivité équitable entre les fournisseurs et améliore ainsi les performances des services et la fiabilité de cloud. Enfin, le chapitre se termine par un résumé de nos contributions et de leurs futures extensions potentielles.

Enfin, le **chapitre 6** présente une conclusion générale pour clore notre mémoire. Celle-ci va nous permettre, de revoir en globalité les travaux accomplis, de mettre en évidence les résultats obtenus, et de donner quelques horizons de recherche qui restent ouverts. Une liste des publications produites pendant cette thèse est fournie à la fin de ce document. Des annexes complètent ce document pour en illustrer certaines parties : l'Annexe A présente le code source de notre framework autonome de surveillance du SLA et l'Annexe B illustre en EBNF la grammaire Xtext.

Mots-clés: Cloud Computing, Qualité de Service (QoS), Contrat SLA (Service Level Agreement), Ingénierie Dirigée par les Modèles, Tiers de confiance, Boucle autonome MAPE-K, Surveillance en temps réel, Violation, Réputation.

Contents

List of Figures	19
List of Tables	23
1 Introduction	1
1.1 Overview	1
1.2 Problem Definition	4
1.3 Research Questions	5
1.4 Motivation	7
1.5 Research Hypothesis	8
1.6 Research Objectives	9
1.7 Thesis Contributions	10
1.8 Thesis Organization	12
2 Background and Literature Review	15
2.1 Cloud Computing	15
2.1.1 An overview of Cloud computing paradigm	16
2.1.1.1 Essential characteristics of cloud computing	17
2.1.1.2 Service delivery models of cloud computing	19
2.1.1.3 Deployment models of cloud computing	20
2.1.1.4 Cloud communication actors	21
2.2 Service Level Agreements	22
2.2.1 QoS Metrics Related to an SLA and their Categorization	25
2.2.2 SLA Life Cycle and uses for Cloud Computing	28
2.2.2.1 Discover - Service Provider	29
2.2.2.2 Define - SLA	30
2.2.2.3 Establish - Agreement	30
2.2.2.4 Monitor - SLA Violation	31

2.2.2.5	Terminate - SLA	31
2.2.2.6	Enforce Penalties for SLA Violation	32
2.2.3	SLA Life Cycle : Functionalities and Use Cases	32
2.2.3.1	SLA Discovery	32
2.2.3.2	SLA Template Management	33
2.2.3.3	SLA Negotiation Scenario	33
2.2.3.4	SLA Establishment	35
2.2.3.5	SLA Monitoring and Violation	35
2.2.3.6	SLA Conclusion	36
2.2.4	Overview of the Existing SLA Definition Languages	38
2.2.4.1	SLA @ Web services	38
2.2.4.2	SLA @ Cloud	44
2.2.5	Comparison of the SLA definition languages	48
2.3	Model-Driven Engineering	50
2.3.1	Model-Driven Engineering and Model-Driven Architecture	50
2.3.2	Domain-Specific Modeling	51
2.3.3	Model, Metamodel and Modeling, Metamodeling	51
2.3.3.1	Models	52
2.3.3.2	Metamodel	53
2.3.4	Model management	54
2.3.4.1	Model Transformation	55
2.3.4.2	Model Comparison	55
2.3.5	Model Driven Engineering Tools	56
2.3.5.1	EMF	56
2.3.5.2	Epsilon	57
2.3.5.3	Epsilon Object Language (EOL)	57
2.3.5.4	Epsilon Comparison Language (ECL)	57
2.3.5.5	Epsilon Merging Language (EML)	58
2.3.5.6	Epsilon Transformation Language (ETL)	61
2.3.5.7	Epsilon Generation Language (EGL)	61
2.4	Trusted third parties in cloud computing	61
2.5	Autonomic Computing	63
2.5.1	The Autonomic Computing paradigm and maturity levels	65
2.5.1.1	The MAPE-K cycle of the AC paradigm	65
2.5.1.2	The maturity levels of the AC paradigm	65
2.6	Comparative Study of Simulators for Cloud Computing	67

2.6.1	Introduction	67
2.6.2	Simulation tools for Cloud Computing	68
2.6.2.1	CloudSim	68
2.6.2.2	CloudNetSim++	70
2.6.2.3	Cloud Analyst	70
2.6.2.4	CDOSim	72
2.6.2.5	EMUSIM	73
2.6.2.6	NetworkCloudSim	73
2.6.2.7	TeachCloud	74
2.6.2.8	DCSim	74
2.6.2.9	iCanCloud	75
2.6.2.10	GreenCloud	77
2.6.2.11	Open Cloud Testbed	78
2.6.2.12	GroudSim	78
2.6.2.13	PerfCloud	79
2.6.2.14	SimMapReduce	79
2.6.2.15	SimIC	79
2.6.3	Comparative Study of Simulators : Characteristics and Limitations .	81
2.7	Chapter Summary	82
3	SLA Contract Meta-model: A Language for the Definition of SLAs for Clouds	85
3.1	Introduction	86
3.2	Related works: Comparison between SLA contract meta-model and the Existing SLA Languages	88
3.3	Constructing the cloud SLA contract meta-model: Modeling languages . .	92
3.3.1	Cloud provider meta-model, offer specification language	93
3.3.2	Cloud consumer meta-model, specification language	94
3.3.3	SLA contract meta-model: definition language	95
3.4	Generation process: for SLA contract establishment generation model . . .	98
3.4.1	General Process of Correspondence and Composition	99
3.4.2	Implementation of Matching Logic in Epsilon	102
3.4.3	Implementation of Composition-Merging Logic with Epsilon Merging Language	105
3.4.4	Xtext: Grammar Xtext of our textual Editor	107
3.5	Validation of SLA contract meta-model with Bunge-Wand-Weber (BWW) representational theory	108
3.5.0.1	Introduction	108

3.5.1	Representation Theory: Bunge-Wand-Weber	109
3.5.2	Evaluation	111
3.5.2.1	Qualitative evaluation	111
3.5.2.2	Quantitative evaluation	115
3.6	Modeling Cloud SLA Contract Requirements Using Model-Driven Engineering	121
3.7	Chapter Summary	125
4	A Novel Penalty Model for Managing and Applying Penalties in the Cloud Computing	127
4.1	Introduction	128
4.2	Background	129
4.2.1	Types of Violations	129
4.2.2	Penalties	130
4.2.3	Penalty Calculation	131
4.2.4	Important Parameters for Service Level Agreements in Cloud Computing	132
4.3	A Survey of Penalty Calculation in Cloud SLA	132
4.3.1	Comparison of Availability Calculation	132
4.3.1.1	Availability Commitment	134
4.3.1.2	Measurement Period	134
4.3.1.3	Service Granularity	134
4.3.1.4	Time Granularity	135
4.3.1.5	Coverage	135
4.3.1.6	Exclusions	135
4.3.2	Comparison of Penalty Calculation	136
4.4	Approaches for definitions of SLA penalty functions	136
4.5	A Novel Penalty Function for Managing and applying penalties in cloud computing	139
4.5.0.1	A business model for cloud providers and consumers	140
4.6	Example Application	141
4.7	Chapter Summary	144
5	An Autonomic SLA Monitoring Framework managed by trusted third party in the Cloud Computing	145
5.1	Introduction	146
5.2	Related works	148

5.2.1	Review of current research efforts in Cloud monitoring systems and architectures : Performance Monitoring and Measurement	148
5.2.2	Review of current research efforts in Trust Reputation Systems . . .	150
5.2.2.1	Trust Reputation Systems	150
5.2.3	Trusted third party for monitoring Service Level Agreements in real time	151
5.3	Autonomic SLA Monitoring Framework managed by a Trusted Third Party	152
5.3.1	General Presentation	152
5.3.2	SLA Establishment module	152
5.3.3	Service Monitoring Module	155
5.3.3.1	Cloud Usage Monitor	155
5.3.3.2	Monitoring Engine	157
5.3.3.3	Analysis component	159
5.3.3.4	Knowledge Database	162
5.3.4	A Novel Trust Reputation Systems Model	163
5.3.5	Framework Mechanism	165
5.4	Evaluation	167
5.4.1	Experimental Testbed	167
5.4.1.1	Prototype	168
5.4.2	Evaluation Scenario	172
5.4.3	Results and Discussion	174
5.4.3.1	Reputation simulation	176
5.5	Chapter Summary	179
6	Conclusion and Future Work	181
6.1	Introduction	181
6.2	Contribution	183
6.3	Limitations of the cloud SLAs contract approach and autonomic monitoring framework	184
6.3.1	Lack of support for SLA negotiation	184
6.3.2	Lack of support All steps of the MAPE-K loop control cycle	184
6.4	Future work	184
6.4.1	Self-establishing a Service Level Agreement within Autonomic Cloud Computing Environment	184
6.4.2	Support for automated negotiation strategy for cloud service negotiation problem	185

6.4.3	Extend CDOSim simulator features to simulate response times and SLA violations	185
Bibliography		187

List of Figures

1.1	Thesis structure and the dependencies among chapters	12
2.1	Cloud Computing	17
2.2	Cloud Computing Patterns	18
2.3	Cloud attributes	19
2.4	The cloud computing service categories and their characteristics	20
2.5	Deployment models of cloud computing	21
2.6	Cloud Actors according to NIST	23
2.7	Summary of QoS metrics classification	27
2.8	SLA Lifecycle	29
2.9	TTP communication	33
2.10	SLA Discovery	34
2.11	Use Case, SLA Template [168]	34
2.12	Use Case, SLA monitoring [168]	35
2.13	Use Case, SLA negotiation [168]	36
2.14	SLA Establishment	37
2.15	WSLA agreement structure	39
2.16	WSLA Meta-model[135]	40
2.17	WS-Agreement agreement structure, as specified by Andrieux et al. [18] . .	41
2.18	WS-Agreement Meta-model[96]	42
2.19	SLA* (SLA@SOI) Meta-model[96]	45
2.20	CSLA Meta-model[96]	47
2.21	Relationships between Model, Metamodel, Modeling and Metamodeling . .	53
2.22	Metamodel of abstract syntax for a simple language [131].	54
2.23	Trusted Third party cloud monitoring services	63
2.24	The MAPE-K cycle of the AC paradigm	66
2.25	Maturity levels towards Autonomic System	66
2.26	CloudSim architecture. (Adapted from [38])	70

2.27	CloudAnalyst architecture. (Adapted from [178])	71
2.28	EMUSIM architecture. (Adapted from [38])	74
2.29	Basic Schema of the Nunez's iCanCloud Architecture [126]	76
2.30	Architecture of the GreenCloud simulator. (Adapted from [86])	77
2.31	Four-layered architecture of SimMapReduce. (Adapted from [167])	80
2.32	Comparative Study of Simulators : Characteristics and Limitations	81
3.1	A provider SLA meta-model	94
3.2	Illustration of provider SLAs offers using the cloud SLA metamodel	95
3.3	A consumer SLA meta-model	96
3.4	Illustration of consumer SLAs requirement using the cloud SLA metamodel	96
3.5	Metamodel of the proposed SLA Contract	97
3.6	General Process of Correspondence and Composition	100
3.7	Correspondences meta-model	101
3.8	Potential representational deficiencies of a modeling technique [149]	110
3.9	Concrete implantation of our Model	116
3.10	Computation time of the generation of the SLA contract when varying the number of customer requirements	120
3.11	SLA Establishment module	121
3.12	Ecore meta-metamodel [66]	122
3.13	Tool examples for a dedicated modeling environment [75]	123
3.14	DSML : Scientific context	123
3.15	Relationships between system, model, metamodel and meta-metamodel	124
3.16	The package view of the generated code based on our meta-model	126
4.1	Detected response time values	143
4.2	Detected availability values	143
5.1	SLA establishment and Monitoring framework managed by TTP	153
5.2	Extract SLA parameters from SLA document	155
5.3	A cloud service consumer sends a request message to a cloud service (1). The monitoring agent intercepts the message (2) to collect relevant usage data before allowing it to continue to the cloud service (3b). While the monitoring agent stores the collected usage data in a log database (3a). The cloud service replies with a response message intercepted by the agent (4), it's sent back to the cloud service consumer.	156
5.4	Functional Architecture of The Monitoring Component	157
5.5	Defined Rules by administrator	158

5.6	Functional Architecture of The Analysis Component	160
5.7	Reputation Model using an adaptive approach	165
5.8	Sequence Diagram for the Proposed Framework	166
5.9	Testbed	168
5.10	Communication Mechanism Overview	169
5.11	Event-processing technologies: collection, aggregation, filtering	172
5.12	Four detected violation events upper than the threshold	176
5.13	<i>The results obtained from the experiments with/without implementations</i> .	177

List of Tables

2.1	Comparison of the SLA languages according to the domain requirements[171][113]	49
3.1	Representation mapping	113
3.2	Evaluating the clarity of SLA contract using constructs redundancy	114
3.3	Results of ontological completeness and clarity of the modeling language	114
3.4	Evaluating the total degree of lack-of-clarity of SLA contract	115
3.5	SLA between Amazon EC2 and Travel Agency	117
3.6	Resource Capacities of the Physical and Virtual Machines	120
4.1	Important parameters for Cloud Services	133
4.2	Availability Calculation [180]	137
4.3	Penalty Calculation	138
4.4	SLA between Amazon EC2 and online bookstore	142
4.5	Used dataset for SLA violation	143
4.6	Calculated penalty Cost	144
5.1	Resource Capacities of the Physical and Virtual Machines	169
5.2	Complex Mapping Rules	170
5.3	Patterns and generated violation	173
5.4	SLA between Amazon EC2 and Bookstore SaaS	173
5.5	Calculated penalty cost	176
5.6	Providers SLA	178
5.7	Calculated evaluation and reputation value	179

List of Abbreviations

SLA	Service Level Agreement
CC	Cloud Computing
TTP	Trust Third Party
QoS	Quality of Service
SLO	Service Level Objective
SaaS	Software as a Service
PaaS	Platform as a Service
IaaS	Infrastructure as a Service
CSP	Cloud Service Provider
CSC	Cloud Service Consumer
NIST	National Institute of Standards and Technology
MDE	Model Driven Engineering
DSML	Domain Specific modeling Language
MDA	Model Driven Architecture
UML	Unified Modelling Language
OMG	Object Management Group
M2M	Model-to-Model
M2T	Model-to-Text
OCL	Object Constraint Language
EMF	Eclipse Modelling Framework
ECL	Epsilon Comparison Language
EML	Epsilon Merging Language
XML	Extensible Markup Language
AM	Autonomic Manager
MAPE-K	Monitor, Analyze, Plan, Execute and Knowledge
REST	Representational State Transfer
API	Application Programming Interface

Chapter 1

Introduction

1.1 Overview

The revolutionary technology of Cloud Computing (CC) offers a scalable and flexible paradigm where infrastructure, platform, and software are offered to users in the form of services. The management of such environment is inherently complex due to the large-scale number and heterogeneity of resources and the increasing number and types of services a cloud must deliver over the Internet. The provisioning of these computing services by cloud providers are regulated by Service Level Agreements (SLAs) [115], which present an important element that provides some degree of assurance. As of now, the differentiating elements between cloud computing solutions are Quality-of-Service (QoS) and the SLAs guarantee provided by the service providers.

Service Level Agreements are one of the most common approaches for specifying some form of mutual understanding about business transactions between a cloud provider (seller) and a cloud consumer (buyer) in the software and telecommunications domain. An SLA is a representation of all features (including the functionality delivered by the service and the quality that the buyer experiences) a consumer should expect to receive by a service [103]. Thus, an SLA represents functional and non-functional properties of services and serves as a way for controlling and managing these properties. Typically, an SLA is a bilateral binding statement signed between a service provider, a service consumer and perhaps a third party, over the agreed terms and conditions of the given service [162]. An SLA also sets out the remedial action and any penalties that could take effect if performance falls below the promised standard. Any SLA mainly describes two things: the different Service Level Objectives (SLO) in terms of values for QoS metrics and the penalties to be applied if the objectives have not been accomplished [110]. These properties need to be measurable and

must be monitored during the provision of the service that has been agreed upon in the SLA [114].

An SLA defines measurable QoS parameters (e.g. availability and response time) and SLOs [115]. QoS parameters might be defined in an SLA with a threshold value (e.g. availability >99.9%); these combined form an SLO. An SLO is a term agreed between the service provider, service consumer and is used as a means of defining and measuring the quality threshold values of a provisioned service. An SLO is an expression over a set of the defined QoS parameters [110]. The QoS parameter threshold values are monitored by at least one of the SLA parties to ensure that the SLOs are not violated. In case of violation, penalties may apply; to be certain that all parties agree to these said penalties, they may also be defined in the SLA.

Until a few years ago, SLA contracts were mostly written using natural expressions; examination of compliance to the agreement also had to be done manually [83]. One attempt to facilitate this process by using SLA templates was limited and unable to specify different service levels for different customers [83]. For this reason, it has become a necessity to automate the procedure through which different SLAs are flexibly described, provisioned and observed [113]. Several SLA specification languages have been developed by researchers within the service provision community to address the previous aspects [83]. Their aim is to simplify the contractual process for the parties involved and to minimize the time and cost included in this process. Many projects and solutions focus on the definition of SLA [41][181][136][79][101][40]. Current approaches for SLA specifications in IT services are not sufficient since SLAs are usually based on templates that are expressed in a natural language, such as the Web Service Level Agreement (WSLA), WS-Agreement, SLAng, SLA*, CSLA, SLAC and SLA@SOI, which are discussed later (see Section 2.2.5, chapter 2), making automated compliance verification and assurance tasks difficult. We note that the poverty of expression is the main limitation of existing languages. We observed that the SLA contract of the different actors in cloud is based on a single metric: availability. On the other hand, the verification of the non-violation SLA can be very tedious for the cloud customer. Yet, we argue that these languages are not able to cope with the set of distinctive characteristics of clouds, such as deployment models and the growing importance of the broker role.

Moreover, managing the cloud computing SLA requires an appropriate means to be consistent with the dynamic nature, autonomous and elasticity of cloud computing (i.e. selfservice, rapid and dynamic resource allocation and release). The automation of the SLA establishment process is addressed to tackle this challenge. We explore the use of MDE to help to automate the establishment and selection processes. There is a lack of

research that uses models for establishment SLA built upon MDE techniques and methods. The DSML/DSL facilitates the specification of provisioning and deployment concerns establishing service level agreements between a cloud provider and cloud customer in the context of a particular service provision.

In the last decades, cloud computing have become an instance of complex systems, in which its characteristics, such as the dynamism, large-scale and heterogeneity, emphasize the complexity of the management process. Moreover, the automation of the management is one of the essential characteristics of the cloud [19]. A prominent approach to cope with this complexity is Autonomic Computing (AC), which aims at equipping computer systems with capabilities to autonomously adapt their behavior and structure according to dynamic operating conditions. Autonomic computing is inspired by the functions of the human nervous system and is aimed at designing and building systems that are self-managing. Autonomic systems are being adopted for self-management of complex large scale distributed system that have become manually unmanageable. Following the IBM Autonomic Computing paradigm we can enrich the cloud computing specification to have dynamic mechanisms that can monitor themselves and maintain a target behavior.

Monitoring plays an important role in determining whether an SLA has been violated, and thereby determine which penalty clause should be invoked as a consequence. From a legal point of view, monitoring appears as a prerequisite for SLAs contract enforcement [143]. Besides, managing and applying penalties has become a critical issue for Cloud Computing. Penalty is important to maintaining stable business relationships, and to preserving SLAs as a useful and meaningful instrument for defining such business relationships.

Within the cloud computing concept, SLA monitoring is a task of paramount importance for both cloud service provider (CSP) to avoid penalties if the SLA terms are violated and for cloud service consumer (CSC) to be aware of the status and quality of their running services. Moreover, a violation of an SLA may cause a cascading effect on the dependent services. Thus, may be affecting the overall composition and degrading the overall system performance. For that reason a trusted third party (TTP) monitoring services is required for this, as a consumer can never prove by itself that an SLA was (partially) or totally violated. The participation of a TTP is necessary in order to resolve conflicts between prospective signatories, likewise to monitor SLA violations in real-time and provide better than best-effort behavior for clouds.

The research in this thesis lies at the intersection between SLAs, cloud computing, Model Driven Engineering (MDE) and autonomic computing. In particular, it aims to investigate how MDE can assist with particular domain challenges of cloud computing, and how cloud consumers of cloud SLA can benefit from the automation support that arises from the use of

MDE tools, particularly for providing support to automated SLA generation and management. Also, the vision explored to enrich the cloud computing with autonomic computing paradigm to produce an autonomic instance of control cycle: monitoring, analysis, planning, execution whereby systems can monitor themselves and maintain a target behavior.

In this context, the research carried out in this work is led to shed some light on the standing problem of modeling/establishing cloud SLAs contract, and autonomic monitoring SLA violations by trusted third party in real-time in the goal to ensure online monitoring cloud services.

Before defining the problem and research motivation, brief introductions to SLAs, cloud computing, MDE and autonomic computing are provided in the following sections.

1.2 Problem Definition

Currently, Service Level Agreements assurance forms one of the major challenges for cloud computing in order to guarantee Quality of Service in real-time and control SLA violations. However, due to the highly dynamic nature of this open environment, it is important to have a binding agreement between all the service parties for ensuring trust while fulfilling the expected QoS. Moreover, managing the cloud computing SLA requires an appropriate means to be consistent with the dynamic nature, autonomous and elasticity of cloud computing (i.e. selfservice, rapid and dynamic resource allocation and release). Current approaches for SLA specifications in IT services are not sufficient since SLAs are usually based on templates that are expressed in a natural language, making automated compliance verification and assurance tasks difficult. We note that the poverty of expression is the main limitation of existing languages and works. We observed that the SLA contract of the different actors in cloud is based on a single metric: availability. On the other hand, the verification of the non-violation SLA can be very tedious for the cloud customer. Yet, we argue that these languages are not able to cope with the set of distinctive characteristics of clouds, such as deployment models and the growing importance of the broker role. There is a lack of research that uses models for establishment SLA built upon MDE techniques and methods.

Actually, the monitoring solutions have focused on some specific quality attributes (e.g., performance) and some of them lack mechanisms to aggregate multiple quality attributes or parameters for a service consumer, which is a critical aspect of monitoring. To the best of our knowledge, there is a need for approaches that autonomically monitor the specific non-functional characteristics of cloud services and that allow the flexibility of adding and modifying monitoring requirements at run-time. A prominent approach to cope with this complexity is Autonomic Computing, which aims at equipping computer systems

with capabilities to autonomously adapt their behavior and structure according to dynamic operating conditions.

According to current studies, it can be seen that there is still room for further research to improve on the technique for monitoring QoS, detection of SLA violation with remedy action and to ensure accountability among CSPs and CSCs. For trustworthy cloud services, effective mechanism for monitoring performances and detection of SLA violations are necessary. This is to increase the end consumers trust level towards the CSPs. To properly operate and manage such complex situations, an effective SLA establishment approach and efficient monitoring techniques is crucial. We argue that SLA establishment and monitoring SLA violations in real-time has become a critical issue for cloud computing. Additionally, in such environment, the participation of a trusted third party, i.e. a mutually trusted cloud broker, is necessary in order to resolve conflicts between prospective signatories, likewise to monitor SLA violations in real-time in the goal to ensure online monitoring cloud services and provide better than best-effort behavior for clouds. For that reason a trusted third party monitoring services is required for this, as a consumer can never prove by itself that an SLA was (partially) or totally violated.

The problems that we address in the thesis are: (i) How to automate the SLA establishment process and describe the SLA terms between cloud providers and cloud consumers, such as service levels, penalties in case of SLA violation, reputation, etc.. (ii) How to provide guarantees on cloud QoS monitoring and detection of SLA violation, and identifies particular penalty clauses that can be used to modify the reputation of a provider over time? How to create a fair competitiveness between providers and hence improve service performance and the reliability in the cloud? How can be the participation of a trusted third party is necessary to operate and controls the communication, in order to guarantees transparency and symmetry with respect to the SLA communication between prospective signatories?

Next, we describe our contributions to the field of automation SLA establishment process and the autonomic SLA monitoring approach as we address the aforementioned questions.

1.3 Research Questions

In this section, we concretize the research questions addressed in this thesis. Nevertheless, Section 1.2 presents the overview of the challenges motivating the research work carried out. Therefore, we concretely address the following questions:

Research Question 1

How to automate the SLA establishment process and describe the SLA terms between cloud providers and cloud consumers?

In order to automate the SLA establishment process and describe the SLA terms between cloud providers and cloud consumers, there is a need for an efficient mechanisms for modeling SLAs contract for cloud. Furthermore, the existing languages are not able to cope with the set of distinctive characteristics of clouds, such as deployment models and the growing importance of the elasticity of cloud computing (i.e. self-service, rapid and dynamic resource allocation and release). For this reason, it has become a necessity to automate the procedure through which different SLAs are flexibly described, provisioned and observed. To resolve this question, the designer has to investigate this issue and propose conception by using Model Driven Engineering approach in cloud computing domain. MDE provides some means of addressing this problematic by considering models as first class items[115]. Another important concept in the MDE is the idea of having specific modeling language (DSML) that are used to describe specific behaviors to a domain in a brief and concise manner. According to this, every model must conform to a specific meta-model, similar to how a program conforms to the grammar of its programming language. In MDE, it's common to have a set of transformation engines and generators that produce various types of artifacts. Practitioners can take advantage of transformation engines to obtain source code, alternative model descriptions, deployment configurations, inputs for analysis tools, and so on. Thus, MDE principles, techniques and tools are necessary.

Research Question 2

How to manage and apply penalties which is become a critical issue for cloud computing?

On the one hand, penalty is important to maintaining stable business relationships, and to preserving SLAs as a useful and meaningful instrument for defining such business relationships. Managing and applying penalties has become a critical issue for cloud computing. On the other hand, we need to investigate this issue and presents the most frequently used definitions of SLA penalty functions by identifying the characteristics of these functions highlighting their strengths and weaknesses. Therefore, to address this issue, a Novel Penalty Model for calculating the penalty cost of the violations are required.

Research Question 3

How can we provide an autonomic SLA monitoring framework managed by TTP following the IBM Autonomic Computing (MAPE-K loop) paradigm to dynamically monitor QoS metrics by detecting SLA violations at runtime?

To provide an autonomic SLA monitoring framework managed by TTP for ensuring accountability and QoS in cloud services is an important management task for the trusted third party. To achieve this goal, the TTP has to follow the IBM Autonomic Computing paradigm by enriching the cloud computing specification to have a dynamic mechanisms that can monitor themselves and maintain a target behavior. Autonomic computing is inspired by the functions of the human nervous system and is aimed at designing and building systems that are self-managing. Therefore, the TTP requires an efficient autonomic SLAs monitoring techniques for this purpose. The monitoring technique should be capable of automatically and dynamically monitor QoS metrics by detecting SLA violations at runtime, and support large scale cloud environments. None of the existing monitoring framework we studied so far posses these capabilities.

1.4 Motivation

Firstly, different studies have been compared, the existing languages and works are not able to cope with the set of distinctive characteristics of clouds, such as deployment models and the growing importance of the elasticity of cloud computing (i.e. self-service, rapid and dynamic resource allocation and release). Hence, existing static SLA templates are unable to cope with the diversity of cloud applications. However, due to diversity in deployment models of cloud computing services (e.g., SaaS, PaaS and IaaS) many other quality parameters need to be considered, such as response time, elasticity, interoperability, etc., are not included, although consumers may be interested in them. There is a problem with this; a significant difficulty is that there is as of yet no standard vocabulary or meta-model for cloud SLAs. Also, there is a lack of research that uses models for establishment SLA built upon MDE techniques and methods.

The motivation of this work (first contribution) is to help cloud consumers and providers precisely, efficiently in expressing SLA requirements and offers, and establishing them automatically. We aim to provide mechanisms to help cloud consumers and providers to automate the SLA establishment process using MDE tools and techniques. Secondly, monitoring plays an important role in determining whether an SLA has been violated, and thereby determine which penalty clause should be invoked as a consequence. From a legal point of view, monitoring appears as a prerequisite for SLAs contract enforcement [143]. Once the SLA agreement has been established, it is mandatory to monitor the SLA terms to determine whether they are achieved or violated. SLA monitoring is essential for both cloud service provider to avoid penalties if the SLA terms are violated and for cloud service consumer to be aware of the status and quality of their running services. Moreover, a violation

of an SLA may cause a cascading effect on the dependent services. Thus, may be affecting the overall composition and degrading the overall system performance. In such situation a trusted third party is required, as a client can never prove by itself that an SLA was (partially) violated. Moreover, the TTP is necessary in order to resolve conflicts between prospective signatories.

Penalty is important to maintaining stable business relationships, and to preserving SLAs as a useful and meaningful instrument for defining such business relationships. To address this issue, we aim to propose a Novel Penalty Model to automatically calculating the penalty cost of the violations in a time interval.

Thirdly, actually the monitoring solutions have focused on some specific quality attributes (e.g., performance) and some of them lack mechanisms to aggregate multiple quality attributes or parameters for a service consumer, which is a critical aspect of monitoring. To the best of our knowledge, there is a need for approaches that autonomically monitor the specific non-functional characteristics of cloud services and that allow the flexibility of adding and modifying monitoring requirements at run-time.

None of the existing monitoring framework we studied so far posses these capabilities that autonomically monitor the specific non-functional characteristics of cloud services and that allow the flexibility of adding and modifying monitoring requirements at run-time. From these studies, it can be seen that there is still room for further research to improve on the technique for monitoring QoS, detection of SLA violation with remedy action and to ensure accountability among CSPs and CSCs. For trustworthy cloud services, effective mechanism for monitoring performances and detection of SLA violations are necessary.

Motivated by this fact we therefore propose an autonomic SLA monitoring framework managed by TTP for ensuring accountability and QoS in cloud services. In the case of penalties, the reputation of CSP concerned is affected. The method of penalty cost calculation, detection of future SLA violations, reputation score is specified in this framework.

1.5 Research Hypothesis

The research presented in this thesis investigates the following hypothesis:

Can MDE principles and tools support the precise modeling of cloud computing SLAs contracts in such a way that cloud stakeholders can define their offers and demands? In addition, can the MDE principles and tools enable automated SLA establishment process for cloud computing SLAs, in order to provide a good support for automated SLA generation and management? Can the IBM Autonomic Computing paradigm enrich the cloud computing

specification to have an autonomic mechanisms that can monitor themselves and maintain a target behavior?

The main characteristics of the above statement are as follows:

1. Modeling and Automation of SLA contract establishment process: By using MDE principles and models of cloud SLAs contract; meta-models can be used to provide a support for automated SLA generation and management, systematically and in a reusable way.
2. The proposed cloud SLA contract language: the language design enables to define extension to the standard language expression scope as part of the SLA document itself.
3. Proposing a Novel Penalty Model for calculating the penalty cost of the violations, can help on managing and applying penalties in the cloud computing efficiently.
4. The vision explored to enrich the cloud computing with autonomic computing paradigm can produce an autonomic instance of control cycle: monitoring, analysis, planning, execution whereby systems can monitor themselves, whereby some steps of the autonomic systems require high-level guidance from humans, e.g. to decide which steps need to be done to keep the system stable.
5. Supporting SLA establishing and autonomic SLA monitoring: By using the results from 1-4 above, we can properly operate and manage such complex situations in the cloud environment; also it helps a lot in resolving conflicts that can occur between the involved parties to reach an appropriate trade-off.

1.6 Research Objectives

Motivated by the problem's definition and hypothesis, this thesis focuses on the following objectives.

1. To provide mechanisms for modeling cloud SLAs contract, by using Model Driven Engineering principles, techniques and tools.
2. To automate the SLA establishment process and describe the SLA terms between cloud providers and cloud consumers.
3. To present a cloud SLA contract language to define SLAs specifically devised to the cloud computing domain.

4. To propose a Novel Penalty Model for managing and applying penalties in the cloud computing.
5. To provide an autonomic SLA monitoring mechanisms following the IBM Autonomic Computing (MAPE-K loop) paradigm to dynamically monitor QoS metrics by detecting SLA violations at runtime to verify compliances for the respective SLAs.
6. The proposed autonomic framework is operated and controlled by a trusted third party, i.e. a mutually trusted cloud broker, that guarantees transparency and symmetry with respect to the SLA communication between prospective signatories. Also, the participation of the TTP is necessary in order to resolve conflicts between prospective signatories, likewise to monitor SLA violations in real-time in the goal to ensure online monitoring cloud services and provide better than best-effort behavior for clouds.
7. To evaluate the above framework with a real experimental testing in a cloud environment as realistically as possible.

1.7 Thesis Contributions

According to the research questions presented in Section 1.3, we highlight in this section the scientific contributions to the state-of-the-art in SLA enforcement and autonomic Cloud management. Our contributions in this thesis have been published in different journals, conferences and workshops. We specify later for each contribution, the references where it has been published.

The main objectives of this thesis are to propose our conception for automation of SLA establishment process, by using Model Driven Engineering to express SLA contract requirements, as a mean for establishing service level agreements between a cloud provider and cloud customer in the context of a particular service provision. Also, to provide an autonomic SLA monitoring mechanisms for real time ensuring accountability and QoS in cloud services.

The key contributions of this thesis can be summarized as follows:

1. Firstly, for modeling SLAs contract and automate the SLA establishment process, we investigate this issue and propose our conception by using MDE approach in CC domain. MDE provides some means of addressing this problematic by considering models as first class items[115]. Another important concept in the MDE is the idea of having specific modeling language (DSML) that are used to describe specific behaviors to a domain in a brief and concise manner. According to this, every model must

conform to a specific meta-model, similar to how a program conforms to the grammar of its programming language. In MDE, it's common to have a set of transformation engines and generators that produce various types of artifacts. Practitioners can take advantage of transformation engines to obtain source code, alternative model descriptions, deployment configurations, inputs for analysis tools, and so on. The cloud SLA contract language we present is a meta-model (consisting of abstract syntax) for defining cloud SLAs specifically devised to the cloud computing domain, that support most of the characteristics of cloud.

3. Secondly, we propose a Novel Penalty Model for calculating the penalty cost of the violations, that can efficiently help for managing and applying penalties in the cloud computing.
4. Lastly, we propose an autonomic SLA monitoring framework managed by trusted third party composed of two modules: the first one SLA establishment module, which aims at providing support for automated SLA generation and management. The second one, a service monitoring module to dynamically monitor QoS metrics by detecting SLA violations at runtime to verify compliances for the respective SLAs, and to propose a mechanism for an adaptive remedy rectification, as a contribution at the third maturity level of the autonomic computing paradigm as defined by IBM. The framework is validated with scenarios on response time and availability, the results obtained are promising. They confirm that our framework manages SLAs in an efficient way, as it detects all violations to be communicated to concerned parties, and identifies particular penalty clauses that can be used to modify the reputation of a provider over time. The TTP framework equipped with such reputation module can provide real-time assessment for consumers informed decision making to continue using a service or to migrate to another service provider in the case of service degradation. This creates a fair competitiveness between providers and hence improves service performance and the reliability in the cloud.

1.8 Thesis Organization

The rest of this thesis consists of five chapters as illustrated in Figure 1.1, and the main contents can be summarized as follows: Chapter 2 provides a relevant literature overview of cloud computing domain, including its features and classifications. The chapter focuses on SLAs content by describing their main components and discusses their specifications, also discusses some SLAs specification languages used to build it. Hence, it provides a

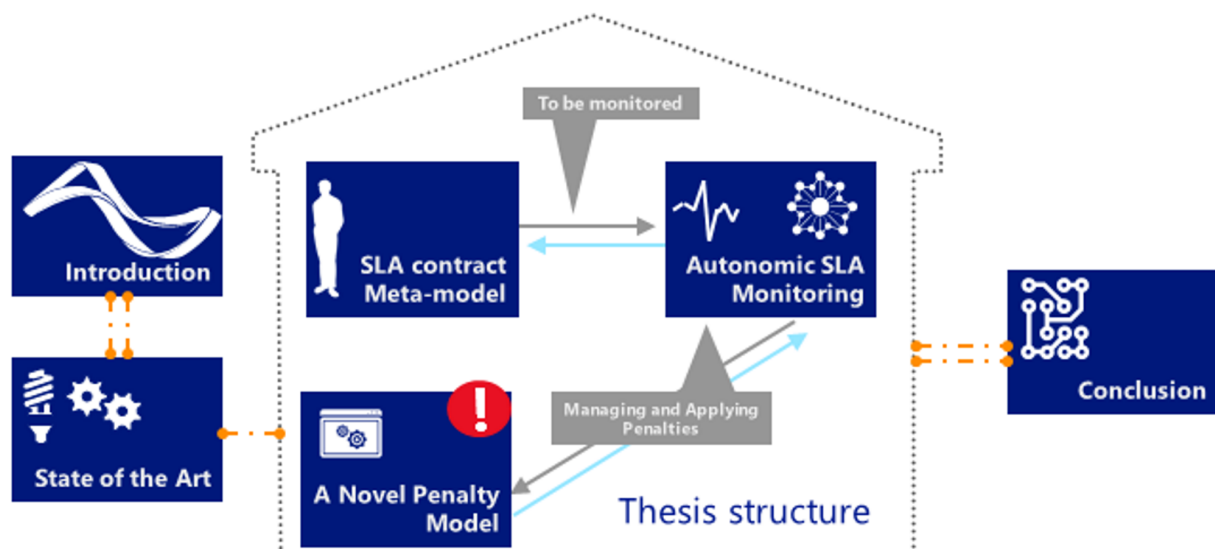


Figure 1.1 Thesis structure and the dependencies among chapters

general overview of the MDE by describing its main concepts and principles related to our objectives. Then it introduces the concept of trusted third party in the cloud computing. Afterward, it demonstrates the importance of autonomic computing for clouds, explains the essential role of the SLA in the context of autonomic systems and of cloud management. Finally, it presents a comparative study of simulators for Cloud Computing by identifying the characteristics of the tools highlighting their strengths and weaknesses.

Chapter 3 analyses and explores the problem and identifies the steps of the research plan to answer and address the hypothesis and objectives of this thesis. These are mentioned earlier in Sections 1.5 and 1.6. This chapter starts with an introductory and motivating example. It presents some related works on SLA establishment and, secondly, states our motivations and outlines our approach. Then, it provides the construction process and describes the cloud SLA contract meta-model used in our proof-of-concept tool support. It gives a description about Bunge-Wand-Weber (BWW) representational theory. Hence, it presents the evaluation of the expressiveness of the proposed modeling languages and the efficiency of the proposed SLA establishment approach. It describes the validation of our meta-model to examines the ability of the proposed SLA establishment approach to generate the SLA document in less time. Finally, it conclude with a summary of our contributions and their future potential extensions.

Chapter 4 presents a Novel Penalty Model for calculating the penalty cost of the violations, and present a formalization for this penalty definition. It also propose a business model for cloud providers to manage their profit. In addition, it gives an overview of the background in the field. Then, survey and analyze various penalty calculation and availability calculation

methods of cloud providers. It presents some approaches for definitions of SLA penalty functions. To validate our proposed model an example application will be presented to demonstrate the effectiveness of the proposed approach. Finally, the chapter notes some challenges and future research directions.

Chapter 5 starts with the related works. It describes the design of our autonomic SLA monitoring framework and we detail its components. It also discusses the experimental evaluations by given an example scenario to demonstrate the effectiveness of the proposed framework. Finally, the chapter is concluded with a summary of our contributions and their future potential extensions.

Chapter 6 presents concluding remarks on this research. This final chapter aims to provide an overview of the research, to analyze some useful development opportunities and to offer some suggestions about how future research on the topics related to it could be carried out.

Chapter 2

Background and Literature Review

This chapter reviews topics that are considered in this research and have influenced the work; it also serves to allow the reader navigate the topics related to this work. This chapter provides a comprehensive overview of cloud computing, SLAs and MDE, which has been used to carry out the research in this thesis and to identify the research gaps in this area. This chapter is divided into five parts. Section 2.1 presents an overview of cloud computing domain, including its features and classifications. Then, in Section 2.2, we present an introduction to SLAs represented by specialized SLA specification languages. SLAs are the core of this thesis, serving as the starting point of all the proposed solutions in various areas that this thesis addresses. Hence, section 2.3 explains MDE and its concepts related to our objectives. Then, section 2.4 introduces the concept of trusted third party in the cloud computing. Section 2.5 demonstrates the importance of autonomic computing for clouds, explains the essential role of the SLA in the context of autonomic systems and of cloud management. Finally, section 2.6 presents a comparative study of simulators for Cloud Computing by identifying the characteristics of the tools highlighting their strengths and weaknesses.

2.1 Cloud Computing

In this section, we provide an overview of cloud computing as a basis for developing models of cloud SLAs. We first provide an overview of the general concepts and characteristics of cloud computing, then outline different deployment mechanisms and service categorization of cloud computing. Finally, we discuss the cloud communication actors terminologies.

2.1.1 An overview of Cloud computing paradigm

The term "Cloud Computing" is currently a hot and highly discussed topic in both technical, economic, and research world. It is used for describing what happens when applications and services are moved into the "Cloud". Currently, Cloud computing is being promoted as the latest outsourcing solution of the new century. Cloud computing (CC) represents the evolution of IT towards an Internet-based computing model explicitly designed to enable the transition from inside-out to outside in organizations. It consists to provide IT services as a service on-demand, accessible from anywhere, anytime and by anyone; see figure 2.1. Through this new computing paradigm, we can now eliminate the need for organizations to build and maintain expensive data centers, moreover we can access to computing resources hosted on the network and benefit from processing capabilities and storage as on-demand services[111]. It enables organizations to build up new systems quickly and easily, and has thus become a platform for innovation. With the CC, we can provide elastic resources that allow applications to scale as needed in response to market demands. It increases business continuity by providing inexpensive disaster-recovery options. Many big companies within the IT industry (e.g., Microsoft, IBM, Google, Amazon,..) are joining to the development of cloud computing, and providing cloud computing services. However, cloud computing definition remains unclear. Many people within the industrial and academic community have attempted to define what "Cloud Computing" really is, and what typical characteristics it presents. Armbrust et al.[59] define a cloud as the *"data center hardware and software that provide services"* and summarize the key characteristics of cloud computing as:(1) *the illusion of infinite computing resources*; (2) *the elimination of an up-front commitment by cloud users*; and (3) *the ability to pay for use as needed "*. Buyya et al. [36] have defined cloud computing as follows: *"Cloud is a parallel and distributed computing system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level-agreements (SLAs) established through negotiation between the service provider and consumers "*. Nevertheless, several definitions of cloud computing exist. Among them, the most cited and widely accepted definition of clouds is provided by the National Institute of Standards and Technology (NIST)¹ [120]. NIST proposes the following definition of cloud computing: *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [120].

¹<http://www.nist.gov/>

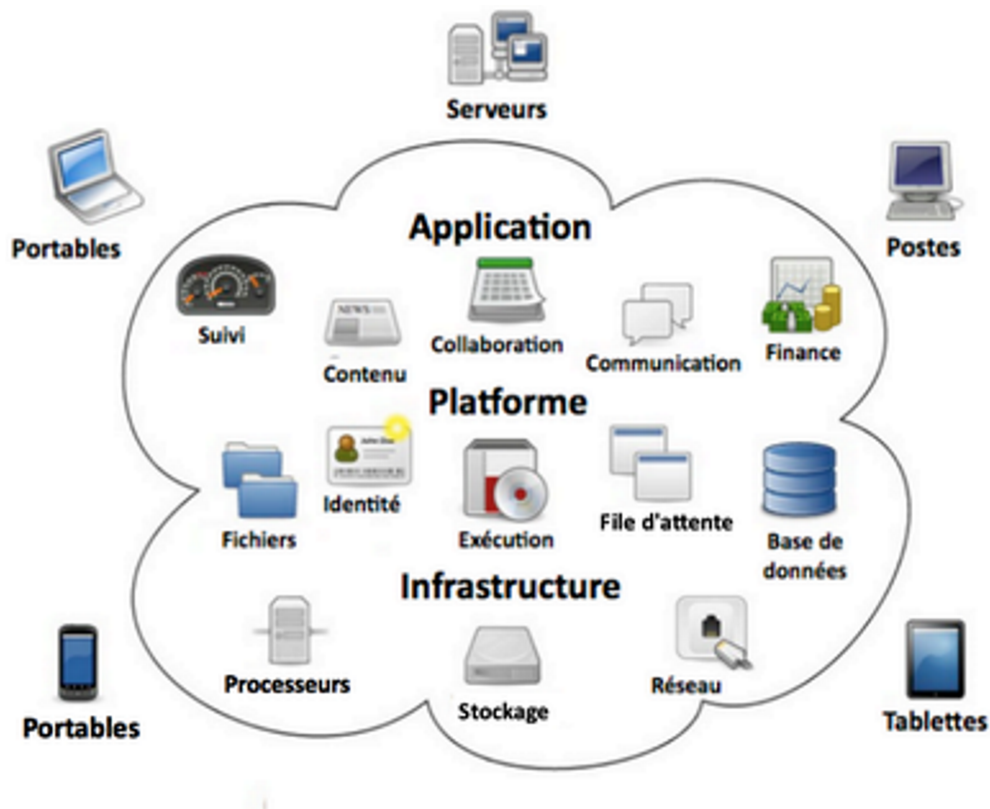


Figure 2.1 Cloud Computing

This cloud model is composed of five essential characteristics, three service models, and four deployment models.

2.1.1.1 Essential characteristics of cloud computing

The five essential characteristics that define cloud computing cited by NIST are presented below:

1. *On-demand self-service*: A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.
2. *Broad network access*: Capabilities are available over the network and accessed through standard mechanisms that promote the use by different types of client platforms (e.g., mobile phones, tablets, laptops, and workstations).
3. *Resource pooling*: The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources

dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify the location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.

4. *Rapid elasticity*: Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly on demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time, figure 2.2.
5. *Measured service*: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate for the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

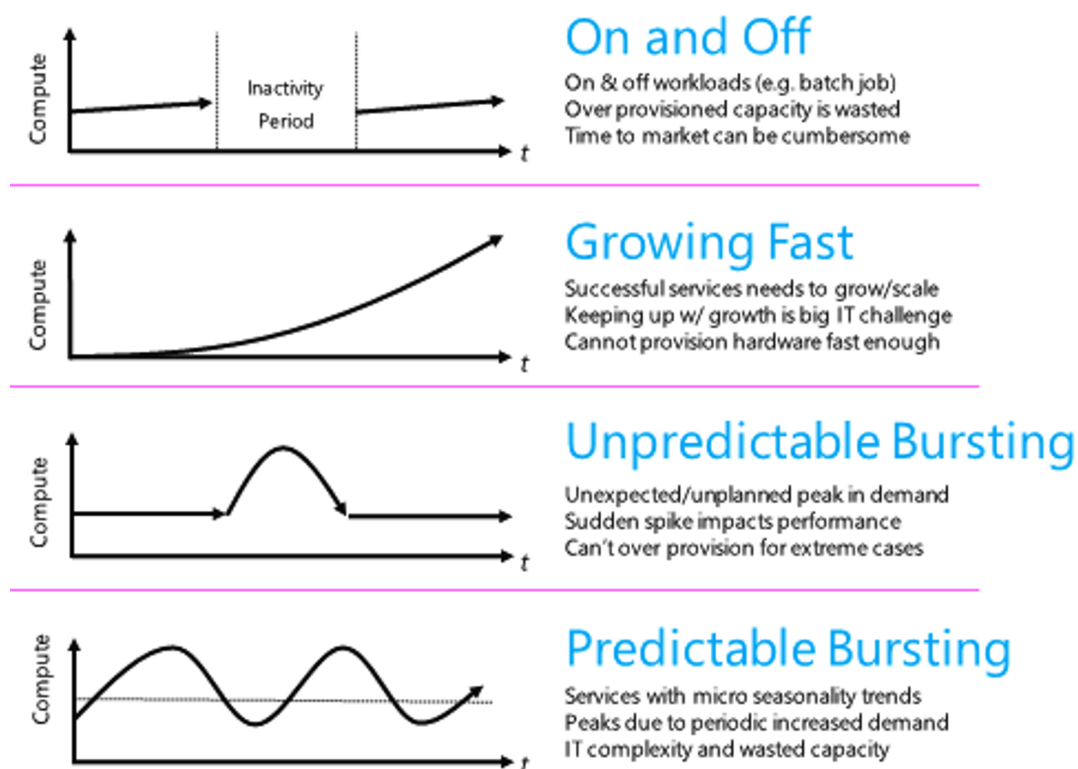


Figure 2.2 Cloud Computing Patterns

Cloud computing provides various services. Clouds have been classified based on the services they provide. It is essential to understand the difference between these services



Figure 2.3 Cloud attributes

and classifications in order to, better plan whether moving to the cloud and achieving the enterprise's goals. The next section discusses the different classifications of cloud computing depending on the service model and deployment model.

2.1.1.2 Service delivery models of cloud computing

The differences between the cloud computing services depend upon the capability of cloud providers, and they are related with the type of service offered, such as: (1) storage and computing capacity, (2) platform for own software development and testing, (3) online software applications. In this context, “service” has a broad meaning and, virtually, everything can be provided as an on-line, on-demand service. Such generic service delivery model is called Everything/Anything as a Service (XaaS) and the most well-known instances of this model (as illustrated in figure 2.4), that NIST has already proposed are:

1. *Software as a Service (SaaS)*: provides services or composition services to users. These services run on a cloud infrastructure and are accessible by various client devices through a client interface such as Web browsers (e.g. Web mail, Google Docs, etc.) or client applications (e.g., iTunes, Picasa, etc.).
2. *Platform as a Service (PaaS)*: offers development platforms where the developers can design, develop, deploy and test their applications and do not necessary need to know about the infrastructure (e.g., processors, memory, storage). There are several IT vendors have developed new PaaS systems such as Google App Engine², Microsoft Azure³ and Amazon Web Service⁴.

²<http://code.google.com/appengine>

³<http://www.microsoft.com/windowsazure>

⁴<http://www.amazon.com/aws>

3. *Infrastructure as a Service (IaaS)*: provides virtualized infrastructure to the user, such as storage, network, processing and other computing resources. Cloud users are able to deploy and run software, which can include operating system and applications. It enables on-demand provisioning of servers with different choices of operating systems and software stacks. IaaS is considered as a bottom layer of cloud computing systems. For example: Amazon Web Services or IBM Smart Cloud mainly offers IaaS.

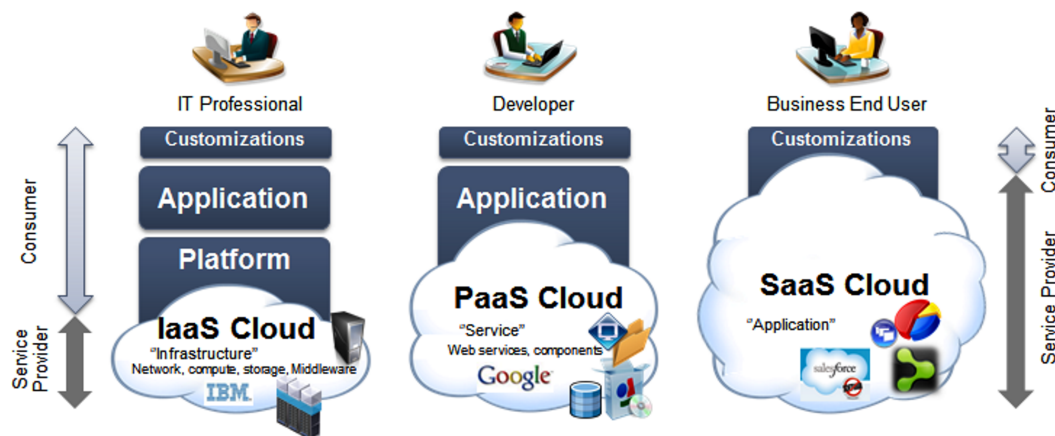


Figure 2.4 The cloud computing service categories and their characteristics

2.1.1.3 Deployment models of cloud computing

Cloud computing has emerged from the combination of public computing utilities. However, depending upon the purpose of cloud services, and physical location or distribution of cloud systems, a cloud can be classified as a public, private, community or hybrid cloud [120], see figure 2.5.

1. *Private cloud*: A cloud system is operated solely for a single organization. In other words, the proprietary network or the internal data center supplies hosted services to a certain group of people, not made available for general public users. For example, Microsoft Azure enables customers to build the foundation for a private cloud infrastructure using Windows Server and System Center family of products with the Dynamic Data.
2. *Public cloud*: A cloud service provider makes resources (e.g., applications and storage) available to the general public over the Internet on a pay-as-you-go basis. For example, the Amazon Elastic Compute Cloud (EC2) allows users to rent virtual computers on which they run their own applications. It provides network infrastructure, data centers and allows customers to pay only for what they use with no minimum fee.

3. *Community cloud*: The cloud system is shared by several organizations with common concerns (e.g., mission, security requirements, policy, and compliance considerations).
4. *Hybrid cloud*: The cloud infrastructure comprises 2 or more clouds (e.g., private, public, or community). In this infrastructure, an organization provides and manages some resources within its own cloud and has other resources provided externally from other clouds.

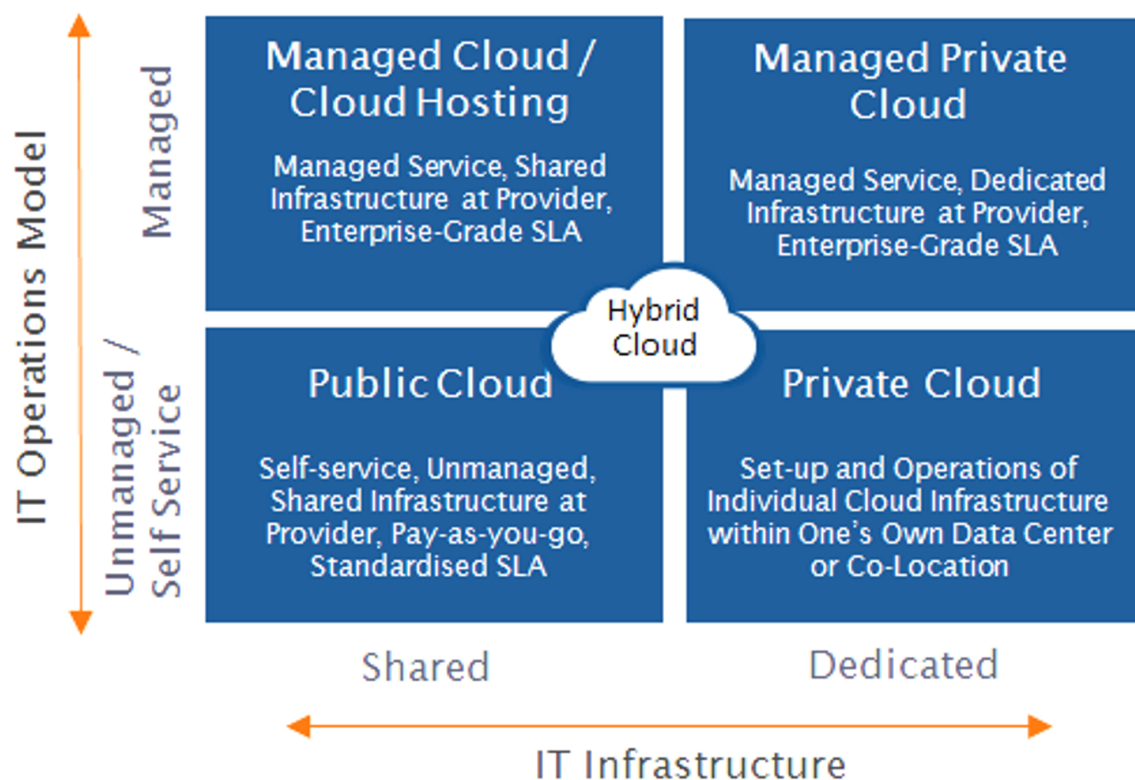


Figure 2.5 Deployment models of cloud computing

2.1.1.4 Cloud communication actors

The NIST cloud computing reference architecture defines five major actors: cloud consumer, cloud provider, cloud auditor, cloud broker, and cloud carrier (see figure 2.6: Cloud Actors). These core individuals have key roles in the realm of cloud computing. Each actor is an entity (a person or an organization) that participates in a transaction or process and/or performs tasks in cloud computing. For example, a Cloud Consumer is an individual or organization that acquires and uses cloud products and services. The purveyor of products and services is the Cloud Provider. Because of the possible service offerings (Software,

Platform or Infrastructure) allowed for by the cloud provider, there will be a shift in the level of responsibilities for some aspects of the scope of control, security and configuration. The Cloud Broker acts as the intermediary between consumer and provider and will help consumers through the complexity of cloud service offerings and may also create value-added cloud services. The Cloud Auditor provides a valuable inherent function for the government by conducting the independent performance and security monitoring of cloud services. The Cloud Carrier is the organization which has the responsibility of transferring the data, somewhat akin to the power distributor for the electric grid.

- *Cloud Consumer* is an entity that uses services from providers;
- *Cloud Provider* is an organization or entity that makes a single or multiple services available to interested parties. Examples of wellknown commercial cloud providers are Amazon, Rackspace and Google;
- *Cloud Carrier* is a third party involved in the provision of the service that provides connectivity and transports the services from the providers to consumers;
- *Cloud Auditor* is an independent party that assesses the service provided in term of, e.g. security level, performance, SLA;
- *Cloud Broker* is an entity that negotiates relationships between providers and consumers, and manages the service use and delivery. Due to its importance in our work, we detail this role in the next section.

2.2 Service Level Agreements

Actually, Service Level Agreements (SLAs) play a pivotal role in the cloud computing, the current technological evolution of information and communications technology. The revolutionary technology of cloud computing offers a scalable and flexible paradigm where infrastructure, platform, and software are offered to users in the form of services[115][111]. The provisioning of these computing services by cloud providers are regulated by SLAs [103]. Service Level Agreements are one of the most common approaches for specifying some form of mutual understanding about business transactions between a cloud provider (seller) and a cloud consumer (buyer) in the software and telecommunications domain. An SLA is a representation of all features (including the functionality delivered by the service and the quality that the buyer experiences) a consumer should expect to receive by a service [103]. Thus, an SLA represents functional and non-functional properties of services and

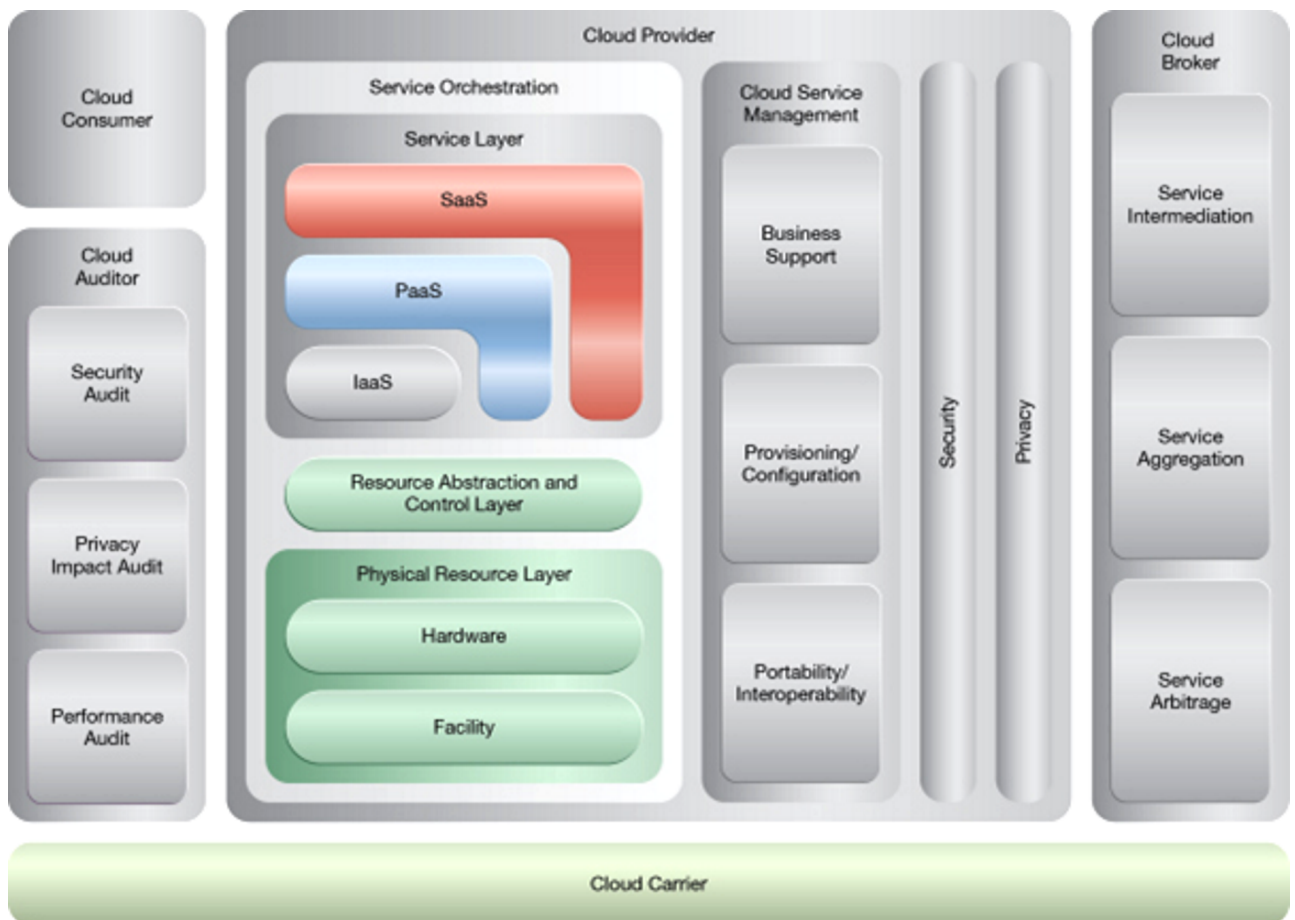


Figure 2.6 Cloud Actors according to NIST

serves as a way for controlling and managing these properties. Typically, an SLA is a bilateral binding statement signed between a service provider, a service consumer and perhaps a third party, over the agreed terms and conditions of the given service [162]. An SLA also sets out the remedial action and any penalties that could take effect if performance falls below the promised standard.

An SLA defines measurable QoS parameters (e.g. availability and response time) and SLOs [115]. QoS parameters might be defined in an SLA with a threshold value (e.g. availability >99.9%); these combined form an SLO. An SLO is a term agreed between the service provider and the service consumer and is used as a means of defining and measuring the quality threshold values of a provisioned service. An SLO is an expression over a set of the defined QoS parameters [110]. The QoS parameter threshold values are monitored by at least one of the SLA parties to ensure that the SLOs are not violated. In case of violation, penalties may apply; to be certain that all parties agree to these said penalties, they may also be defined in the SLA.

An SLA can be used to remove ambiguity between parties by defining the service consumers requirements and specifying the service qualities and responsibilities of each party. The intention behind specifying and using an SLA is that, by providing precise definitions and conditions, the parties can reduce the areas of conflict. As a result, an SLA is used most frequently to identify service consumers requirements and service provider capabilities. There are several possible SLA structures, Figure 2.1 shows an example of such a structure. The main components, as in the example, are:

- *Duration of the SLA*: The agreement specifies the start and end dates regarding which the agreement conditions can be applied.
- *Parties*: This refers to the involved parties who take part in the SLA, these are usually a service provider, a service consumer and perhaps a third party.
- *List of services*: Each SLA: defines the list of services that are covered by the agreement. This is illustrated in Figure x as service scope.
- *Service definition*: This part includes a detailed description of the service and its features and functions.
- *Service Exclusions*: This describes the uncovered conditions and services in the SLA.
- *SLO*: This part defines a threshold level of service that a service provider promises to provide to the service consumer. For example, a service provider might promise to provide a service with 99.9% availability of the service time.
- *Penalty*: This section of the agreement defines actions that might be taken when a service provider fails to meet the promised SLO (violation). Examples of actions that can be taken are: terminating the SLA, or an increasing/decreasing in the agreed payment.
- *Agreement effective date*: specifies the duration of the service.

SLAs contain Quality of Service properties that must be maintained by a provider, generally defined as a set of Service Level Objectives (SLOs). These properties need to be measurable and must be monitored during the provision of the service that has been agreed in the SLA [114]. The SLA must also contain a set of penalty clauses specifying what happens when service providers fail to deliver the pre-agreed quality. This raises the following question: How to describe the SLA terms between prospective signatories, such as service levels, Quality of Service properties, penalties in case of SLA violation, etc. SLA can be represented

by specialized languages for easing SLA preparation, automating SLA negotiation, adapting services automatically according to SLA terms, and reasoning about their composition. Therefore, the aim of this section is to present a comprehensive review of how SLAs are created, managed and used in web services and cloud computing environment. This section also provides a review of SLA languages specification. Among them the WSLA, WS-Agreement, SLAng, SLA*, CSLA and SLAC. Then, a comparison of these languages in terms of the requirements is presented, highlighting their strengths and weaknesses. The rest of this section is organized as follows: Section 2.2.1 provides the QoS metrics used within an SLA along with their categorisation. Section 2.2.2 describes different SLA life cycle and their uses for cloud Computing. Section 2.2.3 gives an overview of SLA life cycle functionalities which are illustrated using UML case diagrams. Section 2.2.4 provides an overview of SLA specification languages. Section 2.2.5 presents a comparative study of the SLA definition languages, characteristics and limitations.

2.2.1 QoS Metrics Related to an SLA and their Categorization

A service has a set of functional and non-functional properties; the non-functional ones are restrictions on the functionality of the service and are referred to as QoS attributes [39][130]. A service can be assigned different QoS attributes which are usually defined for the operations of the service. However, any service object can be assigned a QoS, such as interfaces, attributes, operation parameters and results [98]. Each QoS attribute is measured by a metric and is thus referred to as a QoS metric. A metric is usually used to describe the unit, the type of value that it can take, and the time required to measure this attribute. For this reason, a QoS metric can be considered as one evaluation of the QoS attribute [98].

In this section, we focus on the mechanisms used to model SLAs. We identify the key components of the SLA and how these can be used in creating, establishing and matching different SLAs. QoS parameters are an important part of SLAs and their comparisons; they are typically used in SLA guarantee conditions. The section provides a brief description of QoS.

SLO usually defines thresholds over the QoS parameters, e.g. *response time* ≤ 5 seconds. QoS is often considered as a part of the non-functional requirements. QoS parameters are used in SLAs to represent the level of services guaranteed by the service provider, e.g. *to guarantee that the service will be available 99.9% of the time*. QoS and its parameters have been undertaken in some SLA specification languages, e.g. WSLA [109] and WS-Agreement [18]. QoS has also been used as a basis for web service discovery and selection, as in [83]. Because of the objectives of this thesis, and because QoS parameters are defined in SLAs, we briefly discuss all QoS classifications and subclassifications that are relevant to our work.

We describe research on QoS parameter specifications, common QoS parameters that are defined in the SLA and choice of QoS metrics as a basis for service selection.

In terms of the SLA, QoS metrics are its primary components and are related to the non-functional attributes of the specified service [133][132]. The service customer and provider usually negotiate the desired quality of service metrics to be included in their SLA. In addition, they agree to set the level at which the service has to offer these metrics [71]. A QoS value is a positive or negative number [152], while the QoS level specifies the upper or lower limit a QoS value can reach.

To be able to understand the semantics of the QoS metrics in a particular SLA, the class under which these metrics may be categorised has to be addressed. This is also useful in assigning the monitoring semantics of an SLA contract when it is presented later in Section x. A summary of all QoS classifications and subclassifications are depicted in Figure 2.7. These classifications are described in what follows according to their occurrence in the figure moving from left to right.

The choice of QoS metrics in a particular SLA may vary according to the **Domain** in which they are used, such as **Network** management, **Service** management and **Business** management [133]. For example, in the network management domain, a typical QoS metric is '*bandwidth*', which indicates the network capacity [102]. On the other hand, a typical QoS in the business management domain may be '*revenue*' and '*cost*', indicating the outcome and price of the service [179]. Despite this variety in QoS metric types used in different domains, the most commonly used ones are those relating to service performance (e.g. '*response time*') or to reliability (e.g. '*availability*') [133]. Additional QoS metrics can include '*serviceability*', '*security*' and '*scalability*' [133][132][71]. The aforementioned QoS metrics can be defined in the web service domain, which is the focus of this thesis, as follows [132][117]:

- *Availability*: Indicates the probability that the service is up and working.
- *Throughput*: Indicates the number of requests the service receives during a specific period of time.
- *Scalability*: Indicates the ability of a web service to handle arriving requests even under different workloads.
- *Security*: Indicates the type of mechanism used to authenticate and authorize customers, which is critical to protect their privacy.
- *Response time*: Indicates the time taken to respond to a request once it is received by the service [99].

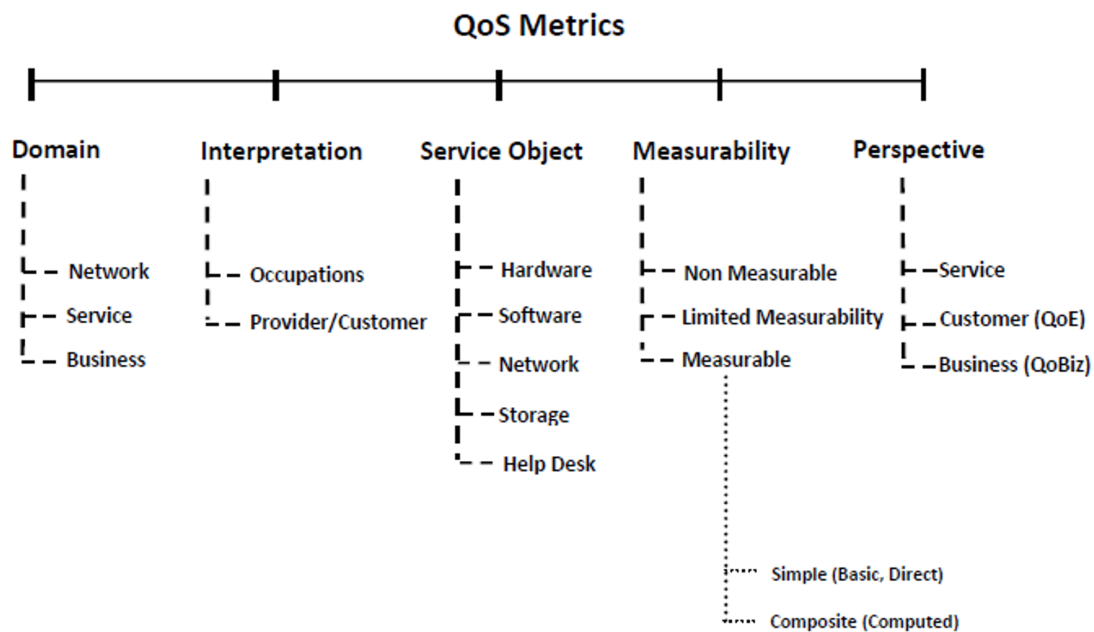


Figure 2.7 Summary of QoS metrics classification

As the type of QoS metric used within an SLA differs from one service domain to another, the **Interpretation** of a single QoS metric varies accordingly [133]. Each **Occupation** may have its own interpretation of a certain QoS metric, just as the service **Provider** and **Customer** may have different perspectives [133]. An example of this diverse understanding of a QoS metric is that of '*availability*'. From the providers perspective, availability means that the hardware is not down (an aspect of infrastructure). However, from the customer's point of view, it could be the ability to serve a request (a service application aspect) or what is called '*successability*' [133].

Another categorization of QoS related metrics can be according to the **Service Object** they are defined for. This might be related to physical **Hardware**, application **Software**, the communication **Network** and its infrastructure, data **Storage** repository or service **Help Desk** or to combination of them [133]. A typical example of a QoS metric related to storage is '*bytes per second*' which reflects the reading and writing speed as specified in [133].

In addition to distinguishing QoS metrics according to their related domain, interpretation, and object type, they can also be classified into three categories according to their **Measurability** [133]. The first category is the Measurable metrics; these, as their name suggests, can be measured automatically from the underlying service, for example '*queue size*'. A metric that cannot be automatically measured is called a **Limited Measurability** metric. This metric is related to customer opinion, for example, the '*degree of customer*

satisfaction', and can be collected using a questionnaire only. Finally, the metric that does not belong to the previous two types is a Non-Measurable metric such as '*staff quality*' [133].

Measurable QoS metrics are the most desirable metrics since they can be quantified and evaluated [172]. These metrics can also be categorised according to simplicity into **Simple** and **Composite** metrics. The simple (basic or direct) metrics are obtained directly from the service by probing or instrumentation using measurement directives such as throughput of '*customer arrival*'. However, composite or computed metrics are derived by applying a function to a set of simple metric values such as '*maximum throughput of customers*' [109][133].

Another way of categorising QoS metrics is according to the **Perspective** of the **Service**, **Customer**, and **Business** [172]. In terms of service, QoS metrics are related to the IT infrastructure of the service from the provider's point of view. This perspective is related to the service itself and does not consider the customer's or business's point of view. An example of such a metric is service '*throughput*'. The customer perspective, on the other hand, is usually referred to as the Quality of Experience (QoE) and is related to the degree of customer satisfaction with the service; this could involve subjective factors. An example of a QoE metric is the '*response time*', as recognized by customers, from sending a request until receiving the response (i.e. including network delays). The last perspective is that of the business, which is referred to as Quality of Business (QoBiz) metrics. These metrics convey how the service provider or customer looks at the service based on the monetary value of the service's properties. An example of a QoBiz metric is the provider estimated financial loss per lost customer.

2.2.2 SLA Life Cycle and uses for Cloud Computing

In this subsection, we discuss and give a brief review of SLA life cycle steps in the cloud computing domain. Each step is described in detail highlighting its objectives and process [112].

The SLA has no widely accepted life cycle. SLA outlines the relationship between parties to understand each other's needs, preferences and expectations. It should include how to perform future service delivery including QoS levels required, performance-tracking techniques, performance reports, managing problems and conflicts, security and termination. During service delivery, SLA requires real-time verification. If violation occurs, appropriate action (e.g., penalty, conflict resolution) should be taken. Ron et al [147] defined SLA life cycle in three high level phases: the creation phase, operation phase and removal phase. Sun Microsystems Internet Data center Group defined a more detailed SLA life cycle in six steps (as shown in Figure 2.8):

- *Discover service providers* where service providers are located according to consumer's requirements.
- *Define SLA* includes definition of services, parties, penalty policies and QoS parameters. In this step it is possible for parties to negotiate to reach an agreement.
- *Establish agreement* an SLA template is established and filled in by specific agreement, and parties are starting to commit to the agreement.
- *Monitor SLA violation* the provider's delivery performance is measured against to the contract.
- *Terminate SLA* SLA terminates due to timeout or any party's violation.
- *Enforce penalties for SLA violation* if there is any party violating contract terms, the corresponding penalty clauses are invoked and executed.

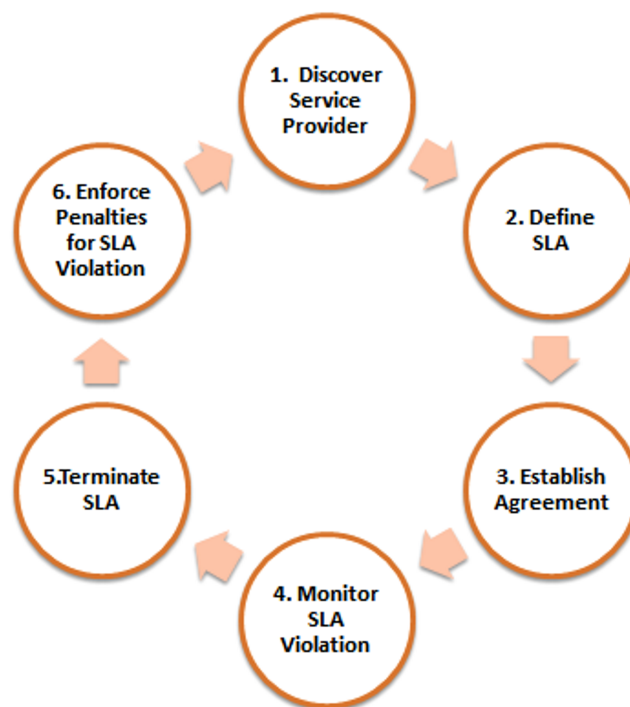


Figure 2.8 SLA Lifecycle

2.2.2.1 Discover - Service Provider

In current utility computing environments, especially Grid and Cloud, it is important to locate resources that can satisfy consumers requirement efficiently and optimally [64]. Such

computing environments contain a large collection of different types of resources, which are distributed worldwide. These resources are owned and operated by various providers with heterogeneous administrative policies. Resources or services can join and leave a computing environment at anytime. Therefore, their status changes dynamically and unpredictably. Solutions for service provider discovery problems must efficiently deal with scalability, dynamic changes, heterogeneity and autonomous administration.

2.2.2.2 Define - SLA

Once service providers have been discovered, it is necessary to identify the various elements of an SLA that will be signed by agreeing metrics. These elements are called service terms and include QoS parameters, the delivery ability of the provider, the performance target of diversity components of user's workloads, the bounds of guaranteed availability and performance, the measurement and reporting mechanisms, the cost of the service, the data set for renegotiation, and the penalty terms for SLA violation. In this stage of the SLA life cycle, measurement metrics and definition of each of these elements is done by a negotiation process between both parties [25] [42].

Other challenges are related to the negotiation process. Firstly, parties may use different negotiation protocols or they may not have the common definition of the same service [29]. Secondly, service descriptions, in an SLA, must be defined unambiguously and be contextually specified by the means of its domain and actor. Therefore, an SLA language must allow the parameterization of service description [106]. Moreover it should allow a high degree of flexibility and enable a precise formalization of what a service guarantee means. Another aspect is how to keep SLA definition consistent throughout the entire SLA life cycle.

2.2.2.3 Establish - Agreement

In this step an SLA template is constructed. A template has to include all aspects of SLA components. In utility computing environments, to facilitate dynamic, versatile, and adaptive IT infrastructures, utility computing systems have to promptly react to environmental changes, software failures, and other events which may influence the system's behavior. Therefore, how to manage SLA-oriented adaptive systems, which exploit self-renegotiation after system failure, becomes an open issue [28]. Although most of the works recognize SLA negotiation as a key aspect of SLA management, recent works only provide little insight on how negotiation (especially automated negotiation) can be realized. In addition, it is difficult to reflect the quality aspects of SLA components in a template.

2.2.2.4 Monitor - SLA Violation

SLA violation monitoring begins once an agreement has been established. It plays a critical role in determining whether SLOs are achieved or violated. There are three main concerns. Firstly, which party should be in charge of this process. Secondly, how fairness can be assured between parties. Thirdly, how the boundaries of SLA violation are defined.

SLA violation means '*un-fulfillment*' of service agreement. According to the Principles of European Contract Law, the term '*un-fulfillment*' is defined as defective performance (parameter monitored at lower level than agreed), late performance (service delivered at the appropriate level but with unjustified de-lays), and no performance (service not provided at all). There are three broad provisioning categories based on the above definition [143]. 'All-or-Nothing' provisioning, characterizes the case in which all SLOs must be satisfied or delivered by the provider. 'Partial' provisioning identifies some SLOs as mandatory ones, and must be met for the successful service delivery by both parties. 'Weighted Partial' provisioning, is the case in which the "provision of a service meets SLO if it has a weight greater than a threshold (defined by the client)" [143]. 'All-or-Nothing' provisioning is used in most cases of SLA violation monitoring, because violation leads to complete failure and negotiation to create a new SLA. An SLA contains mandatory SLOs that must be delivered by the provider. Hence, in 'Partial' provisioning, all parties assign these SLOs the highest priority to reduce violation risk. How much the SLO affects the 'Business Value' a measure of the importance of a particular SLO term. The more important the violated SLO, the more difficult it is to renegotiate the SLA, because any party does not want to lose their competitive advantages in the market.

Monitoring infrastructures are used to measure the difference between the pre-agreed and actual service provision between parties [143]. There are three types of monitoring infrastructures, which are trusted third party (TTP), trusted module on the provider side, and trusted module on the client side. Nowadays, TTP provides most of the functionalities for monitoring in most typical situations to detect SLA violation.

2.2.2.5 Terminate - SLA

There are two scenarios in which an SLA may be terminated. The first is termination due to normal time out. The second one is termination because any party violated its contract terms. Normally, in Clouds, this step is conducted by customers and termination typically is caused by normal time out or the provider's SLA violation. Sometimes, providers also terminate SLAs depending on the task priorities. If the reason for SLA termination is violation, then

the Enforce Penalties for SLA Violation step of the SLA life cycle has to be applied. Usually this step is performed manually.

2.2.2.6 Enforce Penalties for SLA Violation

A penalty clause can be applied to the party who violates SLA terms. First is a direct financial compensation being negotiated and agreed between parties. Second is a decrease in price along with the extra compensation for any subsequent interaction. In other words, this option is according to the value of loss caused by the violation. In this case, TTP is usually used as a mediator (see figure 2.9). The workflow for this option is that clients transfer their deposit, bond, and any other fees into the Third Party's account, and then if the SLOs have been met, the money is paid to provider via TTP. Otherwise, the TTP returns the amount of fees back to the consumer as compensation for SLA violations. The SLA violation has two indirect side impacts on providers. The first is that consumers will use less service from the provider in the future. The second is that provider reputation decreases and it affects other clients willingness to choose this provider subsequently. The major indirect influence on consumer is that future request will be rejected due to bad credit record.

A major issue, in the above discussion, is the variety of laws enforced in different countries. This problem can be solved by a 'choice of law clause', which indicates explicitly which country's laws are applied when a conflict occurs between parties. 'Legal templates' [52] can be used to refine these clauses [143].

2.2.3 SLA Life Cycle : Functionalities and Use Cases

In this subsection, we give an overview of SLA life cycle functionalities. For a better understanding, these functionalities are illustrated using UML use case diagrams [112]. UML breaks the complex system into discrete pieces that can be understood easily. Complex system can be understood by the disparate developers who are working on different platforms. UML model is not a system or platform specific. It unifies all disparate developers under one roof.

2.2.3.1 SLA Discovery

SLA Discovery Use Case enables Cloud customer to access/view to SLA Templates related to a selected service, figure 2.10. The objective of this use case is to allow customers to check Service Level Agreements service and guarantee terms a provider has established for a given service. Multiple SLA templates can be defined depending on the service category.

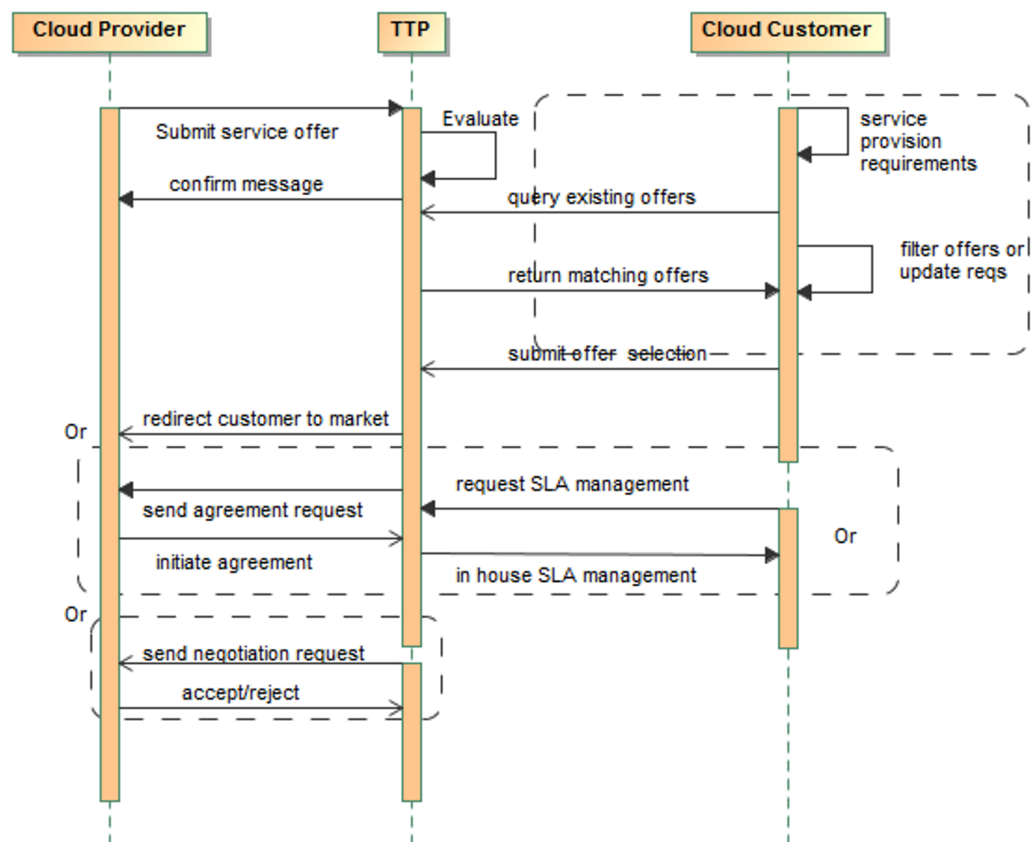


Figure 2.9 TTP communication

2.2.3.2 SLA Template Management

SLA Template Management Use Case considers all necessary operations to allow providers to publish and unpublish SLA Templates, as shown in figure 2.11.

- The Cloud service provider creates an SLA template.
- This template includes the QoS metrics to be monitored, and the boundaries for these metrics.
- The provider can also view, update or delete SLA templates.

2.2.3.3 SLA Negotiation Scenario

SLA negotiation is the most significant stage to extract a vital SLA due to the following: first, it helps a lot in resolving conflicts that can occur between the involved parties to reach an appropriate trade-off. Second, the different service aspects like QoS cannot be prefixed since the consumers needs vary from one to another. Finally, the dynamic change of the



Figure 2.10 SLA Discovery

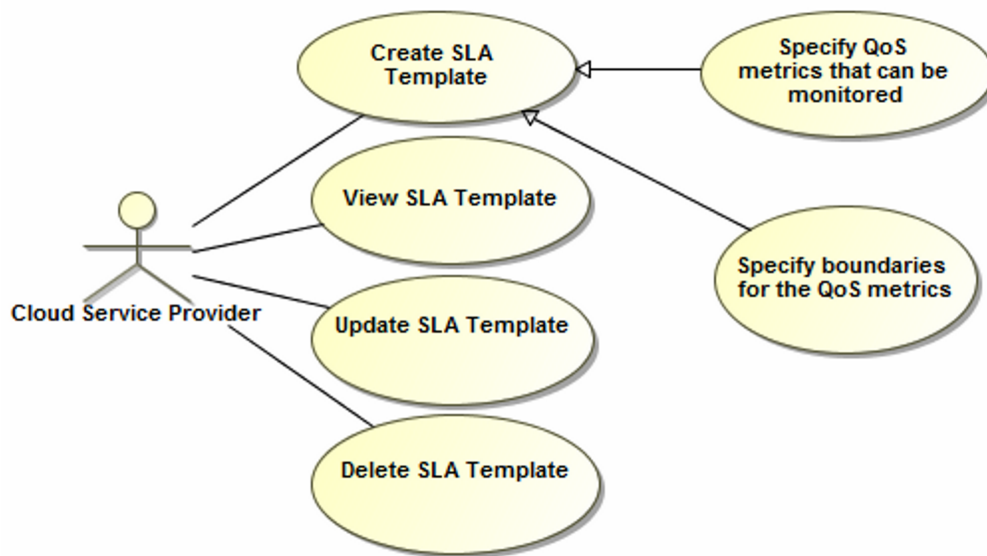


Figure 2.11 Use Case, SLA Template [168]

demand and the supply style that indeed affect in the creation of the SLA [Salem et al.]. SLA negotiation is needed during the stage of establishing SLA offers. SLA negotiation (or renegotiation) is also needed during the stage of enacting SLA. During the SLA establishment, the negotiation will support the service composition activity. During the SLA enactment, the negotiation will support the recovery activity between a cloud service customer (CSC) and Cloud service provider (CSP) for a service/resource. The ideal SLA negotiation flow is illustrated by the use case diagram in Figure 2.13.

1. The Cloud service provider registers a service / resource in the Resource Catalogue.
2. As part of the registration process, the provider defines an SLA template for this resource. The provider specifies the QoS (Quality of Service) metrics that can be monitored, and the QoS terms (boundaries for these metrics).
3. The cloud customer looks for a service / resource in the Resource Catalogue.

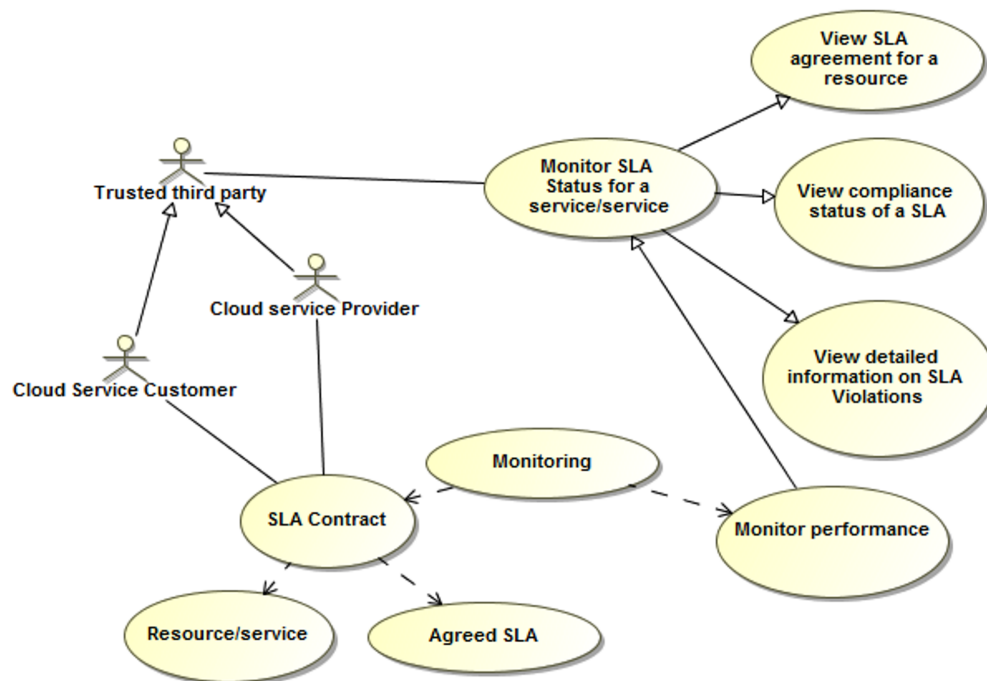


Figure 2.12 Use Case, SLA monitoring [168]

4. The CSC selects a resource and views the SLA definition associated with this service / resource.
5. The CSC accepts or rejects the SLA offer based on whether the SLA terms specified in the SLA offer fits their needs.

2.2.3.4 SLA Establishment

SLA establishment is a phase where an SLA is negotiated between the service consumer and the service provider to potentially reach a mutual agreement. The process of establishing the SLA can be generally classified into three cases. The first case refers to a situation where an agreement is achieved with a simple negotiation, as illustrated in figure 2.14. The second case refers to a situation where an agreement is achieved with a bargaining negotiation. The third case refers to a situation where an agreement cannot be achieved due to the termination state [Salem et al.].

2.2.3.5 SLA Monitoring and Violation

The service provider has to deliver the service that has been contractually approved during the SLA negotiation [107]. Essentially, monitoring, measuring and recording the performance of

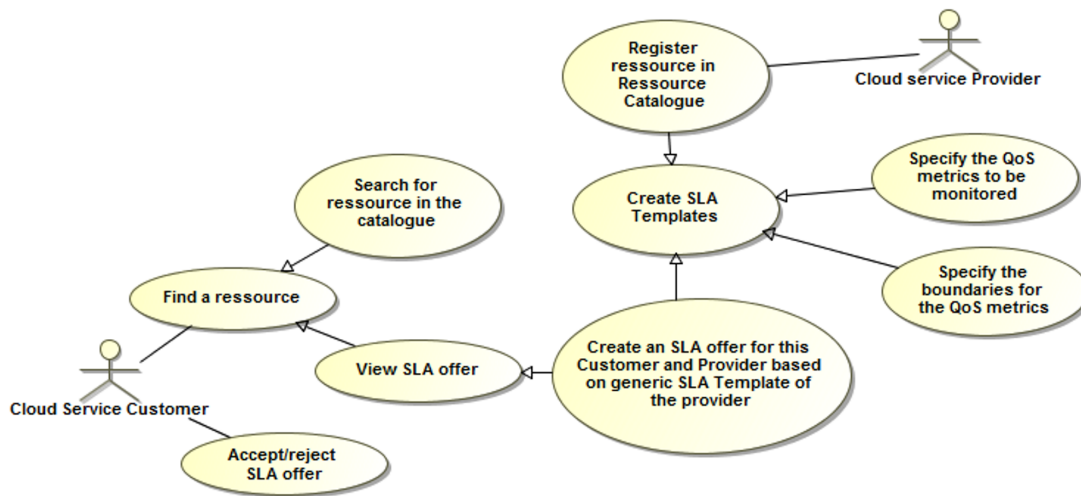


Figure 2.13 Use Case, SLA negotiation [168]

the service in the SLA operation phase play a critical role in determining whether Service Level Objectives (SLOs) are achieved or violated. The monitor service activity monitors the delivered service quality with respect to service levels as defined in the SLA between cloud service customer and cloud service provider. Monitor the status of an SLA Agreement (that was created between a CSC and CSP) for a service/resource. Detect a violation of the QoS in a SLA contract, store it and decide if the penalty should be put, notify the customer and notify the service provider. These decisions depend on the SLA contract agreed for the customer and the boundaries within, see figure 2.12.

1. The actor (Cloud service provider or customer) can view the SLA Agreement for a service / resource.
2. The actor can view the compliance status of the SLA Agreement, i.e. whether the QoS terms in the SLA agreement have been met or not.
3. The actor can view the SLA violations in detail, i.e. which QoS term have not been met, when, etc...
4. The actor can view the performance of the services related with the SLA.

2.2.3.6 SLA Conclusion

During the SLA conclusion phase, signed SLAs are terminated successfully (service delivery concluded or SLA validity period is over) or as violated agreements. The providers use accounting and billing mechanisms in order to provide the required information to the

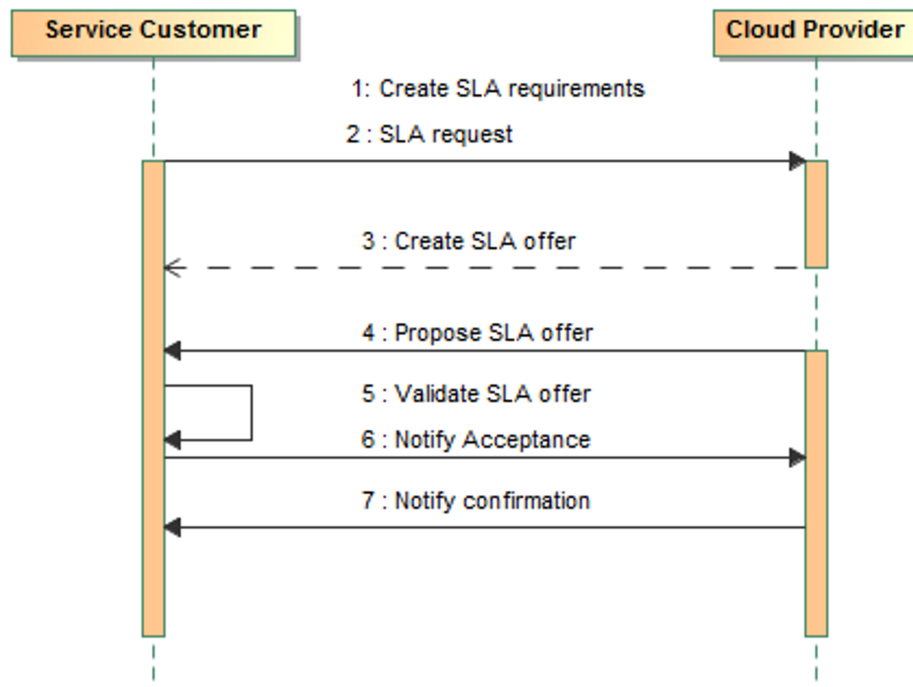


Figure 2.14 SLA Establishment

customers. If the SLAs have been violated, the corresponding compensations/penalties are calculated during the resolution process. Furthermore, in the case of multi-provider environments (e.g. cloud federations or composite services) resolution includes revenue sharing for the engaged providers.

We use XML as a representation format. The following XML (see Listing X) presents an example of a SLA file describing the penalties for SLA between the fore-mentioned provider and customer concerning the service SaaS. For each penalty, a procedure indicates the actor in charge of the violation notification (e.g., provider), the notification method (e.g., email) and the notification period (e.g., 7 days).

```

1 <penalty id="p-Rt" objective="responseTime" description="Rt_not_respected" notificationMethod="e-mail" condition="violation" obligation="provider" unit="dollar/request" notificationPeriod="7_days"/>
2 <penalty id="p-Av" objective="availability" description="Av_not_respected" notificationMethod="e-mail" condition="violation" obligation="provider" unit="dollar/request" notificationPeriod="7_days"/>
3 <penalty id="p-St" objective="Storage" description="St_not_respected" notificationMethod="e-mail" condition="violation" obligation="provider" unit="dollar/request" notificationPeriod="7_days"/>

```

```

4 <penalty id="p-Me" objective="Memory" description="Me_not_respected"
  notificationMethod="e-mail" condition="violation" obligation="
  provider" unit="dollar/request" notificationPeriod="7_days"/>
5 <penalty id="p-IB" objective="Incoming_Bandwidth" description="IB_not
  _respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request" notificationPeriod="7_
  days"/>
6 <penalty id="p-OB" objective="Outgoing_Bandwidth" description="OB_not
  _respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request" notificationPeriod="7_
  days"/>

```

2.2.4 Overview of the Existing SLA Definition Languages

Until a few years ago, SLA contracts were mostly written using natural expressions; examination of compliance to the agreement also had to be done manually [83]. One attempt to facilitate this process by using SLA templates was limited and unable to specify different service levels for different customers [83]. For this reason, it has become a necessity to automate the procedure through which different SLAs are flexibly described, provisioned and observed [113].

Several SLA specification languages have been developed by researchers within the service provision community to address the previous aspects [83]. Their aim is to simplify the contractual process for the parties involved and to minimize the time and cost included in this process. Many projects and solutions focus on the definition of SLA [41][181][136][79][101][40]. We report here only work directly related to the Web services and Cloud services.

In the following subsections, WSLA, WS-Agreement, SLAng, SLA*, CSLA and SLAC are described in details. Then, a comparison of these languages in terms of the requirements of the proposed methodology is presented [113].

2.2.4.1 SLA @ Web services

There are two main specifications for describing a SLA for web services. In what follows we introduce WSLA and WS-Agreement which represent the most successful results.

- ***Web Service Level Agreement (WSLA)***

WSLA [109] was introduced by IBM research as a framework for the definition and monitoring of SLAs for web services. The WSLA framework provides a formal language, an XML schema and a run-time architecture that interprets the language and handles SLA management

tasks. According to WSLA [109], service level agreements are contracts between a service provider and a customer that specify the agreed QoS and define service level objectives (SLOs). An SLA may be monitored by either or both signatories. An SLA depicts anticipated service levels with respect to business and technical level objectives. The same service may be offered in diverse levels. WSLA describes all the aspects that are contracted between the signatory parties, the service supplier and the service consumer, regarding a specific service. The negotiation process for establishing such a document could be accomplished either on-line or off-line through the use of a WSLA template that comprises most of the defined and the agreed upon information needed to create the SLA contract [83].

WSLA represents, in its agreement, all the information that is normally contained within an SLA document. This information is situated, as depicted in figure 2.15, in the following three main parts:

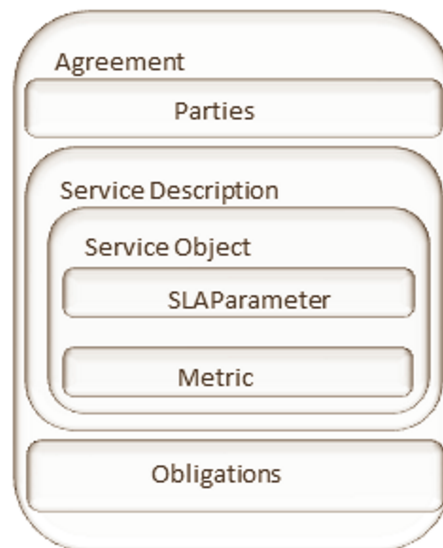


Figure 2.15 WSLA agreement structure

- **Parties:** contains information about the parties engaged in the SLA contract. These are the signatory parties, the service provider and the service consumer, in addition to the supporting parties that may be involved in measuring, monitoring or managing particular parts of the contract [109][83].
- **Service Description:** contains an explanation of the Service Object on which the QoS metrics are defined, its SLAParameter(s), and what Metric(s) are used to compute their values. A metric will be measured from a source by identifying a measurement directive, if it is basic. However, in the case of a composite metric, its value will be

computed using a function that takes other metrics or constants as its operands. The SLA parameters are one of the important parts of an SLA because they correlate the metrics to a particular consumer with specific accepted values [109][83].

- **Obligations:** contains Service Level Objectives the service provider is obliged to maintain. These are the agreed values of SLA parameters during a specific duration and the actions to be taken in the case of a contract violation [109][83].

Figure 2.16 illustrates the major components of WSLA in a UML class diagram. The latter offers the possibility to enrich the directory of performance criteria by creating new metrics. The latest version of the WSLA specification was published in 2003 [109]. Since then, WSLA is integrated into WS-Agreement. Despite specification discontinuation and latest version semi-stable in 2003, WSLA remains one of the most mature defining a SLA specifications. This specification covers the entire life cycle of SLA. However, the management of penalty is limited mainly to the notification in case of violation.

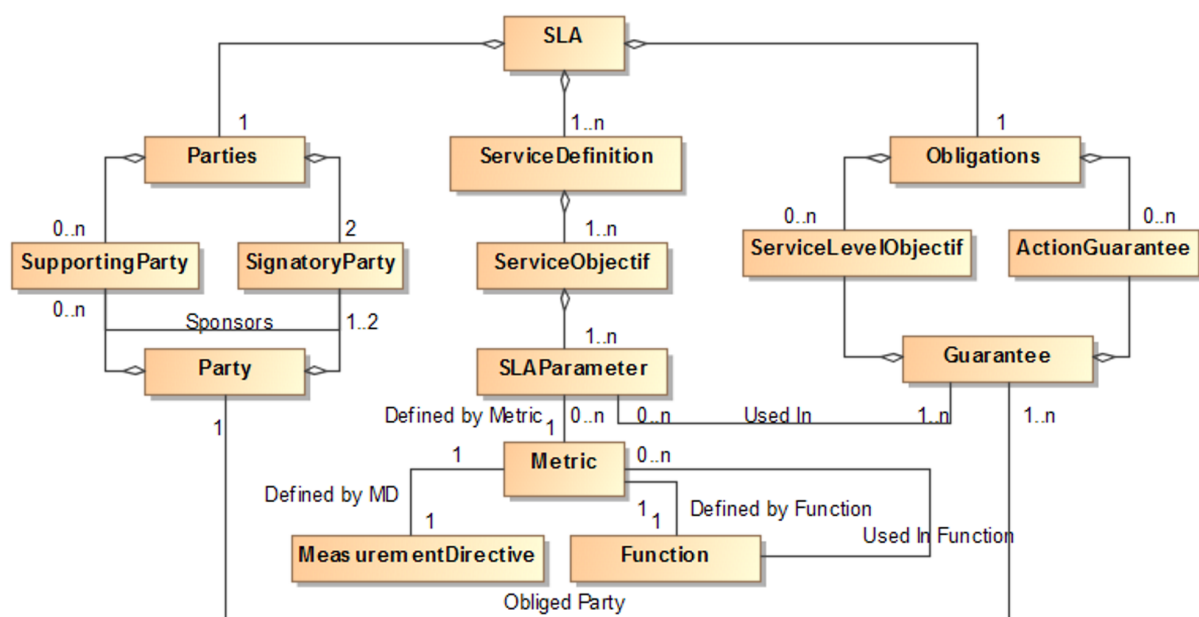


Figure 2.16 WSLA Meta-model[135]

• *Web Service Agreement Specification (WS-Agreement)*

WS-Agreement is an SLA specification presented by the Grid Resource Allocation and Agreement Protocol Working Group of the Compute Area of the Open Grid Forum [18]. It is an XML-based language and web service protocol for: firstly, promoting through templates a

set of possible accepted agreement offers from agreement responders (which could be the service provider or consumer); secondly, generating a proposed agreement offer that the agreement initiator is keen to establish based on one of these templates; thirdly, negotiating this agreement according to specific constraints; fourthly, creating an agreement between the service provider and customer with all conditions and restrictions, and then finally observing its fulfillment [18]. The WS-Agreement is structured in three parts [18] as follows:

1. *Agreement Schema* (agreement creation offer schema): This is used by the agreement initiator to create an offer according to a specific template. The agreement creation offer and the agreement are structurally the same. The agreement offer, as specified in figure 2.17, contains the agreement Name, the Context (this includes the involved parties and the agreement life span), and the Terms, which is the most important part of an agreement offer. Within the Terms, each term contains at least one Service Term and zero or more Guarantee Terms which could be combined using logical operators. These are defined as follows:



Figure 2.17 WS-Agreement agreement structure, as specified by Andrieux et al. [18]

- **Service term:** is used to describe the service to be offered and consists of three parts: Service Description terms (SDTs) for describing the service functionality, Service Reference terms to identify a service, and Service Property terms to identify quantifiable measures (e.g. response time) that are used to define service level objectives.
- **Guarantee terms:** identify the agreed quality level of the offered service that is specified in the service term. The Guarantee term consists of many parts such

as the service scope which names the services for which this guarantee is valid; the qualifying conditions that make the guarantee obligatory; the service level objectives, which define the required quality of the service; and the business value list that includes the SLO's importance and the agreed penalties and rewards.

2. *Agreement Template Schema* : This is used by the agreement responder to promote acceptable agreement offers.
3. *Set of ports type and operations*: These are used for organizing and administrating the agreement life-cycle operations, such as accepting or rejecting the offer, or identifying a type of a document for monitoring the agreement states.

Figure 2.18 illustrates the basic UML metamodel structure of an agreement according to WS-Agreement.

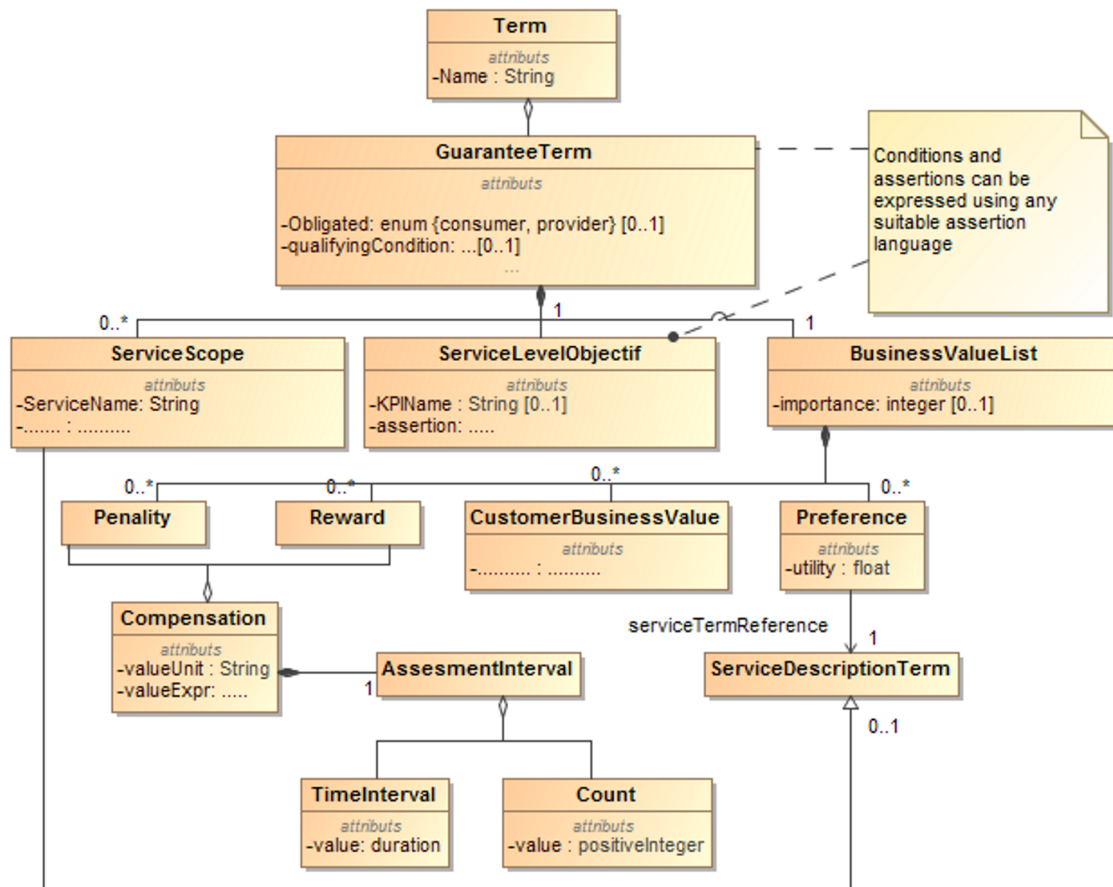


Figure 2.18 WS-Agreement Meta-model[96]

One of the main strengths of WS-Agreement is that it is powerful in terms of extending new elements that are domain-specific [18]. In addition, it contains more information

about the service functional properties than WSLA does. Also, business values related to QoS metrics can be specified even if an accounting procedure is not supported [81]. WS-Agreement does not support a constructive ontology to define QoS metrics which is necessary to understand their exact definitions. Also its semantics are not defined precisely. However, WS-Agreement has recently been extended with SWAPS Extension [18] to overcome this semantic ambiguity.

- ***Service Level Agreement Language (SLAng)***

The Service Level Agreement Language (SLAng) is a language that describes a domain-specific SLA between a service provider and a consumer. It is one of the deliverables of the Trusted and Quality of Service Aware Provision of Application Services project. Its syntax is defined by an XML schema that allows it to be combined with existing service description languages (e.g. WSDL) or other technologies to allow a complete business clarification [157].

The semantics of SLAng are described by producing a UML model of the language rules which is then used in the service's behavioral model (including models for the involved parties). By producing this abstract syntax, the SLA constraints, which are described using OCL (Object Constraint Language, which is part of the UML standard), comprise the semantics of SLAng [157].

The specification of SLAng is derived from a reference model of a distributed system architecture. This reference model consists of three tiers: *Application*, *Middle*, and *Underlying resources* which contain an *Application*, an *Application Service*, *Container*, *Storage* and *Network* providers [157].

An SLAng contract consists of a specification of the involved parties, contract information (contract lifetime), and a Service Level Specification (SLS) which specifies the QoS metrics and their related values, the provider roles, the user roles, and their shared roles [157].

SLAng differs from the other SLA specifications in many aspects [157] such as:

- Unlike most SLA specifications that concentrate on SLAs for web services only, SLAng is used for a wider range of Internet services (such as the application service provision, Internet service provision, and storage service provision).
- SLAng represents service and client behavior in a formal semantic. This decreases vagueness in the meaning of the language and minimises the need to re-check it to eliminate any contradiction or flaw. Moreover, it makes SLAng more user-friendly and simplifies SLA negotiation.

SLAng however suffers from the absence of certain characteristics. For example, it does not contain a description pertaining to the actions the obliged party will take or the charges

that this party will incur if a contract is breached. In addition, unlike WSLA, its structure does not allow the submission of only a part of the SLA to a monitoring party. This causes information about the parties and the service to be revealed to such a party. Furthermore, SLAng is unable to identify new parameter types according to existing ones [157].

2.2.4.2 SLA @ Cloud

Regarding cloud services, NIST [120] proposed a definition of SLA. However, no attempt has been made to implement this definition until now. Similarly, Cloud Standards Customer Council [49] presented a practical guide to SLA which is a collaborative effort that brings together different experiences based on the client. Thus, Rackspace [10] defined a set of metrics through cloud monitoring to establish an SLA contract between two players Cloud. In addition the industrial solutions, several academic projects have been developed to meet the proposed limits by cloud services. In particular the project SLA@SOI offers the new language SLA* [82]. In what follows, we describe two major technologies SLA*, CSLA and SLAC that consider SLA in cloud environments.

• SLA*

The SLA* [82] language is part of the SLA@SOI project, which aims at providing predictability, dependability and automation in all phases of the SLA life cycle. SLA@SOI propose SLA* [82] an abstract syntax for SLA. SLA*, is a domain-independent syntax for machine-readable Service Level Agreements and SLA templates [82]. Historically, SLA* was developed as a generalization and refinement of the web-service specific XML standards: WS-Agreement, WSLA, and WSDL. Instead of web services, however, SLA* deals with services in general, and instead of XML, it is language independent. SLA* provides a specification of SLA(T) content at a fine-grained level of detail, which is both richly expressive and inherently extensible: supporting controlled customization to arbitrary domain-specific requirements. The model was developed as part of the *FP7 ICT* Integrated Project SLA@SOI, and has been applied to a range of industrial use-cases, including; ERP hosting, Enterprise IT, live-media streaming and health-care provision. At the time of writing, the abstract syntax has been realised in concrete form as a Java API, XML-Schema, and BNF Grammar.

The solution is very expressive and extensible. SLA* promote the formalization of SLA in any language for any service by eliminating the restriction of XML. According to SLA@SOI, SLA is an SLA template with an extended attribute specifying the validity of the SLA contract. An SLA template contains five sections see figure 2.19: i) attributes template SLA, ii) the parties to the agreement, iii) service descriptions, iv) variable declarations, and v) the terms of the agreement, specifying the guarantees of Quality of Service. The parties in the

abstract syntax of SLA(T) are identified by their role (supplier, customer). The description of services is defined by the statements of the interface. A declaration interface is used to assign an identifier to a local interface which may be a functional interface or a description of a resource. Variable declarations are provided to improve readability and avoid duplication of content. The terms of the agreement are formalized as guarantees of two types: Action warranty and status. In addition the simple expressions (for example, A constant, or a linear proportionality) SLA* propose a formal model to formalize penalties. The main motivation of this proposal is the opening and applicability to different domains without dependency on specific languages, taxonomies or technologies.

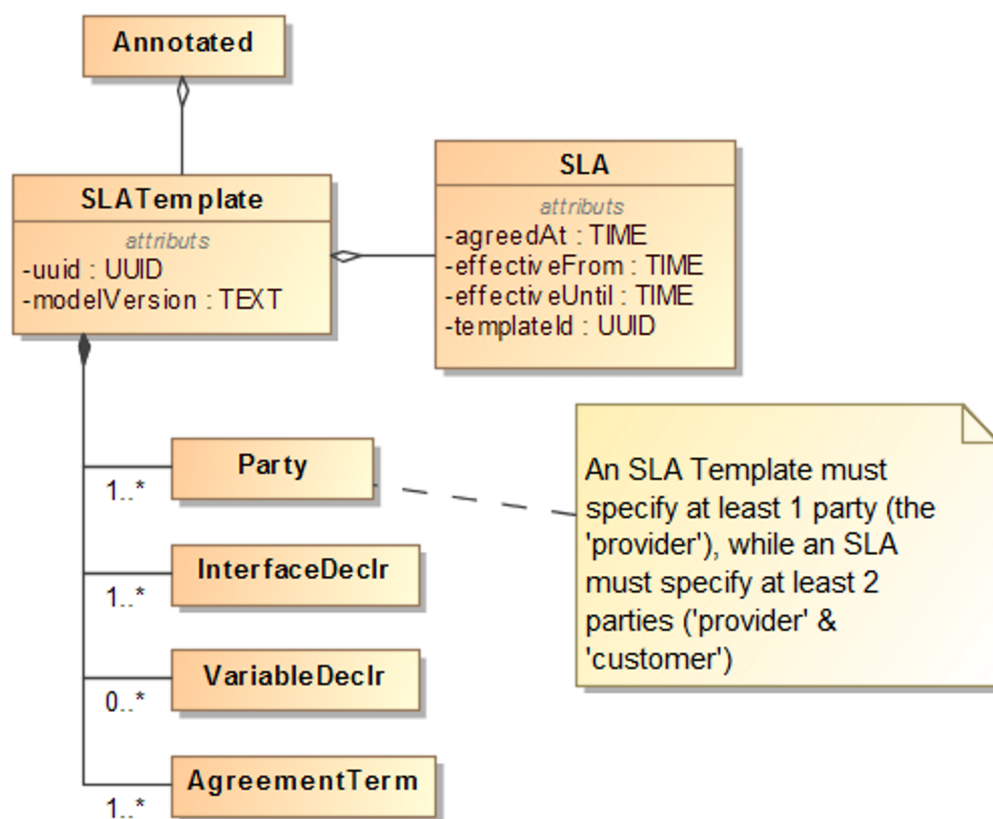


Figure 2.19 SLA* (SLA@SOI) Meta-model[96]

• CSLA

An SLA language to finely express SLA and address SLA violations in the context of Cloud services [97]. CSLA language has been influenced by related work, in particular WSLA and SLA@SOI [109],[82]. It allows to describe the SLA between a cloud service provider and a cloud customer. Thus, it describes QoS guarantees in the form of SLO

clauses. Besides the standard formal definition of contracts – comprising validity, parties, services definition and guarantees, CSLA [154] is enriched with features (QoS/functionality degradation and an advanced penalty model) introducing a fine language support for Cloud elasticity management.

CSLA allows to define SLA in any language for any Cloud service (XaaS). CSLA addresses intrinsically (i) QoS uncertainty in unpredictable and dynamic environment; (ii) the cost model of Cloud computing. Its structure is similar to WS-Agreement. and

Conceptual Model : A SLA in the CSLA language contains three sections see figure 2.20: a section describing the validity of the agreement, a section defining the parties involved in the agreement and a section referencing the template used to create the agreement. The validity defines how long an agreement is valid. CSLA distinguishes two types of parties: Signatory parties, namely service provider and service customer, and Supporting parties (e.g., trusted third party). A CSLA template is like a pattern for SLA. It contains five elements: Cloud services definition, Parameters, Guarantees, Billing and Terminations. A *Cloud service definition* refers to any XaaS service (SaaS, PaaS or IaaS). In order to deal with unpredictable and dynamic environment, we propose the concept of functionality degradation (i.e., normal vs degraded mode) for each SaaS and PaaS services (e.g., 3D vs 2D display). *Parameters* provide a way to define variables in the context of the agreement. *Guarantees* contain a set of guarantees. Each guarantee consists of four elements: Scope, Requirements, Terms and Penalties. CSLA supports two types of *billing*: Pay as You Go and All-in package. Finally, the Agreement starts on effectiveFrom and ends on the earlier of the effectiveUntil or in accordance with *Terminations* section.

• *Service-Level-Agreement for Clouds (SLAC)*

SLAC [171] is a language to define SLAs specifically devised to the cloud computing domain, which is inspired by WS-Agreement and shares many features with the definitions and structure of this language. SLAC was defined as a domain specific language for clouds. The metrics available in the language are pre-defined in the light of the requirements of the cloud domain. The set of characteristics of the SLAC language, such as multi-party, group definition and specification of the involved parties on each term, enables and leverages the definition of SLAs including a broker[171].

This core-language provides the basic ingredients that enable the description of a simple SLA for the cloud computing domain. Differently from most of the existing SLA definition languages, SLAC does not differentiate service description terms and quality requirements. The formal definition of the syntax of the core language is defined in the Extended Backus Naur Form (EBNF)[170].

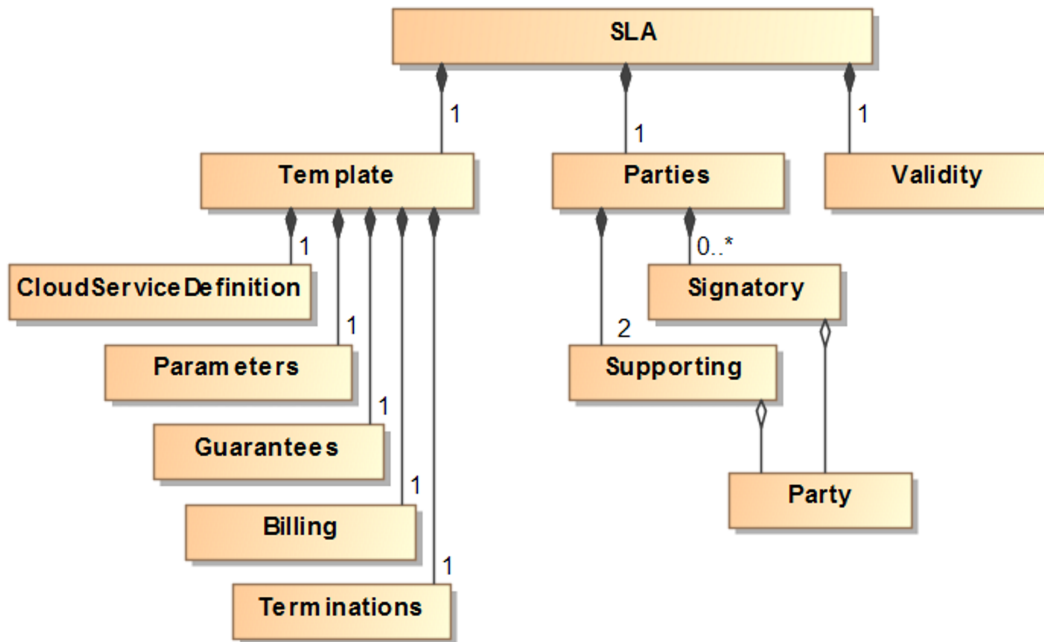


Figure 2.20 CSLA Meta-model[96]

The semantics of the evaluation of a SLA is formulated as a Constraint Satisfaction Problem (CSP) that verifies: (i) at negotiation-time, whether the terms composing the agreement are consistent; and (ii) at enforcement time, whether the characteristics of the service are within the specified values. More specifically, the formal semantics of SLAC is given in a denotational style[170]. Denotational semantics is based on the recognition that the elements of the syntax are symbolic realizations of abstract mathematical objects or functions. Although originally developed to describe the behaviour of imperative programming languages, the denotational approach is also appropriate for declarative languages (as required for SLAs) due to its capacity to translate the static and dynamic elements of the domain. This use of the semantics provides the formalism that is needed to express the meaning of the elements of a SLA definition language.

Finally, it is worth noticing that SLAC focusses on Infrastructure-as-a-Service (IaaS) [171]. Since most requirements of the other service models are also taken into account, we do not envisage any major issue in extending SLAC to cover them, especially the case of Platform-as-a-Service (PaaS). We plan to specifically address the other models in future works.

2.2.5 Comparison of the SLA definition languages

In this section, we present a comparative study of SLA languages highlighting their strengths and weaknesses. Table 2.1 shows the comparison between the SLA definition languages according to the domain requirements. For the sake of simplicity and standardization, we classify the languages in three levels: the ■ symbol represents a feature covered in the language, □ stands for a partially covered feature and - represents a not covered feature[171].

- *Cloud Domain* - This criterion defines when a language was devised for the cloud domain. Languages for a broader domain (e.g. IT Services) are not considered to cover the criterion[170];
- *Cloud Delivery Models* - Fully supported when a language covers IaaS, PaaS and SaaS, and partially supported when only one of these models is covered;
- *Multi-party* - Fully supported when the language enables the use of: (i) all roles defined; (ii) multiple-parties with the same role; and (iii) multiple roles for the same party[170]. Partially supported when at least one of the characteristics is present;
- *Broker support* - To support the broker, a SLA definition language should enable the specification of: (i) the parties involved in each term, which define the parties in charge of providing and consuming the service; (ii) offers and requests[170], which enable the creation of specification of services needed by consumers and of services available by providers; (iii) the role of the broker in the party's definition; and (iv) the actions and metrics which enable the parties to state who is the responsible for the service, who consumes it, who pays for it and the parties involved in each action. When at least two of these characteristics are supported we determine that the language partially fulfils this criterion;
- *Ease of Use* - This criterion is considered fully supported when a language is ready to be used in the domain [171](without or with minimal extensions) and, at the same time, it is easy to understand, also for human actors. If a language has only one of these characteristics it is considered as partially supported;
- *Business Metrics and Actions* - Fully supported when the language provides metrics related to the business aspects, such as performance indicators (KPIs) and enables the use of actions which express common business behaviours, for example the payment of penalties. It is partially supported if at least one of them is available in the language[170];

		WSOL	SLAng	WSLA	WS-Agreement	SLA*	CSLA	SLAC
General	Cloud Domain	-	-	-	-	□	■	■
	Cloud Service Models	-	-	-	-	□	■	□
	Multi-Party	-	-	-	-	-	■	■
	Broker Support	-	-	-	-	-	□	■
	Ease of Use	□	□	-	□	□	■	□
Business	Business Metrics	□	□	□	□	□	■	■
	Price schemes	□	-	□	□	□	■	■
Formal	Syntax	■	■	■	■	■	■	■
	Semantics	-	□	-	-	-	■	■
	Verification	-	□	-	-	-	■	■
Tools	Evaluation	■	■	■	■	■	■	■
	Open-Source	■	■	-	■	■	■	■

Table 2.1 Comparison of the SLA languages according to the domain requirements[171][113]

- *Price schemes* - The language should consider flat and variable pricing schemes and the main models of the domain, which are fixed pricing, bilateral agreement, exchange, auction, posted price and tender. A language offers partial support when it includes at least one type of variable pricing;
- *Formal Syntax* - Supported only when a formal definition of the syntax, e.g using BNF[170], is available;
- *Formal Semantics* - Fully supported if a formal definition of the SLA evaluation process is available. By formal we intend the use of any recognized formal tool (e.g. operational or denotational semantics) to avoid ambiguity on its interpretation. It is partially supported if possibly ambiguous tools are used in its definition;
- *Formal Verification* - Fully supported when the formal verification of agreement exists before execution time. We consider as partially supported if only the syntax is verified;
- *Evaluation Tools* - A language fully supports it when an implementation of a tool which evaluates the SLAs against the services monitoring information is available. If only design time verification is available this characteristic is only partially supported;

Although we specify the criteria for the comparison of the languages, their classification is subjective since they are not quantitative[170]. However, this comparison provides the base to understand to which extent the existing languages support the domain. The results of the comparison among the language show that the existing languages do not support some of the main requirements of the domain.

2.3 Model-Driven Engineering

After presenting an overview of SLAs in section 2.2 and cloud computing concepts in section 2.1, we now discuss Model-Driven Engineering. One the objectives of our thesis is to provide mechanisms for modelling cloud SLAs. In this thesis, we present cloud SLAs as standard artefacts to automate the process of cloud SLA establishment and management. Thus, to achieve these goals and as explained in the introduction, we employ MDE; hence, this section gives an overview of MDE concepts. MDE involves the key concepts of: models, metamodels, model transformations and model management. This section describes these MDE artefacts and relates them to the previous discussion on cloud computing and SLAs.

2.3.1 Model-Driven Engineering and Model-Driven Architecture

- *Model-Driven Engineering*

Model-Driven Engineering (MDE) refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. Models are considered as first class entities, they are better than implementing code for describing about software system and its parts for the stakeholders. MDE can be applied to software, system, and data engineering. It offers a promising approach to address the inability of third generation languages to reduce the complexity of platforms and express domain concepts effectively [153]. Applying MDE approach for the deployment process helps to encapsulate the deployment and management of a cloud system into a series of procedural operations. Schmidt et al. [153] summarized that a promising approach to address the system complexity is to develop MDE technologies that combine 1) domain-specific modeling languages (DSML) whose type systems formalize the application structure, behavior, and requirements within particular domains; and 2) transformation engines and generators that analyze certain aspects of models and then synthesize various types of artifacts, such as source code or alternative model representations. Moreover MDE offers various facilities for building programming frameworks based on the structure of DSML metamodels. These facilities have different degrees of automation, ranging from

fully automatic programming-framework generation (for example, with support for model persistency, model validation, transactions, and so on) to simple Java class generation.

- ***Model-Driven Architecture***

Model-Driven Architecture (MDA) is an MDE approach proposed by the Object Management Group (OMG)⁵. As defined by the OMG, MDA encourages the efficient use of system models in the software-development process, and it supports reuse of best practices when creating families of systems. MDA is a way to organize and manage enterprise architectures supported by automated tools and services for both defining the models and facilitating transformations between different model types [30]. The main goal of MDA is to support engineers to model applications, or systems independently of specific platforms. MDA defines system functionality using a platform independent model (PIM) using an appropriate domain-specific language (DSL). Then, the PIM model is translated into one or more platform-specific model (PSM) that computers can run. This requires mappings and transformations. One of the particular importance to model-driven architecture are the notions of metamodel and model transformation. Metamodels are defined at the OMG using the MOF (Meta Object Facility) standard. A specific standard language for model transformation called QVT has been defined by OMG. The OMG has also defined a model interchange mechanism based on XML called XMI.

2.3.2 Domain-Specific Modeling

Domain-specific modeling (DSM) is a specific software engineering method for designing and developing systems based on the systematic use of a domain-specific modeling language (DSML). DSMLs tend to support higher-level abstractions than general-purpose modeling languages, therefore they require less effort and fewer low-level details to specify a given system. DSMLs are described using metamodels, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively [153].

2.3.3 Model, Metamodel and Modeling, Metamodeling

- **What is a model and metamodel?**

⁵<http://www.omg.org/mda/>

"We use models when we think about problems, and when we talk to each other, and when we construct mechanisms, and when we try to understand phenomena, and when we teach. In short, we use models all the time" [108]. *"A model is a simplified representation of an aspect of the world for a specific purpose. Complex systems typically give rise to more than one model because many aspects are to be handled"* [74]. A model is usually considered as an abstraction, a description, or a specification of (some aspect of) a system [50][60]. While a model is an abstraction of phenomena in the real world; a metamodel is yet another abstraction, highlighting properties of the model itself. A model conforms to its metamodel in the way that a computer program conforms to the grammar of the programming language in which it is written.

- **What is modeling and metamodeling?**

Modeling not only represents a solution at a higher abstraction level than code, but also is a specificity of engineering that engineers build models of artifacts that usually do not exist yet [74]. It is indeed one of the standards of any scientific activity along with validating models with respect to experiments carried out in the real world. Metamodeling is the analysis, construction and development of the frames, rules, constraints, models and theories applicable and useful for modeling a predefined class of problems. As its name implies, this concept applies the notions of meta- and modeling. In short, "Metamodeling" is the construction of a collection of "concepts" (things, terms, etc.) within a certain domain. The relationships between model, metamodel, modeling and metamodeling are illustrated in figure 2.21.

2.3.3.1 Models

Models are the core artefacts in MDE. As in object-oriented technology, where *"Everything is an Object"*, MDE is based on the principle that *"Everything is a model"* [23]. There have been many definitions of the term model: *"A model is a representation of a concept. The representation is purposeful: the model purpose is used to abstract from the reality the irrelevant details"* [175]; *"A model is a purposely abstracted, clear, precise and unambiguous conception"* [57]; *"A model is a purposeful abstraction that allows one to reduce complexity by focusing on certain aspects"* [163]. Thus, a model is an abstraction of reality, which can be a system, phenomenon, object system or a system under study (SUS) [94]. The model represents the main properties of the system under study. By focusing on the main properties and discarding ones, models simplify and reduce the complexity of the system under study. A map is an example of a model; it represents a certain geographical area. For same area, there can be different maps, each of which is represented in a way that supports a certain

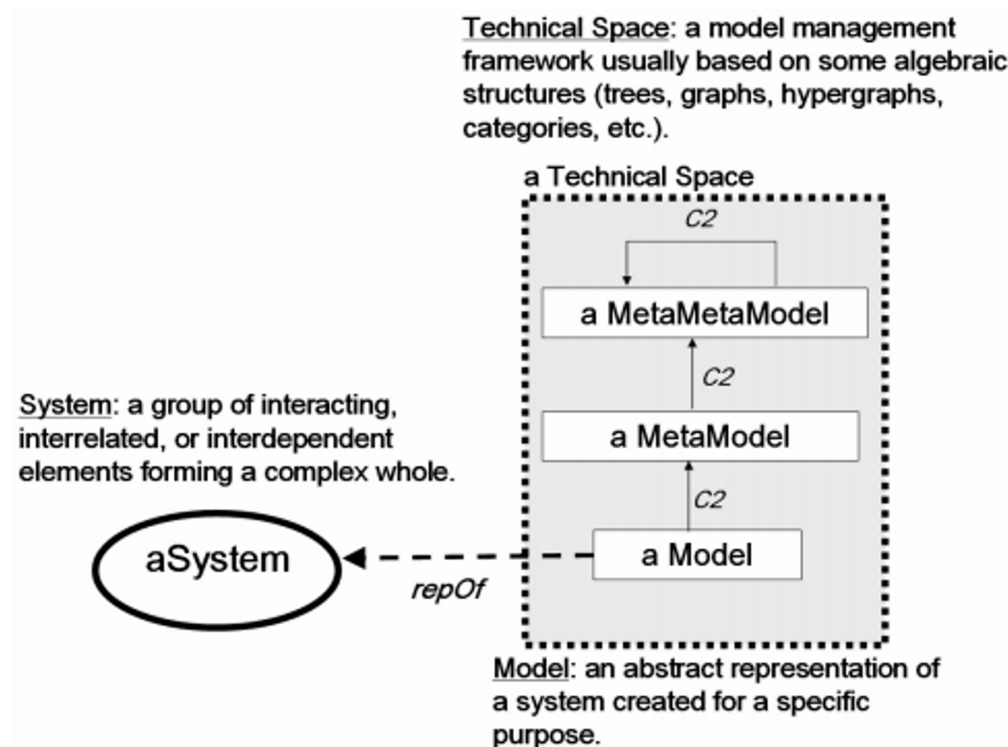


Figure 2.21 Relationships between Model, Metamodel, Modeling and Metamodeling

goal [23], e.g., to describe political boundaries, the geological structure, etc. Models are used widely in the field of science. Some examples of model usage in science are mathematical models, statistical models and biological models. In software development, models are used generally; for example, to describe software architecture or the behaviour of a system. In the context of software development, one of the aims in using MDE is to increase the automation of the software development process [68], [87]. One of the definitions of a model, in the context of MDE, is: *"a description of (part of) a system written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer"* [85]. A model is expressed in a modelling language. A modelling language is defined in terms of metamodels. A metamodel describes the abstract syntax of a language.

2.3.3.2 Metamodel

The relationship between a model and a metamodel is called instantiation. A model *conforms* to a metamodel [23]. A metamodel defines the modelling language in a high level of abstraction (e.g. above the code level). A model conforms to a metamodel, which specifies its elements based on the concepts that are defined in the metamodel, and uses these elements

according to the rules or constraints of the metamodel. A valid model is one that conforms to a metamodel and satisfies the constraints captured by its metamodel [131]. For example, classes, data types and packages could represent the key construct (abstract syntax) of the modelling language. Figure 2.22 provides an example of the metamodel abstract syntax, which defines a model. This model consists of a collection of types, each of which has a name and attributes which also have a type. Concrete syntax is the notation that is used to enable the description and presentation of the model [44]. Concrete syntax can be textual or graphical [44]. A common example of a general purpose modelling language is the Unified Modelling Language (UML) [uml]. Returning to the abstract syntax shown in Figure 2.22, a possible concrete syntax for this abstract syntax is listed in Listing X. Semantics are used to describe the meaning of the concepts in the language.

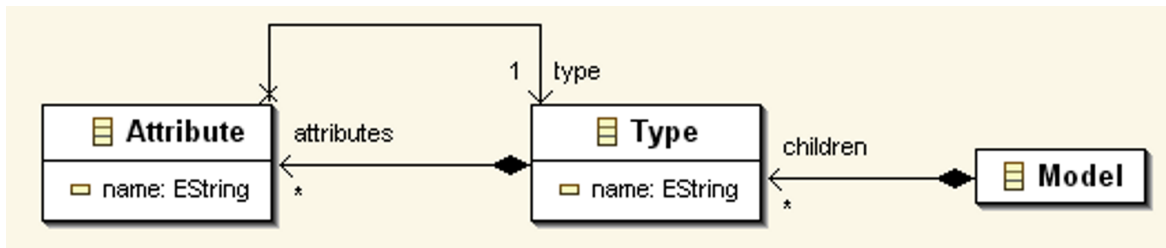


Figure 2.22 Metamodel of abstract syntax for a simple language [131].

```

1 | Type Mail {
2 |   From : User To : User
3 | }
4 | Type User {
5 |   Name : String
6 | }
7 | Type String;

```

Several metamodeling frameworks have been proposed, which can be used to define metamodels and models that conform to their metamodels, such as the OMG Meta Object Facility (MOF) [15] and Eclipse Modelling Framework (EMF) [13].

2.3.4 Model management

The aim of MDE is to raise the level of abstraction and increase the automation in the software development process. A typical MDE process involves various inter-related models that capture different level of abstractions and may use different modelling languages. Model management is the discipline of manipulating models (e.g., via transformation, comparison,

merging) using automated tools. Model management tasks, such as model transformation, validation, are discussed in this section.

2.3.4.1 Model Transformation

Transformation is a fundamental operation in MDE. A transformation transforms a *source* model into one or more *target* models, based on the transformation rules. The rules describe how concepts in the source model relate to those in the target models; these relations are expressed in terms of the source and target metamodels [85], [175]. Transformations can be used to transform models from higher levels of abstraction to lower ones. They can also be used to transform models at the same level of abstraction. Another transformation can be performed from a low level to a higher level (e.g. in reverse engineering). Transformations can also generate code. There are different approaches to transformations such as **Model-to-Model (M2M)**, **Model-to-Text (M2T)** and **Text-to-Model (T2M)**. In M2M, the target model is derived from the source model. The transformation in M2M is usually defined by a set of rules, each of which specifies how the source element(s) is transformed into the equivalent target element(s) [94]. Many M2M transformation languages have been proposed, such as the Epsilon Transformation Language (ETL) [94], Query/View/Transformation (QVT family) [20] and ATL (ATLAS transformation language) [98]. M2M languages can be classified as imperative, declarative or hybrid [94]. In the imperative transformation, the scheduling of the rule execution is defined by users explicitly, whereas in declarative languages, an execution engine decides which rule is to be executed. Hybrid languages are a mix of implicit and explicit rule scheduling. ETL syntax is explained later in section ?? In an M2T transformation, the target is text, e.g. code or non code artefacts such as documentation. Common approaches to M2T transformation are template-based approaches [163]. The Epsilon Generation Language (EGL) is an example of M2T transformation language, as explained later in section 2.3.5.7. The third type, T2M transformation, transforms text into generate structured artefacts that are used to form models. The different transformation approaches are presented in [163].

2.3.4.2 Model Comparison

Model comparison is an operation executed on two models to identify any matching elements between them [33][89]. Model comparison is an important task in MDE; it supports model development processes (e.g. for model versioning) [43], model composition and model transformation testing [93]. For two model elements, we need a *calculation* or *algorithm* in order to be able to compare or match two models [33]. The outcome of this calculation

may identify any differences; e.g., elements for which matching elements do not exist in the opposite [93] model.

Several model comparisons apply different calculations and methods to match two models. [90] Categorized several approaches to model matching. In *Identity-based matching*, elements are matched based on their identities, assuming that each element has a unique, static and persistent identity. This approach finds any differences between two versions of the same model, and so does not work for independent models. The second approach is *signature-based matching*. This is similar to identity-based matching, in that it matches the identities of elements. However, the identity in this matching is not static but calculated dynamically. This approach applies to comparing independent models. A third approach is *similarity-based matching*. This approach seeks to calculate the aggregate similarity of the features of elements. It also specifies a weight for each feature. Finally, in *custom language-specific matching*, users can specify a comparison algorithm using a specific language for comparison. While this approach can provide more accurate matching results, it requires significant effort to develop the comparison algorithm. ECL is an example of this approach. Thus we use Epsilon Comparison Language for Establishing Correspondences, which is explained after.

2.3.5 Model Driven Engineering Tools

A number of model management frameworks exist, including AMMA [163], EMF [163], and Epsilon[12]. In the following sections (2.3.4.1 and 2.3.4.2), we discuss EMF and Epsilon, as these technologies are employed in this thesis.

2.3.5.1 EMF

The Eclipse Modelling Framework (EMF) is built on the Eclipse platform, which is in itself a tool-integration platform for software development. It provides tools such as a graphical editor to define the metamodels and tools to generate model editors from the metamodels. The EMF framework includes three main components: *Ecore*, which is used to specify metamodels, and *EMF.Edit*, which is used for building editors for new modelling languages and *EMF.Codegen*, which is a code generation for facilitating the building of an editor for the EMF model. The EMF model editor consists of; a navigation view to specify the models' elements and property, and, is used in the model examples in Chapter 3. Several tools have been built for use with the EMF, such as the Graphical Modeling Framework (GMF) [14] and Epsilon [12], as described in section 2.3.5.2.

2.3.5.2 Epsilon

We use Epsilon in this thesis to implement model comparison, transformation, management and creation. Epsilon [91] stands for *Extensible Platform of Internet Language for model management*. It is a family of task-specific languages which can be used to manage EMF models. Epsilon can be used to perform different MDE tasks, such as model-to-model transformation, code generation, model merging, comparison and validation. For each of these MDE tasks, Epsilon provides a task-specific language. *EOL* is the core language and other model management languages are built atop it. In the following sections, we explain each of the Epsilon task-specific languages that are used later in this thesis.

2.3.5.3 Epsilon Object Language (EOL)

EOL [eps], [91] is a model management language used to manage models from various technologies such as EMF and XML. It allows the creation, modification and querying of models. EOL contains different features. It supports model querying operations such as `select()`, `collect()`, etc. It has the capability of model modification, such as modifying element's properties, *adding*, *removing* and *deleting* model elements.

Furthermore, EOL supports access to multiple models that are possibly conforming to different meta-models. EOL users can reuse their defined operations, not only in other EOL programs, but also in other Epsilon languages. Finally, it supports common programming constructs such as *while and for loops* and *if.else statements*, and also provides support for user interaction.

2.3.5.4 Epsilon Comparison Language (ECL)

ECL is a rule-based, metamodel-independent language for comparing the arbitrary models of arbitrary meta-models [eps]. As with other Epsilon languages, ECL is built on EOL language. With ECL, users can define comparison rules for the elements in two models. The result of the ECL comparison is a trace that consists of a number of matches. Each match has a reference to the compared elements (left and right) and a Boolean to indicate the matching result. It also has a reference to the match rule that led to the decision.

As ECL is used in this thesis for comparing and matching elements of cloud SLA models, we provide further details of it. The concrete syntax of an ECL rule is displayed in Listing 2.3. ECL rules are specified in modules *ECLModule*. In this module, a user can import other epsilon modules, a set of match *rules* and *pre* and *post* blocks [89]. Statements defined in the *pre* part are executed before executing the match rules, while statements in the *post* part are executed after executing the match rules that are defined in the module. Both the *pre* and *post*

parts are optional. The name of the rule follows the keyword *rule*. This rule declares two parameters *leftParameter* and *right-Parameter*, e.g. two elements from two models. The left parameter is defined after the *match* keyword and the right parameter is defined after the *with* keyword. The rule can optionally extend other rules. These are separated by a comma and defined after the *extends* keyword. Rules that are specified in the extends part are executed before the comparison statement and, if the rules in this part are not satisfied, the compare part is not executed. This extends is an optional part. The rule can optionally specify a *guard* expression or block of statements. The guard statement narrows the set of matched elements; if the guard fails for specific parameters, the compare part is not executed.

After the *compare* keyword, users specify comparison expressions or block of statements. Expressions after *guard* and *compare* follow a column. The block of statements is defined between curly brackets () after the *guard* and *compare* keyword. In this comparison part, a user can specify their matching statements to indicate whether the left and right parameters are matched or not. This compare part returns a Boolean value to indicate whether the two parameters (left and right) are matched or not. If this Boolean value indicates that a match has been found (true value), an optional *do* is executed and if the comparison result is true, a user can specify any additional actions. If the left and right parameters are matched, the *do* part allows a user to define any other actions. An **ECL** rule is executed automatically for the non-lazy and non-abstract rules. The operation *match()* invokes the lazy rule.

```

1 || (lazy)?(abstract)?
2 || (greedy)?rule <name>match <leftParameterName>:<leftParameterType> with
   <rightParameterName>:<rightParameterType>(extends (<ruleName>)*<ruleName>)?
   (guard (:expression)|(statementBlock))? compare (:expression)|(statementBlock)(do
   statementBlock)?

```

2.3.5.5 Epsilon Merging Language (EML)

The Epsilon Merging Language (EML) is a metamodel agnostic language for expressing model compositions. It includes a model comparison and model transformation language as subsets, and is built atop a generic model management language called the Epsilon Object Language (EOL) [6], which is inspired by OCL. An EML specification consists of a set of rules describing how model compositions should be carried out. Rules in EML are of three types: match rules, merge rules, and transform rules. Match rules can be further subdivided into comparison and conformance rules (examples to follow). Each match rule has a unique name and two metaclass names as parameters. The rule itself is composed of a *compare* part and a *conform* part. The rule is executed for all pairs of instances of the metaclasses that appear in the source models. The compare part of a match rule determines whether two

instances match, using a minimum set of (syntactic) criteria. The conform part applies only to instances that satisfy the compare part of a rule; the conformance rule part refines this match. If the conformance part of the rule fails, then an exception is raised. An example is shown in listing below:

```

1 | abstract rule ModelElements
2 | match l: Left!ModelElement
3 | with r: Right!ModelElement
4 | extends Elements {
5 |   compare {
6 |     return l.name = r.name
7 |     and l.namespace.matches(r.namespace);
8 |   }
9 | }
10 | rule Classes
11 | match l: Left!Class
12 | with r: Right!Class
13 | extends ModelElements {
14 |   conform { return l.isAbstract = r.isAbstract; }
15 | }

```

The rule on ModelElements is abstract and provides basic behaviour that is used by rules that *extend* it; EML supports rule reuse via inheritance. The behaviour of this abstract rule is to match model elements that have identical names ($l.name=r.name$) and matching namespaces. A similar match rule is used for classes. Classes match when they obey the rules declared in their parent and when the *conform* part of the rule holds, i.e., when classes are either both abstract or both not abstract.

• *EML Model Element Categorisation*

After the execution of all match rules in an EML specification, all model elements are categorised in four groups: those that match and conform; those that match but do not conform; those that do not match; and those to which no match rule has applied (the last category of element produces warnings). The results of this matching process are used in the merging process. In particular, elements that match and conform will be merged with their identified opposite. The specification of merging is captured in *merge rules*. Elements that do not match will be *transformed* into model elements compatible with the target meta-model. This is captured using *transformation rules*.

• *EML Merge Rules*

Merge rules in EML are used to specify the behaviour necessary to compose two instances of model elements that match and conform. Each merge rule consists of a unique name, two

metaclass-typed parameters, and a list of the model elements that the rule creates in the target model.

For all pairs of matching instances of the two parameters, the rule is executed and the declared empty model element(s) are created in the target model.. The contents of the newly created model element are defined by the body of the merge rule. Two examples of merge rules are shown below:

```

1 rule ModelElements {           \** rule Classes {
2 merge l: Left!ModelElement     \** merge l: Left!Class
3 with r: Right!ModelElement     \** with r: Right!Class
4 into m: Merged!ModelElement    \** into m: Merged!Class
5 extends ModelElements
6 m.name := l.name;              \** m.feature := l.feature.
7 m.namespace:=l.namespace.equivalent()\** includeAll(r.feature). }
8 }                               equivalent();

```

This listing presents two merge rules, one for merging ModelElements and a second for merging UML classes (“Classes”). The first rule applies to all Model Elements and produces a new, merged ModelElement whose name is that of the left original model element, and whose namespace is that of the left original model element. In the second rule, the two parameters, *l:Left!Class* and *r:Right!Class*, are declared; the merge rule is also declared to produce an instance of the Merged!Class metaclass. The “Classes” rule creates a new instance of the Class metaclass, carries out all operations declared in its parent (ModelElements), and sets the feature list of the new class to be the union of all features from the left and right arguments.

There is a slight twist to the merging rule that takes the union of all features from the left and right model elements: the use of the *equivalent()* operation. This operator returns *the equivalent of the model element to which it is applied* in the target model. The equivalent of an element is the result of a merge rule if the element has a matching element in the opposite model; otherwise it is the result of a transform rule. This operator is necessary because the target and source metamodels may differ, and it ensures that all source elements are expressed in the target metamodel.

A key aspect of merging models in EML is that many merge rules can in fact be inferred from the *structure* of the metamodel itself: for example, when merging two classes, a basic merge rule can automatically be inferred that merges the contents of the classes (i.e., behavioural features and attributes). Such inferred rule sets we call *strategies*; further details on them (and on EML) can be found in [24].

2.3.5.6 Epsilon Transformation Language (ETL)

ETL is a hybrid model-to-model transformation language, built on EOL; this allows the transformation language to integrate with other Epsilon languages and share and reuse operations [94]. ETL transforms input models into arbitrary output models. As in ECL, ETL rules are organized in modules *ETL*. ETL modules include also *pre* and *post* blocks, as in the ECL module. Each rule defines a name, a *sourceParameter* and one or more *targetParameter*. The *sourceParameter* is used to specify an element of the source model that is involved in the transformation, while the *targetParameter* specifies the target element(s) of the target model. Similar to ECL, an optional *extends* specifies the rules that it extends, separated by a comma. After the optional *guard* keyword, a user can define simple EOL expressions following column (:) or as a block of statements between curly brackets (fg). The ETL statements are executed if the optional parts extend and guard rules and expressions are satisfied. Finally, a sequence of EOL statements form the body of the transformation rule that is executed, which means a target element(s) is/are created. These statements populate the contents of the created element(s). The ETL is used in this thesis to transform the outcome of the matching process to a correspondence model (Chapter 3).

2.3.5.7 Epsilon Generation Language (EGL)

EGL is a model-to-text transformation language. It is a template-based language for code generation and documentation [148]. It includes standard features, such as decoupling content from the destination, mixing generated and hand-written code, and coordinating templates. EGL defines sections which are: *static* sections, *dynamic* sections and *dynamic output* sections. The contents of a *static* section appear verbatim in the generated text. The executable code (expressed in EOL) is defined in the *dynamic* section and is enclosed between [% %]. The results of evaluating the expressions in dynamic output, which has the syntax [%=expr%], are included in the generated text. EGL is used in this thesis to generate XML documents for the outcome models of the matching process (as described in Chapter 3).

2.4 Trusted third parties in the cloud computing

The origins of the trusted third party mechanism are found in cryptography [160], where a TTP represents an entity that allows secure (trustful) interaction between two distinct parties. It is assumed that both parties consider this third entity a trusted one. A widely used TTP implementation has been that of certificate authorities for web transactions.

Outside of strict cryptographic bounds a TTP can be perceived as a public notary service. According to ISO [138] a TTP has to implement security policies that prevent unauthorized usage of resources by any involved party. In [104] NIST defines cloud actors that typically interact within a cloud eco-system. According to the NIST architecture, service customers can use broker services instead of direct interaction with providers to manage cloud products. According to NIST definition a cloud broker represents an entity that handles how a cloud service performs and how it is dispatched to one or more users. In addition a cloud broker can also assist the role of a negotiator between service providers and customers. Cloud brokering services are mapped into three general areas: intermediation, aggregation and arbitrage. A cloud broker entity can also be realized via a TTP to enhance security and trust in cloud transactions. By definition a TTP acts in a neutral manner as a mutually trusted entity. An intermediary can be neutral and operate based on predefined rules with the goal to balance trust and confidence among transacting parties. A public notary in the cloud can be characterized as a neutral intermediary that enforces a set of security policies. Thus, our TTP notion extends the traditional TTP definition. The TTP entity operates in the cloud computing by monitoring in real time the Service Level Agreements.

From a legal point of view, monitoring appears as a prerequisite for SLA contract enforcement [143]. It is a way to build trust between the CSPs and CSCs. Once the SLA agreement has been established, it is mandatory to monitor the SLA terms to determine whether they are achieved or violated. SLA monitoring is essential for both CSP to avoid penalties if the SLA terms are violated and for CSC to be aware of the status and quality of their running services. An initial assumption of using a TTP is that in general a TTP represents a neutral entity. The TTP does not operate in favor of any of the transacting parties. Accordingly all service description parameters and service level guarantees that are provided in the TTP SLA apply unanimously to any party [161]. The trusted third party provides an independent module that can monitor (and log) all communication between clients and service providers. By default, a TTP entity implements necessary monitoring mechanisms that enable robust check SLA violations and should be able to apply an advanced penalty model that guarantees the performance and the reliability of the cloud [110]. SLA contract will be checked at real time in the monitoring phase to detect any deviations or violations from predefined parameters in the commitment. Mostly a question that frequently arises: where this TTP will be placed ? In our context, outside running on a physically server (offered as a service in the cloud environment) that is separated from the CSPs and CSCs side (as shown in 2.23).

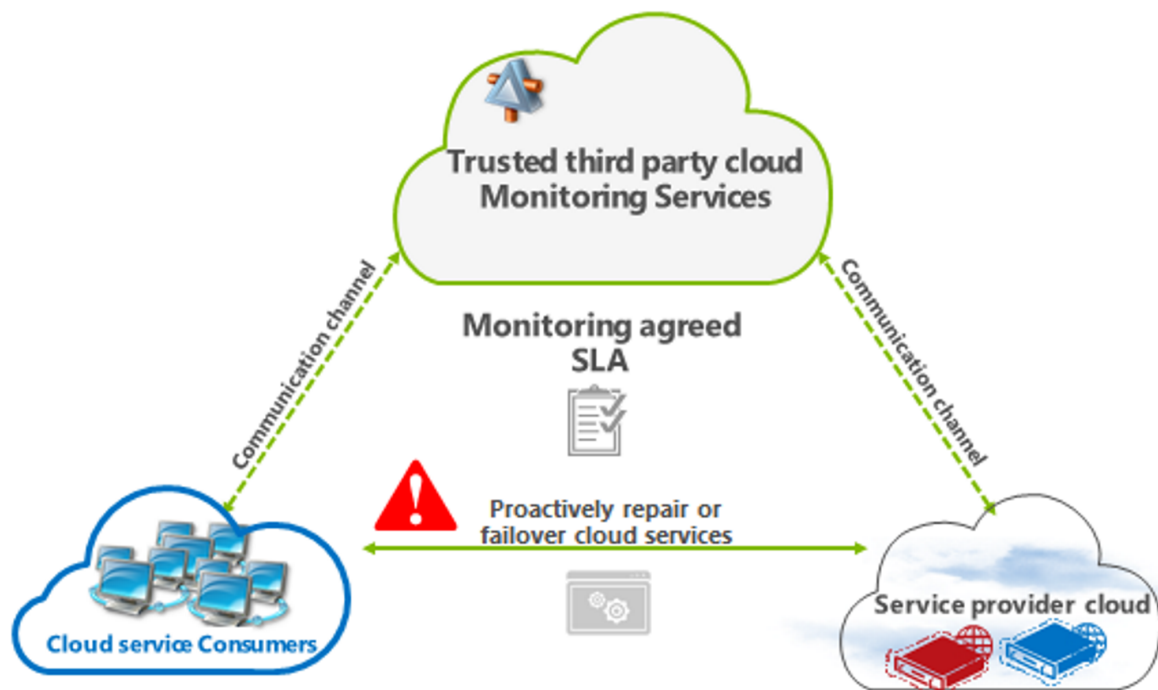


Figure 2.23 Trusted Third party cloud monitoring services

2.5 Autonomic Computing

Cloud computing is an instance of complex systems, in which its characteristics, such as the dynamism, large-scale and heterogeneity, emphasize the complexity of the management process. Moreover, the automation of the management is one of the essential characteristics of the cloud [19]. A prominent approach to cope with this complexity is Autonomic Computing (AC), which aims at equipping computer systems with capabilities to autonomously adapt their behavior and structure according to dynamic operating conditions. Autonomic computing is inspired by the functions of the human nervous system and is aimed at designing and building systems that are self-managing.

Autonomic systems are being adopted for self-management of complex large scale distributed system that have become manually unmanageable. Autonomic systems require high-level guidance from humans, but autonomically decide which steps need to be done to keep the system stable. Such systems constantly adapt themselves to changing environmental conditions. Similar to biological systems, e.g., human body, autonomic systems maintain their state and adjust operations considering changing components, workload, external conditions, hardware, and software failures.

In order to effectively achieve self-management, a system needs the following properties:

- *Self-Configuration*: Installation and configuration of complex systems is an expensive and a fallible process. The self-configuration is executed according to high-level policies. For example, if a new node is added to a cloud infrastructure, an autonomic cloud must install and configure the necessary software stack to integrate this node in the cloud and add agents into this node to collect monitoring information;
- *Self-Optimization*: Provides the capacity of the system in tuning and adjusting policies and configurations to maximize system's performance and resources;
- *Self-Healing*: Assures the effective and automatic recovery when failures are detected;
- *Self-Protecting*: This characteristic enables the system to defend itself from malicious and accidental attacks. For instance, when a user tries to delete an important file of the operating system, the system forbids the operation;
- *Self-Awareness*: Computing systems are composed of a collection of "smaller" systems, such as single computers or processors, depending on the level of abstraction. Therefore, the system has to know itself, i.e. it requires the detailed knowledge of its components and of itself as a whole. For example, a cloud should retrieve data of single nodes but also process the data and extract information, such as the cloud usage and the total number of users;
- *Context-Awareness*: An autonomic system should collect information about important aspects of the environment and the interaction with other systems in order to optimize and react to changes;
- *Openness*: Autonomic systems can not exist in hermetic environments. While independent in its self-management, it must function in heterogeneous environments and implement open standards;
- *Anticipatory*: The system must manage itself pro-actively, anticipating, when possible, the needs and behaviours of the system itself and of the context.

Autonomic computing has been served as a promising concept for the infrastructure management in various areas, e.g., services, Grids, and SLA management[113]. Autonomic control loop is known as MAPEK [19], where (M) stands for monitoring of the managed elements, (A) for their analyzation, (P) for planning actions, and (E) for their execution. MAPE-K loop stores knowledge (K) required for decision-making in a knowledge base (KB).

2.5.1 The Autonomic Computing paradigm and maturity levels

Manual management of QoS requirements is a complex task, especially when the administrator has to consider evolving application-level QoS requirements and dynamic resources deployment. The Autonomic Computing [165] is a key paradigm which takes into account this dynamicity in an autonomous way.

2.5.1.1 The MAPE-K cycle of the AC paradigm

The Autonomic Computing paradigm is based on four main components identifying the so-called MAPE-K loop to guarantee the above properties of the system. Typically, control loops are implemented following MAPE-K (Monitoring, Analysis, Planning, Execution, Knowledge) steps[165]. Figure 2.24 shows these components, which are described below:

- The *M-onitoring* of the managed resources and its current performances associated with its current configuration or QoS satisfaction based on SLA, with symptoms detection based on defined rules;
- The *A-nalysis* of the generated symptoms with the aim to identify the possible causes of them, based on stored information in knowledge base, if changes are required, a request for change is sent to the plan function;
- The *P-lanning* of the actions needed to be set up with the aim to match the targeted goals; it creates or select procedures/plans to enact on the managed entity;
- The *E-xecution* of the elaborated plans to change the behavior of the monitored entity through the effectors.

Within the IBM model, a K-knowledge base allows storing data (metrics, rules, symptoms, policies, topology information, etc.) required for the implementation of the previous steps. The control loop with its components and the Knowledge database makes the autonomic manager to be self-manageable. The managed resources can be software or hardware resources that are given autonomic behavior in accordance with an autonomic manager. Thus, the managed resources can be operating systems, wired or wireless network, CPU, database, servers, routers, application modules, Web services or virtual machines, etc.

2.5.1.2 The maturity levels of the AC paradigm

Implementing an autonomic system able to integrate the MAPE-K loop is also a complex task; it requires passing through five Maturity Levels (MLs) [165] such as figure 2.25:

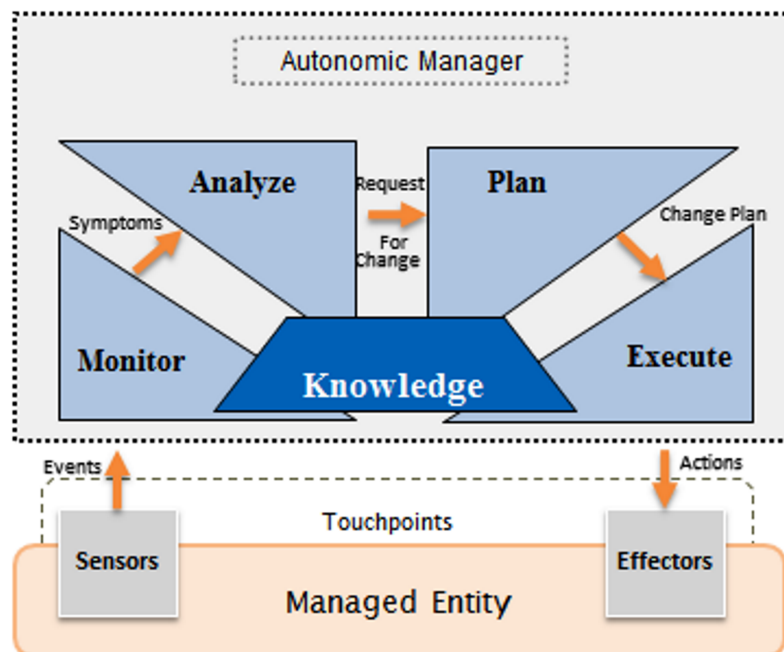


Figure 2.24 The MAPE-K cycle of the AC paradigm

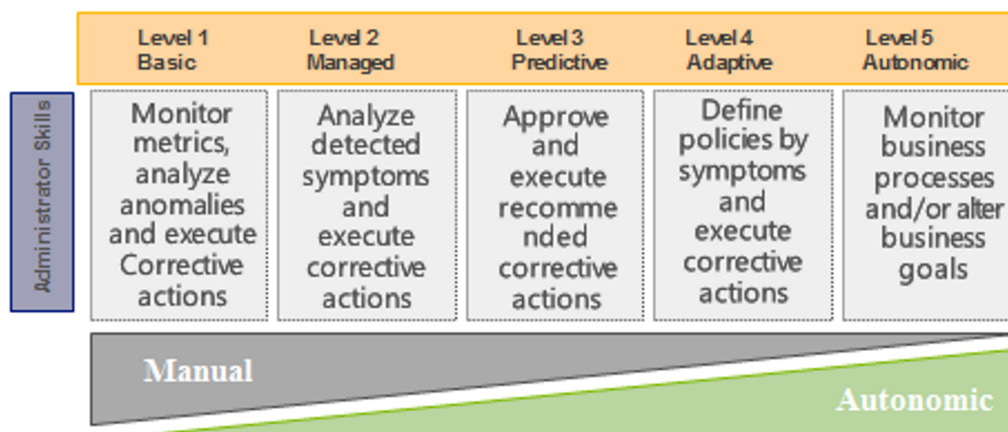


Figure 2.25 Maturity levels towards Autonomic System

- **Level 1 – Basic ML:** the management and setting of the system's elements are done independently by the administrator. Human skills are then required to monitor the system, to analyze the observed metrics and eventually to execute actions depending on the anomalies that have been detected;
- **Level 2 – Managed ML:** monitoring tools and technologies can be used to collect metrics from the system to detect anomalies, helping to reduce the time it takes to collect and synthesize information. Human skills are required to analyze the detected anomalies and execute corrective actions;

- **Level 3 – Predictive ML:** diagnostic and analysis capabilities are introduced in the system to analyze the situations and provide possible corrective actions. Here the final decision as well as the activation of the actions depend on the administrator;
- **Level 4 – Adaptive ML:** the administrator does not have to approve the corrective actions and to activate them; it has just to define policies based on the correlation between symptoms and mechanisms, and let the adaptive environment automatically decide of the adequate actions based on the available information and the knowledge of what is happening in the environment;
- **Level 5 – Autonomic ML:** at the autonomic level, business policies and objectives govern the autonomic manager with the consideration of applications requirements. The administrator interacts with the autonomic manager to monitor business processes, alter the objectives, or both. At the end of this level, the system becomes autonomic.

2.6 Comparative Study of Simulators for Cloud Computing

Recent years have seen a significant increase in cloud computing simulation tools. They are essential to evaluate cloud computing services and applications. Indeed, they can help to test, evaluate, and replicate the assumptions cost-effectively before a cloud platform is deployed in a real environment. It also provides an option to change and control the bottlenecks of the performance prior to implementing on an actual cloud platform. The simulation tools for cloud computing are also important to help users choose which cloud computing service is suitable for their requirements as deploying and operating a cloud platform in a real environment are often expensive, time consuming, or even impractical due to multiple test runs in different conditions and requirements.

The wide variety of cloud computing services and applications makes it difficult for users to compare and find the right service that meet their needs. Hence, cloud computing simulation tools are important for users to evaluate the services offered by different cloud service providers. On the other hand, researchers are often facing difficulty to analyze the performance of new scheduling and provisioning on actual cloud computing platform [139]. Also, researchers need a way to understand and visualize the micro and macro behavior of the systems they are developing. When building a cloud platform is impractical and costly, simulation tools form an appropriate solution for the evaluation of cloud technologies.

The performance, efficiency, and reliability of new algorithms for a cloud platform of a large heterogeneous infrastructure can be evaluated more easily by means of suitable

simulation tools. Indeed, they can offer significant benefits, as they allow cloud developers (i) to test performance of their provisioning and service delivery policies in a repeatable and controllable environment free of cost, and (ii) to tune the performance bottlenecks before real-world deployment on commercial clouds.

The validity and the accuracy of a simulator are greatly reliant on how much detail each component of the simulator can mimic the behavior of the physical platform. Representing the dynamic behavior of a system by another system is the goal of the simulation process. A simulation provides more flexibility than the real system, allows parameters to be set freely, provides better insight into the internal workings, and allow evaluating the hypothesis prior to real deployment in an environment where one can reproduce tests by the replay and repetition of scenarios. It also fundamentally avoids the need to build physical prototypes or experiments, which speeds up development.

In this section we present a review and discussion on the cloud computing simulation tools. Section surveys the various simulation platforms and tools of cloud computing. In section 2.6.3, we identify the characteristics of the tools highlighting their strengths and weaknesses via a comparative study of simulators in order to understand which simulation tools are suitable for which problem. Finally, this section notes the challenges and the future research directions.

2.6.1 Simulation tools for Cloud Computing

This section discusses significant works in the field of simulation tools for cloud computing systems. Moreover, we present an overview of the characteristics for each simulator.

2.6.1.1 CloudSim

Proposed in 2009, CloudSim is a simulation toolkit for enabling continuous modeling, simulation, and experimentation of cloud computing and application services [35]. CloudSim does not provide a ready-to-use environment for execution of a complete scenario with a specific input. Instead, users of CloudSim need to develop their own cloud scenario to evaluate, define the required output, and provide the input parameters. CloudSim scales well for cloud computing and has a low simulation overhead but is arguably the most sophisticated discrete-event simulation (DES). CloudSim enables fast evaluation of scheduling and resource allocation mechanisms within cloud data centers. It represents data center topology in the form of direct graph but no specific network topology is involved for modeling an internal data center network [38]. CloudSim allows researchers to study specific system problems without

worrying low-level details related to cloud-based infrastructures and services. CloudSim has the following advantages [38]:

- It helps in developing new resource distribution policies and scheduling algorithms rapidly.
- It provides modeling and simulation of large-scale infrastructure that involves data centers of multiple physical computing nodes.
- It creates and manages various, independent, and co-hosted virtualized services on a data center node by using a virtualization engine.
- It offers great flexibility due to the switch between space-shared and time-shared allocation of processing cores to virtualized services.

CloudSim is based on DES where the state of the simulation model can only change at discrete points in time, which are referred to as events. This means the model is time based and takes into account resources, constraints, and interactions between the events as time passes. Central to DES are a clock and an event list that tells what steps have to be executed. Therefore, DES in general is able to cut the simulation time significantly compared to packet-level simulation at the expense of simulation accuracy.

The high-level multilayer architecture of CloudSim software framework is illustrated in Figure 2.26.

CloudSim simulates cloud systems following a cloud provider perspective and uses so-called cloudlets to model abstract computations. The modeling and simulation of virtualized cloud-based data center environments including dedicated management interfaces for VMs, memory, storage, and bandwidth are supported through the CloudSim simulation layer. The simulation layer also manages fundamental issues, such as provisioning of hosts to VMs, managing application execution, and monitoring dynamic system state. The user code is situated at the topmost layer in the CloudSim. It exposes basic entities for hosts (number of machines, their specification, and so on), applications (number of tasks and their requirements), VMs, number of users and their application types, and broker scheduling policies [38].

CloudSim is very popular tool to simulate cloud computing environment. Many researchers developed other simulation tools based on CloudSim. For example, CDOSim [58], CloudAnalyst [178], NetworkCloudSim [63], and TechCloud [73] were all built or developed on top of the CloudSim. Werner et al. also proposed an organization theory model for resource management for green cloud computing based on CloudSim [177].

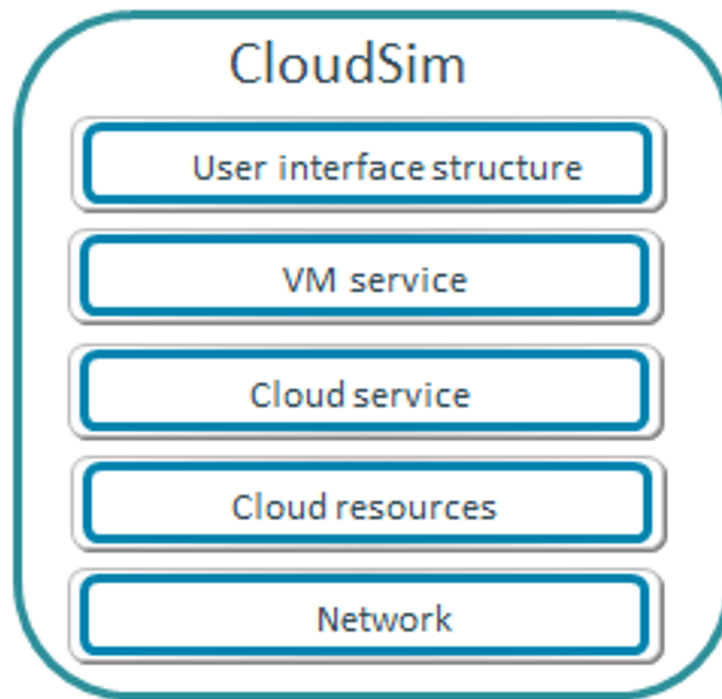


Figure 2.26 CloudSim architecture. (Adapted from [38])

2.6.1.2 CloudNetSim++

The CloudNetSim++ is developed as an extension to OMNeT++, provides a rich GUI to simplify analysis and debugging. Another salient feature of CloudNetSim++ is that addition of data center resources, such as racks, computing servers, routers, and switches is made easy through user interface.

2.6.1.3 Cloud Analyst

CloudAnalyst is developed based on the SimJava and CloudSim toolkit by extending its functionality with the introduction of concepts that model the Internet and Internet application behavior [178]. It is developed on Java platform to simulate large-scale cloud applications with the purpose to study and analyze the behavior of such applications under various deployment scenarios. CloudAnalyst allows to set up repeat simulations easily and displays simulation experiments sequentially with slight parameters variations in a quick and easy way. It also provides many user-friendly capabilities that make CloudAnalyst simulator more flexible and configurable to use. The graphical user interface (GUI) facilitates users to set up and design their experiments easily. The GUI is also convenient for users to combine technology with the easing of extension. Making a simulation implementation separated from a programming implementation is considered one of the key benefits of CloudAnalyst.

Therefore, by using CloudAnalyst users can focus on the simulation problems without being heavily involved in programming. CloudAnalyst allows users to model scenarios where SaaS data centers and users are in different geographic locations. By using CloudAnalyst, application developers or designers are also able to choose the optimum strategy for allocating the limited resources among data centers, strategies for selecting data centers to serve specific requests, and reducing costs related to such operations [178].

There are three algorithms that are implemented in CloudAnalyst including VM load balancing, throttled load balancer, and active monitoring load balancer. The design of CloudAnalyst architecture is illustrated in Figure 2.27. The main components of CloudAnalyst are as follows [178]:

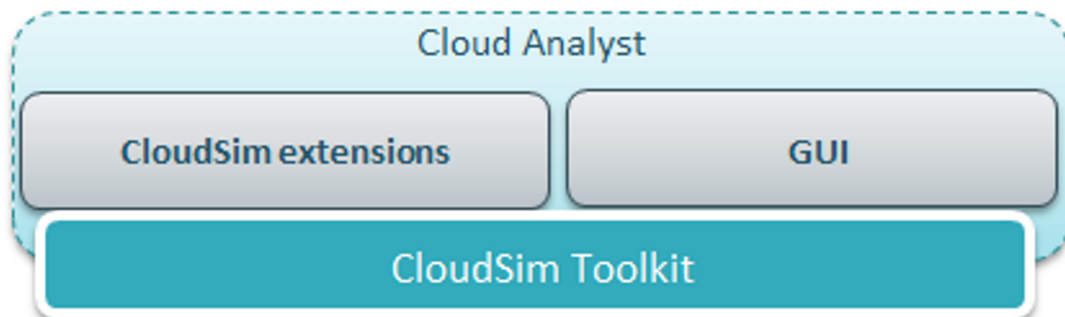


Figure 2.27 CloudAnalyst architecture. (Adapted from [178])

- *GUI* Package to manage GUI.
- *Simulation* to manage creation and execution of the simulation.
- *UserBase* for generating the traffic that represent users.
- *DataCenterController* to control all activities of users.
- *InternetCharacteristics* for defining the Internet characteristics that applied during the simulation.
- *VmLoadBalancer* for displaying the policy of load balance that are used by data centers when serving the requests of allocation.
- *CloudAppServiceBroker* for displaying the service brokers that control traffic routing between user bases and data centers.

2.6.1.4 CDOSim

CDOSim [58], cloud deployment option (CDO) simulator, is built based on CloudSim and extends it to combine with cloud migration framework CloudMIG [61]. CDOSim is able to simulate various scenarios, such as [58]

- *The response times*: The total rating for the response times will be calculated.
- *Service-level agreement (SLA) violations* Counts the number of each SLA violations.
- *Rating* Making ratings for each run and the scale ranges for rating from 1 to 5 (where 1 is the best performance).
- *Costs of a CDO* Presents the overall cost that is the total costs for VM instances and the used bandwidth.

CDOSim is developed with objective to overcome the limitations of other existing cloud simulator that related to CDOs. The CDOSim has the following characteristics [58]:

- It does not expose fine-grained internals of a cloud platform when following the cloud provider perspective; therefore, it focuses on client perspective.
- It keeps a balance between the lack of knowledge of cloud user's and control regarding an underlying cloud platform structure and its effect on a total performance of applications.
- Workload profiles from production monitoring data are used to rerun actual user behavior to simulate CDOs.

Based on these characteristics, CDOSim can be considered as the best simulator tool for evaluating the trade-off between costs and performance or for comparing runtime reconfiguration plans. The main features of CDOSim include the following [58]:

1. *IC derivation*: The derivation is implemented with one of the IC mechanisms.
2. The size of included types is derived to approximate the bandwidth that is used when there are distributed calls to other VM instances.
3. The transformation from the mapping model provided by CloudMIG Xpress to the extended CloudSim metamodel.
4. The actual simulation with CloudSim, the new simulation result is rated relative to the other runs.

2.6.1.5 EMUSIM

A key question raised by potential cloud customers before renting resources from any cloud service provider is how to know the services received will behave in a set of resources, and how much costs are involved when their resource pool has changed. EMUSIM (emulation and simulation) [37] is developed with the objective to answer these questions. It is designed for both emulation and simulation objectives in order to obtain better understanding of service's behavior on cloud platforms. Emulation in EMUSIM allows the simulator to automatically extract information from the application behavior and then uses this information to generate the corresponding simulation model [37].

EMUSIM is illustrated in Figure 2.28 . EMUSIM is built on top of two software systems: automated emulation framework (AEF) for emulation purpose and CloudSim for simulation purpose. As it employs both emulation and simulation, in theory, EMUSIM should improve the accuracy of the simulation. The main advantages of this integration are as follows: Emulation allows execution of the actual application in a small-scale environment that models the actual production infrastructure, and simulation allows assessment of how a system or an application performs in response to various conditions, for example, request arrival times and amount of work, in a controlled manner.

By using EMUSIM, cloud service provider, we are able to know how many resources are required to have a given response time considering a specific user arrival distribution and the effects of the changes in the request arrival rate. There are important requirements as an input when applying EMUSIM: description of the physical environment hosting the emulation, description of the emulated environment, application configuration, and simulation configuration.

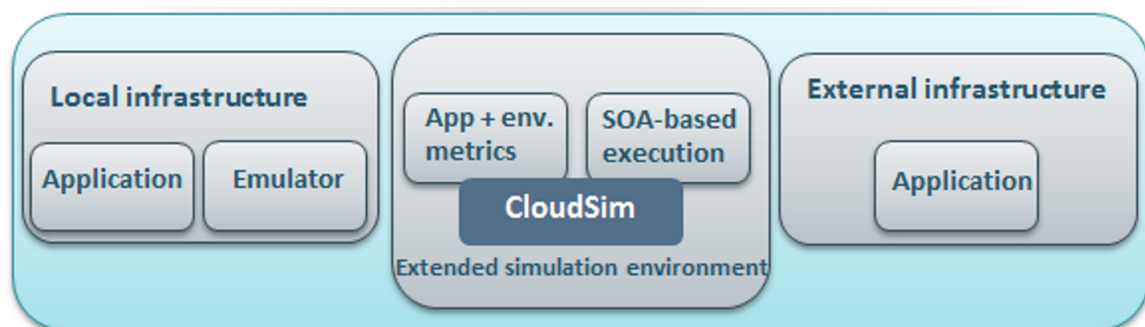


Figure 2.28 EMUSIM architecture. (Adapted from [38])

2.6.1.6 NetworkCloudSim

NetworkCloudSim [63] is an extension of CloudSim that supports a scalable network model of data center and generalized applications such as high-performance computing (HPC), e-commerce, social network, and web applications. NetworkCloudSim claims to have more realistic application models than any other cloud computing simulator. The main components of NetworkCloudSim are as follows: classes to model a network topology and classes for application modeling. NetworkCloudSim has three main entities: Switch, NetworkDatacenter, and NetworkDatacenterBroker. Switch represents a network entity that can be configured as a router or switch. NetworkCloudSim has two levels of scheduling, one at the host level (i.e., scheduling of tasks on VMs) and another at the VM level where real applications are executed.

NetworkCloudSim is capable of simulating cloud data center network and applications with communicating tasks such as MPI with a high degree of accuracy [63].

2.6.1.7 TeachCloud

TeachCloud is an extension of CloudSim, particularly designed for teaching purposes. TeachCloud is equipped with a simple GUI, which enables students and scholars to modify a cloud's configuration and perform simple experiments easily [73]. Students can use TeachCloud to experiment with different cloud components such as processing elements, data centers, networking, SLA constraints, web-based applications, SOA, virtualization, management and automation, and business process management (BPM). TeachCloud introduces many new enhancements on top of CloudSim basic platform, such as providing a GUI toolkit, adding the cloud workload generator to the CloudSim simulator, adding new modules related to SLA and BPM, and adding new cloud network models such as VL2, Bcube, Portland, and Dcell. TeachCloud enables students to reconfigure the cloud system any time to investigate the effects of changes in the overall system performance [73]. It is an open source for students and researchers.

2.6.1.8 DCSim

DCSim or data center simulator is an extensible simulation framework for simulating a multitenant, virtualized data center hosting an IaaS cloud to any clients [169], [139]. DCSim is specifically designed and developed for transactional and continuous workloads such as a web server (WS). Hence, by using DCSim, there is a further capability of modeling that replicates the VMs sharing incoming workload and the dependencies between VMs that are part of a multitiered application. DCSim is especially focused on a virtualized data center

providing IaaS to multiple tenants. This characteristic makes it differ from other simulators such as GreenCloud, MDCSim, and GDCSim, which are not equipped with any modeling of virtualization.

DCSim is classified as an event-driven simulator and is developed in Java to allow rapid development and evaluation of data center management techniques. In addition, the implementation of SLA can be easily measured and directed. The SLA is also available to manage components in the simulation. The main component of DCSim is the DataCenter, which contains hosts, VMs, and different management components and policies. A data center has a set of interconnected hosts that are managed by a set of management policies. Each host consists of a set of resource managers that manage local resource allocation, a CPU scheduler to decide how to run VMs, and a power model that models how much power is being consumed by the host at any point in time.

2.6.1.9 iCanCloud

iCanCloud is a simulation platform for large storage networks cloud infrastructures (thousands of nodes) that has special characteristics such as usability, flexibility, performance, and scalability [126], [127]. iCanCloud is initiated in 2010 as an open source and is developed based on SIMCAN [11], a modular simulation platform that can be configured for modeling a wide range of HPC architectures. The main objective of iCanCloud is to predict the trade-offs between cost and performance of a given set of applications executed in a specific hardware and then provide users useful information about such costs. iCanCloud can be used by a wide range of users, from basic active users to developers of large distributed applications. The main features of iCanCloud are as follows [126], [127]:

- It supports a wide variety of cloud computing architectures.
- It supports GUI for designing and running the experiments but not for modeling the software system.
- It implements a pay-as-you-go method as a policy of charging users.
- It supports parallel processing: run one experiment is run over various machines in parallel.
- Customizable VMs can be used to quickly simulate uncore/multicore systems.
- A flexible cloud hypervisor module provides an easy method for integrating and testing both new and existent cloud brokering policies.

- It provides a wide range of configurations for storage systems, which include models for local storage systems, remote storage systems, like NFS, and parallel storage systems, like parallel file systems and RAID systems.
- New components can be added to the repository of iCanCloud to increase the functionality of the simulation platform.

iCanCloud models the simulated cloud computing scenarios using a set of existent components to simulate the behavior of real components such as storages, memories, networks, and file systems. Those components are hierarchically organized within the repository of iCanCloud, which form the core simulation engine. New components can be added to its repository if necessary. Furthermore, iCanCloud also allows an easy substitution of components for a particular component. Those interchangeable components can differ in the level of detail (to make performance versus accuracy trade-offs), in the functional behavior of the component, or both. Figure 2.29 presents an overview of the architecture of iCanCloud.

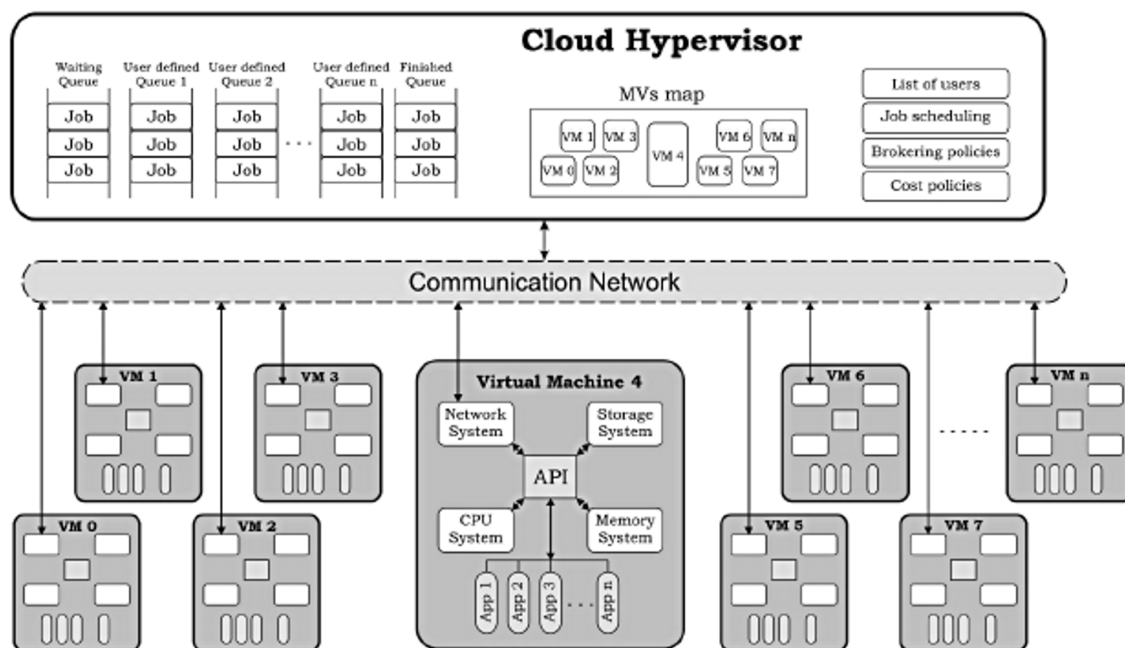


Figure 2.29 Basic Schema of the Nunez's iCanCloud Architecture [126]

2.6.1.10 GreenCloud

GreenCloud is a sophisticated packet-level simulator for energy-aware cloud computing data centers with a focus on cloud communications. It offers a detailed fine-grained modeling of

the energy consumed by the data center IT equipment, such as computing servers, network switches, and communication links. GreenCloud can be used to develop novel solutions in monitoring, resource allocation, workload scheduling, as well as optimization of communication protocols and network infrastructures. It can simulate existing data centers, guide capacity extension decisions, as well as help design future data center facilities [86].

GreenCloud simulator is an extension of the well-known simulator in computer networking, ns-2 [8]. Using ns-2 as the foundation, GreenCloud implements a full TCP/IP protocol reference model, which allows seamless integration of a wide variety of communication protocols, such as IP, TCP, and UDP with the simulation. Similar to ns-2, GreenCloud is also coded in C++ with OTcl. The main drawback of packet-level simulation is on the simulation duration, which is usually longer than DES, but with more accurate results. Typical GreenCloud simulation is about tens of minutes.

Figure 2.30 presents an overview of the architecture of GreenCloud. The framework supports the modelling of three-tier network architectures for data centers.

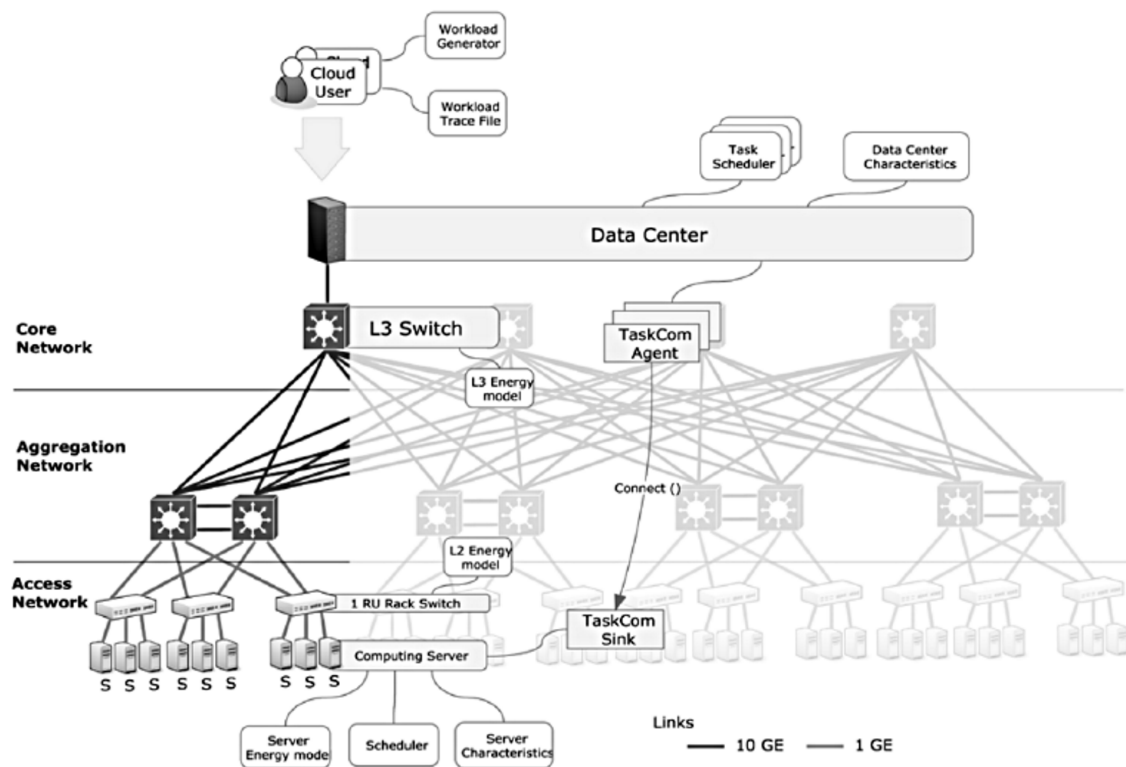


Figure 2.30 Architecture of the GreenCloud simulator. (Adapted from [86])

Cloud users can be modelled as workload generators based on workload trace files. The generated traffic enters into the modelled data centre. The centre consists of task schedulers, a three-tier network architecture, task completion agents, and a server pool. The data centre

entity can be modelled including data centre characteristics. The first network layer represents the core network of the data centre. Depending on the setup, the switches of the core network can be modelled as 10 gigabit or 100 gigabit ethernet switches. Each switching layer can be modelled with a certain energy model. The second network layer represents the aggregation network. The access network of the data centre is represented by the third network layer. This layer connects to the servers within the pool. Each computing server consists of a server energy model, a scheduler, server characteristics, and a task completion sink.

2.6.1.11 Open Cloud Testbed

In 2009 [67][9], Grossman et al. proposed Open Cloud Testbed (OCT) as a smaller-scale testbed with main objectives to benchmark different cloud computing systems and to study their interoperability. The main characteristics of OCT are as follows:

- Different cloud systems and services are installed, including Eucalyptus, Hadoop, CloudStore (KosmosFS), Sector/Sphere, and Thrift in order to make OCT easier to study interoperability and benchmarking, to develop network libraries, monitoring systems, and benchmark suites to support the development and experimental studies of cloud computing stacks.
- The architecture of OCT includes high-performance protocols, services, and infrastructure at all levels. It uses a high-performance 10 Gb/s network based on extremely fast transport protocols supported by dedicated light paths instead of using the Internet.

2.6.1.12 GroudSim

GroudSim [129] is a grid and cloud simulation toolkit for scientific applications based on a scalable simulation-independent discrete-event core. GroudSim is developed in Java environment and can simulate both cloud and grid computing. GroudSim provides a comprehensive set of features for complex simulation scenarios from simple job execution on leased computing resources to calculation of costs and background load on resources. Simulations can be parameterized and is easily extended by probability distribution packages for failures that normally occur in complex environments. GroudSim focuses on IaaS but is easily extendable to support additional models like cloud storage or platform as a service [129].

One of the main characteristics of GroudSim is the definition of failures, that is, failures can be produced in a defined interval for a specific registered resource. Like CloudSim, GroudSim is also a DES. GroudSim also has GroudJobs, which is functionally similar to cloudlets in CloudSim. However, unlike CloudSim, which employs process-based approach

that runs a separate thread for each entity, GroudSim requires one simulation thread only (instead of one thread per entity). The GroudSim framework is integrated as a back end in the ASKALON Grid computing environment, which performs both real and simulated execution of real-world applications using the same integrated development, monitoring, and analysis interface [129].

2.6.1.13 PerfCloud

PerfCloud is a cloud environment built on the top of a GRID system, which allows the user to instantiate virtual clusters (VCs) that become part of the starting GRID. It is a cloud computing simulation framework that provides performance prediction services in an e-science cloud. PerfCloud relies on a set of grid services that are able to create a VC and to predict the performance of a given target application on that VC taking into account the actual amount of computing and communication resources allocated to the VC, as well as the presence of the virtualization layer. The obtained predictions allow the user to evaluate on-the-fly whether the created VC is compatible with the user's performance expectations or not [116].

2.6.1.14 SimMapReduce

SimMapReduce [167] is a simulator designed to be a flexible toolkit and is convenient to inherit or be inherited by other packages. SimMapReduce makes an effort to model a vivid MapReduce environment, considering some special features such as data locality and dependence between map and reduce, and it provides essential entity services that can be predefined in XML format. Furthermore, by using this simulator, modelers can realize multilayer scheduling algorithms on user level, job level, or task level by easily extending preserved classes.

SimMapReduce is very useful for analyzing MapReduce programs in the simulation environment. Figure 2.31 gives the architectural diagram of SimMapReduce. It explains the mapping of user code with the simulation environment.

2.6.1.15 SimIC

In 2013, Stelios Sotiriadis proposed SimIC framework for simulating various inter-cloud activities [159]. It is a discrete event simulator built upon the SimJava package. Recently, many inter-cloud activities are evolved, thus exchanging many services between different cloud data centers and eventually increasing quality of services. None of the above simulators can implement inter-cloud activities without extending distribution package. It has the entities

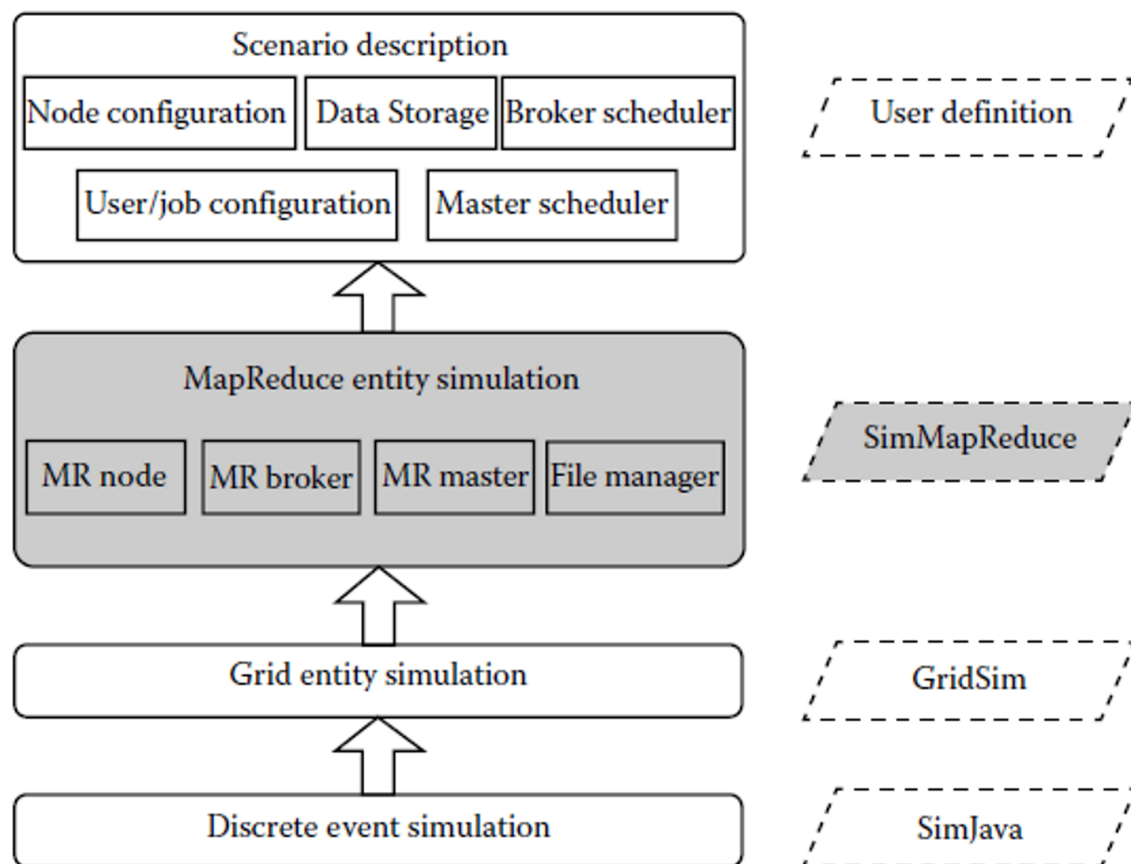


Figure 2.31 Four-layered architecture of SimMapReduce. (Adapted from [167])

of inter-cloud such as users, data center, host, hypervisor, meta-brokers, local brokers and virtual many important features such as energy model for reducing power consumption during message distribution, host-scheduling policy for time sharing, benchmarking with machines (VMs) [159]. SimIC uses ICMS algorithm for intercloud scheduling which depends upon most of the distributed parameters. It also gives as-per-as-you go facility. Using SimIC, Sotiriadis analyzes the performance of ICMS that depend on the varying amount of job submission in related with computational requirement [158]. The authors also tried to use the result for future inter-cloud benchmarking analysis. Many important features such as energy model for reducing power consumption during message distribution, hostscheduling policy for time sharing, benchmarking with CloudSim are still author expected in future work.

2.6.2 Comparative Study of Simulators : Characteristics and Limitations

In this section we present a comparative study of simulators highlighting their strengths and weaknesses. The Table I summarizes the comparative study on these tools in terms of characteristics and limitations, platform, programming language and availability.

Figure 2.32 Comparative Study of Simulators : Characteristics and Limitations

Simulator	Characteristics and Limitations	Programming Language	Platform	Available
CloudSim [2], [3]	• Discrete-event simulation • Comprehensive • Complex scheduling support • No TCP/IP support • Limited communication model (only transmission delay and bandwidth) • Limited GUI support (via CloudAnalyst) • No MPI and workflow applications support • No energy model	Java	GridSim	Open source
Cloud Analyst [5]	• Simulation of large-scale cloud applications • Easy to repeat simulations and to change different parameters • Easy to read output (graphical output) • GUI support • Flexible: easy to combine use with other technologies	Java	CloudSim, SimJava	Open source
CDOSim [4]	• Simulation of response times, SLA violations, rating, and costs of a CDO • No TCP/IP support	Java	CloudSim	Not available
EMUSIM [10]	• Implementation of both emulation and simulation • Assessment of system and application • Improve simulation's accuracy	Java	AEF, CloudSim	Open source
NetworkCloudSim [6]	• Discrete-event simulation • Scalability, flexibility, and reliability • Limited support for MPI and workflow applications	Java	CloudSim	Open source

Simulation tools are becoming more and more important in the evaluation of the cloud computing model. In this section we discussed significant works in the field of simulation tools for cloud computing systems. Moreover, we presented a detailed review of the characteristics for each simulator. Also, we presented a comparative study of simulators highlighting their strengths and weaknesses. It's true that research on the simulation of Cloud computing is not a new topic but it is still in infancy state. Research work is booming in this field. Often issues are too complex and too sophisticated to be investigated by a single simulation tool. In general, there is no single simulation tool that can solve all cloud computing problems. There is no silver bullet. Therefore, co-simulation or hybrid simulation is sometimes a necessity. Co-simulation (or cooperative simulation) is a simulation mechanism that allows individual components to be simulated by different simulation tools running simultaneously and exchanging information in a collaborative fashion [139]. More research should investigate issues related to cooperative simulation tools.

TeachCloud [7]	• GUI support • Cloud system is reconfigurable many times • Easy to add new modules • No TCP/IP support	Java	CloudSim	Open source
DCSim [11]	• Focused on a virtualized data center providing IaaS to multiple tenants • Event-driven simulation • No TCP/IP support • Developed for transactional, continuous workloads • Model of replicated VMs	Java	-	Open source
iCanCloud [12], [13]	• Design for simulate large storage networks • Implementation of detailed modeling of communication aspects of data center • GUI supports • Support to parallel processing • Easy to add new modules	OMNeT, MPI, and C++	SIMCAN	Open source
GreenCloud [23]	• Packet-level simulation • Suitable for modeling data center • Implementation of a full TCP/IP protocol reference model • Implementation of full communication model • Limited GUI support (via Nam) • Precise energy model (servers + network)	C++ and OTcl	ns-2	Open source

(To be continued in the next page)

Open Cloud Testbed (OCT) [17]	• Geographically distributed cloud testbed spanning four data centers • Connected with 10G and 100G network connections • Different cloud systems and services installed • Use to develop new cloud computing software and infrastructure • Focused: developing an OpenFlow-enabled version of Hadoop	-	-	Limited (registration is needed)
GroudSim [18]	• Discrete-event simulation • Simulation of clouds and grids • Provides a comprehensive set of features for complex simulation scenarios	Java	-	Open Source
PerfCloud [19]	• Virtual clusters (VCs) created on user request • Performance prediction of user applications on the newly generated VC in real time	MetaPL	GRID system	Not available
SimMapReduce [20]	• As a parallel programming framework, It's based on Grid Sim • Facilitate research on resource management and performance evaluation • Researchers are free to implement scheduling algorithms and resource allocation policies by inheriting the provided java classes without concerns of implementation details	MetaPL	GRID system	Not available
SimIC [21]	• Discrete-event simulation • Uses ICMS algorithm for intercloud scheduling which depends upon most of the distributed parameters	SimJava	Java	Not available

2.7 Chapter Summary

In this chapter, we described the SLA concepts and SLA specification languages. An SLA forms part of the legal contract between the service consumer and service provider. The main components of SLAs are the QoS parameters and metrics. Different services, service providers and consumer requirements require different QoS parameters. Also, a comparative study of the SLA definition languages, characteristics and limitations have been

presented. This chapter also described how cloud computing technology provides different types of services and how the dynamic nature of clouds requires continuous monitoring and establishing SLA to ensure the required/agreed service level. Different studies defining different QoS Metrics related to an SLA, their Categorization and models have been described. There is a lack of standard specifications for cloud SLAs and the QoS terminologies. We described the MDE principles and tools and the intersection between cloud computing and MDE. We then described the key concepts of MDE that involves: models, metamodels, model transformations and model management. Therefore, we presented cloud SLAs as standard artefacts to automate the process of cloud SLA establishment and management. After that, we discussed the concept of trusted third party in the cloud computing, finally, we demonstrated the importance of autonomic computing for clouds, explaining the essential role of the SLA in the context of autonomic systems and of cloud management. The next chapter (Chapter 3) explores, analysis and discusses the ways to approach thesis hypothesis and objectives, as proposed in Chapter 1.

Chapter 3

SLA Contract Meta-model: A Language for the Definition of SLAs for Clouds

In this chapter, our main objective is to propose an efficient method for automation of SLA contract establishment process. To this end, we investigate this issue and we propose our conception by using MDE (Model Driven Engineering) to express the SLA contract requirements, as a mean for establishing service level agreements between a cloud provider and cloud customer in the context of a particular service provision. The cloud SLA contract language we present is a meta-model for defining cloud SLAs specifically devised to the cloud computing domain, that support most of the characteristics of cloud. We evaluate the completeness and clarity of our SLA contract establishment for Conceptual Modeling with Bunge-Wand-Weber representational theory. The results indicated that, our meta-model is capable of achieving a satisfactory completeness and clarity, witch is very promising to accurately describe the structure and behavior of the real world. We also validate our meta-model to examines the ability of the proposed SLA establishment approach to generate the SLA document in less time. This chapter brings preliminary thoughts opening challenging perspectives. This work remains a key issue to properly solve the trade-off between the benefit of the supplier and the customer satisfaction. This chapter presents not only the meta-model for describing cloud SLAs, but also the process by which it was developed, including the different step of the meta-model that have been elaborated.

3.1 Introduction

Currently, Cloud computing (CC) is being promoted as the latest outsourcing solution of the new century. CC represents the evolution of IT towards an Internet-based computing model

explicitly designed to enable the transition from inside-out to outside in organizations. It consists to provide IT services as a service on-demand, accessible from anywhere, anytime and by anyone. Through this new computing paradigm, we can now eliminate the need for organizations to build and maintain expensive data centers, moreover we can access to computing resources hosted on the network and benefit from processing capabilities and storage as on-demand services[111]. It enables organizations to build up new systems quickly and easily, and has thus become a platform for innovation. With the CC, we can provide elastic resources that allow applications to scale as needed in response to market demands. It increases business continuity by providing inexpensive disaster-recovery options. Along with the development of computing clouds service, an important element that provides some degree of assurance to both users and providers of these cloud resources is the Service Level Agreements.

Actually, Service Level Agreements (SLAs) play a pivotal role in CC. The revolutionary technology of CC offers a scalable and flexible paradigm where infrastructure, platform, and software are offered to users in the form of services. The provisioning of these computing services by Cloud providers are regulated by SLAs [103]. The relationship between cloud providers and cloud consumers is normally governed by SLAs (as with other distributed computing models). The dynamic and complex nature of cloud computing arguably warrants complex SLA specification and management [112]. One of the challenges in CC is defining the SLA specifications "in such a way that has an appropriate level of granularity, namely the trade-offs between expressiveness and complicatedness, so that they can cover most of the consumer expectations and is relatively simple to be weighted, verified, evaluated, and enforced by the resource allocation mechanism on the cloud" [110]. Moreover, managing the cloud computing SLA requires an appropriate means to be consistent with the dynamic nature, autonomous and elasticity of cloud computing (i.e. selfservice, rapid and dynamic resource allocation and release). The automation of the process of establishment an SLA is addressed to tackle this challenge [113][115]. Furthermore, current cloud computing providers provide pre-defined cloud SLAs, which define the resource availability, while other QoS parameters, such as performance and elasticity, are not considered.

Service Level Agreements are one of the most common approaches for specifying some form of mutual understanding about business transactions between a Cloud provider (seller) and a Cloud consumer (buyer) in the software and telecommunications domain. Thus, an SLA represents functional and non-functional properties of services and serves as a way for controlling and managing these properties. Typically, an SLA is a bilateral binding statement signed between prospective signatories, over the agreed terms and conditions of the given service [162]. SLAs contain Quality of Service properties that must be maintained by a

provider, generally defined as a set of Service Level Objectives (SLOs). These properties need to be measurable and must be monitored during the provision of the service that has been agreed in the SLA. An SLA also sets out the remedial action and any penalties that could take effect if performance falls below the promised standard.

We believe that a differentiating element between CC environments will be the QoS and the SLA provided by the cloud. This raises the following questions: (i) How to describe the SLA terms between cloud provider and cloud customer, such as service levels, penalties in case of SLA violation, etc. (ii) How to provide guarantees on cloud QoS and provide better than best-effort behavior for clouds? We investigate this issue and propose our conception by using MDE (Model Driven Engineering) approach in CC domain. MDE provides some means of addressing this problematic by considering models as first class items[115], As discussed in the previous section 2.3. Another important concept in the MDE is the idea of having specific modeling language (DSML) that are used to describe specific behaviors to a domain in a brief and concise manner. According to this, every model must conform to a specific metamodel, similar to how a program conforms to the grammar of its programming language. In MDE, it's common to have a set of transformation engines and generators that produce various types of artifacts. Practitioners can take advantage of transformation engines to obtain source code, alternative model descriptions, deployment configurations, inputs for analysis tools, and so on.

In this context, our idea is to use Model-Driven Engineering to express the SLA contract requirements, as a means for establishing service level agreements between a cloud provider and cloud customer in the context of a particular service provision. SLAs contain Quality of Service properties that must be maintained by a cloud provider, generally defined as a set of Service Level Objectives (SLOs). The main focus of our work is to use MDE technology for the creation and automate establishing of the SLA contract. More specifically, in this work, we propose a method for an SLA establishment guided by QoS for composite XaaS. This method embraces the model-driven engineering (MDE) principles to facilitate the modeling aspect of the problem. In addition, it provides the automatic transformation of the various models (customer requirements, provider resources offers, and SLA). Furthermore, it offers automated processes that assist the cloud provider to provide there offers and identifying those resources that, on the one hand, satisfy the customer requirements and, on the other, increase his benefits.

To provide the modeling aspect, we propose three meta-models: The first meta-model, called CloudCustomer, allows consumers defining their requirements and specify its functional and non-functional requirements for a composite XaaS-based service; The CloudCustomer meta-model reuses the domain specific modeling language (DSML) for the functional

requirements perspective and the QoS requirements. The second meta-model, called Cloud-Provider, enables the provider to describe the QoS of selected services, their composition schema and the underlying IT in order to satisfy the customer requirements. Similar to the CloudCustomer, this meta-model also relied on the DSML; the extensions we propose cover the Cloud environment characteristics. Finally, the third meta-model, called Cloud SLA contract, is used to specify the finale service level agreement.

The rest of this chapter is organized as follows: in section 3.2 we presents some related works on SLA establishment and, secondly, states our motivations and outlines our approach. Next, in section 3.3 we provides the construction process and describes the cloud SLA meta-model used in our proof-of-concept tool support. In section 3.4 we describes the generation process for SLA contract establishment meta-model. Section 3.5 we provides a description about Bunge-Wand-Weber representational theory. We also evaluate the completeness and clarity of our SLA contract establishment for Conceptual Modeling, in section 3.5.2. The results indicated that, our meta-model is capable of achieving a satisfactory completeness and clarity, witch is very promising to accurately describe the structure and behavior of the real world.

3.2 Related works: Comparison between SLA contract meta-model and the Existing SLA Languages

To reveal our motivations and contributions, we find it useful to review some works on the SLA establishment first. Afterward, we evaluate the existing languages for the definitions of SLA based on their support for these requirements and examine the differences among these languages and our proposed cloud SLA contract language we present is a meta-model (consisting of abstract syntax and well-formedness constraints) for defining cloud SLAs, specifically for XaaS clouds.

• *Work on SLA establishment*

Comuzzi et al. [48] present an approach for the establishment and monitoring of SLAs in complex service-based systems. They asserted that a service provisioning infrastructure should allow the establishment of SLA contracts through a coordinated negotiation among service providers and customers. They use WS-Agreement as a specification language and propose a negotiation process of electronic SLA between the interested stakeholders. This approach did not consider the requirements definition phase nor the services and resources discovery phase, both of which are prerequisites for an SLA establishment and monitoring.

Skalkowski et al. [156] present the application of the enterprise service bus (ESB) architecture to monitor a distributed system. This architecture is used to monitor SLA requirements, resources and shared services. The authors identify some issues affecting the efficient SLA enforcement processes such as different technologies for the evaluation of the SLA documents, complex deployment processes and scalability issues. Their SLA enforcement strategy is based on a continuous monitoring of the system to detect violating situations. However, they did not address how to specify and define the SLA.

The BREIN project [181] defines a business framework prototype for electronic business collaborations. Some functionalities of this prototype framework include service selection, SLA negotiation, system monitoring and SLA evaluation. The SLAs are modeled with WS-Agreement.

In [56], the authors propose an ontology-driven approach for automatic establishment of service level agreements (ODACE SLA). This approach involves four modules: the correspondence generation module, the correspondence refinement module, the correspondence evaluation module and the contract generation module. This approach aims to unify the semantic QoS objectives for the development of SLA contracts. These objectives are based on semantic representation intentions of the customer and provider called, respectively, ClientOnt and ProviderOnt. Each intention is expressed freely in the area of competence of these two parties. The unification of these intentions process is based on linguistic, heuristic and semantic correspondence. From the customer point of view, this work does not specify both the functional and non-functional requirements, and it is limited to the non-functional requirement (noted as intention). From the provider's viewpoint, the proposed model, namely ProviderOnt, does not take into account the underlying resource allowing the concrete services execution. Even in the proposed model for the SLA specification, the SLAOnt describes only the concepts constituting an SLA document which are *Parties*, *Obligations* and *ServiceDefinition* neglecting the resources description.

The Web services community has performed significant level of research in SLAs languages. Several languages, such as SLAng [100], a domain specific language for IT services; WSOL, WSLA [109] and WS-Agreement [18] have been proposed for SLA specification using a XML-based language. WS-Agreement is the most flexible language in the level definition of SLA. This means that users are free to define their terms as they want, but instead, a problem of automation is put seen that there is no common definition. All these works have contributed significantly to the emergence of SLA. However, none meets the needs for cloud computing environment and particularly they do not address the SLA violations in this kind of unpredictable and dynamic environment.

More recently, initiatives such as SLA@SOI [11] or Optimis [181] have addressed SLA specification for clouds. The SLA@SOI [11] language (SLA*) is based on the WS-Agreement while the Optimis language (WSAG4J) is a full Java-based implementation of WSAgreement and WS-Agreement Negotiation. Their solutions cover SLA lifecycle. However, the violations management does not reflect cloud characteristics, the model does not support multi-party agreements, does not provide the formal semantics of the language and requires the definition of the vocabulary for the cloud domain. In addition, in SLA*, the description of SLA is just limited to guarantee terms while external files are needed (e.g., Open Virtualization Format) is needed to describe IaaS services. The CSLA [97] language shares motivations with the SLA@SOI project and goes further by taking into account the cross-layer nature of Cloud and proposes the definition of SLAs using a mechanism to support fuzziness in the numeric specification of metrics. However, this approach does not consider multi-party agreements, the deployment models (e.g. community cloud) or the prominent role of broker in the domain. Moreover, the support of CSLA for the delivery models is restricted to a few metrics and terms, which also requires extensions for real-world deployment.

We note that the poverty of expression is the main limitation of existing languages. We observed that the SLA contract of the different actors in cloud is based on a single metric: availability. On the other hand, the verification of the non-violation SLA can be very tedious for the cloud customer. Yet, we argue that these languages are not able to cope with the set of distinctive characteristics of clouds, such as deployment models and the growing importance of the broker role. In addition, the languages and the works presented do not address the concept of dependence between the level SaaS (Web services) and IT resources (IaaS) except for SLA @ SOI. However, this project adopts a semiautomatic solution where the intervention of an expert is needed.

To sum up, we can say that none of the above-mentioned projects discusses the specification of the SLA for the XaaS model. In addition, each of the reviewed works handled different aspects in the SLA lifecycle. So far, no work has dealt with the modeling and the specification of the customer functional and QoS requirements and the modeling of a provider environment so as to facilitate the SLA establishment. Moreover, we find that the solutions in the cloud community are neither complete nor mature enough. Indeed, the definition of the service section remains generally restricted to a single-level XaaS which is the SaaS level.

Current approaches for SLA specifications in IT services are not sufficient since SLAs are usually based on templates that are expressed in a natural language, making automated compliance verification and assurance tasks difficult. In such a context, the use of models becomes particularly relevant: such models can help retrieve data from the running system to

verify SLA compliance and if the desired quality levels are not achieved, drive the dynamic reconfiguration of the cloud services architecture.

There is a lack of research that uses models for establishment SLA built upon MDE techniques and methods. The DSML/DSL facilitates the specification of provisioning and deployment concerns establishing service level agreements between a cloud provider and cloud customer in the context of a particular service provision.

• ***Motivation and overview of our approach for SLA establishment***

In this work, parameters exclusive to cloud SLA contract are identified and defined formally (chapter 4 section.2.3). These parameters are ingrained into proposed SLA contract meta-model to cope with the diversity of cloud applications. Specially mentioned in this work these parameters such as availability, interoperability, reliability, trusted third party and reputation so as to facilitate cloud consumer before availing services provided by a particular cloud provider.

Our work differs considerably from the existing SLA languages or solutions :

- It emphasizes the formal aspects of SLA;
- It considers the particularities of the cloud computing domain (automation task to express the SLA contract requirements);
- It provides the base for dynamism in SLAs.

An SLA establishment succeeds if and only if the provisioning of services is achieved under clearly specified conditions. These conditions must be understood by both cloud providers and customers so that services meet the customer requirements.

Our approach takes these needs into account by offering modeling languages, transformations and automated processes. In terms of support for the SLA establishment, our approach offers an SLA contract meta-model where:

- The Cloud customer defines his requirements according to the CloudCustomer meta-model which is based on the DSML.
- The cloud provider provides his offers according to the CloudProvider meta-model.
- The consumer requirements with the provider offers are automatically matched into a preliminary SLA contract model, which is enriched by our penalty and reputation approach to instantiate some concepts in order to obtain a complete service level agreement.

We next detail the meta-models we propose and the transformations occurring between them. Afterward, we present the SLA contract generation process.

The cloud SLA contract language we present is a meta-model (consisting of abstract syntax and well-formedness constraints) for defining cloud SLAs, specifically for XaaS clouds, that support most of the characteristics of cloud. The purpose of this abstract syntax is to support the construction of well-defined cloud SLAs models. The abstract syntax includes specifications of QoS concepts (which define a number of QoS parameters, specifications of the obligation terms (i.e. SLO and credits) and cloud resource units. Each cloud unit defines the characteristics of unit and its price. The abstract syntax allows both cloud providers and consumers to define a well structured cloud SLA contract for use in systematic composition and selection. This SLA can be used as a starting point for cloud negotiations; in particular, the meta-model makes it possible to define different cloud SLA models, which can then be compared automatically.

3.3 Constructing the cloud SLA contract meta-model: Modeling languages

To establish cloud SLAs contract based on MDE tools and principles, consumer and provider models are required. As discussed before in the introduction, we constructed three meta-models for cloud SLAs: one for cloud provider to define his proposed resources to meet the Cloud customer requirements and the other for cloud consumer to define his functional QoS requirements. However, we ended up with one cloud SLA contract meta-model combining both cloud provider and consumer, which will be detailed later.

As we explain in the remainder of this section, the definition of the meta-models of these three languages relied on existing standards, on the one hand, and on the need to automate the derivation of the SLA contract, on the other. This section presents not only the meta-model for describing cloud SLAs contract, but also the process by which it was developed, including the different versions of the meta-model that have been elaborated.

3.3.1 Cloud provider meta-model offer specification language

Before constructing the CloudProvider meta-model we have analyzed relevant related work in particular [109][83][154][18][82] [97] to determine the accepted ways of specifying SLA concepts and structure. Thus the development of meta-model is inspired by the latest cited works: which it is objective and precise. Providers can express functional and nonfunctional offers that want to be satisfied.

The CloudProvider meta-model will be used by the Cloud provider to model a configuration he might offer to satisfy the functional, nonfunctional and QoS requirements defined by the Cloud customer. Indeed, the CloudProvider meta-model is based on domain specific modeling languages. As shown in Figure 3.1, to these standard concepts, it adds new concepts and relationships pertinent to the Cloud environment and to the services and resource discovery.

The figure 3.1 shows that an SLA meta-class defines different services. There are different categories of services (SaaS, PaaS, IaaS), here we present just SaaS categories composed with: Computing units, Storage or Networking, presented as subclasses of the Service Class to define the service type. Also, to cope up with the change in technology domain and time, many new things should be incorporated. For example, with the wide adoption of cloud computing, nonfunctional parameters should be an integral part of SLA. Section 3 in chapter 4 enlists, key parameters to be included in cloud SLA and gives their formal definition with respect to cloud. Four specifications namely availability, interoperability, reliability and data security related to cloud services are only being taken up in this meta-model. Furthermore, pre-defined cloud providers provide an SLA for each service, e.g. Amazon (EC2 SLA and S3 SLA) and RackSpace (Cloud Server SLA, Cloud Files SLA and Cloud Load Balancer SLA). The Service class refers to the QoS class.

This QoS is a superclass of Availability, Interoperability, Data security, Modifiability and Reliability. These classes are defined to provide the QoS concept. In our context Availability can be calculated in terms of uptime and downtime, which are described as low level parameters. The figure 3.1 illustrates that the attribute of Availability is called the attribute of type AvailabilityAttr. AvailabilityAttr is an enumeration that defines upTime, downTime and incident-Time. The enumerated type AvailabilityAttr specifies elements of the availability quality. Different attributes of Reliability include Mean Time TO Repair (MTTR), Mean Time Between Failure (MTBF) and Max Time To Recover (MaxTTR). Thus a cloud provider define their QoS concepts using specific terminology. The same of Interoperability and Data security, which the attribute of Data security is called the attribute of type datasecurityAttr etc... Further, a CloudProvider meta-model is proposed, which incorporates these parameters and tries to bridge the gap with existing SLA templates providers. Finally, billing is an action subclass, and an attribute of the SLA class. We propose a pay-as-you-go billing mode where the usage and the cost parameters are dynamic.

• *Creating Cloud SLA models using the Cloud SLA abstract syntax*

This subsection describes how the SLAs of cloud provider is defined using the proposed cloud SLA abstract syntax. The purpose of this subsection is to test the applicability of the proposed meta-model, to define pre-defined cloud SLAs as model instances of this meta-model. The

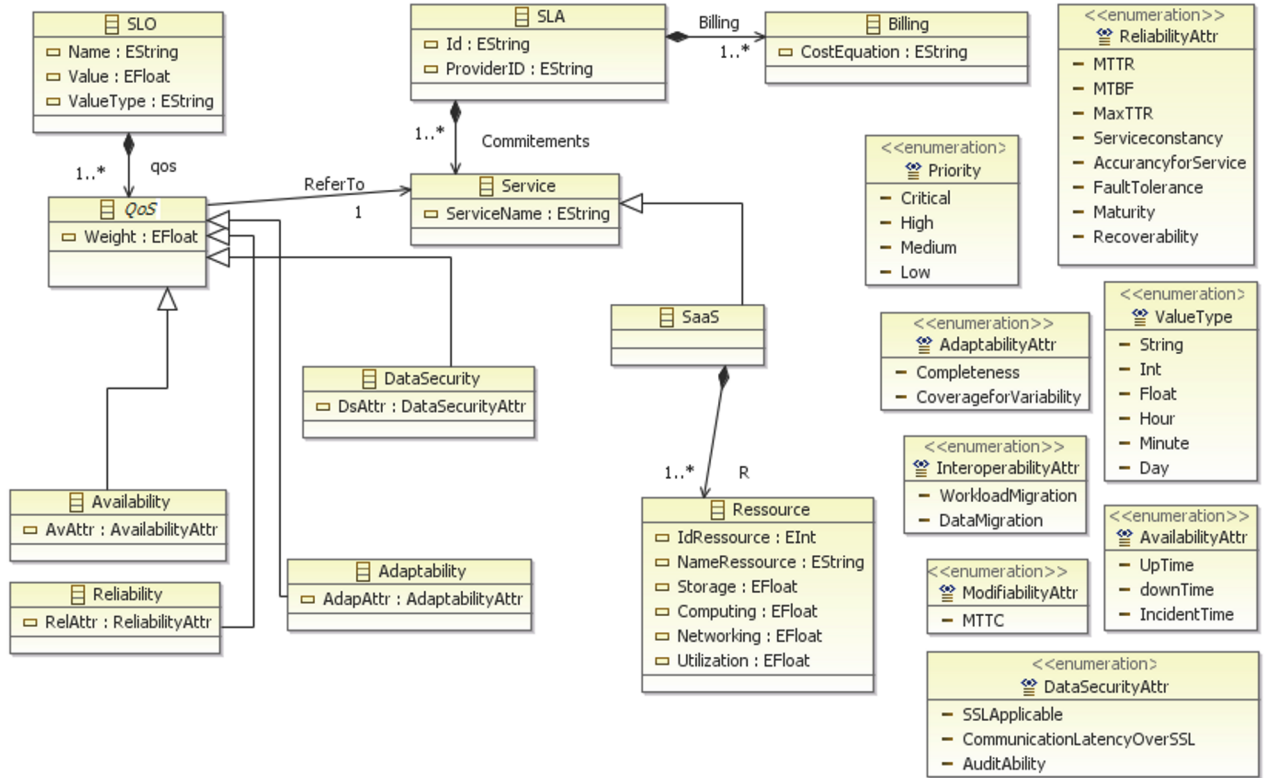


Figure 3.1 A provider SLA meta-model

models from this offer are shown in Figure 3.2. The model were created using a tree-based EMF editor. We have described above the cloud provider SLAs offers and explained how these can be defined using the proposed abstract syntax, as cloud SLA model instances.

3.3.2 Cloud consumer meta-model specification language

Figure 3.3 illustrates our proposed CloudCustomer meta-model for consumer requirements to define his needs for cloud services. A meta-model defines the abstract syntax of the SLA, e.g, it defines the main classes of the SLA, their attributes, constraints and rules to form consumer SLA models. Similar to the CloudProvider meta-model, the CloudCustomer meta-model is also based on the domain specific modeling languages. Therefore, an SLA meta-model is a means for cloud consumers and cloud providers to define and specify their SLA. The meta-model is similar to the provider meta-model. The difference is: the service class (which is not included in the consumer meta-model, largely as a simplification).

• *Creating Cloud SLA model using the Cloud SLA abstract syntax*

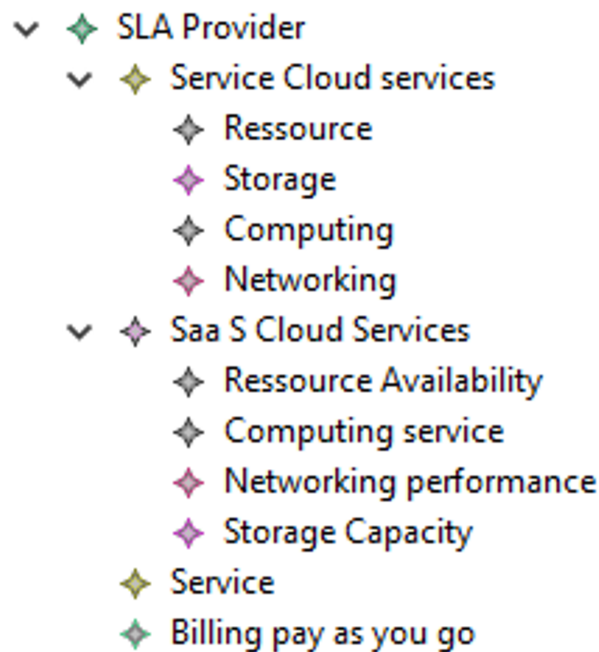


Figure 3.2 Illustration of provider SLAs offers using the cloud SLA metamodel

Figure 3.4 illustrates consumer requirements in the form of an SLA model. This consumer model is used to express his requirements with cloud SLA model. A cloud consumer provides an SLA to find a provider offer that matches their requirements. The cloud consumer SLA is defined, as we did with the cloud offers, by using the proposed cloud SLA metamodel. Our correspondence process starts to find a match between the requested and offered SLAs, will be discussed after.

3.3.3 SLA contract meta-model: definition language

Once the Cloud customer expresses its requirements for a set of abstract services, the Cloud provider provides the right service instance and automatically determines the appropriate resources required from the infrastructure layer in order to respect the QoS requirements of his consumers. In a final step, an agreement is established between these two parties. We extend the meta-model to include other concepts (penalties, reputation and trusted third party features). This agreement is presented in the form of our proposed extended version of the Service Level Agreement for cloud computing (SLAC) [170] which we call SLA4C (Service Level Agreement **for** cloud computing).

Our proposed meta-model describes QoS guarantees in the form of SLO clauses. One of the novelties that it integrates features dealing with QoS uncertainty and cloud fluctuations,

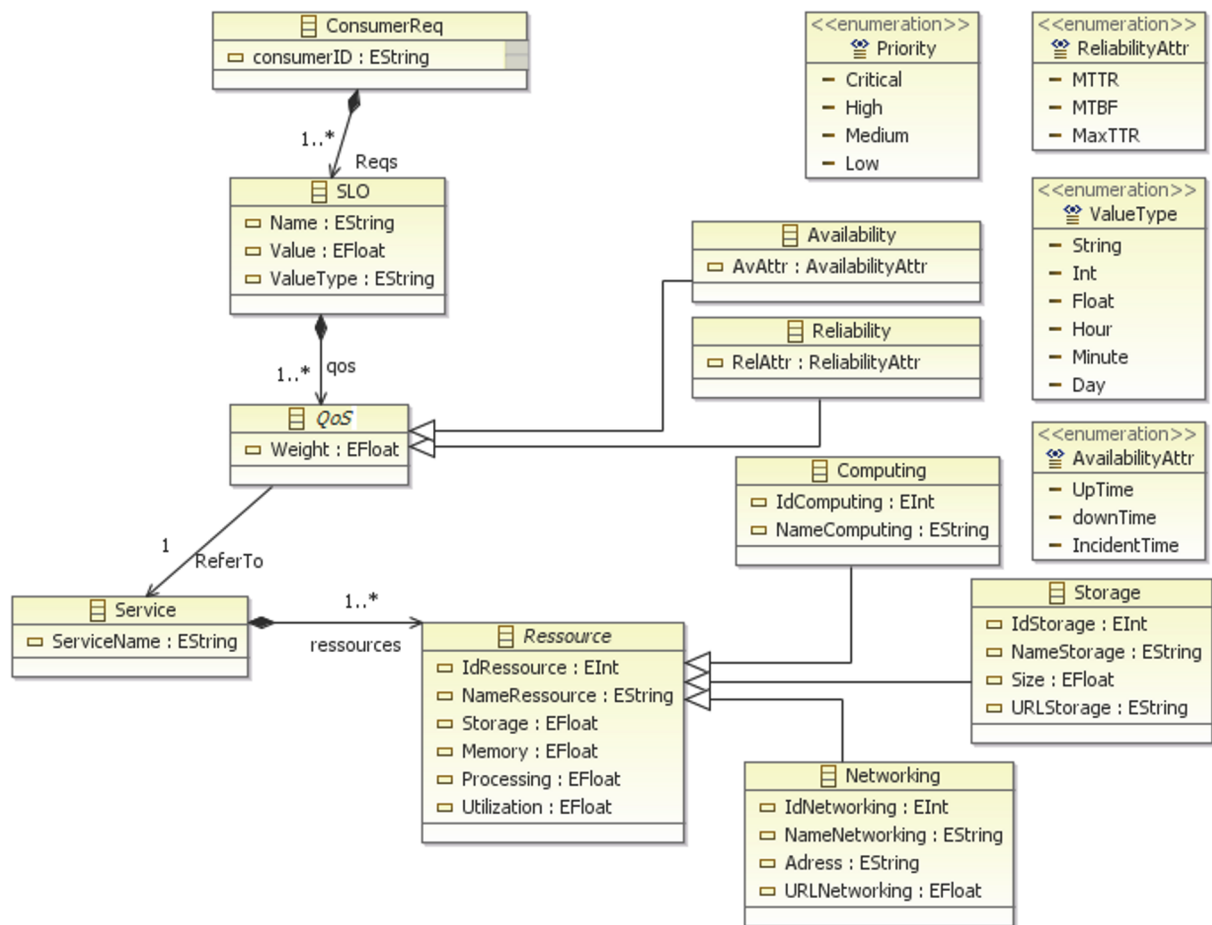


Figure 3.3 A consumer SLA meta-model

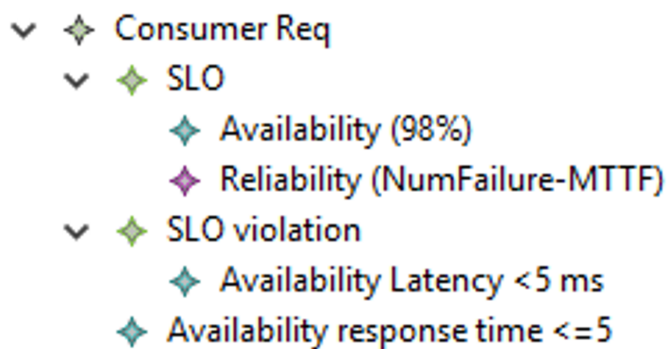


Figure 3.4 Illustration of consumer SLAs requirement using the cloud SLA metamodel

such as penalty. The penalties are applied in case of SLA violations, to compensate cloud service customers. We model the SLA violation penalty by a linear function as described in the following section 4.5.

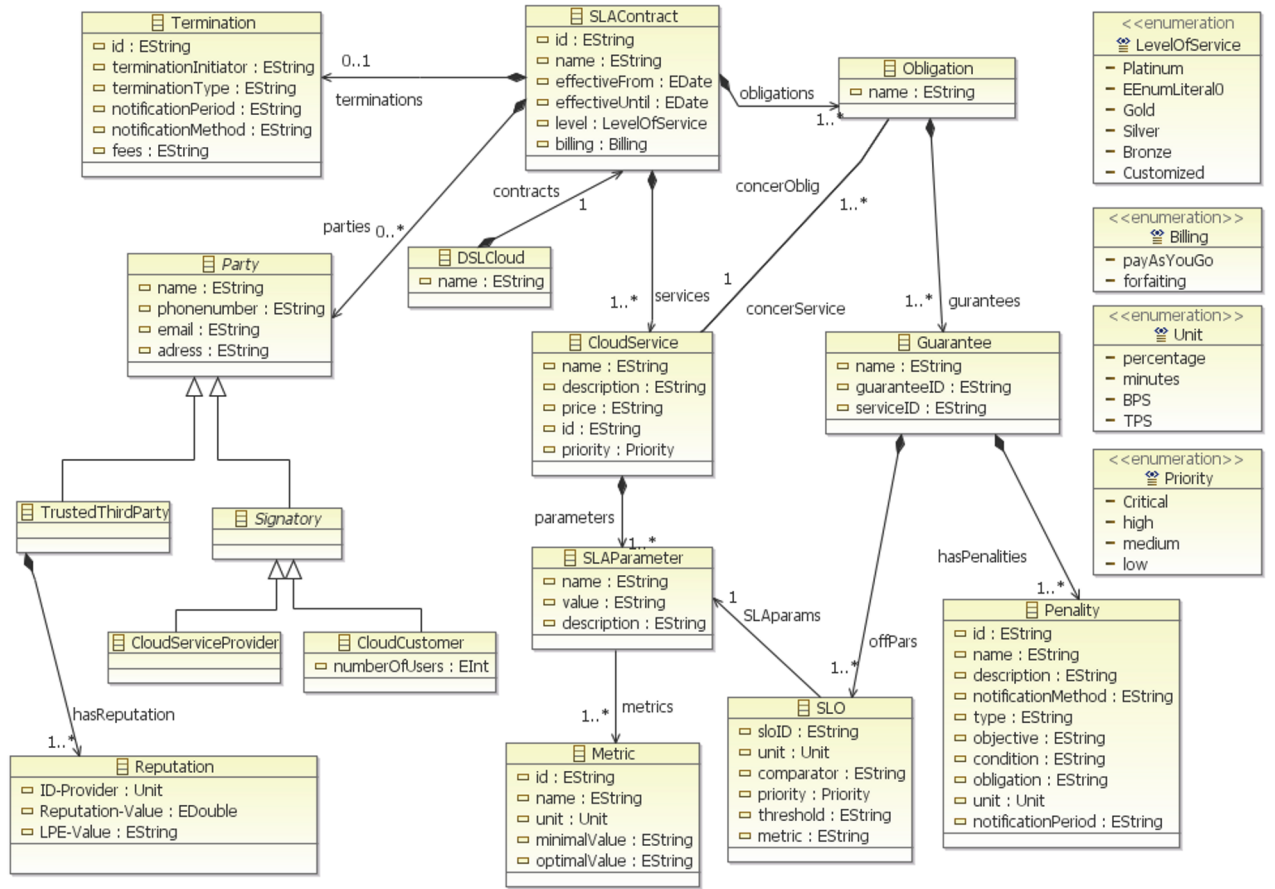


Figure 3.5 Metamodel of the proposed SLA Contract

An *SLA contract* is composed of five principal elements: a section defining the *parties* involved in the agreement, a section describing the *validity* of the agreement, *Cloud services*, *Parameters*, *Guarantees*, *Billing* and *Terminations*.

Firstly, *Party*, an abstract metaclass that defines the parties involved in the agreement: the signatory *parties*, namely Cloud service provider and Cloud customer, and trusted third party involved in monitoring SLAs in real time. Secondly, the *Cloud services* metaclass presents a description of the services defined in the SLA, which refers to any XaaS service (SaaS, PaaS or IaaS). It also contains *Parameters* of Quality of Service that will be used in the specification of the obligations. *Parameters* provide a way to define variables in the context of the agreement that should be used in others sections. Variables refer to a distinct element such as *Metric* and *Schedule*. These *Parameters* are based on metrics evaluated by measurement guidelines. We propose two types of values, optimal and minimal. The difference is that the second one contains the minimal value allowed for a parameter. The third principal metaclass is *Obligation*; it presents the obligations of the SLA. The *Guarantees* metaclass

contain a set of guarantees. Each guarantee consists of two elements: Terms and Penalties. A guarantee term contains one or more Objectives (SLO). Each objective defines an expression that must be met according to a precondition. An expression formulates a constraint and it is characterized by a Metric, a Comparator and a Threshold. We define a Priority for each objective to take into account the customer QoS preferences. The validity defines how long an agreement is valid. In case of SLA violation, penalties are applied to the service provider in order to compensate the consumer for tolerating the service failure. The compensation can be applied either as a constant or variable rate. Our meta-model supports two types of *Billing*: Pay as You Go and All-in package. Finally, the Agreement starts on effectiveFrom and ends on the earlier of the effectiveUntil or in accordance with *Terminations* section.

Reputation metaclass contain the reputation value assigned to the cloud provider which is calculated by our trust reputation Model as described in the section 5.3.4.

The meta-models presented above can be used to produce editors, allowing end-users (e.g., consumers or providers) to specify their requirements or their offerings; we have implemented such editors in Eclipse. The output of the SLA Editor is an SLA Contract expressed in a SLA language, compliant with the SLA meta-model.

3.4 Generation process: for SLA contract establishment generation model

It is important call to mind that our objective is to automate the SLA contract document generation step. As shown in Figure 3.1, this objective is achieved through an MDE transformation method at the meta-model level and through some model enrichment methods. The meta-models was noted as one of the requirements in order to help to specify and better automate the process of establishing SLA between cloud consumer and provider. Our main concern is to find matches between cloud consumer requirements and cloud provider offers based on the QoS parameters. At the meta-model level, there are two essential transformations: correspondence and composition operations between CloudCustomer and CloudProvider meta-model. In the correspondence process that we present focus on finding matches and similarities between the elements of CloudCustomer and CloudProvider meta-model. These transformations are based on rules implemented in our prototype using Epsilon Comparison Language (ECL) and Epsilon Merging Language (EML). Their aim is to automate the creation of SLA contract documents (target model) from the cloud customer requirements and provider offers (source models). The overall intention of our approach is to automate the matching process of consumer requirements with cloud provider offers. In our matching

approach, we first match an SLA model for consumer requirements and an SLA model for provider offerings; this is the core part of our approach.

3.4.1 General Process of Correspondence and Composition

The development of complex systems is based on a varied set of languages, tools and environments that are generally used separately by modeling experts working on different dimensions of a project. In addition, developers are often located in distant geographical areas, as is the case in distributed collaborative development, which complicates their cooperation. Among problems that typically arise in this type of situation, we can mention the fact that different terminologies and terms can be used to represent the same concept or that the same term can be used to express different concepts. In this subsection we describe how to tackle this problem by providing an SLA contract from an SLA consumer model and an SLA provider model. The SLA contract model cannot be constructed in a monolithic manner, it has to be divided into two operations: correspondence and composition. The process below described the different models involved, as shown in Figure 3.6.

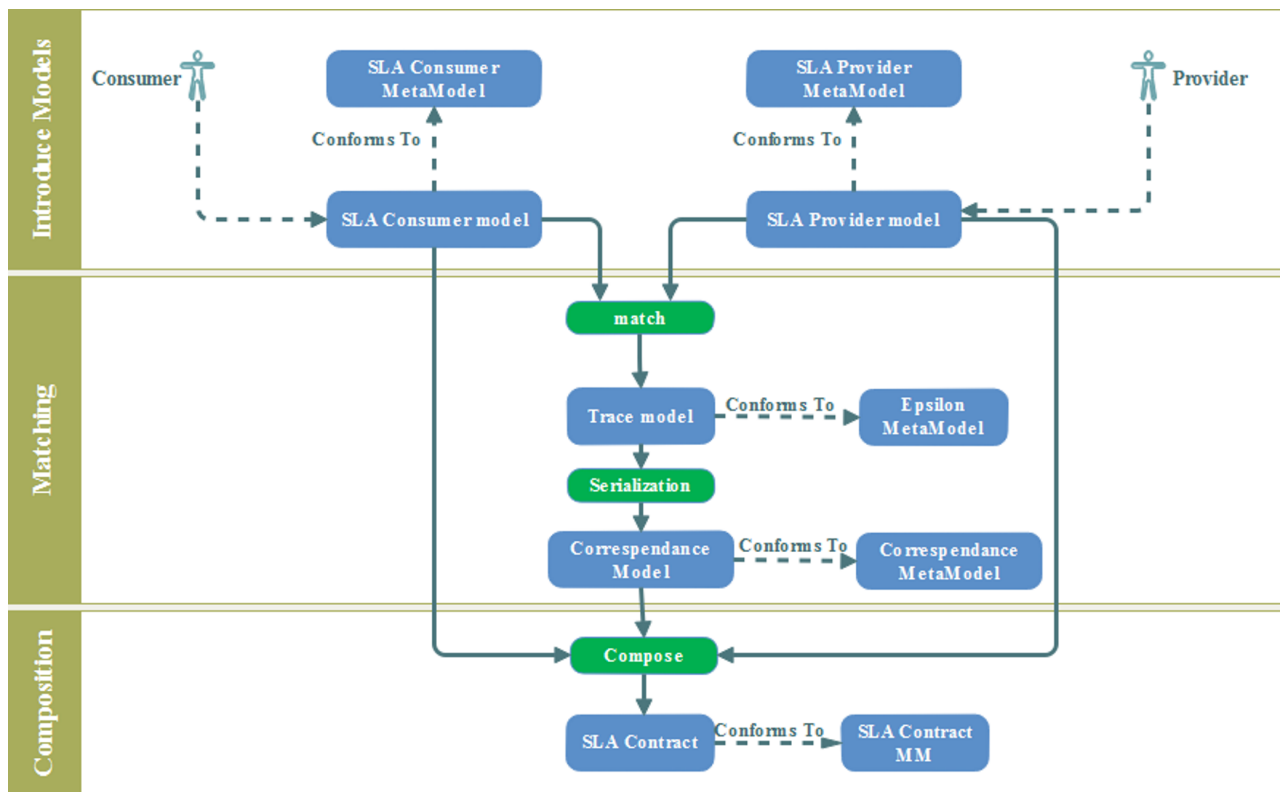


Figure 3.6 General Process of Correspondence and Composition

- *Correspondence process*

The role of the correspondence operation, formally described in [24], is to analyze the input models in order to identify the different relations between them. It consists in defining a correspondences model that captures links between different models. The semantics of these links is defined in a correspondence meta-model.

The correspondence model can be created by different approaches. [24] presents a state of art of this approaches. In Epsilon a match operation is defined using comparison rules (ECL). These rules search for relationships between the models elements. The match operator does not have fixed semantics. The semantic is defined with comparison and conformance rules. Comparison rules determine syntactic similarities between model elements. Conformance rules identify if a subset of syntactically similar elements are semantically compatible. The result of comparing two models with ECL is a trace (MatchTrace) that consists of a number of matches (Match). Each match holds a reference to the objects from the two models that have been compared (left and right), a boolean value that indicates if they have been found to be matching or not [88]. To exploit the precedent trace and so to be able to reuse the result for MDE purposes (e.g. composition), the developer must do a serialization step to transform the traces into a model of correspondences. This latter is conform to a metamodel model of correspondences proposed in [24].

The MMC represented in Figure below is composed of a set of meta-classes. Correspondence Model is composed of correspondences established among at least two (meta)-elements belonging to different (meta)models through their references. The Correspondence meta-class is composed of at least two extremities, that are elements from different (meta)models, and a relationship. The latter is associated with the meta-class RefElement to define n-ary correspondences connecting at least two elements with a relationship. Its bidirectional attribute specifies whether the relationship is bidirectional or not. If the relationship is bidirectional, the concerned (meta-)elements are all source ones and there is no target (meta-)element, as specified by the OCL rule. Relationship is an abstract generalization of DIR (Domain Independent Relationship) and DSR (Domain Specific Relationship) meta-classes. The specialization of the first one allows representing generic relationships that may exist in different domains. However, these relationships may be insufficient for a given domain. In this case, it is possible to represent specific relationships by a specialization of the DSR meta-class.

• *Composition process*

The purpose of this process is to create the global model by combining the elements of the input models using the relationships defined in the correspondence model. The compose operation $M_{AB} = \text{Compose}(M_A, M_B, C_{AB})$ takes two models M_A, M_B and a correspondence model C_{AB} between them as input and combines their elements into a new output model.

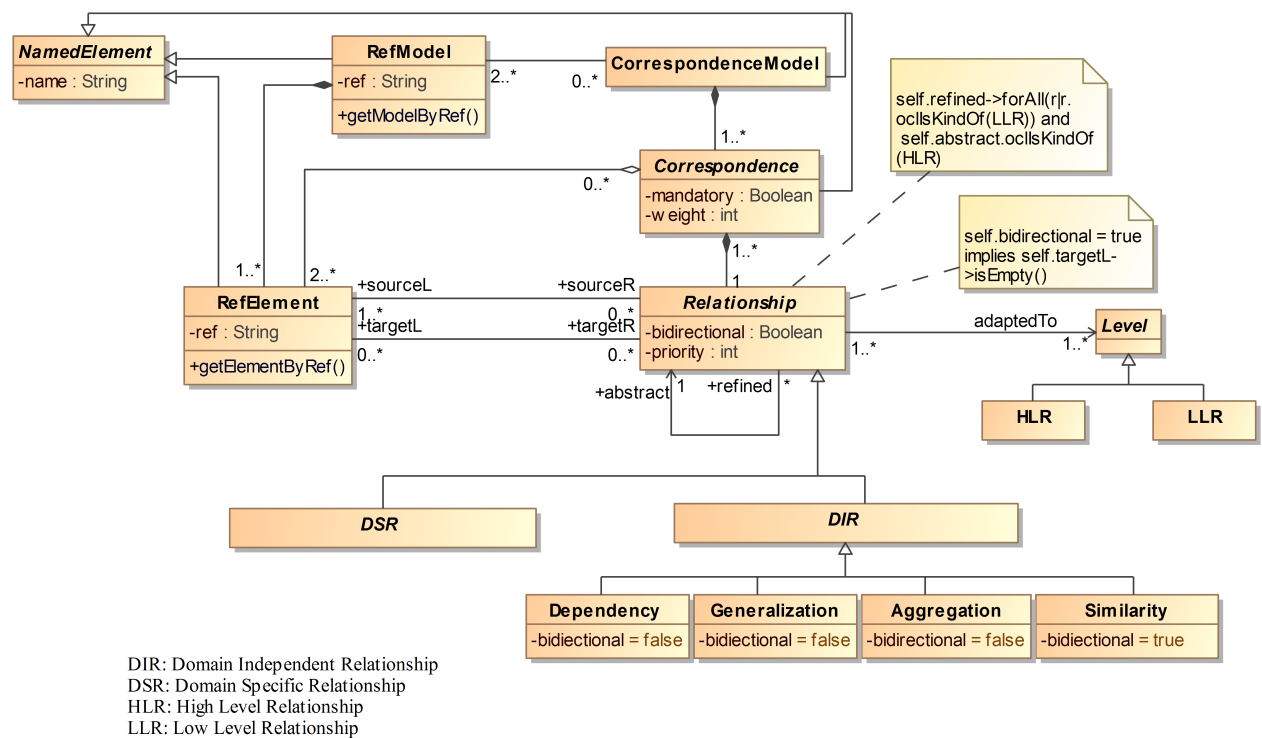


Figure 3.7 Correspondences meta-model

In the three approaches there are some differences in the terminology to specify what a composition is. Besides composition, the second most employed term is *merge*. However it is advisable to separate merge and composition. Composition is a more general operation. The semantic is specified in the different operations by a set of rules, and it varies from case to case. Merge, however, is a special case of model composition. Merge has information preservation constraints, i.e., all the information from the input models should be present in the output models, and no duplicate information.

Merge operation: The merge operation $M_{AB} = \text{Merge}(M_A, M_B, C_{AB})$ takes two models M_A, M_B and a correspondence model C_{AB} between them as input, and returns a model M_{AB} including all the information from M_A and M_B , without duplicate information. The correspondence model is created by the match operation. It specifies the elements that are going to be merged.

The composition process was constructed for model comparisons, i.e. cloud consumer models composed with cloud provider models. The constructed meta-model, finale SLA contract, is used to define both cloud provider and consumer. Thus the composition process provides a comparison for matching providers SLAs when they conform to the same meta-model of consumer SLAs.

3.4.2 Implementation of Matching Logic in Epsilon

The purpose of this section is to describe the matching logics and the outcome models of the matching process. This process matches cloud consumer requirements with cloud providers offers.

After discussing the general process of correspondence and composition conceptually, we present an implementation based on standard model management tasks and tools, particularly Eclipse EMF and Epsilon. The matching implementation is divided into several ECL tasks. In this section we explain an ECL task, which focuses on matching different parts of the SLA models. There are different possibilities available for defining a match between two models. The suitability and appropriateness of a particular logic depends on the user requirements, which are implemented using a set of ECL rules and by using a particular set of input models. Another benefit of this correspondence metamodel is that both the consumer and provider can define their models by using its abstract syntax. It provides a communication language between them.

This correspondence process includes a matching logic that can be applied to two cloud SLAs models. The matching logic is implemented using a model comparison language (ECL). The comparison process can be used to match the concepts of SLA (i.e. matching model elements and properties) as well as matching the values of the SLA concepts. Different

matching algorithms were provided, which can be used to support different correspondence scenarios and satisfy different customer requirements. One of these algorithms is arguably too restrictive (optimal match), while another is too flexible (name-based match); however, these logics are needed in different scenarios, because clouds are differently defined and specify various QoS and resource units, as well as defining their structures differently. This transformation aims to guarantee a reliable management of the SLA. It provides a way to map/match the Cloud customer requirements and Cloud provider view in a coherent way.

Also, we will focus on a general SLA meta-model comparison. In some comparison modules, there may be differences in comparison implementation between different matching logics. The basic pattern in ECL is to compare models by matching pairs of elements. For this, we define a *matching* rule which has a unique name, *leftParameter* and *rightParameter*. In the compare part we define the matching logic for the specific parameters under comparison. The **matches()** operation is used implicitly to determine and invoke the (type-) appropriate matching rule to compare two elements. In the following subsections we describe the implemented rules in each **EclModule** we have created. Each ECL rule defines the *left parameter* and *right parameter* to be matched. In our implementation the left parameter represents an element of the consumer model and the right parameter represents an element of a provider model.

We now provide an example of matching a cloud consumer model with a provider model. Different matching logics are applied. The purpose of this example is to show the differences in the trace models using different logics.

In this general matching process, we match the references of the cloud SLAs meta-models. The process of matching the SLA models consists of five main matching tasks as follows:

- Matching SLAs
- Matching units and prices.
Matching SLAs consists of matching composite elements.
- Matching services
- Matching the definition and QoS parameters
- Matching the obligation and SLO

In all examples, the input model for the matching process is a model defined based on consumer requirements (A). We also make use of a provider model (B), as illustrated in Figure 3.6, and the output model is a finale SLA contract trace model, which is presented as an XML document.

The general structure of the matching logic is illustrated in Listing 3.1 in pseudo code. The consumer and provider SLA models are instances of the meta-model described before. Thus, we use meta-model class names in this algorithm. The comparison logic in Listing 3.1 illustrates the logic of finding matches between the SLA model of consumer requirement and the SLA model of cloud provider offers.

```

1  for each providerSLA model in a set of provider SLA models{
2  p = providerSLA model;
3  c = consumerSLA model;
4  for each element in c {
5  for each element in p {
6  If c.SLA.service = p.SLA.service{
7  apply matching rule for c.DefinitionTerm and
   p.DefinitionTerm
8  ;
9  apply matching rule for c.ObligationTerm and
   p.ObligationTerm
10 ;
11 }
12 apply matching rule for c.CloudUnit and p.CloudUnit;
13 }
14 }
15 return trace of matching elements ;
16 }

```

To match two cloud units, different rules are applied, as illustrated in Listing 3.2: a rule for matching cloud specifications, a rule for matching prices and a rule for matching locations. If all rules return true, then a match for cloud units between the consumer and provider has been found.

Matching cloud units SLO between consumer and the provider SLA model

```

1  p = providerSLA;
2  c = consumerSLA;
3  for each c.cloudUnit{
4  for each p.cloudUnit{
5  apply matching rule for c.cloudUnitSpec and
   p.cloudUnitSpec;
6  apply matching rule for c.price and c.price;
7  apply matching rule for c.location and c.location;

```

```

8   }
9   }

```

Listing 3.3 illustrates a match rule for the elements of the type Value of the SLA metamodel (see Section 4.3). The rule defines ConsumerSLA!Value as a left parameter and ProviderSLA!Value as a right parameter. The rule matches two Value elements of the consumer and provider models. The two values are matched if they define the same value type e.g. both define a value of the string type or both define a value of numeric type and both define the same value of the isPositive attribute.

```

1  p = providerSLA;
2  c = consumerSLA;
3  for each c.cloudUnit{
4  for each p.cloudUnit{
5  apply matching rule for c.cloudUnitSpec and
   p.cloudUnitSpec;
6  apply matching rule for c.price and c.price;
7  apply matching rule for c.location and c.location;
8  }
9  }

```

• *Comparison of Cloud Service Implementation*

This subsection presents the implementation of matching services which is part of the SLA model matching. To match the SLA models, we first match the service defined by the SLA model. We define a rule to match the services defined in two SLA models (e.g. the consumer and provider models). This rule, as illustrated in Listing 3.4, matches the element names, which are: Computing, Storage and Networking. This rule is imported by other ECL modules that match the SLA and QoS. This rule is the same for the three matching logics.

```

1  || greedy rule matchServicesmatch c : ConsumerSLA!Service with p : ProviderSLA!Service
   compare : c.service.EClass.name = p.service.EClass.name

```

By using this matching logic, we achieve an-automation for cloud SLA matching. This matching process matches SLAs one by one, which facilitates, for the providers, the analysis of the similarity and differences between two SLAs. Therefore, we conclude that the proposed approach satisfies automated SLA establishment.

3.4.3 Implementation of Composition-Merging Logic with Epsilon Merging Language

In the merging phase, there are two activities that produce elements in the target model[92]; The elements that have been identified as matching are *merged* into a sequence of model elements in the target model and a selection of the elements for which a match has not been found in the opposite model are *transformed* into elements of the target model. Therefore, EML provides two different types of rules; *merge-rules* and *transform-rules*.

Each merge-rule defines the types of elements it can merge, as well as a list of the elements it produces in the target model. In its body, the merge-rule defines the exact way in which source elements are related to the newly created elements in the target model. Similarly, each transform-rule defines the type of instances it can transform, a list of model elements that it produces in the target model and a body that implements the actual transformation[92].

EML is a hybrid, rule-based language for merging homogeneous or heterogeneous models. As a merging language requires all the features of a transformation language (merging model A with an empty model into model B is equivalent to transforming $A \rightarrow B$), EML reuses the syntax and semantics of ETL and extends it with concepts specific to model merging.

A merging operation is a transformation in a general sense, since it transforms some input (source models) into some output (target models). However, a model merging facility has special requirements (support for comparison, conformance checking and merging pairs of input elements) that are not required for typical *one-to-one* or *one-to-many* transformations [88] and are therefore not supported by contemporary model transformation engines.

Merging Strategy A merging strategy has two methods. *The appliesTo(leftElement : Object, rightElement : Object) : Boolean* decides if the strategy applies to a specific pair of instances by returning a *Boolean* value. *The autoMerge(leftElement : Object, rightElement : Object)* specifies the logic that merges the two instances[92].

Practical merging strategies we have developed include the *CommonMofMetamodelMergingStrategy* and the *CommonEmfMetamodelMergingStrategy* strategies for merging elements of the same type originating from MOF or EMF-based models of a common metamodel[92]. The merging algorithm they follow is quite straightforward; for single-valued features they choose the value of the element from left model while for multi-valued features they perform a union of the left and right values and assign them to the feature of the target element.

Transformation Strategy Similarly to the previous two types of strategies, a transformation strategy can transform a source element into an equivalent element in the target metamodel. An EML module has two associated transformation strategies; the *leftTransformationStrategy* and the *rightTransformationStrategy* that can transform elements from M_A M_B respectively. As an example, a useful concrete transformation strategy is the *Common-*

MofMetamodelTransformationStrategy that creates a deep copy of the source element on which it is applied in the target model[92].

Before merging can be performed, correspondences between elements of the input models need to be established. As discussed throughout the previous section, the correspondences are recovered with the comparison language of Epsilon and stored in an instance of the metamodel of correspondences.

Example: Heterogeneous Model Merging with ECL/EML

```
1 rule MergeEntityWithTerm
2 merge s : Source!Entity
3 with t : Vocabulary!Term
4 into m : Target!Entity {
5
6 m.name = t.name;
7 m.inDomain = true;
8 }
9
10 rule MergeSystemWithVocabulary
11 merge s : Source!System
12 with v : Vocabulary!Vocabulary
13 into t : Target!System {
14
15 t.entity = s.entity.equivalent();
16 }
17
18 rule TransformEntity
19 transform s : Source!Entity
20 to t : Target!Entity {
21
22 t.name = s.name;
23 t.inDomain = false;
24 }
```

3.4.4 Xtext: Grammar Xtext of our textual Editor

Xtext is a text editor development language based on EMF for the Manipulation of models. It provides a grammar definition language similar to (EBNF : Extended Backus-Naur Form [22]) for the description of the syntax of the language. The generated Xtext editor offers a set of features: syntax highlighting, Automatic completion, automatic indentation, navigation, search through the outline and finally validation via a type expression language OCL named Check [95].

The text editor, implemented with Xtext, allows people who are used to working in "source editing" mode to create the matching model. For this, the expert, relying on automatic completion, syntax highlighting, etc., begins by importing the (Meta) models, then creates the match template and defines the different matches. The syntax of the editor is based on a grammar defined in EBNF (Appendix B). The beginning of the grammar allows to import the Ecore metamodel of the MMC. Through this, references are defined to this metamodel in order to specify the concepts corresponding to each textual form.

This grammar definition language gives us the possibility to create a textual editor that allows the provider to define its model offers, also for the consumer to define its requirement model.

3.5 Validation of SLA contract meta-model with Bunge-Wand-Weber (BWW) representational theory

In the previous section, we presented a meta-model to express SLA Contract requirements using MDE, as a mean for establishing service level agreements between a cloud provider and cloud customer in the context of a particular service provision. Our approach for SLA establishment embraces MDE principles to automate the SLA document generation from the customer requirements and provider offers. In this section, we evaluate the completeness and the clarity of our SLA contract establishment for conceptual modeling and examine the ability to generate the SLA document in less time. The results indicate that, our meta-model is capable of achieving a satisfactory completeness and clarity, which is very promising to accurately describe the structure and behavior of the real world.

The rest of this section is organized as follows: Section 3.5.1 provides a description about Bunge-Wand-Weber (BWW) representational theory. Section 3.5.2 presents the evaluation of the expressiveness of the proposed modeling languages and the efficiency of the proposed SLA contract establishment approach. Finally, the section is concluded with a summary of our contributions and their future potential extensions.

3.5.1 Representation Theory: Bunge-Wand-Weber

Bunge Wand Weber representation theory (BWW) [72][149] is a well-known benchmark used to analyze a particular modeling language by making an analysis of the language capability to provide a complete and clear representation of the domain being modeled. Employing the BWW ontology to evaluate modeling languages requires a rigorous methodology to ensure comparability, completeness and objectivity. Such analysis is achieved by relating, or

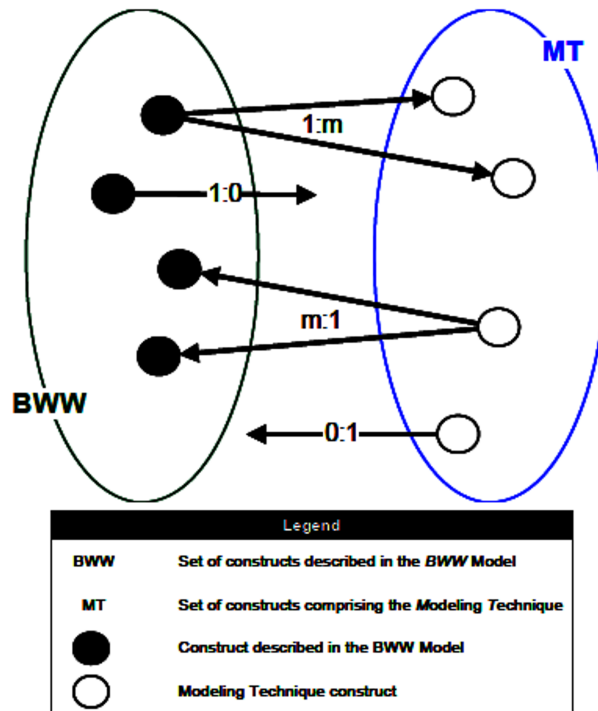


Figure 3.8 Potential representational deficiencies of a modeling technique [149]

mapping, the representational model of BWW and the language constructs to each other. This choice is motivated and justified by three reasons: First, the BWW representation model has reached a considerable level of maturity [144]; therefore, it could be a basis for the theoretical evaluation and comparison in our study. Second, BWW has been successfully used in many analysis projects that spanned various representational grammars [145][144]. Third, the BWW model defines a set of constructs considered as necessary and able to accurately describe the structure and behavior of the real world (requirements and offer). Two main evaluation criteria that may be studied according to the BWW model [145]: ontological completeness and ontological clarity. Ontological completeness is measured by the degree of construct deficit (1:0), i.e., the extent to which a process modeling technique covers completely the constructs proposed in the BWW representation model. On the other hand, ontological clarity is constituted by the degrees of 1) construct overload (m:1), or the extent to which single language constructs cover several BWW constructs, 2) construct redundancy (1:m), or the extent to which a single BWW construct maps to several language constructs, and 3) construct excess (0:1), or the extent of language constructs that do not map to any BWW construct. These criteria can be further categorized into four sub-types (see Figure 3.8).

Following the representation mapping, four measures can be established:

- The *degree of completeness (DoC)* is the ratio between the number of BWW constructs found to have a mapping to language constructs (C) divided by the total number of constructs defined in the BWW representation model (M). It is thus the inverse relation of the extent of construct deficit in a language;
- The *degree of redundancy (DoR)* is the ratio between the number of language constructs found to have a mapping to the same BWW construct (R) divided by the total number of constructs in the modeling language (L);
- The *degree of overload (DoO)* is the ratio between the number of language constructs found to have a mapping to more than one BWW construct (O) divided by the total number of constructs in the modeling language (L);
- The *degree of excess (DoE)* is the ratio between the number of language constructs found not to have a mapping to any BWW construct (E) divided by the total number of constructs in the modeling language (L).

Based on these four types of mappings, representation theory advocates the principle that a ‘good’ modeling language should be ontologically complete, i.e., it should not exhibit construct deficit. Ontological completeness implies that language users can describe all real-world phenomena that they seek to have represented by the information system they model. A ‘good’ modeling language should furthermore be ontologically clear, i.e., it should not exhibit construct overload, redundancy or excess. Ontological clarity implies that users can unambiguously describe the real-world phenomena that they seek to have represented without causing confusion to the end users.

3.5.2 Evaluation

This section exemplifies the investigation with regards to the evaluating of the representational capability (completeness and clarity) of SLA contract meta-model, which carried out based on the representational theory.

The primary goal of this evaluation is to provide a proof of a concept. Therefore, we evaluate two aspects: the expressiveness of the proposed modeling languages and the efficiency of SLA contract establishment approach. The former measures completeness and the clarity of this language (i.e., a qualitative evaluation), whereas the latter examines the ability of the proposed SLA establishment approach to generate the SLA contract in less time (i.e., quantitative evaluation).

3.5.2.1 Qualitative evaluation

The main objective of the qualitative evaluation is to assess qualitatively the expressiveness of the SLA contract offer specification language. Since this language are mainly based on DSML. We propose to evaluate our SLA contract language with BPMN 2.0 and UML QoS Framework (the combination of these languages) using Bunge–Wand–Weber representation theory (BWW)[145]. Two main evaluation criteria can be studied according to BWW: (i) ontological completeness that depicts the extent to which a process modeling language covers completely the constructs proposed in the BWW representation model; and (ii) ontological clarity that denotes the degrees of construct overload, the construct redundancy and the construct excess.

To conduct our evaluation, we followed the reference methodology proposed in [145][27] which implies the use of meta-models created in the same meta-language to easily improve the comparability between the constructs of each model. Hence, meta-models of the different languages (table 3.1 and 3.2) are used to apply the mappings with the BWW meta-model created in [51]. Two kinds of mappings are carried out:

- Representation mapping: Maps each construct within the BWW model to its corresponding construct of the SLA contract specification language. As a result, we can assess the ontological completeness and the redundancy degrees.
- Interpretation mapping: Maps each construct of SLA contract offer specification language to its corresponding construct in the BWW model. This mapping determines the potential degree of construct overload and construct excess.

- **Discussion**

Table 3.1 summarizes the results of the ontological completeness and clarity for our meta-model (SLA contract). From the results shown in this table3.1, we note that the construct deficit of the meta-model specification language is 28.57% which means that its completeness is 71,42%. This value is considered satisfactory in comparison with the scores of other languages [125][16]. For the ontological clarity, despite the value of the construct excess (43.12%) for the meta-model specification language, it becomes balanced thanks to the low value of construct redundancy (10.32%) and overload (30.28 %). Overall, these results demonstrate that the SLA contract have an acceptable ontological expressiveness.

In order to obtain a single value for the total degree of clarity, the numerical values obtained for the three representational deficiencies need to be merged together. Nevertheless, this merging requires the existence of a common source, thereby relative to the common

		c		
		BPMN 2.0	UML QoS Framework	SLA contract
Things	Thing	✓	✓	✓
	Property	✓	✓	✓
	Class	✓	✓	✓
	Kind	✓	✓	✓
States	State	-	✓	✓
	Conceivable state space	-	-	-
	Lawful state space	-	-	-
	State law	-	-	-
	Stable state	-	✓	-
	Unstable state	-	✓	✓
	History	✓	-	✓
Events	Event	✓	-	✓
	Conceivable event space	-	✓	✓
	Lawful event space	-	✓	✓
	External event	✓	✓	-
	Internal event	✓	✓	-
	Well-defined event	✓	✓	-
	Poorly defined event	✓	✓	-
	Transformation	✓	✓	✓
	Lawful transformation	✓	✓	✓
	Coupling	✓	✓	✓
Systems	System	✓	✓	✓
	System environment	✓	✓	✓
	System composition	✓	✓	✓
	System decomposition	✓	✓	✓
	System structure	-	✓	✓
	Subsystem	✓	✓	✓
	Level structure	✓	-	✓
Score		19	23	20

Table 3.1 Representation mapping

		BPMN 2.0	UML QoS Framework	SLA contract
Things	Thing	2	3	2
	Property	2	2	1
	Class	1	2	2
	Kind	x	1	3
States	State		5	3
	Conceivable state space			
	Lawful state space			
	State law			
	Stable state		4	
	Unstable state		1	
	History			2
Events	Event	9	7	8
	Conceivable event space		3	2
	Lawful event space		4	3
	External event	8	6	
	Internal event	8	6	
	Well-defined event	2	2	
	Poorly defined event	1	1	
	Transformation	6	2	7
	Lawful transformation	7	4	2
	Coupling	1	2	1
Systems	System	2	2	2
	System environment	2	1	1
	System composition	2	1	2
	System decomposition			
	System structure	2	4	3
	Subsystem	2	2	2
	Level structure	2		2

Table 3.2 Evaluating the clarity of SLA contract using constructs redundancy

Table 3.3 Results of ontological completeness and clarity of the modeling language

Language	Ontological completeness Deficit (%)	Ontological clarity		
		Redundancy (%)	Overload (%)	Excess (%)
SLA contract specification language	28.57	10.32	30.28	43.12

Table 3.4 Evaluating the total degree of lack-of-clarity of SLA contract

Relative value of the representational deficiency	Calculation	Final value
Degree of construct redundancy, relative to the degree of completeness	(10.32 / 71,42%)	14,45%
Degree of construct excess, relative to the degree of completeness	(30.28 / 71,42%)	42.40%
Degree of construct overload, relative to the degree of completeness	(43.12 / 71,42%)	60.40%
Total degree of lack-of-clarity	$(14,45 + 42.40 + 60.40) / 3$	39.09%

source, the values of the three representational deficiencies can be added together. The ability of our SLA contract to represent any given real world domain characteristic (i.e. completeness) has been considered as the common source, and the values of the three representational deficiencies relative to the total degree of completeness were identified (Table). Since the three representational deficiencies decrease the clarity, those relative values were used to achieve a single value for the lack-of-clarity of the Meta-model. Hence, the final value obtained for the total degree of clarity of SLA contract involves 61%.

3.5.2.2 Quantitative evaluation

In order to validate our SLA establishment approach and to illustrate its applicability, we conducted experimental testing on evaluating our meta-model to examines its ability to generate the SLA contract in less time. To this end, we describe a possible scenario of SLA establishment between a SaaS provider and his customer so as to ease the understanding of the proposed approach.

The goal is to establish an SLA using the proposed meta-model, to define an SLA template that contains all possible QoS attributes that should be established on and included in the final SLA. In this case study, A travel agency established an SLA contract with a web hosting provider Amazon EC2 to deploy its online flight booking services. We used a couple of SaaS services offered by the provider to handle different consumer operations (e.g. book flight, search, delete, list). We specified an SLA for each service, expressed through a set of QoS properties, mainly: response time, and availability.

We first describe the SLA establishment process using our proposed approach, then we present the experimental setup we have used in evaluating our SLA establishment approach. We propose a test scenario of SLA establishment by varying the Cloud customer requests

Table 3.5 SLA between Amazon EC2 and Travel Agency

Service	Metric	Oper.	Optimal Threshold	Minimal Threshold	Penalty
Web hosting	Response time	$\leq$	750 ms	770 ms	0.1\$
	Availability	$\geq$	99.9%	98%	
	Storage	$\geq$	200 GB	190 GB	
	Memory	$\geq$	5 GB	4 GB	
	Incoming Band-width	$\geq$	50 Mbit/s	48 Mbit/s	
	Outgoing Band-width	$\geq$	100 Mbit/s	98 Mbit/s	

(i.e., different QoS values) and the number of resources. Finally we discuss the results we obtained from the execution of this test scenario.

• *SLA Establishment process*

The SaaS provider should guarantee an average response time (R_t) less than or equal to 750ms (present the optimal value) a margin of 20 milliseconds is acceptable as a QoS degradation. Regarding the service availability, the SLA states that at least 99.9% of availability (Av) should be guaranteed, the minimal value of Av must be accepted at 98%. A violation of the response time SLO (resp. the availability) implies a penalty equal to \$US 0.1/per violated request. This SLA will enter into effect on first June 2017 (the Effective Date) for a defined period of two years. The other QoS parameters for web hosting services that should be included are described in Table 3.5. The aimed SLA is planned to be constructed according to the proposed SLA establishing engineering model in this research. (see figure 3.9 for concrete Model).

According to our Scenario, the metamodel is generated with pre-defined parameters to ensure that offered QoS guarantees are realistic and realizable. We use XML as a representation format. The following XML (see Listing 1, 2, 3, 4) presents an example of an SLA file describing the guarantee terms and penalties for SLA between the fore-mentioned provider and customer concerning the service S1 (Software as a service).

The SLA contract starts on 01-06-2017 and ends on the 01-06-2019 or in accordance with Terminations section. The Billing is based on the model Pay as You Go.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <DSLCloud:SLAContract xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"

```

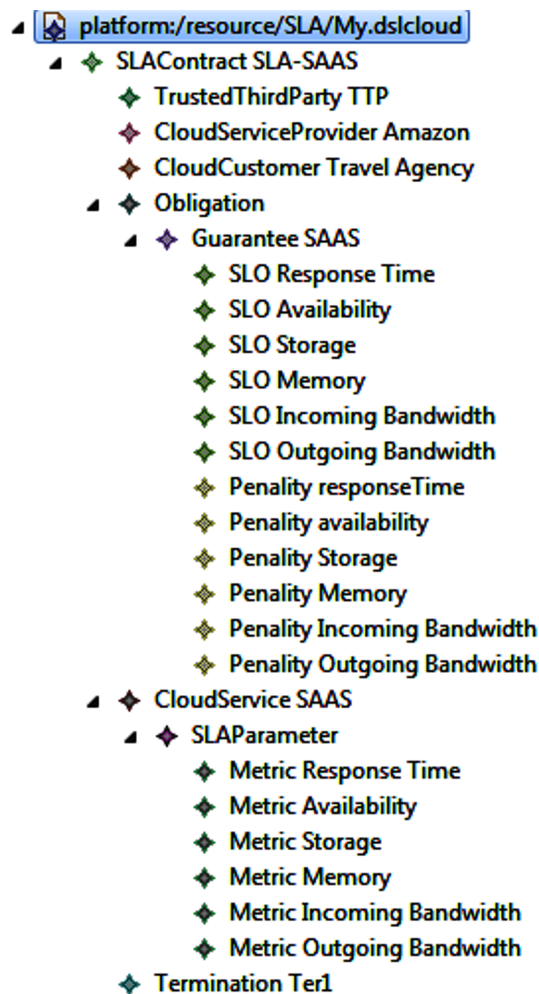


Figure 3.9 Concrete implantation of our Model

```

1  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2  xmlns:DSLCloud="http://MEH.fr/" id="SLA1" name="SLA-SAAS"
   effectiveFrom="2015-07-01T01:00:00.000+0100"
   effectiveUntil="2017-07-01T01:00:00.000+0100" level="Gold"
   Billing="payAsYouGo">
3  <parties xsi:type="DSLCloud:TrustedThirdParty" name="TTP"
   phonenumber="+212" email="ttpemail.ma" address="neutralparty"/><parties
   xsi:type="DSLCloud:CloudServiceProvider" name="Amazon" phonenumber="+33"
   email="CSPemail.fr"
4   address="France"/>
5  <parties xsi:type="DSLCloud:CloudCustomer" name="Travel Agency"
   phonenumber="+212" email="TAgencygmail.com" address="Casablanca"
   numberOfUsers="100"/></parties>

```

In this example, six SLOs are proposed using the "And" operator: a response time, availability, Storage, Memory, Incoming Bandwidth and Outgoing Bandwidth SLO. The response time SLO (lines 8), specifies that the maximum request (Rt) must be below 770 ms. The SaaS provider must guarantee at least 98% of availability, as it is shown in (lines 9). The availability can be defined as the proportion of service uptime, that is, the measure of likelihood to successfully access resources. The remaining SLOs are described in detail in table 3.5, highlighting their optimal and minimal threshold. These objectives should be achieved every working day between 7am and 22pm during the validity of the contract.

```

1 <obligations>
2 <gurantees name="SAAS" guaranteeID="G1" serviceID="S1">
3 <offPars sloID="Response Time" unit="ms" threshold="750"
  minimalValue="770"/>
4 <offPars sloID="Availability" unit="%" threshold="99.9"
  minimalValue="98"/>
5 <offPars sloID="Storage" unit="GB" threshold="200"
  minimalValue="190"/>
6 <offPars sloID="Memory" unit="GB" threshold="5" minimalValue="4"/>
7 <offPars sloID="Incoming Bandwidth" unit="Mbit/s" threshold="50"
  minimalValue="48"/>
8 <offPars sloID="Outgoing Bandwidth" unit="Mbit/s" threshold="100"
  minimalValue="98"/>
9 </gurantees>
10 </obligations>
11 <services id="S1" name="SAAS" description="Software as a service"
  price="100 dollars">
12 <parameters>
13 <metrics id="Rt" name="Response Time" unit="ms"/>
14 <metrics id="Av" name="Availability" unit="%">
15 <metrics id="St" name="Storage" unit="GB"/>
16 <metrics id="Me" name="Memory" unit="GB"/>
17 <metrics id="IB" name="Incoming Bandwidth" unit="Mbit/s"/>
18 <metrics id="OB" name="Outgoing Bandwidth" unit="Mbit/s"/>
19 </parameters>
20 </services>

```

The second part presents the penalties (lines 10- 11). They are applied in case of SLA violations to compensate Cloud service customer, a violation of the response time SLO implies a penalty equal to \$US 0.1/per violated request the same for all SLOs. For each penalty, a procedure indicates the actor in charge of the violation notification (e.g., provider), the notification method (e.g., email) and the notification period (e.g., 7 days).

```

1 <hasPenalties id="p-Rt" objective="responseTime" description="Rt
  not respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request" notificationPeriod="7
  days"/>
2 <hasPenalties id="p-Av" objective="availability" description="Av
  not respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request" notificationPeriod="7
  days"/>
3 <hasPenalties id="p-St" objective="Storage" description="St not
  respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request" notificationPeriod="7
  days"/>
4 <hasPenalties id="p-Me" objective="Memory" description="Me not
  respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request" notificationPeriod="7
  days"/>
5 <hasPenalties id="p-IB" objective="Incoming Bandwidth"
  description="IB not respected" notificationMethod="e-mail"
  condition="violation" obligation="provider" unit="dollar/request"
  notificationPeriod="7 days"/>
6 <hasPenalties id="p-OB" objective="Outgoing Bandwidth"
  description="OB not respected" notificationMethod="e-mail"
  condition="violation" obligation="provider" unit="dollar/request"
  notificationPeriod="7 days"/>

```

Finally, the customer has the right to terminate the SLA contract with termination fees (20 dollars). He just has to notify the supplier with an email in a time notice of 15 days.

```

1 <terminations id="Ter1" terminationInitiator="consumer"
  terminationType="Voluntary Termination" notificationPeriod="15"
  notificationMethod="e-mail" fees="20 dollars"/>
2 </DSLCloud:SLAContract>

```

- *SLA Establishment evaluation*

- **Testing environment**

The testing environment was based on a real Cloud platform (VMware vSphere 6) which provides efficient virtualization for VM Server. The environment consisted of one virtual machine used for the Web service application that simulates online flight booking services to handle different operations and one virtual machine which simulated consumer requests, to simulate requests the Apache JMeter Web Services load testing framework was used. The resource capacities of the virtual machines are depicted in Table 3.6.

Table 3.6 Resource Capacities of the Physical and Virtual Machines

Instance Type	System	CPU	Memory	Storage
Physical Machine	Linux/ESX	12-core 6-CPU 2.8 GHz Intel Xeon E5420 QC	32 GO	1 TO
Virtual Machine	Windows Server 2012	4 VCPU 2.5 GHz Intel Xeon X3440	8 GO	250 GO

We tested 30 times the real case study described above in order to evaluate the efficiency of the SLA contract establishment approach. We mean by efficiency the ability to generate an SLA contract between the Cloud customer requirement, provider offer through the different transformation rules in less time.

During the experiment, we varied the Cloud customer requests (i.e., different QoS values) for different operations (e.g. book flight, search, delete, list). We constructed 30 tests while varying the customer requirements from 5 to 30 while fixing the number of resources belonging to the Cloud provider. The time of the automatic generation of the SLA contract depends on the execution time of the transformation rules. The computation time needed for the generation of the SLA contract is shown in figure 3.10. From this figure, we can see that while the number of concrete services increases, the growth trend of the computational time of the generation of the SLA contract remains low (20 s).

This experiment show that the generation of the SLA contract between the cloud customer meta-model and cloud provider meta-model thanks to the establishment process using MDE approach is not costly in terms of time. This proves the temporal efficiency of our proposed approach.

In this section, we provided a description about Bunge-Wand-Weber representational theory. We also evaluated the completeness and clarity of our SLA contract establishment for Conceptual Modeling. The results indicated that, our meta-model is capable of achieving a satisfactory completeness and clarity, witch is very promising to accurately describe the structure and behavior of the real world. Finally, we validating our meta-model to examines the ability of the proposed SLA establishment approach to generate the SLA document in less time.

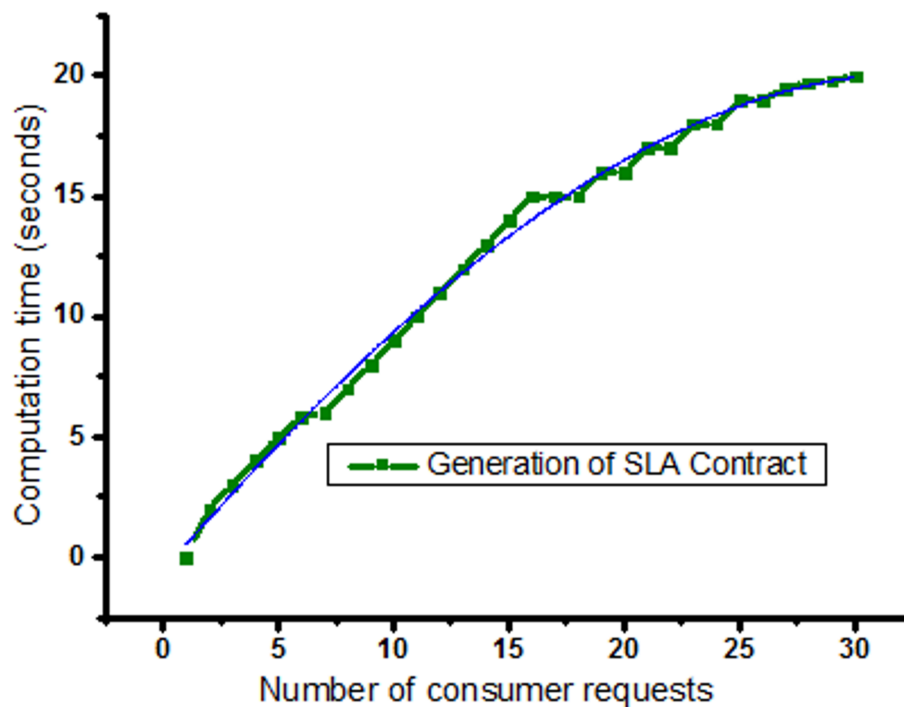


Figure 3.10 Computation time of the generation of the SLA contract when varying the number of customer requirements

3.6 Modeling Cloud SLA Contract Requirements Using Model-Driven Engineering

In the software and systems modeling community, research on domain-specific modeling languages (DSMLs) is focused, especially since the last decade, on providing technologies for developing languages and tools that allow domain experts to develop system solutions efficiently [32]. In this section [110], we present a meta-model to express SLA contract requirements using MDE, as a mean for establishing service level agreements between CSPs and CSCs in the context of a particular service provision. Our approach for SLA establishment embraces MDE principles to automate the SLA contract generation from the customer requirements and provider offers (as shown in 3.11). We propose to use domain specific modeling languages (DSML) to express the SLA contract requirements. Defining a DSML consists in defining an abstract syntax (i.e. the meta-model), a concrete syntax accompanied by a correspondence mapping between them. MDE promotes the usage of models as first class entities. A model is used to represent a given system conforms to a precise meta-model (in our case an SLA Contract [110], see figure 3.12). This latter is just a model that provides the basis for constructing another model. A concrete syntax may be

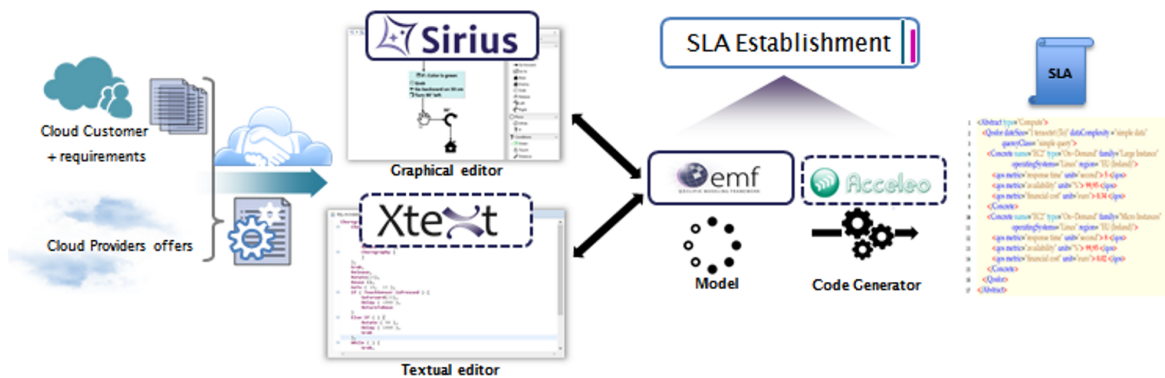


Figure 3.11 SLA Establishment module

textual or graphical. The first one enables users to work with models just as they would work on other text-based programming languages whereas a graphical syntax enables users to work with models using a diagram surface. DSMLs can be used for different purposes (figure 3.13): Editors, Documentation generators, checkers, etc. In this work we use DSML for an Editor purposes to construct a model that we are going to verify its properties to decide the actions to perform, see figure 3.14.

Our proposed meta-model for SLA establishment between a cloud provider and cloud customer in the context of a particular service provision (see figure 3.5), has been influenced by related work in particular [109][83][154][18][82] [97], however, this project adopts a semiautomatic solution where the intervention of an expert is needed. Our approach takes these needs into account by offering modeling languages, transformations and automated processes. Our meta-model describes QoS guarantees in the form of SLO clauses. One of the novelties that it integrates features dealing with QoS uncertainty and cloud fluctuations, such as penalties. This penalties are applied in case of SLA violations, to compensate cloud service customers. Our proposed cloud SLA language we present is a metamodel which enables the definition of SLAs for any type of cloud service. This language is intended for dynamic service provisioning environments. It's based on the Open Cloud Computing Interface (OCCI) and the Cloud Computing Reference Architecture of the National Institute of Standards and Technology (NIST). In our context, we used the Eclipse Modeling Framework (EMF) as a domain specific modeling environment for the definition of meta-models and tools to generate model editors. As we explained in [110], with EMF we were able to define the meta-model of this language relied on existing standards, on the one hand, and on the need to automate the derivation of SLA document, on the other. EMF is a modeling framework and code generation facility for building applications based on a structured data model. In addition, EMF provides the foundation for interoperability with other EMF-based

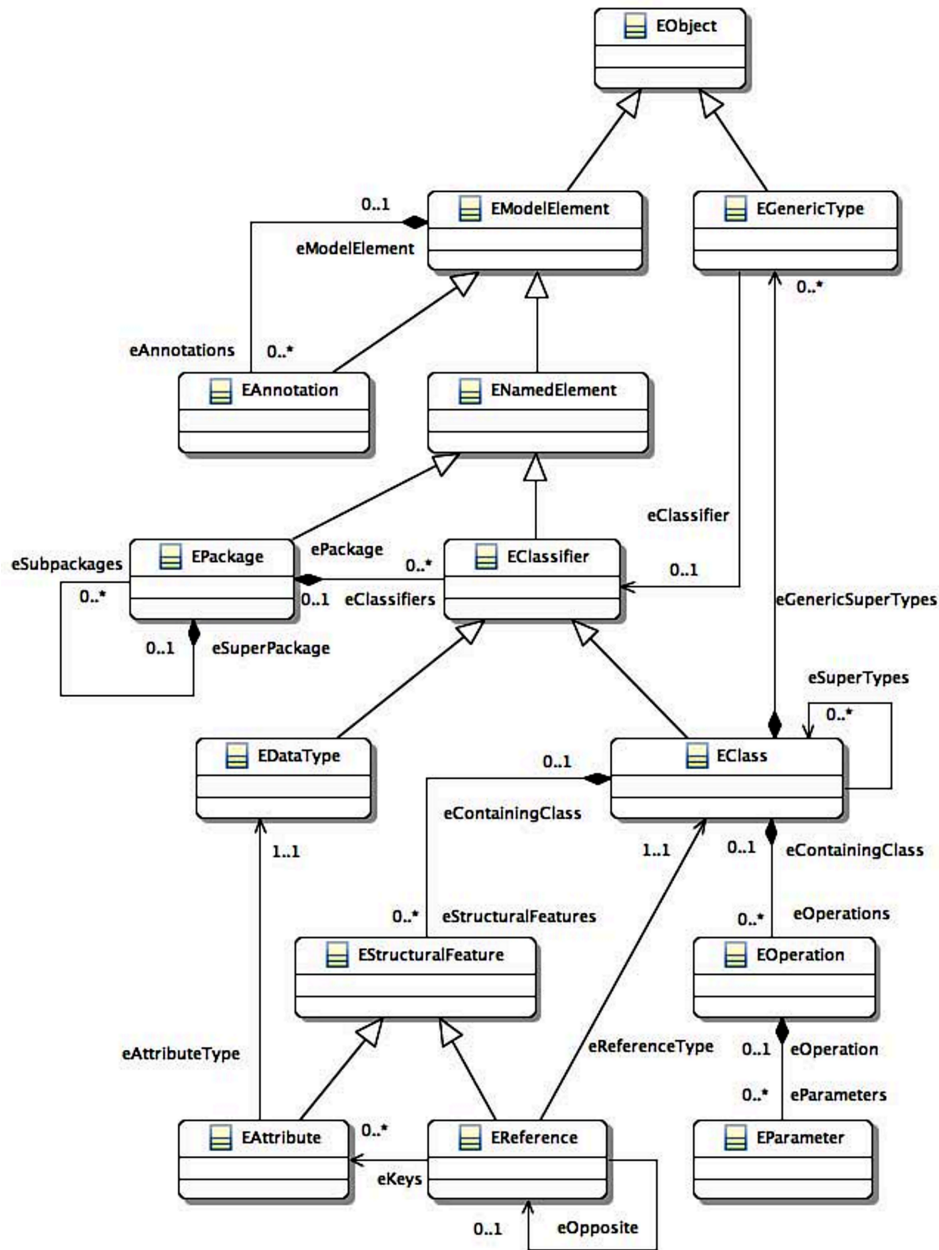


Figure 3.12 Ecore meta-metamodel [66]

tools and applications [164]. It provides an implementation of EMOF (Essential MOF), a subset of MOF, called Ecore. It is based on the ecore meta-meta-model that we construct our meta-model [66]. Once our model has being constructed, a M2T (Model To Text) transformation is performed in order to generate an equivalent of our model in a textual format. For this, we use Acceleo [124], which is an open-source project that aims to provide

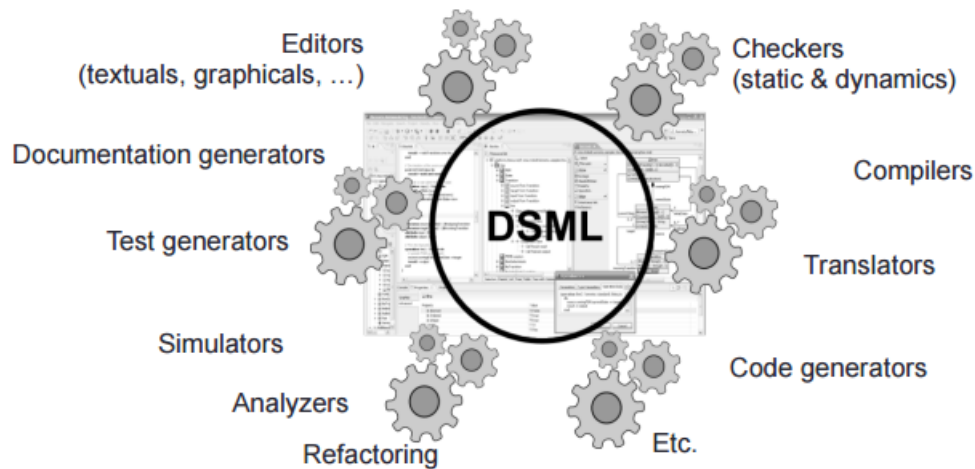


Figure 3.13 Tool examples for a dedicated modeling environment [75]

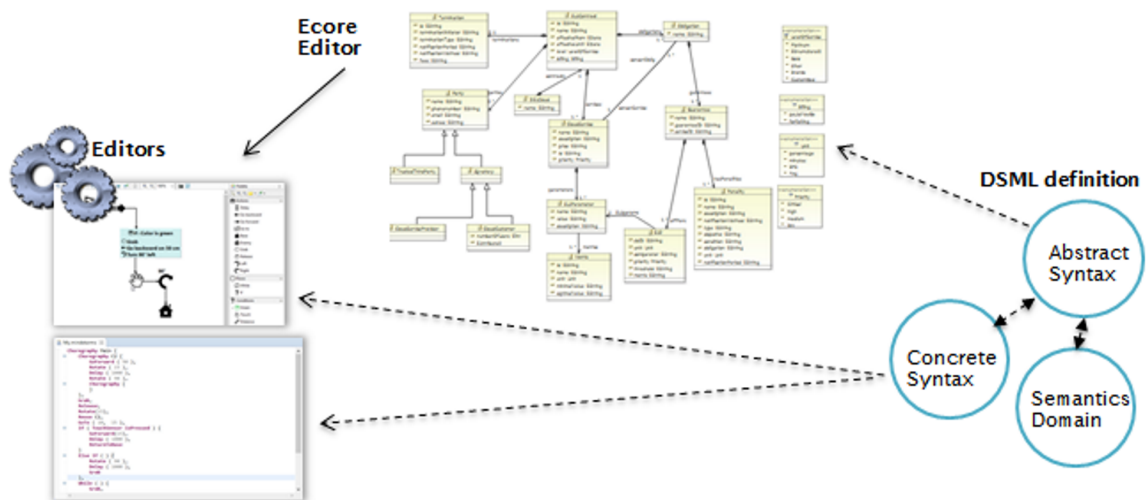


Figure 3.14 DSML : Scientific context

a code generator (transferring models into code) for the people who aim to profit from a Model-driven approach in order to increase their software development productivity.

Our proposed metamodel describes QoS guarantees in the form of SLO clauses. One of the novelties that it integrates features dealing with QoS uncertainty and cloud fluctuations, such as penalty. The penalties are applied in case of SLA violations, to compensate cloud service customers. We model the SLA violation penalty by a linear function as described in the following Chapter 4, section 4.5. EMF offers a set of tools to specify meta-models in Ecore and to generate other representations for instance Java. Figure 3.16 illustrates the package view of the generated code based on our meta-model.

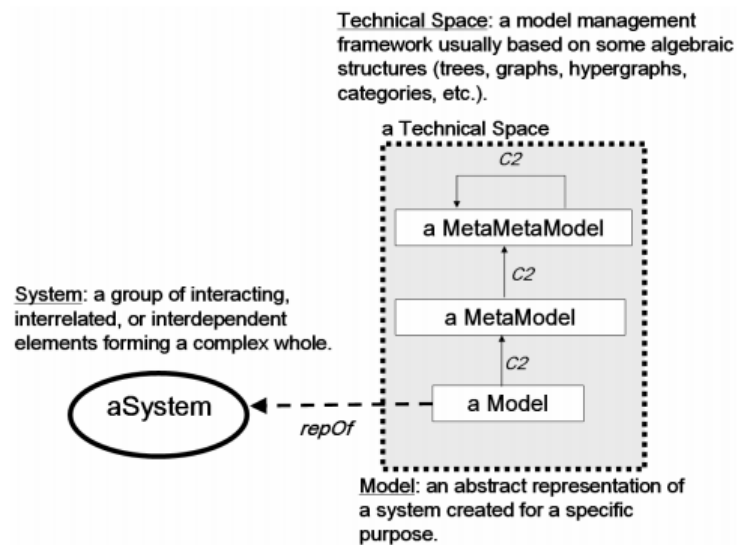


Figure 3.15 Relationships between system, model, metamodel and meta-metamodel

3.7 Chapter Summary

In this chapter, we provided an overview of the existing languages for SLAs definition, and explained how MDE techniques and methods facilitate the establishment of SLA contract requirements between a cloud provider and customer in the context of a particular service provision. The SLA contract meta-model we have defined is built in a way to address the drawbacks that we have found in other approaches. Indeed we have gathered in the same meta-model not only concepts to represent information concerning the SLA contract and the relevant parties but also the concepts that allow us to manage contract variability. Thanks to the model driven engineering based technologies, we can define multiple templates for one model. In other words, we can define from the SLA contract model suitable templates for the supplier and another for the customer and thus meet their needs. We have suggested in this work to combine between two different domains: MDE and cloud computing. Using MDE helped us to overcome the limitations found in the studied cloud computing approaches. We initiated this work by an illustrative example specific to a domain which is travel agency. We provided a description about Bunge-Wand-Weber representational theory. We also evaluated the completeness and clarity of our SLA contract establishment for Conceptual Modeling. The results indicated that, our meta-model is capable of achieving a satisfactory completeness and clarity, which is very promising to accurately describe the structure and behavior of the real world. Finally, we validated our meta-model to examine the ability of the proposed SLA establishment approach to generate the SLA document in less time. This chapter brings

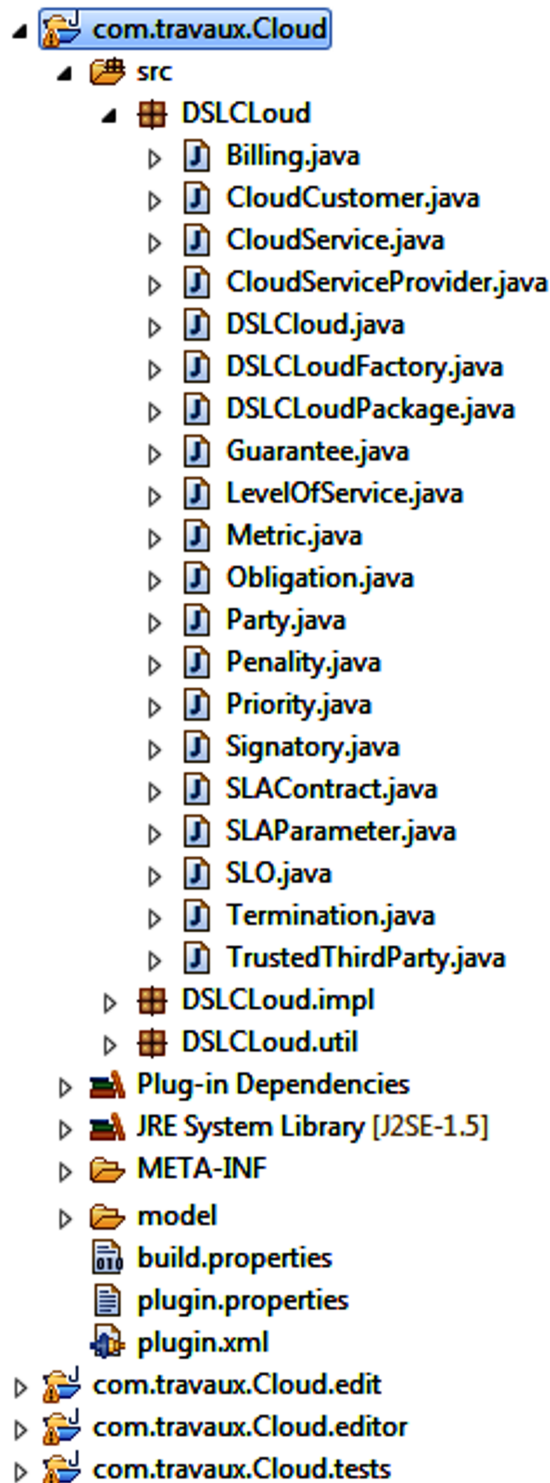


Figure 3.16 The package view of the generated code based on our meta-model

preliminary thoughts opening challenging perspectives. This work remains a key issue to properly solve the trade-off between the benefit of the supplier and the customer satisfaction.

Chapter 4

A Novel Penalty Model for Managing and Applying Penalties in the Cloud Computing

Managing and applying penalties has become a critical issue for cloud computing. In this chapter, we investigate this issue and present the most frequently used definitions of SLA penalty functions. We identify the characteristics of these functions highlighting their strengths and weaknesses. We survey and analyze various penalty calculation and availability calculation methods of cloud providers. Then, we propose a Novel Penalty Model for calculating the penalty cost of the violations, and present a formalization for this penalty definition. We also propose a business model for cloud providers to manage their profit. An example application will be presented to demonstrate the effectiveness of the proposed approach. Finally, the chapter notes some challenges and future research directions. The rest of this chapter is organized as follows: Section 4.2 gives an overview of the background in the field. Section 4.3 survey and analyze various penalty calculation and availability calculation methods of cloud providers. Section 4.4 presents some approaches for definitions of SLA penalty functions. Section 4.5 discusses A Novel Penalty Model for managing and applying penalties in the cloud computing. In Section 4.5.0.1, we propose a business model for cloud providers to manage their profit. Finally, in section 4.6 we give an example scenario to demonstrate the effectiveness of the proposed approach before concluding with future lines of work on the topic.

4.1 Introduction

Cloud computing (CC) environments contain several cloud providers which propose similar cloud services/computing resources and so the consumer has to choose the most suitable provider for his needs [19]. Typically, an SLA is a bilateral binding statement signed between prospective signatories, over the agreed terms and conditions of the given service [162]. SLAs contain Quality of Service properties that must be maintained by a provider, generally defined as a set of Service Level Objectives (SLOs). SLA also includes obligations, details of service pricing and penalties for violations of agreements. These properties need to be measurable and must be monitored during the provision of the service that has been agreed upon in the SLA [114]. An SLA also sets out the remedial action and any penalties that could take effect if performance falls below the promised standard. The violation of such SLAs may involve penalties [115] [110].

Penalty is important to maintaining stable business relationships, and to preserving SLAs as a useful and meaningful instrument for defining such business relationships. Managing and applying penalties has become a critical issue for Cloud Computing. In this chapter we investigate this issue and presents the most frequently used definitions of SLA penalty functions. We identify the characteristics of these functions highlighting their strengths and weaknesses. We survey and analyze various penalty calculation and availability calculation methods of cloud providers. Then we propose a Novel Penalty Model for calculating the penalty cost of the violations, and present a formalization for this penalty definition. We also present a business model for cloud providers to manage their profit. An example application will be presented to demonstrate the effectiveness of the proposed approach. Finally, we note some challenges and future research directions.

4.2 Background

In this section we present a brief overview of violations types. Then, we introduce the concept of penalties and penalty calculation. Finally, we present the principal parameters for Service Level Agreements in the cloud computing.

4.2.1 Types of Violations

One of the main issues that the provider and the consumer will have to agree during the SLA negotiation is the penalty scheme or the sanctionary policies [143]. Since both the service provider and the client are ultimately businesses (rather than consumers), they are free to decide what kind of sanctions they will associate to the various types of SLA breaches,

in accordance with the weight of the parameter that was not fulfilled. According to the Principles of European Contract Law [26], the term unfulfillment is to be interpreted as comprising:

- (1) defective performance (parameter monitored at lower level);
- (2) late performance (service provided at the appropriate level but with unjustified delays);
- (3) no performance (service not provided at all).

Based on these descriptions three broad categories can be defined as follows:

- All-or-nothing provisioning: provisioning of a service meets all the SLOs – i.e., all of the SLO constraints must be satisfied for a successful delivery of a service;
- Partial provisioning: provisioning of a service meets some of the SLOs – i.e., some of the SLO constraints must be satisfied for a successful delivery of a service;
- Weighted Partial provisioning: provision of a service meets SLOs that have a weighting greater than a threshold (identified by the client).

Monitoring can be used to detect whether an SLA has been violated. Typically such violations result in a complete failure – making SLA violations an all-or nothing process. In such an event a completely new SLA needs to be negotiated, possibly with another service provider, which requires additional effort on both the client and the service provider. Based on this all-or-nothing approach, it is necessary for the provider to satisfy all of the SLOs. This equates to a conjunction of SLO terms. An SLA may contain several SLOs, where some SLOs (e.g. at least two CPUs) may be more important than others (e.g. more than 100 MB hard disk space). During the SLA negotiation phase, the importance of the different SLOs may be established. Clients (and service providers) can then react differently according to the importance of the violated SLO. In the WS-Agreement specification [18], the importance of particular terms is captured through the use of a Business Value. Weighted metrics can also be used to ensure a flexible and fair sanctionatory mechanism in case an SLA violation occurs. Thus instead of terminating the SLA altogether it might be possible to renegotiate, i.e., with the same service provider, the part of the SLA that is violated. Again, the more important the violated SLO, the more difficult (if not impossible) it will be to renegotiate (part of) the SLA.

4.2.2 Penalties

Using penalty clauses in SLAs leads to two questions that need to be answered: what types of penalty clauses can be used; and how (if at all) can these be included in SLAs [143]. The focus is on penalty clauses for service providers, since the burden of proof and the interest

in demonstrating that the agreed SLOs have been violated lies on the main beneficiary of the service, that is in the service client. Behind the imposition of any contractual sanctions lies the idea that faulty behavior of a provider should be deterred. As such, it is always possible for the service provider to contest its liability in the unwanted result (SLA breach) and claim that a force majeure situation occurred. Although the situation is impossible to be dealt with through automatic enforcement, monitoring the message exchanges among the provider and the client can give indication whether the SLA violation was the consequence of a misconduct from the provider (either intentional or negligent)[143]. The parties are advised to stipulate either in the SLA or in the associated Collaboration Agreement how they choose to deal with the situation where the provider's faulty behavior cannot be documented, and a force majeure situation did occur. A penalty clause in an SLA may consist of the following:

- a decrease in the agreed payment for using the service, i.e., a direct financial sanction;
- a reduction in price along with additional compensation for subsequent interaction;
- a reduction in usage of service from the provider;
- a decrease of the reputation of the provider.

During the negotiation phase, client and provider can agree on a direct financial sanction. Usually, the amount to be paid depends on the value of the loss suffered by the client through the violation (which should cover entirely) and if agreed, a fix sum of money that has to be paid as fine for the unwanted behavior. Due to the difficulties in proving and documenting the financial value of the loss, during the negotiation phase the parties may choose an agreed payment for non performance that is a fixed sum of money that will have to be paid upon nonperformance, regardless of the fact that no financial loss was suffered by the client. The service provider can deposit the negotiated fine at a trusted third party, that acts as a mediator, before the service provision commences. On successful completion of the service provision (based on the SLA) the TTP returns the deposit to the service provider, otherwise the client receives the deposit as compensation for the SLA violation. Notice that a trusted monitor is required for this, as a client can never prove by itself that an SLA was (partially) violated. However, for this to work properly – and especially automatically – some kind of micro payment [70] system is required – such as Paypal. Another possibility is that a client reduces its usage of services from a provider that violated an SLA. If the economic position of the client is strong enough this can be a valid strategy.

Another kind of penalty clause can lead to a change in the reputation of a provider [80][146]. In such a system the reputation of service providers that violate SLAs will drop. In this case special care needs to be taken that the reputation of a service provider is correctly

determined. Both reputation building, using dummy clients that praise a service provider, and slandering reputations, where dummy clients (unjustly) complain about a service provider, form serious threats in reputation based systems. In the negotiation phase of the SLA service provider and client can agree on which reputation mechanism to use.

4.2.3 Penalty Calculation

When the cloud services can't meet availability commitment in the SLA, providers would pay back service credits to make up for their violation[180]. We list three important items when calculating the penalty.

- **Calculation Method** : There are three methods to count the penalty, all of which are based on different violation level (i.e. unavailability rate or downtime): 1. Method *Ratio of Total Charge*: A certain percentage of total charges paid by consumers in the measurement period based on different violation level; 2. Method *Fixed Value*: fixed value at different violation levels; 3. Method *Ratio of Downtime*: A certain ratio of downtime, usually greater than 1;
- **Penalty Cap** : It is the maximal value of penalty that providers pay back if violation. The method of *Ratio of Total Charge* and *Fixed Value* have inherently penalty cap, while the method of *Ratio of Downtime* will set up a maximal penalty manually;
- **Payment Method** : Most cloud providers claim the penalty can only be applied to pay for future payments in the same service. No real dollar is paid back to consumers accounts.

Besides, cloud providers claim duty of reporting violation belongs to cloud consumers. Consumers are the reporting onus and have to detect violation themselves. They should report server logs or monitor systems within a time restriction according to current SLAs. Though providers monitor their services themselves and may be aware of violation, they won't make compensation without consumers reports.

4.2.4 Important Parameters for Service Level Agreements in Cloud Computing

In this section, we aim at finding an initial set of parameters that should be present in a Service Level Agreement to guarantee the Quality of Service the cloud consumer is minimally expecting. We do that by reviewing the concept of functional and non-functional requirements (NFRs)[174] in cloud computing, as shown in table 4.1. Some parameters that are important

for consumers are left out and others are very vaguely described. We try to find out an initial set of SLA parameters necessary for cloud computing. SLA, as defined before, is a formal contract that includes a set of QoS metrics in addition to policies concerns, conditions, rules, regulations and penalties for exceeding the expectations. We gathered all possible parameters and QoS metrics from multiple resources which can be included and negotiated in any SLA.

The SLA parameters are specified by metrics. These metrics define how cloud service parameters can be measured and specify values of measurable parameters. In the cloud computing architecture, there are three types of services which providers can present to consumers. These services are: infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). The description and the values of these parameters will vary from one cloud service to another.

4.3 A Survey of Penalty Calculation in Cloud SLA

This section survey and analyze various penalty calculation and availability calculation methods of cloud providers [180]. The results are presented in Table 4.2 and 4.3.

4.3.1 Comparison of Availability Calculation

In this subsection, we present the Availability Calculation service, are listed as follows :

4.3.1.1 Availability Commitment

Committed availability among these cloud SLAs ranges from 99.9% to 100%, which means no more than 43.2-minute downtime per month. Five of eight cloud SLAs reach up to more than 99.95%, which is 21.6 minute downtime per month. However, considering the measurement period, time granularity and exclusions, not all availability commitments are as high as expected. We will discuss it in the following paragraphs [180].

4.3.1.2 Measurement Period

Most clouds define a calendar month or a billing month as their Measurement Period [113]. It is reasonable to set a short period to gain a stringent availability commitment. IBM Softlayer has a one-billing-year measurement period. This period is long and inappropriate for two reasons: Firstly, consumers can't get timely penalty payment in such a long term, which will largely influence consumers economic interests. Another reason is the long period can't meet the flexibility and pay-as-you-go manner of current clouds[180]. We also prefer billing

Table 4.1 Important parameters for Cloud Services

Parameters	Description	Metrics
Availability	The amount of time the system is operational and accessible when required for use. In cases of down-time service providers pay penalties [128].	Response time
		Accuracy
		Flexibility
Scalability	Is the ability of services to function properly when the system needs to be changed in size[128]. Scalability can follow through horizontal scaling, vertical scaling or stateless services.	Average of assigned resources among the requested resources
Reliability	Is the power of a service to remain functional with time without malfunction[20]. The capability of the service to keep on operating with a certain degree of efficiency timely.	Service Constancy
		Fault Tolerance
		Recoverability
Efficiency	The resources employed for services when providing the wanted operation and also therefore the amount of efficiency, underneath settled situation.	Utilization of Resource
		Ratio of waiting time
		Time behaviour
Reusability	Known as the level to which a application component or even other work system may be used in a number of program or application[20].	Readability
		Coverage of variability
		Publicity
Modifiability	Is the ability to make changes to the service efficiently and with low costs[62].	MTTC (Mean Time To Change)
Maintainability	The ability to maintain the service.	Maintenance cost
		Prediction of failures
		Downtimes for maintenance
Interoperability	Is the ability of communication entities to share specific information and operate on it according to an agreed-upon operational semantics. [31].	Workload migration
		User authentication
		Data migration
Price	In cloud computing is a unit price per hour for usage of the service. The "Pay as you go" concept[20].	Pay as you go
		Cost over a Fixed Time (\$/year)
		Billing
Security	Unauthorized access to the service should not be allowed. By that the service provider can guarantee confidentiality of the consumer's data.	Authenticity - Integrity Auditing - Integrity
		Confidentiality
		Privacy

month to calendar month, because billing-month measurement period is closely related with penalty payment and renewal of services.

4.3.1.3 Service Granularity

Virtual Machine is the most common service granularity. This indicates whichever running instance is down, the situation will be considered as unavailable and counted into downtime. However, Amazon EC2 and Google GCE affirm their services are down only if all running instances are unavailable in different zones. Windows Azure has a looser criterion, which only needs more than one VM unavailable. Because some failures related with hypervisor or operating system may affect a single VM rather than a rack or a whole zone, a looser criterion implies a higher service quality. Rackspace, IBM Softlayer, Salesforce and SAP HANA Cloud haven't defined their service granularity. They make a commitment of the availability in infrastructure facilities (e.g. network, power, host, HVAC, etc.). Rackspace also ensures the availability of hypervisor, which is included in the availability of hosts. However, these commitments are indirect and immeasurable for consumers. Consumers have no idea of what incident happens with provider's facilities except their own instances [113]. It's unreasonable to commit some items that can't be measured by consumers, especially when consumers have the responsibility to report the SLA violation.

4.3.1.4 Time Granularity

Time granularity ranges from 1 minute to 1 hour. A greater time granularity means a commitment with less actual availability rate, because it will neglect a small period of downtime. However, Rackspace and IBM Softlayer have the 2nd longest time granularity, which is NC. This means their 100% availability is not actual. When downtime comes to 29 minutes, it won't be counted into downtime or compensated eventually. If we define one minute as a standard time granularity, their actual availability commitment is 99.93% (1-30min. /30days), which means their commitments are different from what they guarantee.

4.3.1.5 Coverage

The indicator of instance availability. In most cases, cloud providers view availability equivalent to internet connectivity of a running instance. But in practice, internet connectivity of instance is just one of prerequisites for instance availability [113]. Moreover, if we focus on the whole cloud services, not only the running instances, but other relevant services like cloud management (e.g. using dashboard to create, destroy instances) should also be taken into availability.

4.3.1.6 Exclusions

All cloud SLAs excludes period of scheduled maintenance and upgrade from availability calculation. ITIL offers a practical calculation of Availability:

$$availability = \frac{AgreedServiceTime(AST) - Downtime}{AgreedServiceTime}$$

In this equation, ITIL emphasizes the Agreed Service Time in contrast to Service Time. This period affects the actual availability of cloud services. During the scheduled maintenance and upgrade, instances are unavailable to consumers, so actual availability commitment is lower than the claimed one in every current cloud SLAs. The gap between actual and committed availability depends on the frequency and duration of maintenance for each cloud. Even with a warning, consumers could suffer loss of business from the maintenance and upgrade [180]. IBM Softlayer takes methods to make up their loss by guaranteeing that hardware replacement and upgrade of its cloud servers shall be completed within 2 hours. Most cloud SLAs also exclude some incidents outside of providers reasonable control, including force majeure, social incidents and external network failure. For instance, external network failures which bring enormous loss to consumers are partially under providers control. Cloud providers can have a few alternatives on external network, for example, selecting a high-quality ISP, or building a reliable network themselves or preparing a backup network for emergency. Moreover, external network failure will bring enormous loss to consumers and Internet attack is a similar issue. Cloud providers should not be blamed for internet attack, but it's better for them to undertake responsibility for preventing cloud servers from hackers attacks.

4.3.2 Comparison of Penalty Calculation

We investigate SLA penalty calculation service and violation reporting in (Table 4.3). Firstly, we measure the penalty paid by cloud providers when violation occurs. Because all cloud providers apply the penalty payment only to future payment for their services, we calculate the payment in the form of service credit percentage. Here we define penalty degree as rate of penalty cloud provider should pay to consumers according to availability rate, which is largely different among various cloud providers. Some providers won't pay much for a quite long period of downtime (SAP HANA Cloud, GoGrid), while some pay almost all income for penalty (Salesforce, GoGrid). SAP HANA Cloud pays more than its income, which is viewed as a punitive compensation.

Secondly, we also notice that all cloud providers claim that penalty can only be paid for future service which isn't met the providers commitments, and the penalty (service credit) can't be applied to any other services.

Table 4.2 Availability Calculation [180]

Cloud Provider	Availability Commitment	Measurement Period	Service Granularity	Time Granularity	Coverage	Exclusions
Amazon EC2	99.95%	Billing month	All VMs in more than one Availability Zone (same Region)	1 min	Network connection	(1)(3)(4)
Windows Azure	99.95%	Billing month	At least 2 VMs in the same Availability Set	1 min	Network connection	(3)(4)
Rackspace	100%	Billing month	N/A	30 min, NC	Network connection, consumer portal, host (including hypervisor), power, HVAC	(1)(2)(8)
Google Compute Engine	99.95%	Billing month	All VMs in more than one zones	5 min, NC	Network connection, persistent disk access	(1)
IBM SoftLayer	100%	Billing year	N/A	30 min, NC	Network connection, consumer portal, power, HVAC	(1)(2)(3)
GoGrid	100%	Billing month	VM	15 min, NC	Hardware and hypervisor layers	(1)(3)(8)(9)
Salesforce	99.97%	Billing month	N/A	5 min, NC	Network connection, power, HVAC	(1)(4)(5)(9)
SAP HANA Cloud	99.96%	Billing month	N/A	20 min, NC	Hardware and hypervisor layers	(3)(4)(5)

(1) scheduled maintenance (2) scheduled upgrade (3) force majeure (4) external network failure (5) internet attack (6) host migration

(7) social incident (e.g. war, political turmoil, administrative action, etc.) (8) emergency maintenance (unforeseen and needed to ensure security or reliability) (9) consumer portal

NC: if less than the time granularity, the period shall not be counted as downtime

Thirdly, though consumers may not be the best part to report violation, most international cloud providers require consumers to be the onus to violation report. Meanwhile, all of the surveyed cloud providers haven't defined clearly the onus to report violation yet, which will raise the problem of unclear responsibility. Besides, some providers would punish those consumers who report a false violation (e.g. IBM Softlayer) [180].

4.4 Approaches for definitions of SLA penalty functions

Currently, there are various ways to express penalties, some simpler (e.g. a flat rate for the entire SLA, or a linear proportionality per guarantee) and others more complex. In this section, we present an overview of some approaches for definitions of SLA penalty functions. These various approaches, however, do not satisfy all of the requirements for formulating complex penalty expressions in a single unambiguous model.

In [21], Becker et al. propose a price function over achieved QoS. Subtracting from the agreed price provides the penalty, so the price function can also be considered to be the penalty function. Rewards are also possible using their approach. The main drawback, it is not possible to express penalties that involve more than one QoS property or these properties being correlated. Jurca and Faltings [78] suggest a method for calculating penalties based on a reputation mechanism, where all customers evaluate the quality of the service they received. The authors take measures to avoid false voting for price reduction or other fraud. This approach, however, can only be combined with classes of service.

In [143], Rana et al. discuss monitoring and reputation mechanisms for SLAs. The authors look into EU contract law, taking relevant points into account, and then define three broad penalty categories: all-or-nothing, partial and weighted partial. However, a complete mathematical model is not present. Kosinski et al. present in [46] a mathematical formulation of penalty functions applied specifically to networking. The paper does capture the relationships between different properties, and policies are defined depending on such relationships and their respective combinations. These policies are captured within 'subcontracts' (Kosinski et al. use this term to refer to sections of a single SLA). Kosinski et al. calculate failures using a pre-specified taxonomy (number of violations, amount of violations, etc).

In [154], Serrano et al. model the SLA violation penalty as a linear function depending on the penalty rate and delay time. Finally, in [54][53][55] Emeakaroha et al. present the utility function that considers on the one hand the provider profit and on the other hand the cost associated with the effort of detecting SLA violations and the penalty cost of the violations.

Table 4.3 Penalty Calculation

Cloud Provider	Penalty Account	Calculation Methods	Penalty Cap	Payment Method	Report onus	report time restriction
Amazon EC2	<99.95% 10% <99% 30%	Ratio of Total Charge	N/A	Future payment	Consumer	N/A
Windows Azure	<99.95% 10% <99% 25%	Ratio of Total Charge	N/A	Future payment	Consumer	N/A
Rackspace	1. Data Plane: 5% of fees for each 30 minute downtime 2. Control Plane: <99.9% 10% <99.5% 20% <99% 30%	1. Ratio of Downtime 2. Ratio of Total Charge	100%	Future payment	Consumer	Within 30 days
Google Compute Engine	<99.95% 10% <99% 25% <95% 50%	Ratio of Total Charge	50%	Future payment	Consumer	N/A
IBM Soft-Layer	5% of the fees for each 30 minute downtime	Ratio of Downtime	N/A	Future payment	Consumer	Within 7 days
GoGrid	100*downtime	Ratio of Downtime	100%	N/A	N/A	N/A
Salesforce	10*downtime	Ratio of Downtime	100%	Future payment	Consumer	Within 2 days
SAP HANA Cloud	<99.92% 10% <99% 25% <95% 50%	Ratio of Total Charge	50%	Future payment	Consumer	Within 20 days

4.5 A Novel Penalty Function for Managing and applying penalties in cloud computing

Based on the approaches previously studied and mentioned in [21][78][143] [46][154][54][110], we propose a novel approach for calculating the penalty cost of the violations in a time interval, based on the following concepts:

- The global penalty is the sum of the penalties associated to each SLO (metrics). Each SLO penalty fee is evaluated according to its priority;
- Each penalty fee varies depending on the magnitude of the violation;
- The penalty fee shall not exceed a certain ceiling. For example we can consider a ceiling 10 % of the total amount of the service;

The penalty cost can be formulated using the following equation:

$$P(t) = \min(\text{ceiling}, \sum_{i=1}^m \sum_{j=1}^n P_i(V_j)) \quad (4.1)$$

Knowing that:

$$P_i(V_j) = V_j \cdot P_i \quad (4.2)$$

Equation (1) becomes :

$$P(t) = \min(\text{ceiling}, \sum_{i=1}^m \sum_{j=1}^n V_j \cdot P_i) \quad (4.3)$$

Where :

- $P(t)$: the global penalty in a time interval (t);
- m : present de number of SLOs parameters (metrics);
- n : present de number of violations in t;
- P_i : the penalty associated to an SLO parameter i;
- V_j : violation ratio;
- ceiling : the maximum penalty fee.

4.5.0.1 A business model for cloud providers and consumers

We propose a business model to manage more finely the violations of the SLA contract. In what follows, we explain, the profit of the service provider. The supply of a service generates a revenue, which is based on a expense. The profit of the provider is defined by the equation :

$$Profit = income - expense$$

The *income* is calculated by equation :

$$income = \sum_{i=c}^C PrServ_c$$

Where C is the number of service customers and $PrServ_c$ is the amount paid by the customer c .

The expense of a service is composed by the expenses of allocated *resources*, paid *penalties* and miscellaneous expenses (*Others*) related to the administration of the service [110].

$$expense = Resources + Penalties + Others$$

The cost of resources depends on the resource type and allocation duration. To calculate the penalties associated with a service, we propose the following equation :

$$Penalties = \sum_{i=p}^P Penalty_p$$

Where P is the total number of violations, $Penalty_p$ is the penalty cost p .

In addition to the resources costs and penalties, we accumulate an additional cost (*Others*) related to the administration service. It is calculated using the equation :

$$Others = \sum_{i=o}^O others_o$$

Where O is the number of various additional expenses and *others* is the associated cost to each expense.

4.6 Example Application

This section describes a possible scenario of a Cloud application between a SaaS provider and his customer so as to ease the understanding of the proposed approach. In this case study, an online bookstore established an SLA contract with a web hosting provider like Amazon EC2 to deploy its online bookstore service.

We assume that the SLA negotiation and establishment have been performed. After the definition of an SLA template that contains all possible QoS attributes that should be

Table 4.4 SLA between Amazon EC2 and online bookstore

Parameter (SLO)	Optimal Threshold	Minimal Thre.	Penalty
Response time	500 ms	570 ms	0.2\$
Availability	99.9%	97%	
Storage	300 GB	290 GB	
Memory	6 GB	5 GB	
Incoming Bandwidth	50 Mbit/s	48 Mbit/s	
Outgoing Bandwidth	100 Mbit/s	98 Mbit/s	

negotiated and included in the final SLA. Upon contract violations, penalties should be paid back to clients as compensation.

The SaaS provider should guarantee an average response time (Rt) less than or equal to 500ms (present the optimal value) a margin of 70 milliseconds is acceptable as a QoS degradation. Regarding the service availability, the SLA states that at least 99.9% of availability (Av) should be guaranteed, the minimal value of Av must be accepted at 97%. This SLA will enter into effect on first July 2015 (the Effective Date) for a defined period of two years. The other QoS parameters for web hosting services that should be included are described in table 4.4, highlighting their optimal and minimal threshold. These objectives should be achieved every working day between 7am and 10pm during the validity of the contract. In this example, six SLOs are proposed : a response time, availability, storage, memory, incoming bandwidth and outgoing bandwidth.

To give a better understanding of our approach, we use the dataset described in table 4.5 and shown in figures 5.12 and 4.2. The sampling frequency is of one sample for every 5 minutes. We use a 15-minutes interval, from 8h10 to 8h25, as the reference measurement interval to capture all SLA violations for the proposed scenario. We consider a maximum penalty fee of 2\$US/per time interval. The penalties are applied in case of SLA violations to compensate cloud service customer.

Violation of SLO implies penalty cost calculated according to equation (3) presented before, as illustrated in table 5.5. The values of the violation ratio obtained in table 5.5 are calculated as follows :

$$Violationratio = \frac{Violationdetected - Threshold}{Threshold}$$

- 600 ms - 570/570 ms= 0,05
- 580 ms - 570/570 ms= 0,01

Table 4.5 Used dataset for SLA violation

Intervals	Availability (in %)	Response Times (ms)
08h00	99%	Rt 500ms
08h05	98%	Rt 500ms
08h10	95%	Rt 530ms
08h15	96%	Rt 600ms
08h20	97%	Rt 530ms
08h25	98%	Rt 580ms
08h30	99.9%	Rt 530ms

Table 4.6 Calculated penalty Cost

SLA parameter	Violations detected	Violation Ratio	SLO Penalty	Total penalty cost
Reponse time	600 ms 580 ms	0,05 0,01	0,012\$	0,018\$
Availability	95% 96%	0,02 0,01	0,006\$	

- $|95\% - 97/97\%| = 0,02$
- $|96\% - 97/97\%| = 0,01$

In the other hand, we calculated SLO Penalty using the following formula:

SLO Penalty = The penalty associated to an SLO parameter i * Violation ratio

- $(0,05 * 0,2\$) + (0,01 * 0,2\$) = 0,012\$$
- $(0,02 * 0,2\$) + (0,01 * 0,2\$) = 0,006\$$

We use XML as a representation format. The following XML (see Listing 1) presents an example of a SLA file describing the penalties for SLA between the fore-mentioned provider and customer concerning the service SaaS. For each penalty, a procedure indicates the actor in charge of the violation notification (e.g., provider), the notification method (e.g., email) and the notification period (e.g., 7 days).

```
1 <penalty id="p-Rt" objective="responseTime" description="Rt not
  respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request"
  notificationPeriod="7 days"/>
```

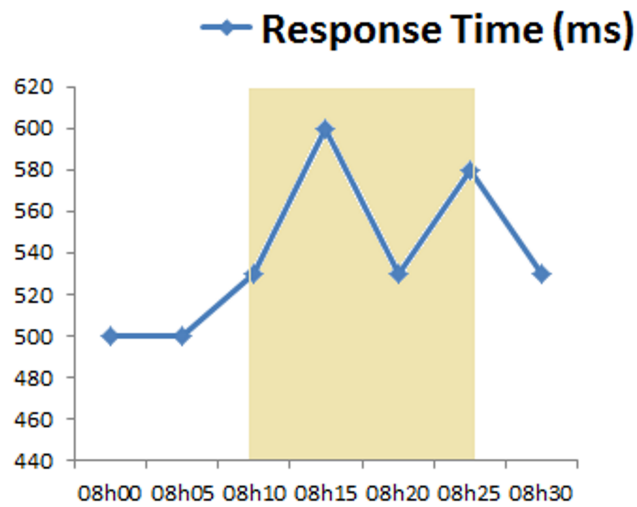


Figure 4.1 Detected response time values

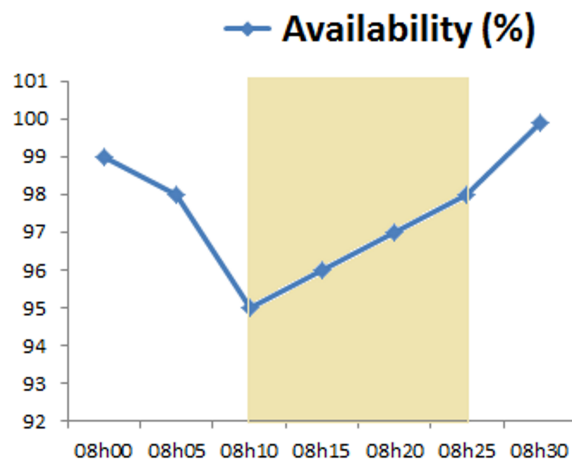


Figure 4.2 Detected availability values

```

2 <penalty id="p-Av" objective="availability" description="Av not
  respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request"
  notificationPeriod="7 days"/>
3 <penalty id="p-St" objective="Storage" description="St not
  respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request"
  notificationPeriod="7 days"/>
4 <penalty id="p-Me" objective="Memory" description="Me not
  respected" notificationMethod="e-mail" condition="violation"
  obligation="provider" unit="dollar/request"
  notificationPeriod="7 days"/>

```



```
5 <penalty id="p-IB" objective="Incoming Bandwidth" description="IB  
  not respected" notificationMethod="e-mail"  
  condition="violation" obligation="provider"  
  unit="dollar/request" notificationPeriod="7 days"/>  
6 <penalty id="p-OB" objective="Outgoing Bandwidth" description="OB  
  not respected" notificationMethod="e-mail"  
  condition="violation" obligation="provider"  
  unit="dollar/request" notificationPeriod="7 days"/>
```

4.7 Chapter Summary

In this chapter, we presented an overview of some approaches for defining SLA penalty functions. We surveyed and analyzed various penalty calculation and availability calculation methods of cloud providers. Then we proposed A Novel Penalty Model for calculating the penalty cost of the violations, and presented a formalization for this penalty definition. We also proposed a business model for cloud providers to manage their profit. An example application was demonstrated to validate the effectiveness of the proposed approach.

Chapter 5

An Autonomic SLA Monitoring Framework managed by trusted third party in the Cloud Computing

Currently, Service Level Agreements (SLAs) assurance forms one of the major challenges for Cloud Computing (CC) in order to guarantee Quality of Service (QoS) in real-time and control SLA violations. However, due to the highly dynamic nature of this open environment, it is important to have a binding agreement between all the service parties for ensuring trust while fulfilling the expected QoS. To properly operate and manage such complex situations, an effective and efficient monitoring is crucial. The participation of a trusted third party (TTP) is necessary in order to resolve conflicts between involved parties. In this chapter, we propose an autonomic SLA monitoring framework managed by TTP composed of two modules: the first one SLA establishment module, which aims at providing support for automated SLA generation and management. The second one, a service monitoring module to dynamically monitor QoS metrics by detecting SLA violations at runtime to verify compliances for the respective SLAs, and to propose a mechanism for an adaptive remedy rectification, as a contribution at the third maturity level of the autonomic computing paradigm as defined by IBM. The framework is validated with scenarios on response time and availability, the results obtained are promising. They confirm that our framework manages SLAs in an efficient way, as it detects all violations to be communicated to concerned parties, and identifies particular penalty clauses that can be used to modify the reputation of a provider over time. The TTP framework equipped with such reputation module can provide real-time assessment for consumers informed decision making to continue using a service or to migrate to another service provider in the case of service degradation. This creates a fair competitiveness between providers and hence improves service performance and the reliability in the cloud.

The rest of this chapter is organized as follows: Section 5.2 discusses the related works. Section 5.3 describes the design of our autonomic SLA monitoring framework and we detail its components. In section 5.4, we discuss the experimental evaluations by given an example scenario to demonstrate the effectiveness of the proposed framework. Finally, the chapter is concluded with a summary of our contributions and their future potential extensions.

5.1 Introduction

The revolutionary technology of Cloud Computing offers a scalable and flexible paradigm where infrastructure, platform, and software are offered to users in the form of services. The management of such environment is inherently complex due to the large-scale number and heterogeneity of resources and the increasing number and types of services a cloud must deliver over the Internet. Combined with a rapidly increasing number of cloud-based services, this complexity generated a pressing request for effective and efficient monitoring solutions. The provisioning of these computing services by cloud providers are regulated by Service Level Agreements [115], which present an important element that provides some degree of assurance. As of now, the differentiating elements between CC solutions are Quality-of-Service and the SLAs guarantee provided by the service providers. Any SLA mainly describes two things: the different Service Level Objectives (SLO) in terms of values for QoS metrics and the penalties to be applied if the objectives have not been accomplished [110]. These properties need to be measurable and must be monitored during the provision of the service that has been agreed upon in the SLA [114].

Within the CC concept, SLA monitoring is a task of paramount importance for both cloud service provider (CSP) to avoid penalties if the SLA terms are violated and for cloud service consumer (CSC) to be aware of the status and quality of their running services. Moreover, a violation of an SLA may cause a cascading effect on the dependent services. Thus, may be affecting the overall composition and degrading the overall system performance. For that reason a trusted third party monitoring services is required for this, as a consumer can never prove by itself that an SLA was (partially) or totally violated. The participation of a TTP is necessary in order to resolve conflicts between prospective signatories, likewise to monitor SLA violations in real-time in the goal to ensure online monitoring cloud services and provide better than best-effort behavior for clouds. We argue that establishing and monitoring SLA violations in real-time has become a critical issue for CC. This raises the following questions: (i) How to describe the SLA terms between CSP and CSC, such as service levels, penalties in case of SLA violation, etc. (ii) How to provide guarantees on cloud QoS monitoring and

detection of SLA violation ? How to create a fair competitiveness between providers and hence improve service performance and the reliability in the cloud ?

To address the above challenges, we propose an autonomic SLA monitoring framework for real time ensuring accountability and QoS in cloud services. The main functional components of this framework managed by a trusted third party, are two modules: (1) SLA establishing; and (2) dynamic service monitoring module. The first module comprises the SLA establishment process aspect to express SLA contract requirements using Model Driven Engineering (MDE) to automate the SLA document generation [110], it is intended to provide agreed SLAs to the second part, which comprises the service monitoring aspect. This monitoring module is responsible for managing and monitoring the QoS by detecting SLA violations at runtime and proposing a mechanism for an adaptive remedy rectification. This module can efficiently manage QoS by collecting data of all the available resources in the cloud environment based on the established and agreed SLAs. It has also a knowledge database that stores this information which is required by the other components in the framework. Dynamic SLAs monitoring is actually an instance of autonomic control cycle: monitoring, analysis, planning, execution whereby systems can monitor themselves and maintain a target behavior. The vision explored in this module is to enrich the CC specification, with the aim to provide it with autonomic and dynamic QoS-monitoring capabilities using Complex Event Processing (CEP). In this aspect, following the IBM Autonomic Computing (AC) paradigm which aims at equipping computer systems with capabilities to autonomously adapt their behavior and structure according to dynamic operating conditions. AC is inspired by the functions of the human nervous system and is aimed at designing and building systems that are self-managing. Autonomic systems are being adopted for self-management of complex large scale distributed system that have become manually unmanageable.

The method of penalty cost calculation, detection of violations, reputation score is specified in our autonomic SLA monitoring architecture. In the case of violations, the penalty mechanism is automatically activated to calculate penalty cost and decline reputation. Furthermore, the reputation of CSP concerned is affected. The new reputation value that will be assigned to the cloud provider is calculated by our trust reputation model.

This work focuses on identifying particular penalty clauses that can be associated with an SLA, and how such penalty clauses may be used to modify the reputation of a provider over time. The results of this reputation scores generated by the TTP is sent to consumers with a detailed dashboard that contain 'Real time' reporting with Alarm list (violations, penalties, value of reputation). The contributions of this paper are threefold and focused on a maturity level that may be considered between the fourth (adaptive) and the fifth (Autonomic) levels and can be summarized as follows :

- (i) the first consists in the proposition and describing implementation choices of our functional architecture model for a monitoring component allowing metrics observation and real-time violation detection related to QoS degradation;
- (ii) the second contribution, consists on proposing a novel trust reputation model;
- (iii) the last one, consists in a proof of concept through a real case study of the benefits that could be performed from the monitoring component through the dynamic and autonomous activation of our mechanism depending on the detected violation, whereby appropriate actions are taken, to prevent/correct SLA violation situations and eventually applicate penalties and decrease reputation.

5.2 Related works

To reveal our motivations and contributions, we find it useful to review some research works in cloud monitoring systems, frameworks and architectures. Afterward, we give a global view of research initiated in Trust Reputation Systems. Finally, we discuss some approaches about Trusted third party for monitoring Service Level Agreements in real time.

5.2.1 Review of current research efforts in Cloud monitoring systems and architectures : Performance Monitoring and Measurement

Currently, there are various recent research and industry efforts have focused on developing monitoring techniques, platforms, and frameworks that can assist cloud parties in tracking the SLA violations of certain service quality requirements. In this section, we selected and presented the most relevant works on monitoring systems and architectures. A few studies deal with monitoring in cloud computing environment.

In view of this, Grati et al. [65] presented a Business Process Execution Language, BPEL-based monitoring framework for monitoring Web services in the Cloud environment. This framework collects information from the Cloud, analyzes the information and then takes corrective actions when SLA violations are detected. The framework's run-time monitoring is based on workflow patterns as composed in BPEL. However, this framework is limited to monitoring the response time QoS requirement only.

Another study in [121] presented a framework that combines the advantages of client- and server-side QoS monitoring. The framework uses event processing to inform users the current QoS values and possible violations of the SLA. These events then trigger adaptive behavior such as hosting new service instances if the QoS is not as desired. However, to automatically react to SLA violations, such as deploying new service instances on-the-fly or

dynamically increase certain virtual machine capabilities are yet to be implemented in the future.

In [154], Serrano et al. introduced a SLA-aware-Service (SLAaaS) Cloud model that integrates QoS and SLA into the Cloud services. Their experiments on online Cloud services through various case studies have successfully demonstrated the capability of the model to provide request response time, availability, resource usage and resource cost guarantees. Other metrics such as service throughput and energetic cost are yet to be evaluated in the future.

A cloud application SLA violation detection model, CASViD, is proposed in [54]. This model uses resource allocation, scheduling, and deployment tools to monitor and detect SLA violations at the Application Layer. Being tested in a real Cloud environment, the model has demonstrated its capability in monitoring, detecting SLA violations, and suggesting effective measurement intervals for various workloads, however it's only efficient with a single application SLA violation situation. More experiments are yet to be conducted on multi-tier applications in the future. However, this approach does not have a flexible means to change the requirements and metrics to be monitored at run-time.

Research in [141], Rak et al. introduced a framework whose approach is based on mOSAIC: Open source API and platform for multiple clouds (mOSAIC). This framework consists of a monitoring and warning system capable of monitoring Cloud resources and application components. Additionally, it will verify the compliance of a service through a set of rules, hence discover warning conditions. However, performance evaluation of the framework is yet to be determined in the future. Müller et al. [123] have designed and implemented SALMonADA, which is a service-based system for monitoring and analyzing SLAs in order to provide an explanation for violations. The SLA is described using a monitoring management document (MMD) to be consumed by the monitoring infrastructure; however, these authors do not help stakeholders in selecting alternative operationalizations depending on the platform, and require advanced users with knowledge about metrics and specific platforms.

Finally, in [155], Shao et al. proposed a runtime model for cloud monitoring (RMCM), which denotes a representation of a running cloud by focusing on common monitoring concerns. However, they do not mention specific non-functional characteristics for SaaS cloud environments and their metrics (e.g. scalability, availability, elasticity, etc.). Furthermore, they do not provide an SLA violation report and leave addressing non-functional requirements from SLA as future work. Overall, these monitoring solutions have focused on some specific quality attributes (e.g., performance) and some of them lack mechanisms to aggregate multiple quality attributes or parameters for a service consumer, which is a critical aspect of

monitoring. To the best of our knowledge, there is a need for approaches that autonomically monitor the specific non-functional characteristics of cloud services and that allow the flexibility of adding and modifying monitoring requirements at run-time. From these studies, it can be seen that there is still room for further research to improve on the technique for monitoring QoS, detection of SLA violation with remedy action and to ensure accountability among CSPs and CSCs. For trustworthy cloud services, effective mechanism for monitoring performances and detection of SLA violations are necessary. This is to increase the end consumers trust level towards the CSPs. Motivated by this fact we therefore propose an autonomic SLA monitoring framework managed by TTP for ensuring accountability and QoS in cloud services. In the case of penalties, the reputation of CSP concerned is affected. The method of penalty cost calculation, detection of future SLA violations, reputation score is specified in our framework.

5.2.2 Review of current research efforts in Trust Reputation Systems

Reputation is generally said or believed to be about a person's or thing's character or standing. Related to products and services, it is the subjective opinion based on feelings, past experiences and the viewpoint of a circle of "trustful" people. Reputation is often used in the sense of the community's general reliability and trustworthiness evaluation of a service entity [122] [140] [84]. Therefore, this trust reputation needs to be gathered, collected and filtered in order to generate the most trustful reputation associated to a service, a product or a user. The process of formalizing trust reputation can be done from different perspectives. To do that, Trust Reputations Systems (TRS) are available tools that can work on trust reputation using algorithms with the purpose to calculate the most reliable evaluation.

5.2.2.1 Trust Reputation Systems

Researchers propose models that handle reputation concepts and introduce trust management from a social approach. Human societies is the most important source of inspiration for researchers in this field. But the way the trust assessments is managed differs from one trust model to another. There are many computational trust models in the literature [166][47]. Different classifications of trust models have been established based on different criteria such as the paradigm type [137], and the representation of trust [173]. When classifying by paradigm type, models are classified as cognitive [150] or probabilistic [134]. While for the trust representation criteria, model is classified following the composition of the trust assessment component. Some models use a single value that represent the trust assessment [118], while others propose more than one value, such as Beta Reputation System [77] that

formulate a trust assessment as a vector of three value : belief, disbelief, and uncertainty, in addition to some constraints and formulas used while calculating those values. Guha et al. also propose a model that handle an assessment of trust and distrust, and put forward an algorithm for the propagation of those values inside the system. Also, many other work in this area, such as [76] [105] [140] [84] propose in their work several Trust Reputation Systems (TRS) architectures with different algorithms to calculate reputation score related to the providers. Those classifications shows how many trust model exist in literature and how existing trust models differs from one another. But none of these approaches give a dynamic trust reputation model that take into account the cloud computing environment.

In contrast to those previous works on TRS, we propose a novel trust reputation model managed by trusted third party. Our model is based on the calculation formulas proposed by [143] with extension by adding other parameters and function that did not already exist in order to calculate the most trustful reputation score for the cloud service provider.

5.2.3 Review of current research efforts about Trusted third party for monitoring Service Level Agreements in real time

In the literature, various researches have introduced the notion of third party regarding different problems [69][176][17][142]. In their work on the issue of real-time QoS evaluation of a service Providers, Hammadi et al. [69] addressed service level management issue by introducing a Third-Party Service Level Agreement monitor (TP SLA Monitor) which consists of a trust assessment module and a risk assessment module. Qian Wang et al [176], studied the problem of ensuring the integrity and security of data storage in Cloud Computing. They consider the task of allowing a third party auditor (TPA) to evaluate the service quality from an objective and independent perspective, on behalf of the cloud client, to verify the integrity of the dynamic data stored in the cloud. The introduction of TPA eliminates the involvement of client through the auditing of whether his data stored in the cloud is indeed intact, which can be important in achieving economies of scale for Cloud Computing. Al Falasi et al [17] also proposed the use of a third party in their work on a framework for Cloud service composition that aims to overcome the issues caused by the open and flexible nature of Cloud services. They proposed to incorporate some trusted third-party entities that assures real-time QoS measures of the service, hence reducing the load on the Cloud provider in view of governing and optimizing the service composition process. Rana et al. [142] describe how SLOs are impacted by the choice of specific penalty clauses. They present a model, where a customer may assign weights to SLOs over others. Moreover, the authors propose a monitoring architecture that uses a trusted mediator (i.e. a TTP) for the resolution

of conflicts between involved parties. The monitoring scheme defines the communication exchange between the TTP and the two signatories. The authors assume that the monitoring infrastructure is located at the service provider's side and that the TTP can guarantee the impartiality of the monitoring and auditing process.

As we can see, the introduction of a third party have been proposed addressing many issues in the cloud computing but little work have explored the effectiveness of the introduction of a trusted third party for monitoring in real time the Service Level Agreements.

5.3 Autonomic SLA Monitoring Framework managed by a Trusted Third Party

5.3.1 General Presentation

In this section, we present in details the design descriptions of our proposed Autonomic SLA Monitoring framework ¹ in the cloud computing architecture at different levels (Saas-Paas-Iaas). This framework is managed by a trusted third party. We describe its components in more depth, and how they interact with one another. Figure 5.1 depicts the main functional components of our framework managed by trusted third party. There are two core components or modules of the framework namely: (1) SLA establishing; and (2) dynamic service monitoring. The first module comprises the SLA establishment process aspect and it is intended to provide an agreed SLAs to the second part, which comprises the service monitoring aspect. Service monitoring module: is responsible for managing and monitoring the QoS by detecting SLA violations at runtime and the proposed mechanism for an adaptive remedy rectification. This module can efficiently manage QoS by collecting data of all the available resources in the cloud environment based on the established and agreed SLAs. It has also a knowledge database that stores this information which is required by the other components in the framework. Dynamic SLAs monitoring is actually an instance of an autonomic control cycle: monitoring, analysis, planning, execution whereby systems can monitor themselves and maintain a target behavior.

5.3.2 SLA Establishment module

In our previous work [110], we presented a meta-model to express SLA contract requirements using MDE, as a mean for establishing service level agreements between CSPs and CSCs in

¹The Autonomic SLA Monitoring Framework is a free, open-source software; it can be downloaded from <http://code.google.com/p/slac-language/>

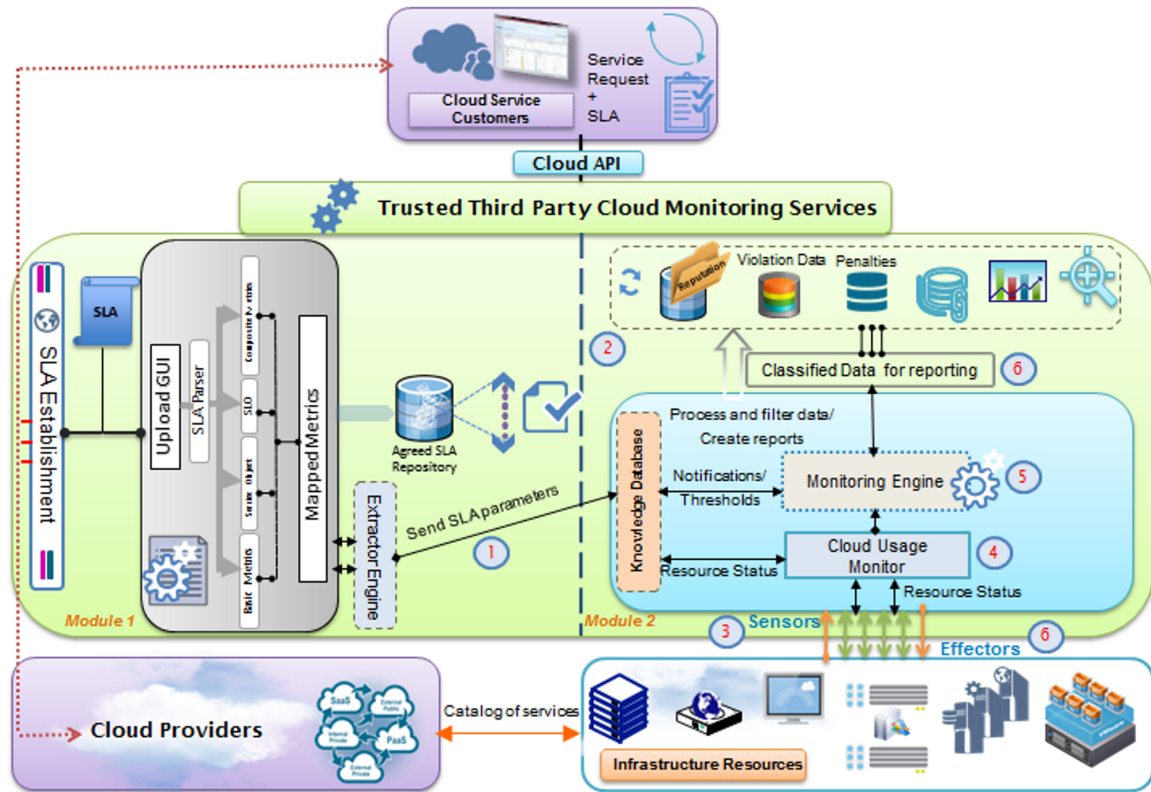


Figure 5.1 SLA establishment and Monitoring framework managed by TTP

the context of a particular service provision. Our approach for SLA establishment embraces MDE principles to automate the SLA document generation from the customer requirements document.

We propose to use domain specific modeling languages (DSML) to express the SLA contract requirements. Defining a DSML consists in defining an abstract syntax (i.e. the meta-model), a concrete syntax accompanied by a correspondence mapping between them. MDE promotes the usage of models as first class entities. A model is used to represent a given system conforms to a precise meta-model (in our case an SLA Contract [110]). This latter is just a model that provides the basis for constructing another model. A concrete syntax may be textual or graphical. The first one enables users to work with models just as they would work on other text-based programming languages, where as a graphical syntax enables users to work with models using a diagram surface.

DSMLs can be used for different purposes : Editors, Documentation Generators, Checkers, Simulators, Compilers, Translators, etc. In this work we use DSML for an Editor purposes to construct a model that we are going to verify its properties to decide the actions to perform.

Our meta-model for SLA establishment has been influenced by related work in particular [109][83][154][18][82] [97], however, this project adopts a semiautomatic solution where the intervention of an expert is needed. Our approach takes these needs into account by offering modeling languages, transformations and automated processes. Our meta-model describes QoS guarantees in the form of SLO clauses. One of the novelties that it integrates features dealing with QoS uncertainty and cloud fluctuations, such as penalties. This penalties are applied in case of SLA violations, to compensate cloud service customers. Our proposed Cloud Service Level Agreement language enables the definition of SLAs for any type of cloud service. This language is intended for dynamic service provisioning environments. It's based on the Open Cloud Computing Interface (OCCI) and the Cloud Computing Reference Architecture of the National Institute of Standards and Technology (NIST).

In our context, we used the Eclipse Modeling Framework (EMF) as a domain specific modeling environment for the definition of meta-models and tools to generate model editors. As we explained in [110], with EMF we were able to define the meta-model of this language relied on existing standards, on the one hand, and on the need to automate the derivation of SLA document, on the other. EMF is a modeling framework and code generation facility for building applications based on a structured data model. In addition, EMF provides the foundation for interoperability with other EMF-based tools and applications [164]. It provides an implementation of EMOF (Essential MOF), a subset of MOF, called Ecore. It is based on the ecore meta-meta-model that we construct our meta-model [66]. Once our model has being constructed, a M2T (Model To Text) transformation is performed in order to generate an equivalent of our model in a textual format. For this, we use Acceleo [124], which is an open-source project that aims to provide a code generator (transferring models into code) for the people who aim to profit from a Model-driven approach in order to increase their software development productivity. In terms of support for the SLA establishment, our approach offers a framework deployed in a cloud environment that emphasizes the steps bellow:

1. A Cloud customer represented in a model, defines the requirements according to a CloudCustomer language;
2. A Cloud provider represented in a model, defines the offers in a CloudProvider language;
3. A SLA contract is generated based on a M2M (Model To Model) transformation. This latter takes as input the Cloud consumer and the Cloud provider models. Then, it maps all the concepts used by the Cloud customer and the Cloud provider.

In conclusion, once an SLA is established between CSPs and CSCs an SLA document is generated, it is passed to the online monitoring by trusted third party cloud controller system described in the following subsection. An integration with external component is needed in order extract the SLA parameters and their corresponding objective values from the SLA document and store them in the knowledge database. This *extractor engine* uses a XML parser using the well-known open source SAX API [119](as shown in figure 5.2).

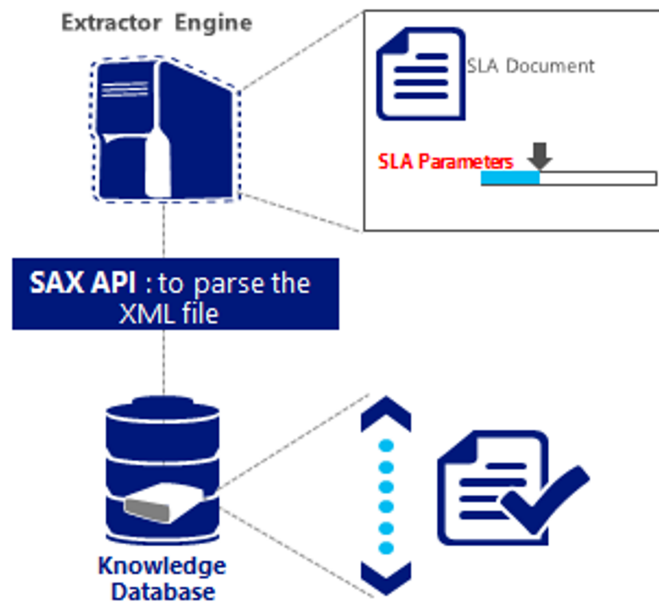


Figure 5.2 Extract SLA parameters from SLA document

5.3.3 Service Monitoring Module

This subsection describes in detail the monitoring module, its components, and how the components interact with one another. We will describe the architecture of this module and later we will explain its implementation in details. As presented in figure 5.1, the monitoring module is composed of four sub modules: the Cloud Usage Monitor, Monitoring Engine, analysis component and Knowledge Database.

5.3.3.1 Cloud Usage Monitor

The cloud usage monitor mechanism is a lightweight and autonomous software program responsible for collecting and processing IT resource usage data. Depending on the type of usage metrics they are designed to collect and the manner in which usage data needs to be collected, cloud usage monitors can exist in different formats. The upcoming paragraph

describes an agent-based implementation format. It's designated to forward collect usage data to a log database for post-processing and reporting purposes.

The Monitoring Agent is an intermediary, event-driven program that exists as a service agent and resides along existing communication paths to transparently monitor data-flows as shown in figure 5.3. This type of cloud usage monitor is commonly used to measure network traffic and message metrics. The agent plays the role of a sensor.

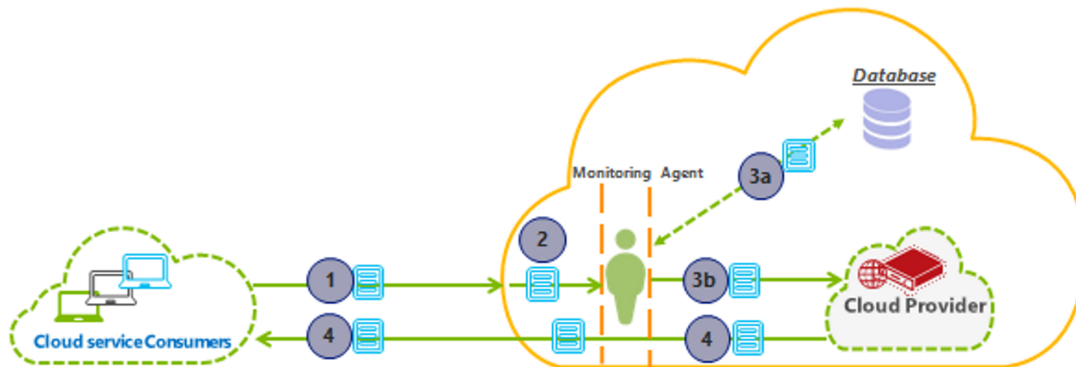


Figure 5.3 A cloud service consumer sends a request message to a cloud service (1). The monitoring agent intercepts the message (2) to collect relevant usage data before allowing it to continue to the cloud service (3b). While the monitoring agent stores the collected usage data in a log database (3a). The cloud service replies with a response message intercepted by the agent (4), it's sent back to the cloud service consumer.

The Monitoring Agent could monitor at different levels in the system Cloud on each Compute Node:

- Host OS/Hypervisor. Here the Monitoring Agent could capture all traditional operating system metrics at the hardware level, e.g., percentage CPU time per VM, memory usage, disk I/O, network I/O, etc.
- Guest OS. Monitoring here enables the Agent to collect operating system information specific to one VM.
- Application Level. As opposed to the previous two levels, this requires that the application be modified to provide a monitoring interface whereby application-level performance metrics can be obtained.

After this data is initially captured, it can be used at different times. All of the monitored data is archived in a database that can be sent at run time to the monitoring engine (ME). The ME can periodically apply a rule set from the SLA information, provided by the knowledge database, to determine if an SLA is being violated, or requires any corrective action. Given

the amount of data in the database, the ME could also do long-term trending analyses that could not be identified from single metrics or events.

5.3.3.2 Monitoring Engine

The Monitoring Engine is used to specifically observe and analyze the runtime performance of cloud services to ensure that they are fulfilling the contractual QoS requirements published in SLAs. This system can pro-actively repair or fail-over cloud services when exception conditions occur, such as when the MA reports a cloud service as "down". Figure 5.4 provides an UML description of the functional architecture of this monitoring engine, making obvious the high level functional components allowing the targeted maturity level. The main

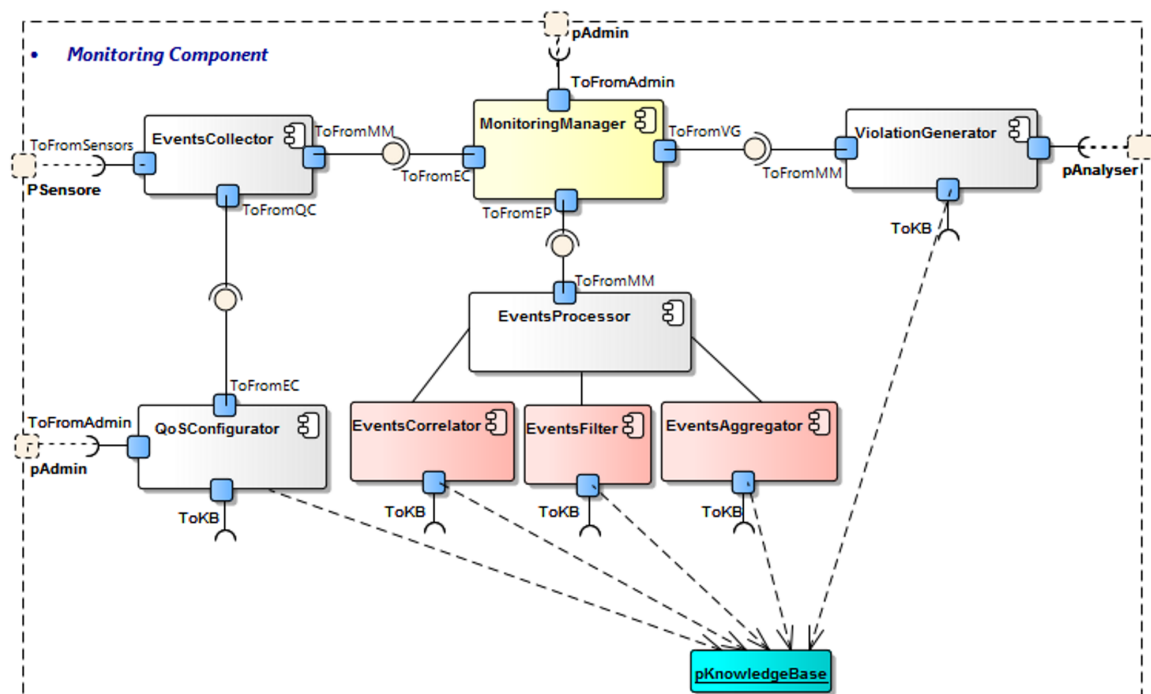


Figure 5.4 Functional Architecture of The Monitoring Component

components of the monitoring engine are :

- The **MonitoringManager** it is responsible for the configuration and deployment of the various components and the interaction between them; it also ensures the interface with the administrator. As can be seen, the monitoring component exposes four external interfaces. A simplified form of the web subsystem provides a web interface for administrator, enabling them to interact with the system (e.g. add new patterns, define EPL statements, receive alerts, etc.) (as shown in figure5.5). This created event patterns expression and EPL statements incoming from the administrative interface (at runtime

configurations) are received by the REST interface which is exposed by the receiver service (API). Subsequently, this rules are transferred to the persistence service which stores them in the Knowledge database. Communication with receiver service and database utilizes the Java DataBase Connectivity (JDBC) interface overlaid by the Hibernate technology;

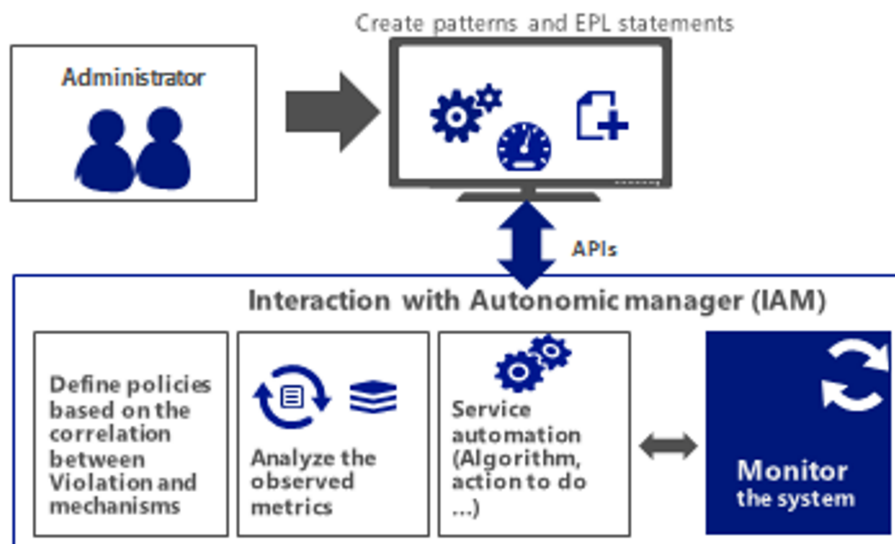


Figure 5.5 Defined Rules by administrator

- The **QoSConfigurator** mechanism designed to interact with the knowledge database (KB) for the retrieval of the agreed SLAs between cloud service providers and consumers, also contain the description of the appropriate sensors; After that, it interacts with the *EventCollector* for the specification of the metrics to collect depending on the agreed SLA;
- The **EventsCollector** this component is responsible for the collection of metrics at runtime of cloud services through interaction with the corresponding sensors. The description of this interaction is specified by the QoSConfigurator that asks the KB. Interaction between EventsCollector and sensors integrated in the managed entity can be done through periodic request/response way or through a listening way with notifications from sensors when a change. After data is initially collected, it can be used at different times. All of the monitored data is archived in a KB that can be queried by the MonitoringManager;
- The **EventsProcessor** it processes arriving events by applying defined patterns stored in the KB; it can be filtering (EventsFilter), correlation (EventsCorrelator) or aggrega-

tion (EventsAggregator) events. Additionally, the correlation means (link between the metrics), aggregation (group different level metrics) and filtering (selection of specific patterns) of collected data are needed to identify or predict more complex patterns that can be modal, temporal, etc. such as a sequence of response time values exceeding a defined threshold and following an increasing tendency. As soon as a pattern is detected, it notifies the MonitoringManager.

- The **ViolationGenerator** it is invoked by the MonitoringManager when one or several events match a defined pattern. It then generates adequate violation correlated to non-satisfaction of QoS requirements via the catalog of violation included in the KB. In our case study, we consider three type of violation associated to the considered metric, that means we need to be warned when 3 types of violations are detected:
 - NOTIFICATION : Two event upper than a threshold, the priority of the generated violation is Low;
 - WARNING : WARN us if we have 3 consecutive events above a certain threshold;
 - CRITICAL : ALERT us if we have 4 consecutive events above a certain threshold.

Thereafter, monitoring engine sends then the description of the related violation to the analysis component. We propose an algorithm to identify the violation type and achieve a better selection of inappropriate events.

5.3.3.3 Analysis component

The main goals of the Analyze component are to process detected SLA violation received from the monitoring engine and to correlate these data in accordance with the knowledge base policies (QoS parameter violation, failure, congestion, etc.) to produce a diagnostic. The output of the component is a request for change that guides the definition of the adaptation processes. Figure 5.6 provides an UML description of the functional architecture of this Analysis Component.

The main components of the analysis phase are:

- The **AnalysisManager** it is responsible for the configuration and deployment of the various components and the interaction between them; it also ensures the interface with the administrator needed to analyze detected violations and to initiate the actions to be executed;

Algorithm 1: Get Violation State

Input : Event list (Metrics : ex. Rt, Av) in a time interval, Context; Where
Event-list-size = 4

Output : Violation State

Let Nb-event = 0;

for $i : 1$ To 4 **do**

if ($Event - list(i) > threshold$) **then**

 Nb-event = Nb-event + 1;

end

end

if $Nb - event \leq 1$ **then**

 Violation-State = Normal ;

else if $Nb - event == 2$ **then**

 Violation-State = Notification ;

else if $Nb - event == 3$ **then**

 Violation-State = Warning ;

else if $Nb - event > 3$ **then**

 Violation-State = Critical ;

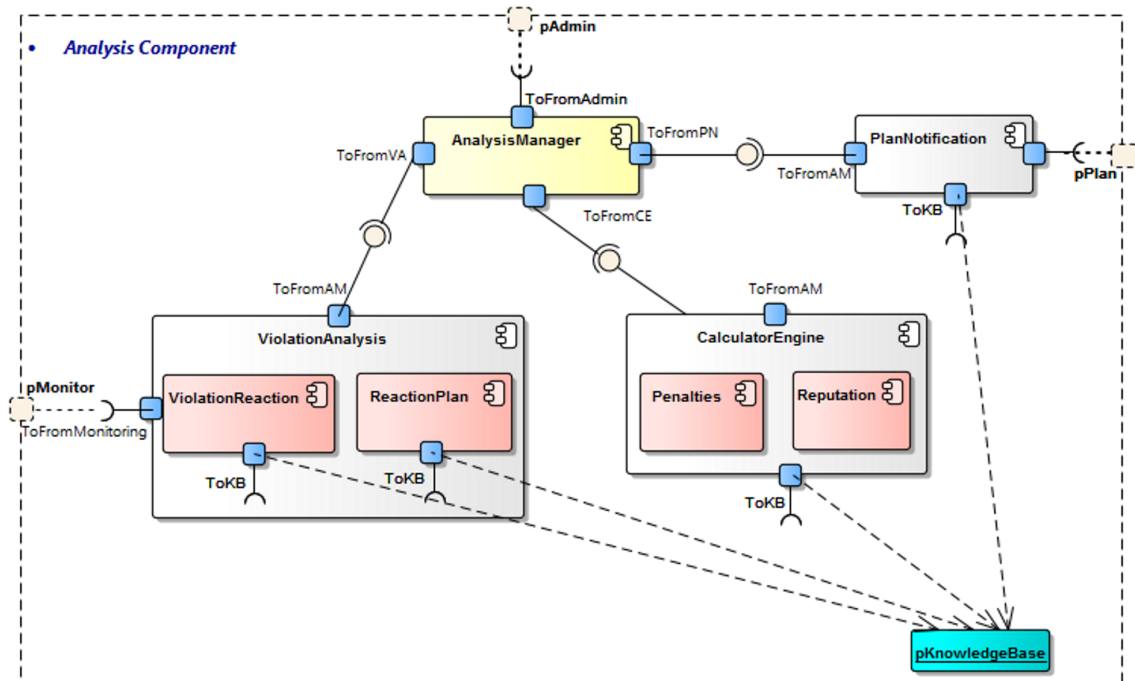


Figure 5.6 Functional Architecture of The Analysis Component

- The **Violation Analysis** engine processes the received violation. It provides an interface for receiving the monitored information from the monitoring phase. It analyzes

and classify violation depending on the priority and criticality (preferred, acceptable, unacceptable). This mechanism presents the backbone of the Analyze component.

Also, the Violation Analysis engine contains two subcomponents: the *Violation Reaction*, as illustrated in algorithm figure2, and *Reaction Plan*:

Algorithm 2: Violation Reaction Algorithm

Input : Violation-State

if *Violation – State == Notification* **then**

 Notify-CSP-CSC-To(Analyse the situation);

else if *Violation – State == Warning* **then**

 Notify-CSP-CSC-To(Alert for further investigation);

 Send recommendation according to Rule-based system;

else if *Violation – State == Critical* **then**

 Compute-Violation(described in section:4.5);

 Compute-Penalty(discussed in section:4.5);

 Decrease-Reputation(explained in section:5.3.4);

 Update-Knowledge-base();

Three Reaction (3R) strategies are defined for Violation Reaction based on the received Violation state:

- In the case of a violation type notification, it dispatches the notification messages to the customers/providers to alert about the situation.
 - In the case of a Warning violation type that means a degraded situation, with this information the system can react quickly to avert the violation threat and save the provider from costly SLA violation penalties. In order to react before critical SLA violations happen. We adopted the Rule-based system method for the interpretation of violation state with the goal of preventing SLA violations by triggering appropriate actions. A rule-based system as Jess [Jes] or Drools [JBo] contains rules in the “IF Condition THEN Action” format, e.g.,
1. IF $R_t < MV_{IB}$ THEN Add new virtual (Optimal Value(OV), Minimal Value (MV)).
 2. IF $R_t < OV_{IB}$ THEN Increase R_t share by 10% for VM.
 3. IF $A_v < MV_{Av}$ THEN Add 2 computer nodes to the cloud.
 4. IF $A_v < OV_{Av}$ THEN Outsource application to other cloud.

- In case of the violation is critical cannot be averted and the real critical situation persists without action by providers, the system logs the necessary information for calculating the appropriate SLA violation penalties in order to compensate the consumers for tolerating the service failure. After that the *Reaction Plan* mechanism notifies the Calculator Engine in order to take a decision based on the received critical violation state.

Thus, at the specified time points, the *Reaction Plan* engine are invoked by the rules (receives the notifications) as inputs if any support actions need to be taken and should output an action that can be executed in order to prevent an SLA violation.

- The **Calculator Engine** it is invoked by the Violation Analysis component. The goal of this engine is to automatically calculating the penalty cost of the violations in a time interval based in our Penalty model for managing and applying penalties, for more details see the function already cited in section 4.5. Therefore, in the case of penalties, the reputation of CSPs concerned is affected. The new reputation value that will be assigned to the cloud provider is calculated by our trust reputation Model as described in the section 5.3.4. The results of this reputation scores generated by the TTP is stored in the reputation System, which will send a recommendation to the customers about the provider's ranking.
- The **Plan notification component** it is invoked by the AnalysisManager to define and structure the set of actions to be executed when a request for change has been produced by the Analysis component in order to provide an adequate scalability and QoS level. The actions are chosen in the Analysis phase in correspondence with the knowledge base. The next steps are planning the order and timing of the actions (Plan phase) and finally executing them (Execution phase) with the help of actuators communicating with the self-management interface of the responsible services.

5.3.3.4 Knowledge Database

The term Knowledge Database in our context following the approach proposed in [55], means intelligent usage of measured data, obtained by monitoring, for the decision making process to satisfy application performance goal defined in SLA agreements while optimizing the computational resource usage. This KB integrates various policies (QoS parameter violation, failure, thresholds, algorithms for best cloud resource selection, SLA violation, etc..). Also contain the catalog of violations that will be used to guide the autonomic behaviour (e.g. diagnostic models, association of issues and specific corrective actions, etc.).

The planning and executing steps of the MAPE-K loop will have to be explored in details in our remains future work, particularly when several adaptation mechanisms will have to be considered.

5.3.4 A Novel Trust Reputation Systems Model

We propose a novel trust reputation model. For that, we extend the calculation formulas proposed by [143] by adding other parameters and function that did not already exist in order to calculate the most trustful reputation score for the cloud service provider. There are several possible sources of this final trustful reputation score, such as trusted third party QoS monitoring services comprises the last past-interaction experience and the current feedback reputation value from existing cloud service consumers. The TTP module collects all those values and stores them in the QoS repository, after processing and calculating methods we give the final score value of reputation. We discuss the reputation calculation methodology of the TTP in the following paragraphs.

Formally, an SLA S consists of a collection of metrics $\mathbf{m} = \langle m_1, \dots, m_n \rangle$ and time interval δt representing the period over which the SLA is valid: $S = \{ \mathbf{m}, \delta t \}$. For this purpose, we propose in our model four different violation state levels (normal, notification, warning or critical). For each metric in an SLA the evaluation can take one of those violation levels. The function Evaluation Value E performs the following action:

- NORMAL : The quality of services is respected according to SLA;
- NOTIFICATION : Low quality impact;
- WARNING : Medium quality impact;
- CRITICAL : Critical quality impact.

$$success(m) \begin{cases} m \text{ is Normal} \rightarrow 3 \\ m \text{ is Notification} \rightarrow 2 \\ m \text{ is Warning} \rightarrow 1 \\ m \text{ is Critical} \rightarrow 0 \end{cases}$$

The *Evaluation Value* E of an SLA S can be defined as the number of successful metrics relative to the total number of metrics n : Each metric m_i where $(i \in [1, n])$ can be weighted according to its importance. Each weight, w_i , can have a value between 1 and 10, where 10

means very important and 1 represents lowest importance to the consumer. This gives the evaluation value of the cloud provider, $E(S)$:

In our trust reputation model we have used a reputation function that calculates the current reputation value of one cloud provider. Consequently, this function takes as an input the past evaluation experiences already received from cloud consumers and stored in the Reputation Database (RDB). Then the function calculates the new reputation value that will be assigned to the cloud provider. That means quantify each of these factors individually and, combine these factors to produce a final reputation value for a service provider. Note that according to this equation, reputation value will always be at most 1. The function based on the following concepts [34] acts by triggering the update of reputation value according to the following equation:

$$\mathbf{R}(S) = (1 - \beta) * \sum_{i=1}^q \frac{E_i(S) * \beta^{q-i}}{1 - \beta^q}$$

Where :

- E_i : the evaluation value in an instant (i);
- q : is the number of past experiences;
- β : presents a constant of forgiveness, it's value is within [0,1], it is used as weight for past experience, the k-passed experience has as weight β^{q-k} .

Initially, a service provider will obtain some default value for its reputation equal to a value defined by the TTP, and used as initial value, for the seek of simplicity of our TRS, we have used a mean value of current cloud provider's reputations already calculated by our TTP. As the number of agreements served increases its reputation will more accurately be reflected by $\mathbf{R}(S)$. In what follows we describe the mechanism of our proposed Reputation Model.

Once a cloud service provider and consumer have established an SLA, the service provided is monitored by a trusted third party. This TTP is a trusted module executing on a server that is physically separated from the cloud provider. In either case, the service provider must not have any access to the TTP, outside of the interactions that are properly defined.

The Reputation queries are exchanged between the CSC, CSP and TTP. Figure 5.7 gives a representation of the message exchange in the system. The TTP maintains a database of reputations (RDB), updating it whenever an SLA is signed and carried out takes place. The RDB is useful to import the reputation score at the beginning of a service provision and

export it back at the end of the service provision so it can be used by other possible CSC. A CSC may also hold a local reputation value associated with a provider – based on monitoring the SLAs that it holds with a provider.

Whenever a CSP provides a service to CSC, the TTP evaluate the QoS according to agreed SLA then generate and update the new reputation of the involved CSP in the RDB. The TTP sends a message to both the CSP and Cc notifying them about the update. For example, if a service provider consistently provides poor service (that is, violating its SLAs), then, its reputation will be declined.

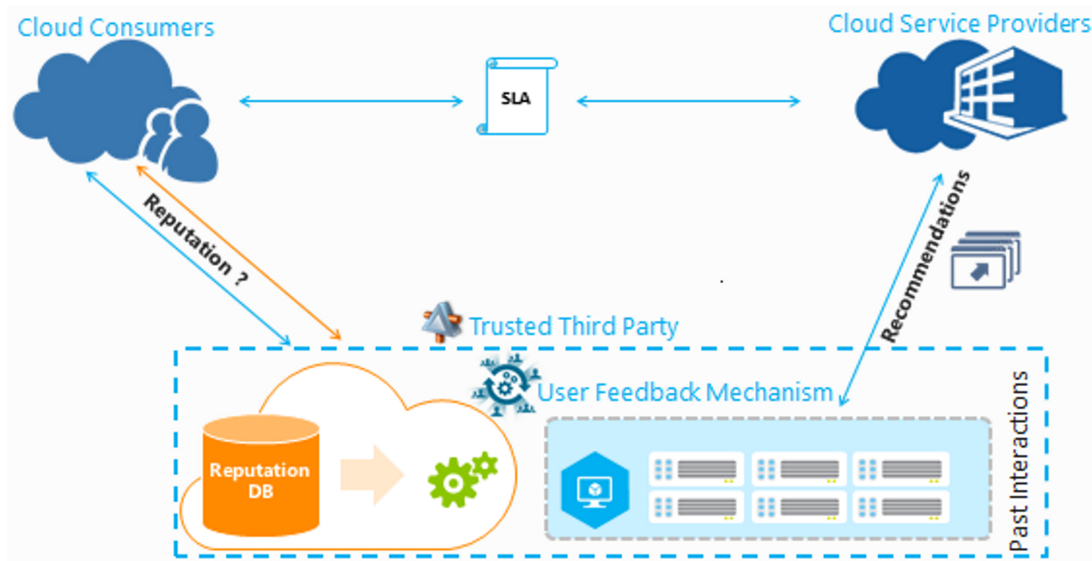


Figure 5.7 Reputation Model using an adaptive approach

5.3.5 Framework Mechanism

The following figure 5.8, illustrates our proposed framework design as a sequence diagram which includes; the CSC and CSP as actors and the various objects related to its events. This sequence diagram provides a complete view regarding the information exchanged between all parties in real time.

The first step is to establish an SLA, the cloud customer define their requirements and the provider send their offers about cloud services. The TTP uses SLA establishment module which is based on MDE principles to generate the SLA document. Constructing and accomplishing SLA is a necessary target for all parties to preserve their rights and solves the conflict issues among them. The TTP sends the SLA contract with defined SLOs parameters to both customer and provider, which in turn will be monitored in the monitoring engine. In this phase, we assumed that the SLA establishing process is completed and the

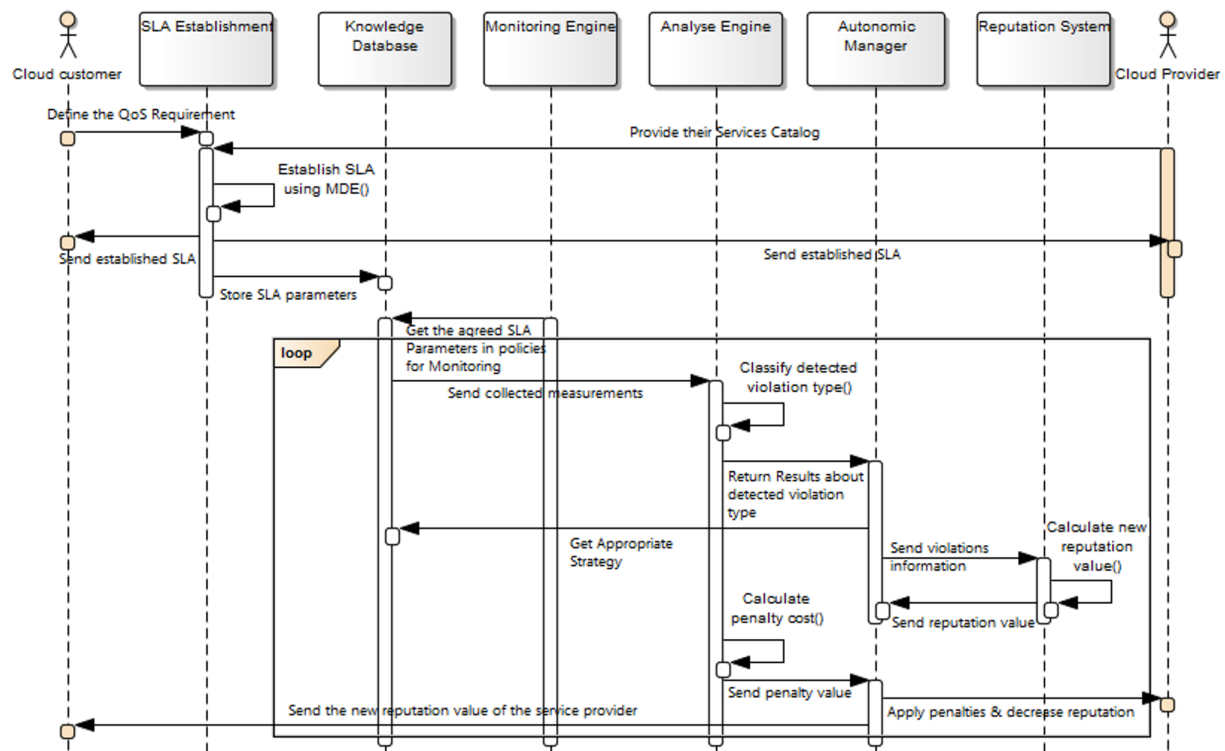


Figure 5.8 Sequence Diagram for the Proposed Framework

agreed SLAs are stored in the knowledge database for service provisioning as presented in figure 5.1. Beside the SLAs, the predefined threat thresholds are also stored in the KB. After integrating various policies in its KB (thresholds, algorithms, policies, etc.), the TTP (autonomic manager) begins with the monitoring phase. The monitoring engine then collects measurement information to be calculated.

The collected data is passed to the analysis component to correlate these data in accordance with the KB policies. The results are then provided by the analysis engine which checks whether the parameters are equal, under, or over the threshold defined in the SLA and whether any violations exist. The strategy of detecting SLA violations is based on the use of predefined violation thresholds, their detection is based on the processing of collected events, and on its comparison with patterns identifying the considered violation type. With this information the system can react quickly to avert the violation threat and save the provider from costly SLA violation penalties. In case of the violation threat cannot be averted and the real violation situation persists, the system logs the necessary information for calculating the appropriate SLA violation penalties in order to compensate the consumer for tolerating the service failure.

Therefore, in the case of penalties, the penalty mechanism is automatically activated to calculate penalty cost and decline reputation. The CSP is accountable for this violation and thus, penalty is imposed by TTP. The new reputation value that will be assigned to the cloud provider is calculated by our reputation model. The results of this reputation scores generated by the TTP is stored in the reputation database. Finally, the new reputation value and penalty cost are sent to the consumer.

5.4 Evaluation

The goal of this evaluation is to validate the functional architecture of our autonomic SLA monitoring framework by detecting SLA violations at runtime and the proposed mechanism for an adaptive remedy rectification. This experimental validation is based on the implementation of the monitoring engine, the dynamic detection of violation related to the response time (Rt) and availability (Av), the autonomic activation of our proposed mechanism, and then the observation of the induced benefits of this action.

We first describe the experimental setup we have used in evaluating our monitoring approach, then we depict the test scenario of SLA monitoring, and we briefly describe the cloud services we have used in the experiments. Finally, we discuss the results obtained from the execution of this test scenario.

5.4.1 Experimental Testbed

In order to validate our proposed framework, we need to perform its experimental testing in a cloud environment as realistically as possible. In this section, we describe the testbed used in the experiment, as it is depicted in figure 5.9. We use Apache CloudStack [45]. CloudStack is an open source cloud computing software for creating, managing, and deploying infrastructure cloud services. It uses existing hypervisors for virtualization. The VMware's hypervisor called vSphere⁶, is used in our evaluation. Table 5.1 shows the resource capacities of the physical and virtual machines being used in the proposed Open Cloud testbed.

SaaS application: we developed a Web service application that simulates a bookstore SaaS services to handle different file operations (e.g. download, upload, search, delete, list). We specified the same SLA for each service, expressed through a set of QoS properties, mainly: Rt, and Av.

Infrastructure: This scenario is performed over an emulation testbed, which allows testing a real bookstore web application, confronted to an emulated traffic. The emulation platform (5.10) is provided as a set of web services consisting of injectors generating traffic,

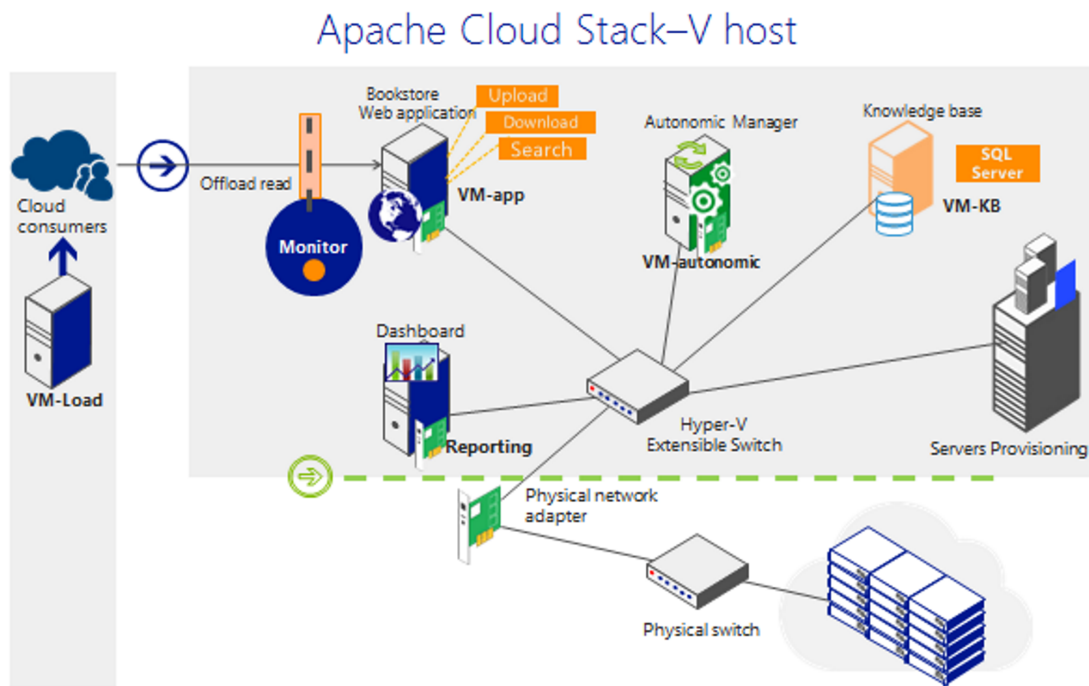


Figure 5.9 Testbed

and of a controller aimed at configuring the injectors following a defined scenario. The traffic is sent to an autonomic monitoring engine.

During the evaluation, we consider four kinds of VMs: The first one, called *vm-autonomic*, hosts the autonomic manager of the system providing interfaces to monitor and act on the system. The second one, named *vm-load*, is used for load injection purposes. It contains an Apache JMeter⁴ application that is used to simulate a load variation on the business application and to analyze overall performance under different types of workload. The third one, named *vm-app* contains our developed bookstore web application, where an Apache server responding to the incoming HTTP requests is installed. The last one, called *vm-KB* is an instance for the database which back-up all transactions.

5.4.1.1 Prototype

This paragraph aims at presenting the prototype including autonomic manager of the system providing interfaces to monitor metrics about the system state, analyze them and act on the system at runtime (as shown in 5.9).

In order to evaluate system behavior under high load, we used a customized Apache JMeter [Act] by introducing bean shell code adapted to our scenario for load injection to generate concurrent/simultaneous requests towards our implemented Web application

Table 5.1 Resource Capacities of the Physical and Virtual Machines

Instance Type	System	CPU	Memory	Storage	Network
Physical Machine	Linux/Ubuntu	12-core 6-CPU 2.5 GHz Intel Xeon E5420 QC	32 GO	1,5 TO	Infiniband 20G Ethernet
Virtual Machine	Windows Server 2012	4 VCPU 2.5 GHz Intel Xeon X3440	8 GO	250 GO	10 Gbit Ethernet
Cluster	Win-Server 2012 R2	2-core 1-CPU 2.5 GHz Intel Xeon X5025	10 GO	500 GO	10 Gbit Ethernet

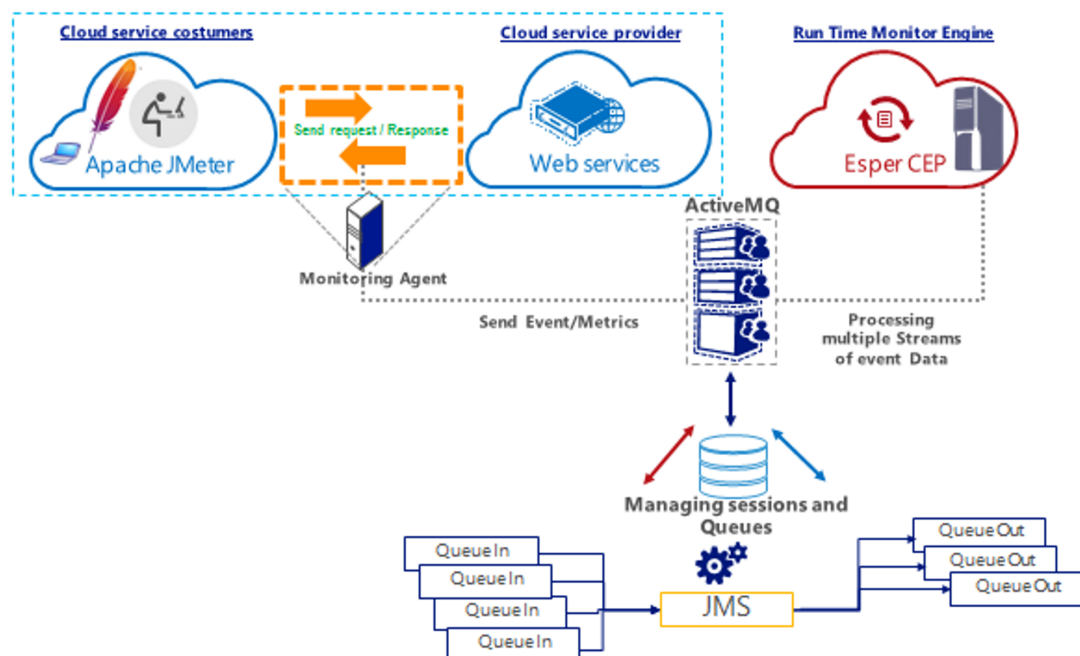


Figure 5.10 Communication Mechanism Overview

to simulate a load variation and to analyze overall performance under different types of workload, and report the measurements values. To measure these QoS we extended an agent to allow measurement of response time, and availability. The Monitoring Agent (MA): is a Java application that interacts with the SaaS application and JMeter, it's positioned in the middle to supervise the traffic between each actors in the resource pool to monitor the metrics of declared services. In general, this agent is placed in the cloud system to gather measurements about the different quality of service aspects stated in the SLA. The collected monitored metrics by this personalized agent are sent to the Monitoring Engine. But before

Table 5.2 Complex Mapping Rules

Resource Metrics	SLA Parameter	Mapping Rule
<i>downtime, uptime</i>	Availability (A)	$A = 1 - \frac{downtime}{uptime + Downtime} * 100$
<i>inbyte, outbytes, packet-size, bandwidthin, bandwidthout</i>	Response Time (R_{total})	$R_{total} = R_{in}(s) + R_{out}(s)$

that, this monitored metrics are periodically mapped to the high level SLA parameters. We take in consideration the approach discussed in [55] to deals with mapping of low-level resource metrics to high-level SLA parameters. In this step, we describe the mapping between the ME values and the values of the SLA parameters. Resource metrics – being monitored by the MA – include e.g., downtime, up-time, available incoming and outgoing bandwidth are mapped to the SLA parameters based on the predefined mapping rules stored in the database monitored metrics. The mapped SLAs are stored in the mapped metric database for usage during the monitoring phase.

We adopted the calculation formulas proposed by [55] in order to calculate the obtained values of the total response time and the availability. For example, the SLA parameter availability R_t is calculated using the resource metrics *downtime* and *uptime* as defined by the following mapping rule (as shown in table 5.2): $availability = \frac{AgreedServiceTime(AST) - Downtime}{AgreedServiceTime}$. In the mapping rules presented in the same table 5.2, the downtime variable represents the *mean time to repair* (MTTR), which denotes the time it takes to bring a system back online after a failure situation and the uptime represents the *mean time between failure* (MTBF), which denotes the time that system was operational between the last system failure to the next. The unit of the availability equation is in percentage. In this equation, we assume that the uptime and downtime variables are greater than or equal to zero. For response time R_t , R_{in} is the response time for sending a request in the inwards communication direction and is calculated as $\frac{packet\ size}{bandwidthin - inbytes}$ in seconds. The *packet size* is the size of the query sent in and its unit is in megabit. The *bandwidth in* is the total bandwidth for communicating in the inwards direction and its unit is in megabit per second. The inbytes is the amount of bandwidth already in use on this channel and its unit is in megabit per second. R_{out} is the response time for receiving an answer in the outwards communication directions and is calculated as $\frac{packet\ size}{bandwidthout - outbytes}$ in seconds. All incoming measurement data from MA are sent to the ME using our implemented communication model, as shown in 5.11. Our communication model exploits the capabilities of the Java Messaging Service (JMS) API, which is a Java message oriented middleware for sending message between two or more

clients. In order to use JMS, there is a need for a JMS provider that is capable of managing the sessions and queues. We used the well-established open source Apache ActiveMQ [Act] for this purpose. Thereby, the ActiveMQ allows the fast transportation of messages using the JMS which enables the communication using several protocols such as Stomp or OpenWire. The implemented communication model is a sort of queuing network. It realizes an interprocess communication for passing messages between the MA and the ME. This queue makes the communication mechanism highly efficient and scalable. The processing of the measurement/monitored data is carried out in a separate host. The chosen data exchange format is the JSON due to its lightweight data exchange format which requires less resources when compared to certain XML formats.

Real-time SLA violation detection is performed using techniques such as complex event processing (CEP) for the processing of collected events. Violation detection is the second main function of the monitoring engine. CEP is implemented based on the open-source solution ESPER [ESP]. Esper is an open source framework that uses the Event Processing Language (EPL), it can be the correlation of some events or aggregation of multiple events to constitute useful information, filtering by selecting specific events from all events, as it is depicted in figure 5.11; thus they constitute the right tool for executing the operations mentioned before. This engine provides high throughput and supports an expressive declarative language for complex event queries. The engine is wrapped as a service and receives all incoming sensor data from the queue in a push-based manner. Incoming measured-metric (ResponseTimeEvent, or AvailabilityEvent) data are encoded in JSON format. The wrapper maps the data to an internal Plain Old Java Object (POJO) representation of raw events, and a sensor stream is created for each sensor type. CEP queries can be defined over these streams or any combination of them via the ESPER query language. Query results are continuously pushed to listeners, which transform the results into a JSON representation and push them to the database.

The received metrics are stored in MySQL DB from where the mapping routine accesses them and applies the appropriate mappings. The agreed service SLA is also stored in the same DB accessible to the monitoring engine. Violations are detected when one or several events match a defined patterns, as you can see in table 5.3. Also, this engine provides a filter to remove identical monitored values so that only changed values between measurements are delivered for further processing. In this work, we use simple patterns aimed at illustrating what could be a violation. These patterns are described in Event Processing Language (EPL). Three simple cases may be defined depending on the received events, as shown in table 5.3. We use Hibernate to realize an interface between the monitoring engine classes and MySQL database. Hibernate is a high-performance Object persistence and query service. The most

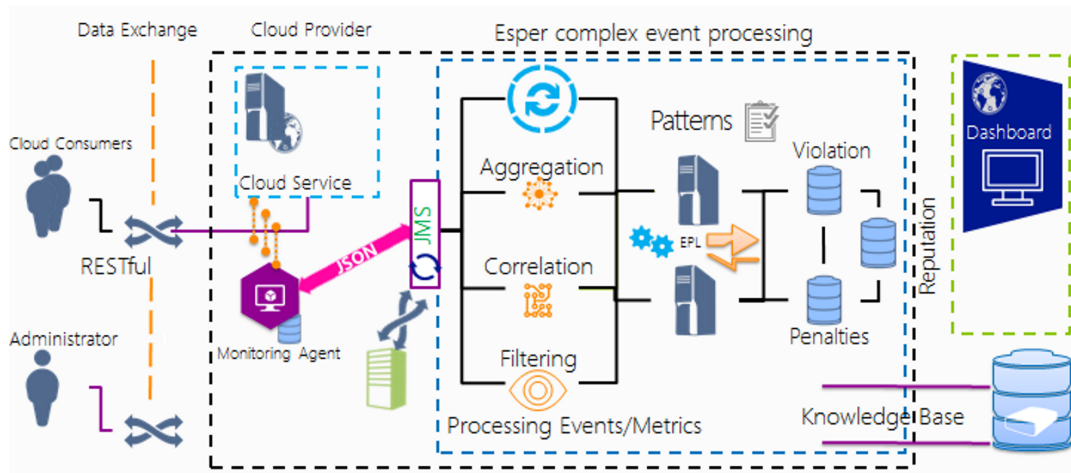


Figure 5.11 Event-processing technologies: collection, aggregation, filtering

flexible and powerful Relational solution on the market. Hibernate takes care of mapping from Java classes to database tables and from Java data types to SQL data types. It provides data query and retrieval facilities that significantly reduce development time [Hib]. With this database, the trusted third party now has the capability to create a graphical interface for displaying the monitored metrics values and the SLA objective status including the reported violations threats and real violations situations. The database is integrated with the web subsystem, providing a graphical interface for visualization of measurements and alerts.

5.4.2 Evaluation Scenario

In this real case study, an online bookstore established an SLA contract with a web hosting provider : Amazon EC2 or Microsoft Azure to deploy its online services like (Market analysis, Testing, Order execution). The SaaS provider should guarantee an average response time (Rt) less than or equal to 800 ms (present the agreed value) a margin of 50 milliseconds is acceptable as a QoS degradation. Regarding the service availability, the SLA states that at least 99.9% of availability (Av) should be guaranteed. A violation of the response time SLO (resp. the availability) implies a certain penalty equal to \$US 0.2, and decrease the reputation of the provider. We assign a reputation value of (0,8) to the cloud provider, this value fluctuates according to our trust reputation model as described in the previous section 5.2.2. The method of violations detection and cost penalties calculation are already specified in our autonomic monitoring engine. This SLA will enter into effect on first Mars 2017 (the Effective Date) for a defined period of two years. The other QoS parameters for web hosting services that should be included are described in table 5.4. Once the SLA has been established, various performance metrics start being collected.

Table 5.3 Patterns and generated violation

Pattern	Violation
select * from Event match_recognize (measures A as Rt1, B as Rt2, C as Rt3, D as Rt4 pattern (A B C D) define A as $A.value < Threshold$, B as $(B.value < A.value)$, C as $(C.value > Threshold)$, D as $(C.value < D.value)$)	[id=1, value=Rt-preferred, Violation=Notification]
select * from Event match_recognize (measures A as Rt1, B as Rt2, C as Rt3, D as Rt4 pattern (A B C D) define A as $A.value > Threshold$, B as $(A.value < B.value)$, C as $(B.value < C.value)$, D as $(D.value < Threshold)$)	[id=2, value=Rt-acceptable, Violation=Warning]
select * from Event match_recognize (measures A as Rt1, B as Rt2, C as Rt3, D as Rt4 pattern (A B C D) define A as $A.value > Threshold$, B as $(A.value < B.value)$, C as $(B.value < C.value)$, D as $(C.value < D.value)$ and $D.value > (2 * A.value)$)	[id=3, value=Rt-unacceptable, Violation=Critical]

Table 5.4 SLA between Amazon EC2 and Bookstore SaaS

Service	Metric	Oper.	Agreed Threshold	Penalty
Book store	Response time	$\leq$	800 ms	0.2\$
	Availability	$\geq$	99.9%	
	Storage	$\geq$	200 GB	
	Memory	$\geq$	5 GB	
	Incoming Bandwidth	$\geq$	50 Mbit/s	
	Outgoing Bandwidth	$\geq$	100 Mbit/s	

We compared the Bookstore application behavior on two cases with/without our proposed Autonomic SLA Monitoring framework (AM). We have studied a scenario that consists of a single peak of workload: 5 threads are started and maintained for 15 min, whereupon we simulate a peak of workload passing from 5 to 20 threads (consumers). The observation interval and evaluation window were fixed in 30 seconds and 30 minutes, respectively. It means that the availability of the SaaS provider at a given observation interval is calculated by the percentage of time the resources are accessible within the concerned interval, whereas the response time is calculated by the average of all the requests within one 30-seconds interval. We can imagine that a strike manifestation affecting the Bookstore service has led to this workload peak. Each thread simulated 10 consumers with two parameters (thus, every simulated consumers sent two types of measurements). The threads sent measurements periodically. Such high frequency of requests enabled evaluation of system behavior during peak load times.

5.4.3 Results and Discussion

This section presents the results obtained from this experiments. Figure fig. 5.13, depicts the results for the experiments with two implementations with/without Autonomic Monitoring (with-AM and without-AM) considering the SLA given in table 5.4. As shown in the graphs, a variable number of consumers was used for these experiments (initially 5, then 20 and finally 5 again). Every 30 seconds, the number of concurrent consumers, the average request response time and the availability are monitored. Figures fig. 5.13a fig. 5.13c, show respectively the QoS in terms of Rt, Av of the SaaS application for the two implementations according to workload variation. When workload increases to 20 threads/s, the request response time (see fig. 5.13a green curve) also increases above the SLO but detects just one critical violation using the with-AM implementation. This implementation detects that the capacity (number of VMs) would lead to violations of the SLA and recommend an action at provider level to avoid them. As we can see in fig. 5.13b red curve, the response time for the without-AM implementation exceed 800 ms in several times during peak. The number of violation detected without-AM implementation is three times higher than the implementation using AM. Figure fig. 5.13c shows the service availability over the execution of the experiment when the load varies. We note that when the workload increases the availability decreases. As we can see fig. 5.13d, the implementation using AM is less degraded when the workload attends its peak, whilst the implementation without AM is highly affected by the increased of charge.

When the number of concurrent consumers begins to rise at $t = 15$ mn, a violation type notification is detected with this information our mechanism dispatches the notification

messages to the customer/provider to alert them about the situation. Between $t = 16$ mn and $t = 17$ mn, a warning violation is detected that means a degraded situation, with this information the system can react quickly to avert the violation threat and save the provider from costly SLA violation penalties. We propose some remedy recommendation with the goal of preventing SLA violations by triggering appropriate actions that could be performed by the CSP (increase memory and/or the number of processors, add new virtual machine). At $t = 19$ mn a critical violation is detected. In fact, four burst of events upper than the threshold of 800 ms are detected [Event-71: 820, Event-72: 860, Event-73: 900, Event-74: 920]. Figure 5.12 illustrate this fact. This violation is due to excessive user load.

In this case, the violation cannot be averted and the real critical situation persists without action by provider. The system logs the necessary information for calculating the appropriate SLA violation penalties in order to compensate the consumer for tolerating the service failure. The CSP is accountable for this violation and thus, penalty is imposed by TTP. The penalty mechanism is automatically activated to calculate and decline reputation. Furthermore, the reputation of CSP concerned is affected, as seen in figure section 5.4.3. The new reputation value that will be assigned to the cloud provider is calculated by our trust reputation model as described before. The results of this reputation scores generated by the TTP is stored in the reputation database. At this step, the TTP presents to consumer a detailed dashboard that contains 'Real time' reporting with Alarm list (violations, penalties, value of reputation).

To give a better understanding of our approach, we use the data extracted from the simulation, as shown in figures fig. 5.13a and fig. 5.13c. The sampling frequency is of one sample for every 30 seconds. We use a 30-minutes interval, as the reference measurement interval to capture all SLA violations for the proposed scenario. We consider a maximum penalty fee of 2\$US/per time interval. The penalties are applied in case of SLA violations to compensate cloud service customer.

Violation of SLO implies penalty cost calculated according to equation presented before, as illustrated in table 5.5. The values of the violation ratio obtained in table 5.5 are calculated as follows :

$$Violationratio = \frac{Violationdetected - Threshold}{Threshold}$$

- Violation ratio of Rt: $|((820+860+900+920)/4 - 800)/800| = 0,09$
- Violation ratio of Av: $|((97 - 99,9)/99,9)| = 0,03$

In the other hand, we calculated SLO Penalty using the following formula:

$$SLO\ Penalty = \text{The penalty associated to an SLO parameter } i * \text{Violation ratio}$$

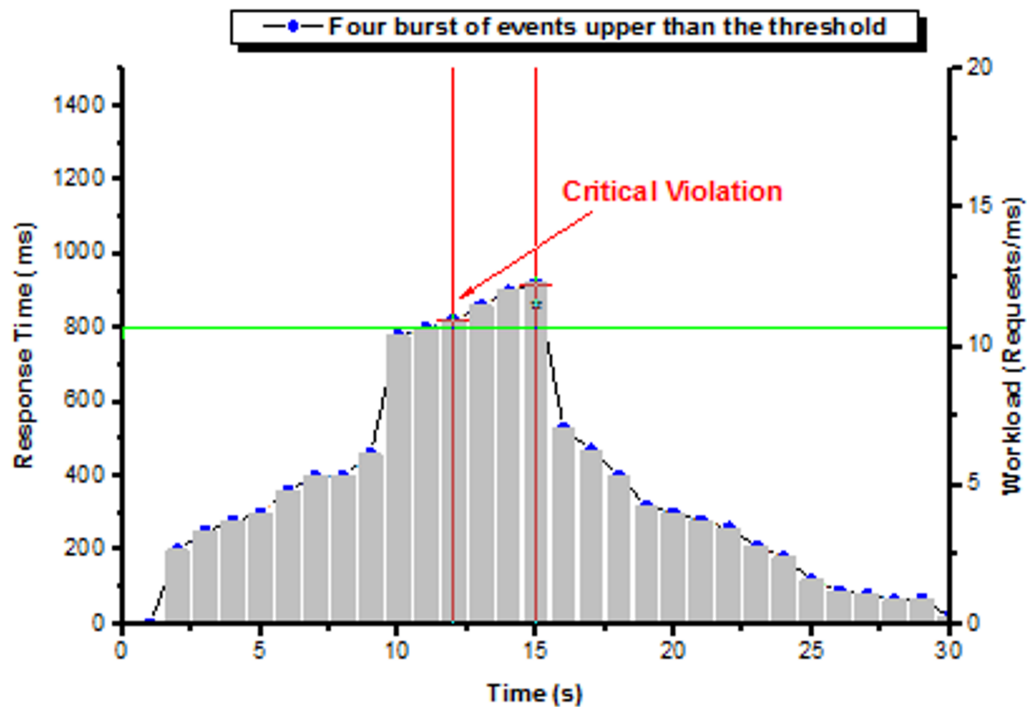


Figure 5.12 Four detected violation events upper than the threshold

Table 5.5 Calculated penalty cost

SLA parameter	Violations detected	Violation Ratio	SLO Penalty	Total penalty cost
Reponse time	875 ms	0,09	0,018\$	0,024\$
Availability	97%	0,03	0,006\$	

- $(0,09 * 0,2\$) = 0,018\$$
- $(0,03 * 0,2\$) = 0,006\$$

The monitored value 875 ms and 97% presents the average value of the four events in terms of response time, availability respectively.

5.4.3.1 Reputation simulation

We describe two sets of scenarios. The purpose of these scenarios is to verify the proposed approach can be used to modify the reputation of a provider over time. Initially, we have set a default reputation value of the cloud provider equal to 0,8. As the number of experiences of provider increases, its reputation will more accurately be reflected by **E**. As described in

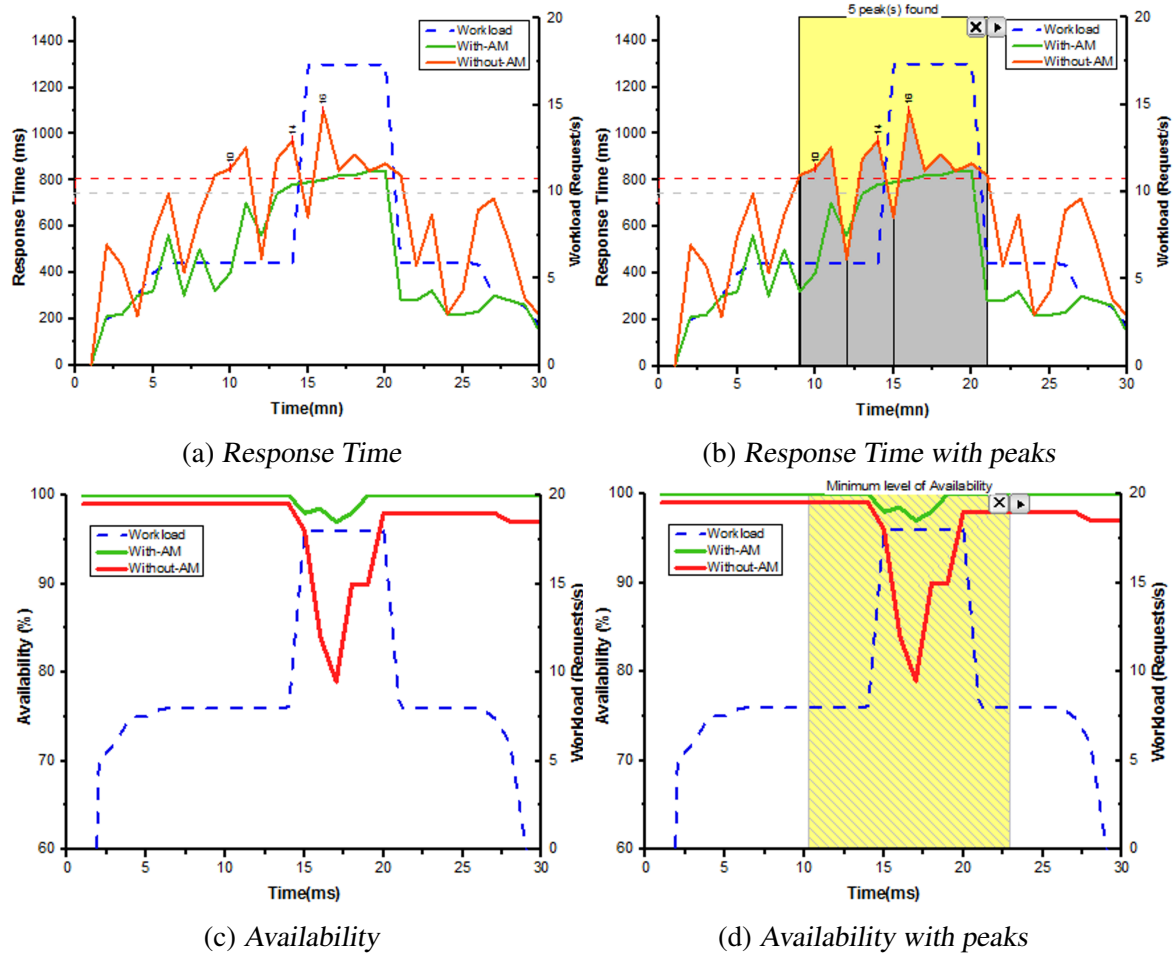


Figure 5.13 The results obtained from the experiments with/without implementations

function mentioned in section 5.3.4, we consider 0,6 as a default value of the constant of forgiveness. In this simulation, we have used two cloud providers (P1, P2) each of those propose its own SLA with specifics parameters and features, see table 5.6. We have also, two cloud consumers who will request cloud services from providers.

These consumers establish various experiences with these two providers by requesting a service. In order for a consumer to choose a provider, he should contact TTP entity that uses a reputation mechanism to give it the reputation score of each provider in order to choose between the most suitable provider for his needs. We consider that consumer 1 demands a service from provider P1 and consumer 2 from provider P2. After completion of the service, the TTP provides information about SLA metrics namely: response time, availability and bandwidth utilization. The information includes interaction with provider during the experiment period. Based on these reports, we can calculate the evaluation value of the provider and consequently his reputation score. Once processing of this information

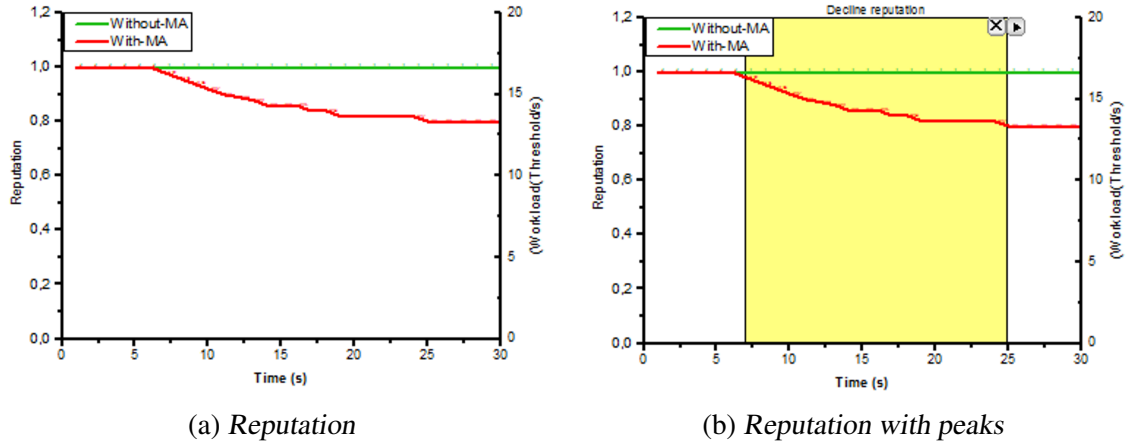


Table 5.6 Providers SLA

Providers SLA	Metrics	Provider P1	Provider P2
Proposed SLA contract	Response time	800 ms	750 ms
	Availability	99.9%	99%
	Bandwidth	50 Mbit/s	45 Mbit/s

is done, it can proceed to the reputation storing step which will allow retrieving reputation information on a later use. The *Evaluation Value* for our two providers obtained in table 5.7 are calculated as follows : First, we calculated Success rate of each metric using the following formula:

$$success(m_i) = 1 - \frac{Metricvalue - Threshold}{Threshold}$$

- Success rate of Rt: $1 - |((800-875)/800)| = 0,9$
- Success rate of At: $1 - |((99,9-97)/99,9)| = 0,97$
- Success rate of BW: $1 - |((50-45)/50)| = 0,9$

$$E(S) = \sum_{i=1}^n \frac{success(m_i) * w_i}{\sum_{j=1}^n w_j} ; E_1 = \frac{(0,9*8) + (0,97*9) + (0,9*3)}{8+9+3} = 0,93 ; E_2 = \frac{(0,9*8) + (0,97*9) + (0,9*3)}{8+9+3} = 0,91$$

$E_1 = 0,93$; $E_2 = 0,91$: With our trust reputation model we have used a reputation function that calculate the current reputation value of one provider. The new reputation value that will be assigned to the cloud provider is generated by the TTP and stored in the reputation system.

Table 5.7 Calculated evaluation and reputation value

List of Evaluations	Metric	Weight	Metric value	Success rate	Final evaluation value	New Reputation value by TTP
Evaluation for P1	Response time	8	875	0,9	0.93	0.93
	Availability	9	97	0,93		
	Bandwidth	3	45	0,9		
Evaluation for P2	Response time	8	900	0,8	0.91	0.91
	Availability	9	97	0,98		
	Bandwidth	3	45	1		

$$\mathbf{R}(\mathbf{S}) = (1 - \beta) * \sum_{i=1}^q \frac{E_i(S) * \beta^{q-i}}{1 - \beta^q} ; \mathbf{R}_1 = (1 - 0,6) * \sum_{i=1}^1 \frac{E_1(0,93) * 0,6^{1-1}}{1 - 0,6^1} = 0,93 ; \mathbf{R}_2 = 0,91$$

In summary, the results of executing the above scenarios, evaluated some key features of our proposed autonomic SLA monitoring scheme, which are: its ability to monitor SLAs violations in an efficient way and the ability to calculate new reputation value that will be assigned to the cloud provider according to QoS.

5.5 Chapter Summary

Establishing and monitoring SLA violations in real-time has become a critical issue for cloud computing. In this chapter, we introduced the concept of trusted third party in the CC. Then, we presented a brief overview of violations types and penalties. Also, we presented the principal parameters for Service Level Agreements in CC. We discussed the Autonomic Computing paradigm and maturity levels. We proposed an autonomic SLA monitoring framework managed by TTP composed of two modules: (1) SLA establishment; and (2) dynamic service monitoring module. The proposed framework is already integrated into a real cloud based on the Apache CloudStack platform. Its experimental evaluation on a real case study showed that our approach can monitor cloud environments efficiently. Results confirmed that our framework manages SLAs in an efficient way, as it detects all violations to be communicated to concerned parties, and calculated new reputation value that will be assigned to the cloud provider according to QoS. This work remains a key issue to properly solve the trade-off between the benefit of the supplier and the customer satisfaction.

Chapter 6

Conclusion and Future Work

The revolutionary technology of Cloud Computing offers a scalable and flexible paradigm where infrastructure, platform, and software are offered to users in the form of services. The management of such environment is inherently complex due to the large-scale number and heterogeneity of resources and the increasing number and types of services the cloud must deliver over the Internet. However, due to the highly dynamic nature of this open environment, it is important to have a binding agreement between all the service parties for ensuring trust while fulfilling the expected QoS. The provisioning of these computing services by cloud providers are regulated by Service Level Agreements, which present an important element that provides some degree of assurance. In this context, SLA contract form an appropriate solution to specify these QoS guarantees. It specifies one or more service level objectives (SLO), to guarantee that the delivered QoS satisfies the consumer expectations. Combined with a rapidly increasing number of cloud-based services, this complexity generated a pressing request for effective SLA contract establishment and efficient monitoring solutions. To properly operate and manage such complex situations, an efficient method for automation of SLA contract establishment process, and an autonomic SLA monitoring mechanisms managed by a trusted third party for real time ensuring accountability and QoS in cloud services are crucial.

In this context, the work conducted in this research had as an objective to propose an efficient method for automation of SLA contract establishment process, and to provide an autonomic SLA monitoring mechanisms managed by a trusted third party for real time ensuring accountability and QoS in cloud services. To this end, our contributions to the field of Service Level Agreement focus on the areas of Model Driven Engineering, autonomic computing paradigm and trusted third party. This research aimed to investigate the hypothesis described in section 1.5, specifically:

Can MDE principles and tools support the precise modeling of cloud computing SLAs contracts in such a way that cloud stakeholders can define their offers and demands? In addition, can the MDE principles and tools enable automated SLA establishment process for cloud computing SLAs, in order to provide a good support for automated SLA generation and management?

Can the IBM Autonomic Computing paradigm enrich the cloud computing specification to have autonomic mechanisms that can monitor themselves and maintain a target behavior? The hypothesis detailed above is investigated by defining the research objectives presented in Section 1.6.

1. To provide mechanisms for modeling cloud SLAs contract, by using Model Driven Engineering principles, techniques and tools.
2. To automate the SLA contract establishment process and describe the SLA terms between cloud providers and cloud consumers.
3. To present a cloud SLA contract language to define SLAs specifically devised to the cloud computing domain.
4. To propose a Novel Penalty Model for managing and applying penalties in the cloud computing.
5. To provide an autonomic SLA monitoring mechanisms following the IBM Autonomic Computing (MAPE-K loop) paradigm to dynamically monitor QoS metrics by detecting SLA violations at runtime to verify compliances for the respective SLAs.
6. The proposed autonomic framework is operated and controlled by a trusted third party, i.e. a mutually trusted cloud broker, that guarantees transparency and symmetry with respect to the SLA communication between prospective signatories. Also, the participation of the TTP is necessary in order to resolve conflicts between prospective signatories, likewise to monitor SLA violations in real-time in the goal to ensure online monitoring cloud services and provide better than best-effort behavior for clouds.
7. To evaluate the above framework with a real experimental testing in a cloud environment as realistically as possible.

The following sections summarize the contributions of this research regarding the thesis hypothesis and research objective. Then, the limitations of the proposed approach and future work are discussed in the following sections.

6.1 Contributions

The contributions of the thesis are summarized in this section.

Firstly, for automating SLAs contract establishment, we investigated this issue and we proposed our conception by using MDE (Model Driven Engineering) to express the SLA contract requirements, as a mean for establishing service level agreements between a cloud provider and cloud customer in the context of a particular service provision. The cloud SLA contract language we presented is a meta-model for defining cloud SLAs specifically devised to the cloud computing domain, that support most of the characteristics of cloud. We evaluated the completeness and clarity of our SLA contract establishment for Conceptual Modeling with Bunge-Wand-Weber representational theory. The results indicated that, our meta-model is capable of achieving a satisfactory completeness and clarity, which is very promising to accurately describe the structure and behavior of the real world. Finally, we validated our meta-model to examines the ability of the proposed SLA establishment approach to generate the SLA document in less time.

Secondly, we proposed a Novel Penalty Model for calculating the penalty cost of the violations, that can efficiently help for managing and applying penalties in the cloud computing.

Lastly, we proposed an autonomic SLA monitoring framework managed by trusted third party composed of two modules: the first one SLA establishment module, which aims at providing support for automated SLA generation and management. The second one, a service monitoring module to dynamically monitor QoS metrics by detecting SLA violations at runtime to verify compliances for the respective SLAs, and to proposed a mechanism for an adaptive remedy rectification, as a contribution at the third maturity level of the autonomic computing paradigm as defined by IBM. The framework is validated with scenarios on response time and availability, the results obtained are promising. They confirmed that our framework manages SLAs in an efficient way, as it detects all violations to be communicated to concerned parties, and identifies particular penalty clauses that can be used to modify the reputation of a provider over time. The trusted third party framework equipped with such reputation module can provide real-time assessment for consumers informed decision making to continue using a service or to migrate to another service provider in the case of service degradation. This creates a fair competitiveness between providers and hence improves service performance and the reliability in the cloud.

6.2 Limitations of the cloud SLAs contract approach and autonomic monitoring framework

6.2.1 Lack of support for SLA negotiation

The consumers compete to attain a service with desired functional and nonfunctional requirements. There is also a competition space among the providers to sell their services to the consumers. Since there are a dynamic competition market and conflicting requirements between consumer and provider, it is essential that automated negotiation mechanisms be proposed to reach an agreement and provide maximum utility for both trading parties. The overall intention of our approach is to automate the establishment SLA contract process between cloud consumer requirements and cloud provider offers. But, the proposed approach does not support the negotiation strategy for cloud service negotiation problem which can efficiently match the requirements of consumers and providers offers. Therefore, automating the negotiation strategy of the cloud SLA management process is an interesting area for future work.

6.2.2 Lack of support All steps of the MAPE-K loop control cycle

In this thesis, our second main objective is to propose an efficient autonomic SLA monitoring mechanisms managed by a trusted third party for real time ensuring accountability and QoS in cloud services.

Our proposed autonomic SLA monitoring framework is actually an instance of an autonomic control cycle: but just support the monitoring and analysis steps whereby systems can monitor themselves and maintain a target behavior. The planning and executing steps of the MAPE-K loop will have to be explored in details in our remains future work, particularly when several adaptation mechanisms will have to be considered.

6.3 Future work

6.3.1 Self-establishing a Service Level Agreement within Autonomic Cloud Computing Environment

The proposed approach defines an autonomic SLA establishment framework, for establishing functional and nonfunctional QoS parameters between a Cloud Service consumer and a Cloud Service Provider, which ensures a self-establishing of the proposed SLA using autonomic

cloud managers. Several research works have been proposed for the SLA establishing in cloud computing, but not autonomic SLA establishment. Therefore, autonomic SLA establishment of the cloud SLA management process is an interesting area for future work. This can be done, by for example, following the IBM Autonomic Computing (AC) paradigm which aims at equipping computer systems with capabilities to autonomously adapt their behavior and structure according to dynamic operating conditions. AC is inspired by the functions of the human nervous system and is aimed at designing and building systems that are self-managing. Autonomic systems are being adopted for self-management of complex large scale distributed system that have become manually unmanageable. The vision who will be explored in this approach is to enrich the SLA specification, with the aim to provide it with autonomic and dynamic establishment capabilities using IBM Autonomic Computing. Finally, we intend to enable the established SLA self management using the MAPE-K control loop.

6.3.2 Support for automated negotiation strategy for cloud service negotiation problem

The negotiation is an important phase of the SLA life cycle and must be fully considered. The proposed approach defines a Novel automated negotiation strategy for cloud service negotiation problem which efficiently match the requirements of consumers and providers offers. Therefore, automating the negotiation strategy of the cloud SLA management process is an interesting area for future work. This can be done, by investigates frameworks, techniques, protocols, and strategies proposed by the Cloud Computing Service Negotiation researches.

6.3.3 Extend CDOSim simulator features to simulate response times and SLA violations

Recent years have seen a significant increase in cloud computing simulation tools. Simulation tools are becoming more and more important in the evaluation of the cloud computing model. They are essential to evaluate cloud computing services and applications. Indeed, they can help to test, evaluate, and replicate the assumptions cost-effectively before a cloud platform is deployed in a real environment. It also provides an option to change and control the bottlenecks of the performance prior to implementing on an actual cloud platform. The simulation tools for cloud computing are also important to help users choose which cloud computing service is suitable for their requirements as deploying and operating a cloud

platform in a real environment are often expensive, time consuming, or even impractical due to multiple test runs in different conditions and requirements.

We think that it will be interesting to extend CDOSim to support three new entities. The first one is a Cloud service consumer entity and the second is a Cloud Broker (trusted third party) entity, instead of DatacenterBroker entity. In addition, we can add a Cloud provider entity that interconnects the Broker TTP, CSPs (DC) and CSC using BRITE topology for modeling link bandwidth and latencies. The use of Sensor class would be adaptable for monitoring resources and specifying thresholds and the SimEvent class for specifying notifications. Therefore, the control loop algorithms at each entity constitute the MAPE-K IBM Autonomic Computing to enable the self management.

Bibliography

- [Act] *ActiveMQ. Messaging and integration pattern provider.*
- [eps] *The epsilon Book.*
- [ESP] *ESPER. Event stream processing.*
- [Hib] *Hibernate. Relational persistence for java and .net.*
- [JBo] *JBoss. JBoss Drools Expert Rule Engine.*
- [Jes] *Jess. Jess Rule Engine.*
- [uml] *Unified Modeling Language.*
- [8] (2013a). The network simulator—ns-2. <http://www.isi.edu/nsnam/ns/>.
- [9] (2013b). Oct. the open cloud testbed. <http://www.opencloudconsortium.org>.
- [10] (2013c). Rackspace. <http://www.rackspace.com/cloud/>.
- [11] (2013d). Sla@soi. <http://www.sla-at-soi.eu/>.
- [12] (2016). Epsilon. <http://www.eclipse.org/epsilon/>.
- [13] (2017a). Eclipse modeling framework project (emf). <http://www.eclipse.org/modeling/emf/>.
- [14] (2017b). Graphical modeling project (gmp). <http://www.eclipse.org/modeling/gmp/>.
- [15] (2017c). Omg’s metaobject facility. <http://www.omg.org/mof/>.
- [16] Agedal, J. and Ecklund, E. (2002). Modelling qos: towards a uml profile. *UML 2002—The Unified Modeling Language*, pages 65–75.
- [17] Al Falasi, A. and Serhani, M. A. (2011). A framework for sla-based cloud services verification and composition. In *Innovations in Information Technology (IIT), 2011 International Conference on*, pages 287–292. IEEE.
- [18] Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., and Xu, M. (2007). Web services agreement specification (ws-agreement). In *Open Grid Forum*, volume 128.

- [19] Anithakumari, S. and Sekaran, K. C. (2014). Autonomic sla management in cloud computing services. In *Recent Trends in Computer Networks and Distributed Systems Security*, pages 151–159. Springer.
- [20] Bardsiri, A. K. and Hashemi, S. M. (2014). Qos metrics for cloud computing services evaluation. *International Journal of Intelligent Systems and Applications (IJISA)*, 6(12):27.
- [21] Becker, M., Borrisov, N., Deora, V., Rana, O. F., and Neumann, D. (2008). Using k-pricing for penalty calculation in grid market. In *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, pages 97–97. IEEE.
- [22] Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.
- [23] Bézivin, J. (2005). On the unification power of models. *Software & Systems Modeling*, 4(2):171–188.
- [24] Bézivin, J., Bouzitouna, S., Del Fabro, M. D., Gervais, M.-P., Jouault, F., Kolovos, D., Kurtev, I., and Paige, R. F. (2006). A canonical scheme for model composition. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 346–360. Springer.
- [25] Blythe, J., Deelman, E., and Gil, Y. (2004). Automatically composed workflows for grid environments. *Intelligent Systems, IEEE*, 19(4):16–23.
- [26] Bonell, M. J. (1996). Unidroit principles of international commercial contracts and the principles of european contract law: Similar rules for the same purposes, the. *Unif. L. Rev.*, 1:229.
- [27] Boukadi, K., Grati, R., and Ben-Abdallah, H. (2015). Toward the automation of a qos-driven sla establishment in the cloud. *Service Oriented Computing and Applications*, pages 1–24.
- [28] Brandic, I., Music, D., and Dustdar, S. (2009). Service mediation and negotiation bootstrapping as first achievements towards self-adaptable grid and cloud services. In *Proceedings of the 6th international conference industry session on Grids meets autonomic computing*, pages 1–8. ACM.
- [29] Brandic, I., Venugopal, S., Mattess, M., and Buyya, R. (2008). Towards a meta-negotiation architecture for sla-aware grid services. In *Workshop on Service-Oriented Engineering and Optimizations*, pages 17–20. Citeseer.
- [30] Brown, A. (2004). An introduction to model driven architecture.
- [31] Brownsword, L. L., Carney, D. J., Fisher, D., Lewis, G., and Meyers, C. (2004). Current perspectives on interoperability. Technical report, DTIC Document.
- [32] Bruel, J.-M. and France, R. B. (1998). Transforming uml models to formal specifications. *The Unified Modeling Language, UML*.
- [33] Brun, C. and Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34.

- [34] Burete, R., Bădică, A., and Bădică, C. (2010). Reputation model with forgiveness factor for semi-competitive e-business agent societies. In *International Conference on Networked Digital Technologies*, pages 402–416. Springer.
- [35] Buyya, R., Ranjan, R., and Calheiros, R. N. (2009a). Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 1–11. IEEE.
- [36] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009b). Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616.
- [37] Calheiros, R. N., Netto, M. A., De Rose, C. A., and Buyya, R. (2013). Emusim: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. *Software: Practice and Experience*, 43(5):595–612.
- [38] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., and Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50.
- [39] Canfora, G., Di Penta, M., Esposito, R., and Villani, M. L. (2005). An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1069–1075. ACM.
- [40] Carpentier, J., Gelas, J., Lefevre, L., Morel, M., Mornard, O., and Laisne, J. (2012). Compatibleone: Designing an energy efficient open source cloud broker. In *Cloud and Green Computing (CGC), 2012 Second International Conference on*, pages 199–205. IEEE.
- [41] Cascella, R. G., Blasi, L., Jegou, Y., Coppola, M., and Morin, C. (2013). *Contrail: Distributed application deployment under SLA in federated heterogeneous clouds*. Springer.
- [42] Chu, X., Nadiminti, K., Jin, C., Venugopal, S., and Buyya, R. (2007). Aneka: Next-generation enterprise grid platform for e-science and e-business applications. In *e-Science and Grid Computing, IEEE International Conference on*, pages 151–159. IEEE.
- [43] Cicchetti, A., Di Ruscio, D., and Pierantonio, A. (2007). A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185.
- [44] Clark, T., Sammut, P., and Willans, J. (2008). Applied metamodeling: a foundation for language driven development.
- [45] Cloudstack, A. (2014). Apache cloudstack: Open source cloud computing. *Acessado em*, 18(06).
- [46] Comarch, S. (2008). Definition and evaluation of penalty functions in sla management framework.

- [47] Commerce, B. E., Jøsang, A., and Ismail, R. (2002). The beta reputation system. In *In Proceedings of the 15th Bled Electronic Commerce Conference*. Citeseer.
- [48] Comuzzi, M., Kotsokalis, C., Spanoudakis, G., and Yahyapou, R. (2009). Establishing and monitoring slas in complex service based systems. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 783–790. IEEE.
- [49] Council, C. S. (2012). Practical guide to cloud service level agreements version 1.0.
- [50] Czarnecki, K., Eisenecker, U. W., Goos, G., Hartmanis, J., and van Leeuwen, J. (2000). Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, 15.
- [51] Davies, I. G., Green, P. F., Milton, S., and Rosemann, M. (2003). Using meta models for the comparison of ontologies.
- [52] Dinesh, V. (2004). Supporting service level agreements on ip networks. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, volume 92, pages 1382–1388.
- [53] Emeakaroha, V. C., Calheiros, R. N., Netto, M. A., Brandic, I., and De Rose, C. A. (2010). Desvi: an architecture for detecting sla violations in cloud computing infrastructures. In *Proceedings of the 2nd international ICST conference on Cloud computing (CloudComp'10)*. Citeseer.
- [54] Emeakaroha, V. C., Ferreto, T. C., Netto, M. A. S., Brandic, I., and De Rose, C. A. (2012a). Casvid: Application level monitoring for sla violation detection in clouds. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 499–508. IEEE.
- [55] Emeakaroha, V. C., Netto, M. A., Calheiros, R. N., Brandic, I., Buyya, R., and De Rose, C. A. (2012b). Towards autonomic detection of sla violations in cloud infrastructures. *Future Generation Computer Systems*, 28(7):1017–1029.
- [56] Fakhfakh, K., Chaari, T., Tazi, S., Jmaiel, M., and Drira, K. (2010). Odace sla: Ontology driven approach for automatic establishment of service level agreements. *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, 1(3):1–20.
- [57] Falkenberg, E., Hesse, W., Lindgreen, P., Nilsson, B., Oei, J., Rolland, C., Stamper, R., Assche, F., Verrijn-Stuart, A., and Voss, K. (1998). A framework of information systems concepts. the frisco report, ifip wg 8.1 task group frisco (web edition).
- [58] Fittkau, F., Frey, S., and Hasselbring, W. (2012). Cdosim: Simulating cloud deployment options for software migration support. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*, pages 37–46. IEEE.
- [59] Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., and Stoica, I. (2009). Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009.

- [60] France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society.
- [61] Frey, S., Hasselbring, W., and Schnoor, B. (2013). Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software: Evolution and Process*, 25(10):1089–1115.
- [62] Galster, M. and Bucherer, E. (2008). A taxonomy for identifying and specifying non-functional requirements in service-oriented development. In *Services-Part I, 2008. IEEE Congress on*, pages 345–352. IEEE.
- [63] Garg, S. K. and Buyya, R. (2011). Networkcloudsim: Modelling parallel applications in cloud simulations. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 105–113. IEEE.
- [64] Gong, Y., Dong, F., Li, W., and Xu, Z. (2003). Vega infrastructure for resource discovery in grids. *Journal of Computer Science and Technology*, 18(4):413–422.
- [65] Grati, R., Boukadi, K., and Ben-Abdallah, H. (2012). A qos monitoring framework for composite web services in the cloud. In *In The Sixth International Conference on Advanced Engineering Computing and Applications in Sciences (Advcomp12)*.
- [66] Gronback, R. (2009). Eclipse modeling project: a domain-specific language toolkit. the eclipse series.
- [67] Grossman, R., Gu, Y., Sabala, M., Bennet, C., Seidman, J., and Mambratti, J. (2009). The open cloud testbed: A wide area testbed for cloud computing utilizing high performance network services. *arXiv preprint arXiv:0907.4810*.
- [68] Hailpern, B. and Tarr, P. (2006). Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451–461.
- [69] Hammadi, A. M. and Hussain, O. (2012). A framework for sla assurance in cloud computing. In *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 393–398. IEEE.
- [70] Hauser, R., Steiner, M., and Waidner, M. (1996). *Micro-payments based on iKP*. Citeseer.
- [71] Hiles, A. N. (1994). Service level agreements: Panacea or pain? *The TQM Magazine*, 6(2):14–16.
- [72] Indulska, M., Recker, J. C., Rosemann, M., and Green, P. (2008). Representational deficiency of process modelling languages: measures and implications.
- [73] Jararweh, Y., Alshara, Z., Jarrah, M., Kharbutli, M., and Alsaleh, M. N. (2013). Teach-cloud: a cloud computing educational toolkit. *International Journal of Cloud Computing*, 2(2):237–257.
- [74] Jézéquel, J.-M. (2007). Modeling and aspect weaving. In *Methods for Modelling Software Systems (MMOSS)*. Internationales Begegnungs-und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

- [75] Jézéquel, J.-M., Combemale, B., and Vojtisek, D. (2012). *Ingénierie Dirigée par les Modèles: des concepts à la pratique...* Ellipses.
- [76] Jøsang, A., Hayward, R., and Pope, S. (2006). Trust network analysis with subjective logic. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, pages 85–94. Australian Computer Society, Inc.
- [77] Josang, A. and Ismail, R. (2002). The beta reputation system. In *Proceedings of the 15th bled electronic commerce conference*, volume 5, pages 2502–2511.
- [78] Jurca, R. and Faltings, B. (2005). Reputation-based service level agreements for web services. In *Service-Oriented Computing-ICSOC 2005*, pages 396–409. Springer.
- [79] Kamateri, E., Loutas, N., Zeginis, D., Ahtes, J., D’Andria, F., Bocconi, S., Gouvas, P., Ledakis, G., Ravagli, F., Lobunets, O., et al. (2013). Cloud4soa: A semantic-interoperability paas solution for multi-cloud platform management and portability. In *Service-Oriented and Cloud Computing*, pages 64–78. Springer.
- [80] Kamvar, S. D., Schlosser, M. T., and Garcia-Molina, H. (2003). The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 640–651. ACM.
- [81] Karaenke, P. and Kirn, S. (2007). Service level agreements: An evaluation from a business application perspective. In *Proceedings eChallenges e-2007 Conference*, volume 44.
- [82] Kearney, K. T., Torelli, F., and Kotsokalis, C. (2010). Sla*: An abstract syntax for service level agreements. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 217–224. IEEE.
- [83] Keller, A. and Ludwig, H. (2003). The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11(1):57–81.
- [84] Khazalah, F., Malik, Z., and Rezgui, A. (2015). Automated conflict resolution in collaborative data sharing systems using community feedbacks. *Information Sciences*, 298:407–424.
- [85] Kleppe, A. G., Warmer, J. B., and Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional.
- [86] Kliazovich, D., Bouvry, P., and Khan, S. U. (2012). Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283.
- [87] Kolovos, D. (2008). *An extensible platform for specification of integrated languages for model management*. University of York.
- [88] Kolovos, D., Rose, L., Garca-Domnguez, A., and Paige, R. (2011). The epsilon book. online.

- [89] Kolovos, D. S. (2009). Establishing correspondences between models with the epsilon comparison language. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 146–157. Springer.
- [90] Kolovos, D. S., Di Ruscio, D., Pierantonio, A., and Paige, R. F. (2009). Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, pages 1–6. IEEE.
- [91] Kolovos, D. S., Paige, R. F., and Polack, F. A. (2006a). The epsilon object language (eol). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer.
- [92] Kolovos, D. S., Paige, R. F., and Polack, F. A. (2006b). Merging models with the epsilon merging language (eml). In *International Conference on Model Driven Engineering Languages and Systems*, pages 215–229. Springer.
- [93] Kolovos, D. S., Paige, R. F., and Polack, F. A. (2006c). Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 13–20. ACM.
- [94] Kolovos, D. S., Paige, R. F., and Polack, F. A. (2008). The epsilon transformation language. In *International Conference on Theory and Practice of Model Transformations*, pages 46–60. Springer.
- [95] Koster, V. (2007). Implementation and integration of a domain specific language with oaw and xtext. *MT AG, Ratingen*.
- [96] Kouki, Y. (2013). *Approche dirigée par les contrats de niveaux de service pour la gestion de l'élasticité du 'nuage'*. PhD thesis, Ecole des Mines de Nantes.
- [97] Kouki, Y., Oliveira, F. A. d., Dupont, S., and Ledoux, T. (2014). A language support for cloud elasticity management. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 206–215. IEEE.
- [98] Kritikos, K. and Plexousakis, D. (2009). Requirements for qos-based web service description and discovery. *IEEE Transactions on Services Computing*, 2(4):320–337.
- [99] Kuyoro Shade, O., Frank, I., Awodele, O., and Okolie Samuel, O. (2012). Quality of service (qos) issues in web services. *IJCSNS International Journal of Computer Science and Network Security*, 12(1):94–97.
- [100] Lamanna, D. D., Skene, J., and Emmerich, W. (2003). Slang: a language for service level agreements.
- [101] Ledoux, T. and Kouki, Y. (2012). Sla-driven capacity planning for cloud applications. In *Proceedings of the 2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 135–140. IEEE Computer Society.
- [102] Lee, H.-J., Kim, M.-S., Hong, J. W., and Lee, G.-H. (2002). Qos parameters to network performance metrics mapping for sla monitoring. *KNOM Review*, 5(2):42–53.

- [103] Leff, A., Rayfield, J. T., and Dias, D. M. (2003). Service-level agreements and commercial grids. *IEEE Internet Computing*, 7(4):44–50.
- [104] Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., and Leaf, D. (2011a). Nist cloud computing reference architecture. *NIST special publication*, 500:292.
- [105] Liu, Y., Zhang, J., and Zhu, Q. (2011b). Design of a reputation system based on dynamic coalition formation. In *International Conference on Social Informatics*, pages 135–144. Springer.
- [106] Loyall, J. P., Schantz, R. E., Zinky, J. A., and Bakken, D. E. (1998). Specifying and measuring quality of service in distributed object systems. In *Object-Oriented Real-time Distributed Computing, 1998.(ISORC 98) Proceedings. 1998 First International Symposium on*, pages 43–52. IEEE.
- [107] Lu, K. (2015). *Planning and Optimization During the Life-Cycle of Service Level Agreements for Cloud Computing*. PhD thesis, Göttingen, Georg-August Universität, Diss., 2015.
- [108] Ludewig, J. (2003). Models in software engineering—an introduction. *Software and Systems Modeling*, 2(1):5–14.
- [109] Ludwig, H., Keller, A., Dan, A., King, R. P., and Franck, R. (2003). Web service level agreement (wsa) language specification. *IBM Corporation*, pages 815–824.
- [110] Maarouf, A., El Hamlaoui, M., Marzouk, A., and Haqiq, A. (2015a). Mde approach for the establishment of a service level agreements monitoring by trusted third party in the cloud computing. *International Journal of Grid and High Performance Computing (IJGHPC)*, 7(4):1–20.
- [111] Maarouf, A., Marzouk, A., and Haqiq, A. (2014a). Automatic control of the quality of service contract by a third party in the cloud computing. In *Complex Systems (WCCS), 2014 Second World Conference on*, pages 599–603.
- [112] Maarouf, A., Marzouk, A., and Haqiq, A. (2015b). Practical modeling of the sla life cycle in cloud computing. In *Intelligent Systems Design and Applications (ISDA), 2015 15th International Conference on*, pages 52–58. IEEE.
- [113] Maarouf, A., Marzouk, A., and Haqiq, A. (2015c). A review of sla specification languages in the cloud computing. In *Intelligent Systems: Theories and Applications (SITA), 2015 10th International Conference on*, pages 1–6. IEEE.
- [114] Maarouf, A., Marzouk, A., and Haqiq, A. (2015d). Towards a trusted third party based on multi-agent systems for automatic control of the quality of service contract in the cloud computing. In *Electrical and Information Technologies (ICEIT), 2015 International Conference on*, pages 311–315. IEEE.
- [115] Maarouf, A., Marzouk, A., Haqiq, A., and El Hamlaoui, M. (2014b). Towards a mde approach for the establishment of a contract service level monitoring by third party in the cloud computing. In *Proceedings of the 2014 Tenth International Conference on Signal-Image Technology and Internet-Based Systems, SITIS '14*, pages 715–720, Washington, DC, USA. IEEE Computer Society.

- [116] Mancini, E. P., Rak, M., and Villano, U. (2009). Perfcloud: Grid services for performance-oriented development of cloud computing applications. In *Enabling Technologies: Infrastructures for Collaborative Enterprises, 2009. WETICE'09. 18th IEEE International Workshops on*, pages 201–206. IEEE.
- [117] Mani, A. and Nagarajan, A. (2002). Understanding quality of service for web services. *IBM developerWorks*, 1.
- [118] Massa, P. and Avesani, P. (2004). Trust-aware collaborative filtering for recommender systems. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 492–508. Springer.
- [119] Megginson, D. et al. (2014). Sax : The simple api for xml. *SAX project*.
- [120] Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing.
- [121] Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. (2009). Comprehensive qos monitoring of web services and event-based sla violation detection. In *Proceedings of the 4th international workshop on middleware for service oriented computing*, pages 1–6. ACM.
- [122] Mifrah, Y., En-Nouaary, A., and Dahchour, M. (2017). Toward a practical method for introducing and evaluating trust learning models in open multi-agent systems. In *Advances in Ubiquitous Networking 2*, pages 573–586. Springer.
- [123] Müller, C., Oriol, M., Franch, X., Marco, J., Resinas, M., Ruiz-Cortés, A., and Rodríguez, M. (2014). Comprehensive explanation of sla violations at runtime. *IEEE Transactions on Services Computing*, 7(2):168–183.
- [124] Musset, J., Juliot, É., Lacrampe, S., Piers, W., Brun, C., Goubet, L., Lussaud, Y., and Allilaire, F. (2006). Acceleo user guide. See also <http://acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf>, 2.
- [125] Notation, B. P. M. (2011). Version 2.0. *OMG Final Adopted Specification. Object Management Group*.
- [126] Nuñez, A., Vázquez-Poletti, J. L., Caminero, A. C., Carretero, J., and Llorente, I. M. (2011). Design of a new cloud computing simulation platform. In *Computational Science and Its Applications-ICCSA 2011*, pages 582–593. Springer.
- [127] Núñez, A., Vázquez-Poletti, J. L., Caminero, A. C., Castañé, G. G., Carretero, J., and Llorente, I. M. (2012). icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209.
- [128] O'Brien, L., Merson, P., and Bass, L. (2007). Quality attributes for service-oriented architectures. In *Proceedings of the international Workshop on Systems Development in SOA Environments*, page 3. IEEE Computer Society.
- [129] Ostermann, S., Plankensteiner, K., Prodan, R., and Fahringer, T. (2011). Groudsim: an event-based simulation framework for computational grids and clouds. In *Euro-Par 2010 Parallel Processing Workshops*, pages 305–313. Springer.

- [130] O'Sullivan, J., Edmond, D., and ter Hofstede, A. H. (2005). Formal description of non-functional service properties. *Centre for Information Technology, Queensland University of Technology, Tech. Rep.*
- [131] Paige, R. F., Brooke, P. J., and Ostroff, J. S. (2007). Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(3):11.
- [132] Papazoglou, M. (2008). *Web services: principles and technology*. Pearson Education.
- [133] Paschke, A. and Schnappinger-Gerull, E. (2006). A categorization scheme for sla metrics. *Service Oriented Electronic Commerce*, 80(25-40):14.
- [134] Patel, J., Teacy, W. L., Jennings, N. R., and Luck, M. (2005). A probabilistic trust model for handling inaccurate reputation sources. In *International Conference on Trust Management*, pages 193–209. Springer.
- [135] Patel, P., Ranabahu, A. H., and Sheth, A. P. (2009). Service level agreement in cloud computing.
- [136] Petcu, D., Di Martino, B., Venticinque, S., Rak, M., Máhr, T., Lopez, G. E., Brito, F., Cossu, R., Stopar, M., Šperka, S., et al. (2013). Experiences in building a mosaic of clouds. *Journal of Cloud Computing*, 2(1):1–22.
- [137] Pinyol, I. and Sabater-Mir, J. (2013). Computational trust and reputation models for open multi-agent systems: a review. *Artificial Intelligence Review*, 40(1):1–25.
- [138] POLEMI, D. K. (2003). Reshen, a best practice approach for secure healthcare networks in europe. *Advanced Health Telematics and Telemedicine: The Magdeburg Expert Summit Textbook*, 96:51.
- [139] Pranggono, B., Alboaneen, D., and Tianfield, H. (2014). 11 simulation tools for cloud computing. *Simulation Technologies in Networking and Communications: Selecting the Best Tool for the Test*, page 311.
- [140] Rahimi, H. and El Bakkali, H. (2013). A new reputation algorithm for evaluating trustworthiness in e-commerce context. In *Security Days (JNS3), 2013 National*, pages 1–6. IEEE.
- [141] Rak, M., Venticinque, S., Echevarria, G., Esnal, G., et al. (2011). Cloud application monitoring: The mosaic approach. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 758–763. IEEE.
- [142] Rana, O., Warnier, M., Quillinan, T. B., and Brazier, F. (2008a). Monitoring and reputation mechanisms for service level agreements. In *Grid Economics and Business Models*, pages 125–139. Springer.
- [143] Rana, O. F., Warnier, M., Quillinan, T. B., Brazier, F., and Cojocarasu, D. (2008b). Managing violations in service level agreements. In *Grid Middleware and Services*, pages 349–358. Springer.

- [144] Recker, J., Rosemann, M., Green, P. F., and Indulska, M. (2011). Do ontological deficiencies in modeling grammars matter? *MIS Quarterly*, 35(1):57–79.
- [145] Recker, J., Rosemann, M., Indulska, M., and Green, P. (2009). Business process modeling-a comparative analysis. *Journal of the Association for Information Systems*, 10(4):1.
- [146] Resnick, P., Kuwabara, K., Zeckhauser, R., and Friedman, E. (2000). Reputation systems. *Communications of the ACM*, 43(12):45–48.
- [147] Ron, S. and Aliko, P. (2001). Service level agreements. *Internet NG project*.
- [148] Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. (2008). The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 1–16. Springer.
- [149] Rosemann, M. and van der Aalst, W. M. (2007). A configurable reference modelling language. *Information Systems*, 32(1):1–23.
- [150] Sabater, J., Paolucci, M., and Conte, R. (2006). Repage: Reputation and image among limited autonomous partners. *Journal of artificial societies and social simulation*, 9(2).
- [Salem et al.] Salem, H. S., El Yamany, H. F., and El-Tawel, G. S. Towards service level agreements engineering process in cloud computing.
- [152] Sathya, M., Swarnamugi, M., Dhavachelvan, P., and Sureshkumar, G. (2010). Evaluation of qos based web-service selection techniques for service composition. *International Journal of Software Engineering*, 1(5):73–90.
- [153] Schmidt, D. C. (2006). Guest editor’s introduction: Model-driven engineering. *Computer*, 39(2):0025–31.
- [154] Serrano, D., Bouchenak, S., Kouki, Y., Ledoux, T., Lejeune, J., Sopena, J., Arantes, L., and Sens, P. (2013). Towards qos-oriented sla guarantees for online cloud services. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 50–57. IEEE.
- [155] Shao, J., Wei, H., Wang, Q., and Mei, H. (2010). A runtime model based monitoring approach for cloud. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 313–320. IEEE.
- [156] Skalkowski, K., Sendor, J., Slota, R., and Kitowski, J. (2010). Application of the esb architecture for distributed monitoring of the sla requirements. In *Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on*, pages 203–210. IEEE.
- [157] Skene, J., Lamanna, D. D., and Emmerich, W. (2004). Precise service level agreements. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 179–188. IEEE.
- [158] Sotiriadis, S., Bessis, N., and Antonopoulos, N. (2013a). Towards inter-cloud simulation performance analysis: Exploring service-oriented benchmarks of clouds in simic. In *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, pages 765–771. IEEE.

- [159] Sotiriadis, S., Bessis, N., Antonopoulos, N., and Anjum, A. (2013b). Simic: Designing a new inter-cloud simulation platform for integrating large-scale resource management. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 90–97. IEEE.
- [160] Stamou, A. (2014). *Systematic Service Level Agreement (SLA) data management*. PhD thesis, University of Geneva.
- [161] Stamou, K., Aubert, J., Gateau, B., and Morin, J.-H. (2013). Preliminary requirements on trusted third parties for service transactions in cloud environments. In *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, pages 4976–4983. IEEE.
- [162] Stanoevska-Slabeva, K., Wozniak, T., and Ristol, S. (2009). *Grid and cloud computing: a business perspective on technology and applications*. Springer Science & Business Media.
- [163] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.
- [164] Steinberg, D., Budinsky, F., and Paternostro, M. (2009). Merks, emf: Eclipse modeling framework 2.0.
- [165] Sterritt, R., Parashar, M., Tianfield, H., and Unland, R. (2005). A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3):181–187.
- [166] Teacy, W. L., Patel, J., Jennings, N. R., and Luck, M. (2006). Travos: Trust and reputation in the context of inaccurate information sources. *Autonomous Agents and Multi-Agent Systems*, 12(2):183–198.
- [167] Teng, F., Yu, L., and Magoules, F. (2011). Simmapreduce: a simulator for modeling mapreduce framework. In *Multimedia and Ubiquitous Engineering (MUE), 2011 5th FTRA International Conference on*, pages 277–282. IEEE.
- [168] TID, L. M. C., Cossu, G., Gerola, M., Murphy, R. S., and ILB, S. M. (2013). D3. 7: Xifi infrastructure network adaptation mechanisms api v2.
- [169] Tighe, M., Keller, G., Bauer, M., and Lutfiyya, H. (2012). Dcsim: A data centre simulation tool for evaluating dynamic virtualized resource management. In *Network and service management (cnsm), 2012 8th international conference and 2012 workshop on systems virtualization management (svm)*, pages 385–392. IEEE.
- [170] Uriarte, R. B. (2015). Supporting autonomic management of clouds: Service-level-agreement, cloud monitoring and similarity learning.
- [171] Uriarte, R. B., Tiezzi, F., and Nicola, R. D. (2014). Slac: A formal service-level-agreement language for cloud computing. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 419–426. IEEE Computer Society.
- [172] Van Moorsel, A. (2001). Metrics for the internet age: Quality of experience and quality of business. In *Fifth International Workshop on Performability Modeling of Computer and Communication Systems, Arbeitsberichte des Instituts für Informatik, Universität Erlangen-Nürnberg, Germany*, volume 34, pages 26–31. Citeseer.

- [173] Victor, P., Cornelis, C., and De Cock, M. (2011). *Trust networks for recommender systems*, volume 4. Springer Science & Business Media.
- [174] Vision, A. C.-C. (2012). Parameters for service level agreements generation in cloud computing. In *Advances in Conceptual Modeling: ER 2012 Workshops CMS, ECDM-NoCoDA, MODIC, MORE-BI, RIGIM, SeCoGIS, WISM, Florence, Italy, October 15-18, 2012, Proceedings*, volume 7518, page 13. Springer.
- [175] Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2013). *Model-driven software development: technology, engineering, management*. John Wiley & Sons.
- [176] Wang, Q., Wang, C., Li, J., Ren, K., and Lou, W. (2009). Enabling public verifiability and data dynamics for storage security in cloud computing. In *Computer Security—ESORICS 2009*, pages 355–370. Springer.
- [177] Werner, J., Geronimo, G., Westphall, C., Koch, F., and Freitas, R. (2011). Simulator improvements to validate the green cloud computing approach. In *Network Operations and Management Symposium (LANOMS), 2011 7th Latin American*, pages 1–8. IEEE.
- [178] Wickremasinghe, B., Calheiros, R. N., and Buyya, R. (2010). Cloudanalyst: A cloudsims-based visual modeller for analysing cloud computing environments and applications. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 446–452. IEEE.
- [179] Wolter, K., van Moorsel, A., and Qoe, Q. (2001). The relationship between quality of service and business metrics: monitoring, notification and optimization. *Hewlett-Packard Labs Technical Report HPL-2001-96*.
- [180] Xiaoyong, Y., Ying, L., Tong, J., Tiancheng, L., and Zhonghai, W. (2015). An analysis on availability commitment and penalty in cloud sla. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 914–919. IEEE.
- [181] Ziegler, W., Jiang, M., and Konstanteli, K. (2011). Optimis sla framework and term languages for slas in cloud environment. *OPTIMIS Project Deliverable D, 2*.

List of publications

International Journals:

- Adil MAAROUF, Mahmoud EL HAMLAOUI, Abderrahim MARZOUK, Abdelkrim HAQIQ, "MDE Approach for the Establishment of a Service Level Agreements Monitoring by Trusted Third Party in the Cloud Computing," International Journal of Grid and High Performance Computing (IJGHPC), vol. 7, no. 4, pp. 1–20, 2015. Available from: DOI: 10.4018/IJGHPC.2015100101
- Adil MAAROUF, Bouchra EL QACIMY, Abderrahim MARZOUK, Abdelkrim HAQIQ, "Defining and Evaluating A Novel Penalty Model for Managing Violations in the Cloud Computing," International Journal of Grid and High Performance Computing (IJGHPC), vol. 9, no. 2, pp. 1–22, 2017. Available from: DOI: 10.4018/IJGHPC.2017040103
- Adil MAAROUF, Abderrahim MARZOUK, Abdelkrim HAQIQ, "An Autonomic SLA Monitoring Framework managed by trusted third party in the Cloud Computing," International Journal of Cloud Applications and Computing (IJCAC), "under publication" 2017.

International conferences:

- Adil MAAROUF, Mahmoud EL HAMLAOUI, Abderrahim MARZOUK, Abdelkrim HAQIQ, "Towards a MDE approach for the establishment of a contract service level monitoring by third party in the cloud computing," in Proceedings of the 2014 Tenth International Conference on Signal-Image Technology and Internet-Based Systems, ser. SITIS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 715–720. [Online]. Available: <http://dx.doi.org/10.1109/SITIS.2014.30>
- Adil MAAROUF, Abderrahim MARZOUK, Abdelkrim HAQIQ, "Automatic control of the quality of service contract by a third party in the cloud computing," in Complex Systems (WCCS), 2014 Second World Conference on, Nov 2014, pp. 599–603. Available from: DOI: 10.1109/ICoCS.2014.7060961
- Adil MAAROUF, Bouchra EL QACIMY, Abderrahim MARZOUK, Abdelkrim HAQIQ, "A novel penalty model for managing and applying penalties in cloud computing," in Computer Systems and Applications (AICCSA), 2015 IEEE/ACS 12th International Conference of. IEEE, 2015, pp. 1–6. Available from: DOI: 10.1109/AICCSA.2015.7507243
- Adil MAAROUF, Abderrahim MARZOUK, Abdelkrim HAQIQ, "A review of SLA specification languages in the cloud computing," in Intelligent Systems: Theories and Applications (SITA), 2015 10th International Conference on. IEEE, 2015, pp. 1–6. Available from: DOI: 10.1109/SITA.2015.7358406

- Adil MAAROUF, Abderrahim MARZOUK, Abdelkrim HAQIQ, "Comparative study of simulators for cloud computing," in Cloud Technologies and Applications (CloudTech), 2015 International Conference on. IEEE, 2015, pp. 1–8. Available from: DOI: 10.1109/CloudTech.2015.7336989
- Adil MAAROUF, Abderrahim MARZOUK, Abdelkrim HAQIQ, "Practical modeling of the sla life cycle in cloud computing," in Intelligent Systems Design and Applications (ISDA), 2015 15th International Conference on. IEEE, 2015, pp. 52–58. Available from: DOI: 10.1109/ISDA.2015.7489170

National workshops:

- Adil MAAROUF, Abderrahim MARZOUK, Abdelkrim HAQIQ, " Ontological Evaluation of SLA Contract based on MDE for Conceptual Modeling," In the First Edition of International Workshop on New services and Networks WNSN'16 organised by The National School of Applied Sciences of Khouribga, Morocco (on the 23rd and 24th of May 2016).
- Adil MAAROUF, Abderrahim MARZOUK, Abdelkrim HAQIQ, " ASSURANCE DU CONTRAT DE NIVEAU DE SERVICE PAR UN PRESTATAIRE TIERS DANS LE CLOUD COMPUTING", JDTIC'14, ENSIAS, Rabat, Maroc.
- Adil MAAROUF, Abderrahim MARZOUK, Abdelkrim HAQIQ, " CONTROLE AUTOMATIQUE DU CONTRAT DE QUALITE DE SERVICE PAR UN PRESTATAIRE TIERS DANS LE CLOUD COMPUTING", JSIM'14, Université Ibn Tofail, Kénitra, Maroc.

Annexes: Source code of our Autonomic Framework

Annexe A: Autonomic SLA Monitoring Framework managed by a Trusted Third Party

This annexes contains source code of the simulation described in our published paper: An Autonomic SLA Monitoring Framework managed by trusted third party in the Cloud Computing.

Source code of: StartDemo–Launch of the program

```
package org.autonomic.monitoring.framework;
import org.autonomic.monitoring.framework.trustmodel.Feature;
import org.autonomic.monitoring.framework.trustmodel.ForgiveFactor;
import org.autonomic.monitoring.framework.util.CDataConstant;
import org.autonomic.monitoring.framework.util.RandomResponseTimeEventGenerator;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
/**
 * Entry point for the Demo. Run this from your IDE, or from the command line using 'mvn exe
 */
public class StartDemo {
    /** Logger */
    private static Logger LOG = LoggerFactory.getLogger(StartDemo.class);
    /**
     * Main method - start the Demo!
```

```

*/
public static void main(String[] args) throws Exception {
    LOG.debug("Starting...");
    CDataConstant.PROVIDER_REPUTATION = CDataConstant.PROVIDER_REPUTATION_INITIAL;
    LOG.debug("PROVIDER REPUTATION START SIMULATION = " + CDataConstant.PROVIDER_REPUTATION);
    long noOfResponseTimeEvents = 100;
    if (args.length == 1) {
        noOfResponseTimeEvents = Long.valueOf(args[0]);
    } else {
        LOG.debug("No override of number of events detected - defaulting to " + noOfResponseTimeEvents);
    }
    // Load spring configuration
    ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(new String[] {
        "beans.xml"
    });
    BeanFactory factory = (BeanFactory) appContext.getBeanFactory();
    Feature responseTime = Feature.getInstance("responseTime");
    responseTime.setId("responseTime");
    responseTime.setWidth(6);
    responseTime.setSLAValue(0.8f);
    Feature availability = Feature.getInstance("availability");
    availability.setId("availability");
    availability.setWidth(9);
    availability.setSLAValue(0.95f);
    // Start Demo
    RandomResponseTimeEventGenerator generator = (RandomResponseTimeEventGenerator) factory.getBean("generator");
    generator.startSendingResponseTimeReadings(noOfResponseTimeEvents);
}
}

```

Source code of: Declared Patterns for CRITICAL Violation with Esper (CEP) Monitoring Engine

```

**
* Wraps Esper Statement and Listener. No dependency on Esper libraries.
*/
@Component
public class CriticalEventSubscriber implements StatementSubscriber {

```

```
/** Logger */
private static Logger LOG = LoggerFactory.getLogger(CriticalEventSubscriber.class);
/**
 * If the last event in a critical sequence is this much greater than the first - issue a
 * critical alert.
 */
private static final String CRITICAL_EVENT_MULTIPLIER = "1.5";
/**
 * {@inheritDoc}
 */
public String getStatement() {

    // Example using 'Match Recognize' syntax.
    String criticalEventExpression = "select * from ResponseTimeEvent "
+ "match_recognize ( "
+ "    measures A as temp1, B as temp2, C as temp3, D as temp4 "
+ "    pattern (A B C D) "
+ "    define "
+ "        A as A.responseTime > " + CDataConstant.RESPONSE_TIME_THRESHOLD + ", "
+ "        B as B.responseTime > " + CDataConstant.RESPONSE_TIME_THRESHOLD + ", "
+ "        C as C.responseTime > " + CDataConstant.RESPONSE_TIME_THRESHOLD + ", "
+ "        D as D.responseTime > " + CDataConstant.RESPONSE_TIME_THRESHOLD + ") ";

    return criticalEventExpression;
}
/**
 * Listener method called when Esper has detected a pattern match.
 */
public void update(Map<String, ResponseTimeEvent> eventMap) {

    // 1st ResponseTime in the Critical Sequence
    ResponseTimeEvent temp1 = (ResponseTimeEvent) eventMap.get("temp1");
    // 2nd ResponseTime in the Critical Sequence
    ResponseTimeEvent temp2 = (ResponseTimeEvent) eventMap.get("temp2");
    // 3rd ResponseTime in the Critical Sequence
    ResponseTimeEvent temp3 = (ResponseTimeEvent) eventMap.get("temp3");
```

```
// 4th ResponseTime in the Critical Sequence
ResponseTimeEvent temp4 = (ResponseTimeEvent) eventMap.get("temp4");

StringBuilder sb = new StringBuilder();
sb.append("*****");
sb.append("\n* [ALERT] : CRITICAL VIOLATION DETECTED! ");
sb.append("\n* " + temp1 + " > " + CDataConstant.RESPONSE_TIME_THRESHOLD);
sb.append("\n* " + temp2 + " > " + CDataConstant.RESPONSE_TIME_THRESHOLD);
sb.append("\n* " + temp3 + " > " + CDataConstant.RESPONSE_TIME_THRESHOLD);
sb.append("\n* " + temp4 + " > " + CDataConstant.RESPONSE_TIME_THRESHOLD);
sb.append("\n*****");
CDataConstant.TOTAL_NUMBER_CRITICAL_EVENT ++;
LOG.debug(sb.toString());
}
}
```

Source code of: Data Constant

```
package org.autonomic.monitoring.framework.util;
import java.util.Date;
public class CDataConstant {
/** Used as the MAX RANDOM */
public static final int RESPONSE_TIME_MAX = 1500;

/** Used as the minimum starting threshold for a violation event. */
public static final int RESPONSE_TIME_THRESHOLD = 800;

/** If 2 consecutive responseTime events are greater than RESPONSE_TIME_THRESHOLD this - iss
public static final int RESPONSE_TIME_THRESHOLD_NOTIFICATION = 2;
/** If 3 consecutive responseTime events are greater than RESPONSE_TIME_THRESHOLD this - iss
public static final int RESPONSE_TIME_THRESHOLD_WARNING = 3;
/** If 4 consecutive responseTime events are greater than RESPONSE_TIME_THRESHOLD this - iss
public static final int RESPONSE_TIME_THRESHOLD_CRITICAL = 4;

public static final int TOTAL_NUMBER_NOTIFICATION_EVENT = 0;
public static final int TOTAL_NUMBER_WARNING_EVENT = 0;
public static int TOTAL_NUMBER_CRITICAL_EVENT = 0;
```

```

public static final double PROVIDER_REPUTATION_INITIAL = 0.8;
public static final double PROVIDER_REPUTATION_DECREASE_STEP = 0.1;
public static final double PROVIDER_REPUTATION_INCREASE_STEP = 0.1;
public static double PROVIDER_REPUTATION = 0;

/** Used as the minimum starting threshold for a FAILURE SERVICE. */
public static final int FAILURE_SERVICE_THRESHOLD = 1490;
public static int TOTAL_DURATION_OF_FAILURES = 0;
public static int TOTAL_NUMBER_OF_FAILURES = 0;
/** Used as the minimum starting threshold for a AVAILABILITY SERVICE. */
public static final Double AVAILABILITY_SERVICE_THRESHOLD = 98.00;
}

```

Source code of: Our Trust reputation mechanism

The new reputation value that will be assigned to the cloud provider is calculated by our trust reputation model as described before. The results of this reputation scores generated by the TTP is stored in the reputation database. At this step, the TTP presents to consumer a detailed dashboard that contains 'Real time' reporting with Alarm list (violations, penalties, value of reputation). Forgive Factor model.

```

package org.autonomic.monitoring.framework.trustmodel;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
/**
 * @author Adil.Maarouf
 */
@Component
@Scope(value = "singleton")
public class ForgiveFactor {
    private float forgivenessFactor = 0;
    public ForgiveFactor() {
        this.forgivenessFactor = 0.7f;
    }
}

```



```

}
@Autowired
private TrustKnowledge trustKnowledge;

public void updateTrustKnowledge(FeatureEvaluation evaluatedFeature) {
    List<FeatureEvaluation> passedEvaluationsOfCurrentFeature = getTrustKnowledge().getPassedEvaluationsOfCurrentFeature();
    if (passedEvaluationsOfCurrentFeature == null) {
        passedEvaluationsOfCurrentFeature = new ArrayList<FeatureEvaluation>();
        getTrustKnowledge().getPassedExperiences().put(evaluatedFeature.getFeature().getName(), passedEvaluationsOfCurrentFeature);
    }
    FeatureEvaluation currentEvaluationOfTheFeature = getTrustKnowledge().getEvaluation(evaluatedFeature.getFeature().getName());
    float updatedEvaluation = evaluatedFeature.getEvaluation();
    int nbrPassedEval = passedEvaluationsOfCurrentFeature.size();
    for (int i = 0; i != nbrPassedEval; ++i) {
        FeatureEvaluation f = passedEvaluationsOfCurrentFeature.get(i);
        updatedEvaluation += Math.pow(forgivenessFactor, nbrPassedEval - i) * f.getEvaluation();
    }
    updatedEvaluation *= (1 - forgivenessFactor);
    updatedEvaluation /= (1 - Math.pow(forgivenessFactor, nbrPassedEval+1));
    currentEvaluationOfTheFeature.setEvaluation(updatedEvaluation);
    currentEvaluationOfTheFeature.incrementUpdateCount();
    passedEvaluationsOfCurrentFeature.add(evaluatedFeature);
}

public String getTrustMetricName() {
    return "ForgiveFactor";
}

public TrustKnowledge getTrustKnowledge() {
    return trustKnowledge;
}

public void setTrustKnowledge(TrustKnowledge trustKnowledge) {
    this.trustKnowledge = trustKnowledge;
}
}

```

Source code of: Simulation Results–Dashboard

```
package org.autonomic.monitoring.framework.util;
```

```
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Date;
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import org.autonomic.monitoring.framework.event.ResponseTimeEvent;
import org.autonomic.monitoring.framework.handler.ResponseTimeEventHandler;
import org.autonomic.monitoring.framework.trustmodel.ForgiveFactor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
/**
 * Just a simple class to create a number of Random ResponseTimeEvents and pass them off to
 * ResponseTimeEventHandler.
 */
@Component
public class RandomResponseTimeEventGenerator {
    /** Logger */
    private static Logger LOG = LoggerFactory.getLogger(RandomResponseTimeEventGenerator.class);
    /** The ResponseTimeEventHandler - wraps the Esper engine and processes the Events */
    @Autowired
    private ResponseTimeEventHandler responseTimeEventHandler;
    @Autowired
    private ForgiveFactor forgiveFactorModel;
    /**
     * Creates simple random ResponseTime events and lets the implementation class handle them
     */
    public void startSendingResponseTimeReadings(final long noOfResponseTimeEvents) {
        ExecutorService xrayExecutor = Executors.newSingleThreadExecutor();
        xrayExecutor.submit(new Runnable() {
            public void run() {

                LOG.debug(getStartingMessage());
                int count = 0;
```

```

Date oStartDateOfTreatment = new Date();
while (count < noOfResponseTimeEvents) {
    int iResponseTime = new Random().nextInt(CDataConstant.RESPONSE_TIME_MAX);
    ResponseTimeEvent ve = new ResponseTimeEvent(iResponseTime, new Date());
    responseTimeEventHandler.handle(ve);
    count++;
    try {
        Thread.sleep(iResponseTime);
        // Thread.sleep(1);
    } catch (InterruptedException e) {
        LOG.error("Thread Interrupted", e);
    }
}
Date oEndDateOfTreatment = new Date();
LOG.debug("TOTAL NUMBER OF CRITICAL VIOLATION EVENT = " + CDataConstant.TOTAL_NUMBER_CRITI
if (CDataConstant.TOTAL_NUMBER_CRITICAL_EVENT == 0){
    // increase reputation
    CDataConstant.PROVIDER_REPUTATION = CDataConstant.PROVIDER_REPUTATION + CDataConstant.PRO
}else{
    // decrease reputation
    CDataConstant.PROVIDER_REPUTATION = CDataConstant.PROVIDER_REPUTATION - (CDataConstant.TC
}
// double dProviderReputation = CDataConstant.PROVIDER_REPUTATION < 0.0 ? 0: CData
double dProviderReputation =forgiveFactorModel.getTrustKnowledge().getTotalEvaluation();
LOG.debug("PROVIDER REPUTATION = " + dProviderReputation + "\n");
LOG.debug(forgiveFactorModel.getTrustKnowledge().toString());

//nombre de défaillances
LOG.debug("TOTAL NUMBER OF FAILURES = " + CDataConstant.TOTAL_NUMBER_OF_FAILURES);
// somme des temps de défaillance
LOG.debug("SUM OF FAILURE TIMES = " + CDataConstant.TOTAL_DURATION_OF_FAILURES);
// Somme des Temps de Bon Fonctionnement = temps d'execution - somme des temps de défaillance
double iSumOfOperatingTimes = (Math.abs(oEndDateOfTreatment.getTime() - oStartDateOfTre
LOG.debug("SUM OF OPERATING TIMES = " + iSumOfOperatingTimes);
// MTBF = Somme des Temps de Bon Fonctionnement / nombre de défaillances (http://chohmann.f
double iMTBF = iSumOfOperatingTimes/CDataConstant.TOTAL_NUMBER_OF_FAILURES;

```

```

LOG.debug("MTBF = " + iMTBF);

// MTTR = Temps d'arrêt Total / nombre d'arrêts
double iMTTR = CDataConstant.TOTAL_DURATION_OF_FAILURES/CDataConstant.TOTAL_NUMBER_OF_FAILURES;
LOG.debug("MTTR = " + iMTTR);

// Disponibilité= MTBF / ( MTTR + MTBF )
Double oAvailability = (iMTBF / (iMTTR+iMTBF))*100;
Double oAvailabilityTruncated = BigDecimal.valueOf(oAvailability).setScale(2, RoundingMode.DOWN);
LOG.debug("AVAILABILITY = " +oAvailabilityTruncated +"%");

// check if Availability is respected
if (CDataConstant.AVAILABILITY_SERVICE_THRESHOLD > oAvailabilityTruncated){
LOG.debug("AVAILABILITY IS NOT RESPECTED = " +CDataConstant.AVAILABILITY_SERVICE_THRESHOLD);
}else{
LOG.debug("AVAILABILITY IS RESPECTED = " +CDataConstant.AVAILABILITY_SERVICE_THRESHOLD);
}
}
});
}

private String getStartingMessage(){
StringBuilder sb = new StringBuilder();
sb.append("\n\n*****");
sb.append("\n* STARTING - ");
sb.append("\n* PLEASE WAIT - RESPONSE TIME ARE RANDOM SO MAY TAKE");
sb.append("\n* A WHILE TO SEE NOTIFICATION, WARNING AND CRITICAL VIOLATION EVENTS!");
sb.append("\n*****\n");
return sb.toString();
}
}

```

Annexe B: Illustrates in EBNF the Grammar Xtext of our textual Editor

```
// automatically generated by Xtext
```

grammar appDSL.xtext.ProviderDSL with org.eclipse.xtext.common.Terminals

```
import "http://Providermetamodel/1.0"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
```

Root returns Root:

```
{Root}
'Root'
'::='
('SLOs' '{' SLOs+=SLO ( "," SLOs+=SLO)* '}' )?
('SLAs' '{' SLAs+=SLA ( "," SLAs+=SLA)* '}' )?
';';
```

SLO returns SLO:

SLO_Impl | Availability | Reliability;

SLA returns SLA:

SLA_Impl | Computing;

QoS returns QoS:

Availability | Reliability | Modifiability | Adaptability | DataSecurity | Interoperability

Service returns Service:

Service_Impl | SaaS;

Ressource returns Ressource:

Ressource_Impl | Computing | Storage | Networking_Impl;

SLO_Impl returns SLO:

```
'SLO'
'::='
('Name:' Name=EString)?
('Value:' Value=EFloat)?
('ValueType:' ValueType=EString)?
'qos' '{' qos+=QoS ( "," qos+=QoS)* '}'
```

```
';;
```

SLA_Impl returns SLA:

```
'SLA'
'::='
('Id:' Id=EString)?
('ProviderID:' ProviderID=EString)?
'Commitements:' '{' Commitements+=Service ( "," Commitements+=Service)* '}'
'Billing:' '{' Billing+=Billing ( "," Billing+=Billing)* '}'
';;
```

EString returns ecore::EString:

```
STRING | ID;
```

EFloat returns ecore::EFloat:

```
'-'? INT? '.' INT (('E'|'e') '-'? INT)?;
```

Availability returns Availability:

```
'Availability'
'::='
('Name:' Name=EString)?
('Value:' Value=EFloat)?
('ValueType:' ValueType=EString)?
('Weight:' Weight=EFloat)?
('AvAttr:' AvAttr=AvailabilityAttr)?
'ReferTo:' ReferTo=[Service|EString]
'qos:' '{' qos+=QoS ( "," qos+=QoS)* '}'
';;
```

Reliability returns Reliability:

```
'Reliability'
'::='
('Name:' Name=EString)?
('Value:' Value=EFloat)?
('ValueType:' ValueType=EString)?
('Weight:' Weight=EFloat)?
```

```
( 'RelAttr:' RelAttr=ReliabilityAttr)?
'ReferTo:' ReferTo=[Service|EString]
'qos:' '{' qos+=QoS ( "," qos+=QoS)* '}'
';';
```

Service_Impl returns Service:

```
'Service'
'::='
('ServiceName' ServiceName=EString)?
'ressources' '{' ressources+=Ressource ( "," ressources+=Ressource)* '}'
';';
```

Modifiability returns Modifiability:

```
'Modifiability'
'::='
('Weight:' Weight=EFloat)?
('ModAttr:' ModAttr=ModifiabilityAttr)?
'ReferTo:' ReferTo=[Service|EString]
';';
```

Adaptability returns Adaptability:

```
'Adaptability'
'::='
('Weight:' Weight=EFloat)?
('AdapAttr:' AdapAttr=AdaptabilityAttr)?
'ReferTo:' ReferTo=[Service|EString]
';';
```

DataSecurity returns DataSecurity:

```
'DataSecurity'
'::='
('Weight:' Weight=EFloat)?
('DsAttr:' DsAttr=DataSecurityAttr)?
'ReferTo:' ReferTo=[Service|EString]
';';
```

Interoperability returns Interoperability:

```
'Interoperability'
'::='
('Weight:' Weight=EFloat)?
('InterAttr:' InterAttr=InteroperabilityAttr)?
'ReferTo:' ReferTo=[Service|EString]
';';
```

Ressource_Impl returns Ressource:

```
{Ressource}
'Ressource'
'::='
('IdRessource:' IdRessource=EInt)?
('NameRessource:' NameRessource=EString)?
('Storage:' Storage=EFloat)?
('Computing:' Computing=EFloat)?
('Networking:' Networking=EFloat)?
('Utilization:' Utilization=EFloat)?
';';
```

SaaS returns SaaS:

```
'SaaS'
'::='
('ServiceName:' ServiceName=EString)?
'ressources:' '{' ressources+=Ressource ( "," ressources+=Ressource)* '}'
// 'R' '{' R+=Ressource ( "," R+=Ressource)* '}'
';';
```

EInt returns ecore::EInt:

```
'-'? INT;
```

Computing returns Computing:

```
'Computing'
'::='
('IdRessource:' IdRessource=EInt)?
('NameRessource:' NameRessource=EString)?
```



```

('Storage:' Storage=EFloat)?
('Computing:' Computing=EFloat)?
('Networking:' Networking=EFloat)?
('Utilization:' Utilization=EFloat)?
('IdNetworking:' IdNetworking=EInt)?
('NameNetworking:' NameNetworking=EString)?
('Adress:' Adress=EString)?
('URLNetworking:' URLNetworking=EFloat)?
('Id:' Id=EString)?
('ProviderID:' ProviderID=EString)?
('IdComputing:' IdComputing=EInt)?
('NameComputing:' NameComputing=EString)?
'Commitements:' '{' Commitements+=Service ( "," Commitements+=Service)* '}'
'Billing:' '{' Billing+=Billing ( "," Billing+=Billing)* '}'
';';

```

Storage returns Storage:

```

{Storage}
'Storage'
'::='
('IdRessource:' IdRessource=EInt)?
('NameRessource:' NameRessource=EString)?
('Storage:' Storage=EFloat)?
('Computing:' Computing=EFloat)?
('Networking:' Networking=EFloat)?
('Utilization:' Utilization=EFloat)?
('IdNetworking:' IdNetworking=EInt)?
('NameNetworking:' NameNetworking=EString)?
('Adress:' Adress=EString)?
('URLNetworking:' URLNetworking=EFloat)?
('IdStorage:' IdStorage=EInt)?
('NameStorage:' NameStorage=EString)?
('Size:' Size=EFloat)?
('URLStorage:' URLStorage=EString)?
';';

```

```

Networking_Impl returns Networking:
{Networking}
'Networking: '
'::='
('IdRessource: ' IdRessource=EInt)?
('NameRessource: ' NameRessource=EString)?
('Storage: ' Storage=EFloat)?
('Computing: ' Computing=EFloat)?
('Networking: ' Networking=EFloat)?
('Utilization: ' Utilization=EFloat)?
('IdNetworking: ' IdNetworking=EInt)?
('NameNetworking: ' NameNetworking=EString)?
('Adress: ' Adress=EString)?
('URLNetworking: ' URLNetworking=EFloat)?
';';

```

```

Billing returns Billing:
{Billing}
'Billing'
'::='
('CostEquation: ' CostEquation=EString)?
';';

```

```

enum ModifiabilityAttr returns ModifiabilityAttr:
MTTC = 'MTTC';

```

```

enum AdaptabilityAttr returns AdaptabilityAttr:
Completeness = 'Completeness' | CoverageforVariability = 'CoverageforVariability';

```

```

enum DataSecurityAttr returns DataSecurityAttr:
SSLApplicable = 'SSLApplicable' | CommunicationLatencyOverSSL = 'CommunicationLatencyOverSSL';

```

```

enum InteroperabilityAttr returns InteroperabilityAttr:
WorkloadMigration = 'WorkloadMigration' | DataMigration = 'DataMigration';

```

```

enum AvailabilityAttr returns AvailabilityAttr:

```

```
UpTime = 'UpTime' | downTime = 'downTime' | IncidentTime = 'IncidentTime';
```

```
enum ReliabilityAttr returns ReliabilityAttr:
```

```
MTTR = 'MTTR' | MTBF = 'MTBF' | MaxTTR = 'MaxTTR' | Serviceconstancy = 'Serviceconstancy' | A
```