# Formal Methods meet Security in a Cost Aware Cloud Brokerage Solution

Salwa Souaf

# INSA CENTRE VAL DE LOIRE

***ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES***

# THÈSE présentée par :

**Salwa SOUAF**

soutenue le : **15 Décembre 2020**

pour obtenir le grade de : **Docteur de l'INSA Centre Val de Loire**

Discipline/ Spécialité : Informatique

# Formal Methods meet Security in a Cost Aware Cloud Brokerage Solution

**THÈSE co-dirigée par :**

Pascal Berthomé, INSA Centre Val de Loire
Frédéric Loulergue, Université d'Orléans

**RAPPORTEURS :**

| | |
|---|---|
| **Catherine Dubois** | Professeur, ENSIIE |
| **Fabrice Mourlin** | MCF HDR, UPEC |

---

**JURY :**

| | |
|---|---|
| **Benjamin Nguyen** | Professeur, INSA Centre Val de Loire |
| **Maryline Laurent** | Professeur, Télécom SudParis |
| **Hélène Coullon** | MCF, IMT Atlantique |
| **Bertrand Cambou** | Professeur, Northern Arizona University |

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS

# Acknowledgements

Firstly, I would like to thank the reviewers for taking the time to read this document and their helpful remarks that have greatly helped its quality.

I am also grateful to both my supervisors for having given me the opportunity of carrying this PhD and guided me through it. Frédéric Loulergue for his impressive theoretical knowledge, his trust and kindness. To Pascal Berthomé, for his tireless proofreading, his insight – but mostly, for his support. With the hope of carrying our collaboration in the future.

I also thank all the SDS research team members. Sabine, Laurent and Fergal for being there with me through the final steps of my PhD. To Sara and Sadjad that joined later on, but who merit no less mention. I wish you the best for the completion of your theses. To Patrice for being the initiator of this journey. To Benjamin, Cedric and to all other colleagues that I have failed to name.

A special mention to ALL the people I have met during my stay at Northern Arizona University. To some of the most hardworking professors: Bertrand and Fatima. To some of the most genuinely nice and caring people I will ever meet: Cassie, Roo, Tommy, Chase, Jolan and Katie. To all the lifetime friendships I have made: Demarara, Mounia and Houda.

I also have debts of gratitude to my family. My parents for their love, prayers, caring and sacrifices for educating and preparing me for my future. My brother, sister, sister in law and nephews for filling my heart with love, encouraging and supporting me through my journey.

But finally, and foremost, to Radouane. The one who has my back. He has and still is always pushing me to reach my full potential. It wouldn't have been the same without your love and support.

# Résumé En Français

## Introduction

Le Cloud computing a prouvé son approche révolutionnaire en fournissant divers services informatiques au cours des dernières années. Il offre de multiples avantages par rapport à un modèle "in-house computing" traditionnel, à savoir, une mise à l'échelle à la demande, le multiplexage des ressources, un accès haut débit au réseau et un modèle de payement "pay-as-you-go". En effet, un consommateur ne paiera que les coûts d'utilisation de ses applications, qui s'avèrent bien inférieurs au coût de construction et de maintenance de sa propre infrastructure.

Avec la demande croissante sur les ressources Cloud, vient l'augmentation du nombre de fournisseurs de services Cloud. Cette augmentation induit deux freins majeurs à l'adoption du Cloud. Le premier étant la difficulté de choisir le meilleur fournisseur et la meilleure offre pour ses besoins. Le deuxième réside dans les appréhensions liées aux problèmes de sécurité [26] lors de l'utilisation du Cloud.

Les clients recherchent des solutions qui *assureraient* leur sécurité. Des propositions détaillées d'exigences relatives aux approches de responsabilité pour les services et outils sont présentées dans [65].

Plusieurs travaux de recherche ont été menés pour remédier à ces problèmes, certains de ces travaux sont présentés dans le chapitre 3 consacré à un état de l'art des solutions de courtage Cloud existantes. Une solution possible consiste en la combinaison de deux scénarios d'interopérabilité Cloud: le *courtage Cloud* et le *multi-Cloud*. Avoir un certain niveau d'interopérabilité entre les différents fournisseurs de Cloud pourrait être très bénéfique à la fois pour

les clients et les fournisseurs de Cloud. Certaines des motivations et avantages d'une telle interopérabilité, ainsi que des définitions détaillées des concepts mentionnés seront présentées dans le chapitre 1.

La solution proposée dans cette thèse intègre deux aspects, l'assurance de la sécurité du Cloud et la variété des offres Cloud. Le courtier Cloud proposé, est une entité tierce qui fait l'intermédiaire entre les clients et les fournisseurs Cloud. Le courtier guidera le client tout au long du processus d'intégration du Cloud. Notre courtier est conceptualisé pour le modèle de déploiement **IaaS** du Cloud computing. Il prend en considération les exigences fonctionnelles (c'est-à-dire la quantité et la description des ressources) et non fonctionnelles (c'est-à-dire les propriétés de sécurité) du client dès le premier contact. Après avoir reçu la description de la demande du client, notre courtier commence par vérifier sa cohérence. Dans le cas d'incohérence de la demande, il met en évidence les éventuelles causes de celle-ci. Nous utilisons des méthodes formelles couplées avec de la programmation linéaire pour vérifier la cohérence et trouver le placement approprié dans une fédération de Clouds. L'outil fait une proposition de déploiement de l'architecture au client. Nous présentons une solution *transparente*, les utilisateurs décrivent leurs exigences sécurité personnalisées dès les premiers stades. De plus, le mécanisme que nous proposons peut être facilement documenté, car ses fondements reposent sur un formalisme facile à comprendre qui est la logique relationnelle de premier ordre.

Dans la suite de ce résumé, je vous présente la première version de la solution du courtage proposée et publiée dans [74]. Les modifications et améliorations apportées à cette solution, en particulier la modélisation de la fédération, des fournisseurs et la recherche de placement. Puis, un résumé du travail qui consiste en la traduction du langage Alloy vers la syntaxe Coq afin de prouver l'exactitude des propriétés de modèles écrites en Alloy. Cela était motivé par les limitations rencontrées lors de l'utilisation du langage Alloy. Ce travail a été publié dans [75].

## Une première solution de courtage Cloud

Le chapitre 4 résume les travaux menés comme amélioration du courtage Cloud solution présentée par Guesmi et al. [37]. Dans sa thèse [36], Guesmi a posé les premières bases de l'utilisation de la vérification formelle dans une solution de courtage Cloud. Elle a réussi à modéliser les utilisateurs et fournisseurs Cloud, mais aussi à définir des contraintes de sécurité personnalisées en utilisant le langage et l'outil *Alloy*. Alloy est un langage de spécification permettant d'exprimer des contraintes structurelles et des comportements complexes des systèmes. L'outil Alloy (aussi appelé l'analyseur Alloy) est un outil de modélisation structurelle simple basé sur une logique de premier ordre. La solution proposée a cependant laissé un certain nombre de questions sans réponse. Bien que ses résultats finaux aient présenté des limites évidentes, nous ne pouvons pas nier le potentiel de cette approche, d'où la décision de reprendre le travail et d'essayer de le développer davantage.

Le fait que le projet soit limité aux architectures de type IaaS n'est pas considéré comme une limitation bloquante. Une décision a été prise de conserver ce type d'architecture, car il permet les manipulations de ressources que nous recherchions. En effet, notre approche peut facilement être étendue aux architectures de type PaaS, en modifiant légèrement les modèles et en ajoutant des contraintes spécifiques à ces architectures. L'une des limites de l'approche adoptée dans [36], est que la taille des problèmes qui peuvent être traités efficacement est limitée. Une seconde limitation est l'impossibilité de répondre à la dynamicité des besoins des clients, des offres fournisseurs ou des architectures déployées.

Le contenu détaillé dans le chapitre 4 est basé sur l'article publié dans [74]. Le but de ce travail était de trouver un moyen d'introduire des fonctionnalités plus dynamiques dans le modèle existant. Des fonctionnalités qui ont comme but de prendre en compte l'évolution des besoins des clients, des offres fournisseurs ainsi que des architectures déjà déployées. Le changement majeur qui a été apporté est le changement de l'outil de modélisation de *Alloy* à *KodKod*.

Kodkod est un solveur de contraintes basé sur SAT pour la logique du premier ordre. Contrairement aux autres solveurs, Kodkod autorise des modèles partiels, en permettant à l'utilisateur de spécifier une solution partielle qu'il pourra compléter ensuite.

## Optimisation de coût dans un courtier Cloud

La solution proposée précédemment, a montré un grand potentiel pour aider les clients dans l'intégration du Cloud tout en tenant compte de leurs besoins fonctionnels et de sécurité personnalisés. Elle possède malgré tout quelques limites. En effet, les modèles, plus précisément ceux des fournisseurs de services et de la fédération, sont assez simplistes et ne reflètent pas un scénario réaliste. La façon dont on a conçu notre première solution repose sur l'existence d'un standard pour décrire les infrastructures des fournisseurs de services Cloud, ainsi que la fédération qui les regroupe. Malheureusement, ces attentes sont irréalistes en raison des problèmes d'interopérabilité entre les fournisseurs et de l'inexistence de standardisation de la description des infrastructures des fournisseurs. Afin d'avoir une solution de courtage plus en lien avec les problèmes actuels, nous avons décidé de modifier les descriptions des fournisseurs et fédérations. Une autre limite rencontrée est liée à l'étape de recherche d'une stratégie de placement approprié de l'architecture des clients. En fait, notre solution de courtage renvoyait la première stratégie trouvée sans prendre en considération des critères de classement spécifiques. D'après nos recherches, le coût reste l'un des critères les plus importants pour les clients potentiels. En conséquence, nous avons décidé de proposer dans la nouvelle version de notre solution une stratégie de placement qui optimisera le coût d'intégration du Cloud.

Dans le chapitre 5, je présente les modifications apportées aux modèle et architecture de la solution précédente, ainsi que l'implémentation du courtier. Le courtier commence par la vérification de la demande du client en utilisant l'API KodKod, ensuite le problème linéaire est résolu afin de trouver une stratégie

de placement qui répond aux exigences fonctionnelles et non-fonctionnelles du client tout en optimisant son coût.

## Transformation de Alloy à Coq

La méthode formelle Alloy nous a permis de modéliser les différents acteurs de notre problème, nos exigences de sécurité et de les vérifier facilement. Cependant, nous avons fait face à certaines de ses limites lors de la première phase d'implémentation. Cela a motivé le travail résumé dans cette section.

Il existe de nombreuses méthodes formelles, allant des outils entièrement automatisés, comme les analyseurs statiques [73], aux prouveurs de théorèmes interactifs qui nécessitent beaucoup de travail humain. Souvent, les utilisateurs de ces outils doivent fournir une spécification du système analysé. L'analyse de cette spécification peut alors être automatique ou interactive. Alloy et l'analyseur Alloy [41] entrent dans la première catégorie. Alloy était et est encore utilisé dans de nombreux domaines différents, par exemple en génie logiciel [32] et en sécurité [67]. Des applications plus spécifiques de celui-ci, telles que présentées par Torlak et al. dans [85], sont la modélisation et l'analyse de systèmes logiciels, la vérification bornée de programme et la génération de cas de test. Plusieurs systèmes ont été étudiés en utilisant Alloy, à savoir, le système de fichiers flash [43], le porte-monnaie électronique Mondex [68], une machine de proton-thérapie [70], une bibliothèque de système d'information [29], *etc.*

En ce qui concerne la vérification bornée des programmes (i.e. *bounded verification* en anglais), deux travaux connexes ont été présentés en détail dans [85]. L'outil Jalloy [40] vérifie une méthode Java par rapport à une spécification de son comportement. Il commence par traduire la méthode en langage Alloy puis en invoquant un premier prototype de l'analyseur Alloy sur les contraintes résultantes. Le deuxième travail a été construit sur le travail précédent et s'appelle Forge [22]. Il a utilisé une nouvelle traduction du code procédural en logique relationnelle impliquant une exécution symbolique, en utilisant l'API

KodKod [83]. Comme expliqué dans le chapitre 4, nous utilisons le langage Alloy et l'API KodKod en arrière-plan de notre solution de courtage Cloud pour décrire et vérifier la cohérence des modèles. Alloy a également été exploité dans de nombreux outils de génération de cas de test, à savoir, TestEra [52] et Whispec [72]. Alors que TestEra [52] utilise Alloy dans un cadre de boîte noire basé sur les spécifications pour tester les programmes Java, Whispec [72] suit une approche de test boîte blanche basée sur les spécifications avec Kodkod. KodKod [24], qui est au cœur du moteur de l'analyseur Alloy est également utilisé dans Niptick [13] un chercheur de contre-exemples pour l'assistant de preuve Isabelle.

Alloy est une méthode formelle légère, car elle repose sur l'hypothèse "*small scope*" : l'examen de tous les petits cas est susceptible de produire des contre-exemples intéressants. Cependant, l'analyseur Alloy ne peut pas montrer l'absence d'erreurs. D'autres outils formels tels que les prouveurs de théorèmes interactifs Coq [80] et Isabelle [61] ont été utilisés pour fournir des garanties très solides sur les logiciels vérifiés, y compris un compilateur C [46] et le noyau d'un système d'exploitation [44].

Nous pensons qu'il est très utile d'utiliser des méthodes formelles légères. En pratique, si l'on veut utiliser un outil tel qu'Alloy dans un premier temps, puis que l'on veut utiliser un outil plus lourd comme Coq dans un second temps, la formalisation effectuée en premier est perdue. Pour accompagner la transition Alloy vers Coq, nous proposons, dans le chapitre 6 un traducteur des modèles Alloy vers du code Coq. Le résultat de ce travail a été publié dans [75].

## Conclusion

De nombreuses solutions de courtage Cloud proposées dans le milieu académique ainsi que celles commercialisées abordent des problèmes liés à la performance, au coût, à la qualité de service et au respect des contrats SLA (i.e. Service Level Agreement est un contrat par lequel le fournisseur de service s'engage à fournir un ensemble de services). Ceux qui intègrent l'aspect

de sécurité ne considèrent que les niveaux de sécurité de base, i.e. les services sécurité proposés par les fournisseurs Cloud. À part le travail proposé par A. Guesmi's dans [37], qui est la solution initiale sur laquelle est basée cette thèse, nous n'avons pas trouvé d'autres travaux qui tiennent compte des besoins personnalisés du client du point de vue de la sécurité. Dans cette thèse, nous proposons une conception et un outil de courtage Cloud qui vise à s'affranchir des limitations du travail de A. Guesmi.

Notre solution donne la possibilité aux clients, souhaitant migrer leur application vers le Cloud, de décrire leur demande sous forme d'une architecture basée sur des composants, de préciser leurs descriptions fonctionnelles ainsi que les relations de sécurité les reliant. Le courtier vérifie ensuite la cohérence de cette demande et trouve une stratégie de placement qui respecte à la fois leurs exigences fonctionnelles et non fonctionnelles tout en optimisant le coût. Le reste du document présente en détail nos contributions.

## Liste des publications

- Salwa Souaf, Pascal Berthomé, and Frédéric Loulergue. A cloud brokerage solution: Formal methods meet security in cloud federations. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orléans, France, July 16-20, 2018*, pages 691–699. IEEE, 2018. `https://hal.archives-ouvertes.fr/hal-02317089`.

- Salwa Souaf and Frédéric Loulergue. A first step in the translation of alloy to coq. In *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings, volume 11852 of Lecture Notes in Computer Science*, pages 455–469. Springer, 2019. `https://hal.archives-ouvertes.fr/hal-02317118`.

# Contents

# List of Figures

# Introduction & Background

1

Contents



## 1.1 Introduction

With the increasing demand on Cloud computing resources and growth of the Cloud market, comes the increase of the number of Cloud services providers (CSP). It makes it harder for potential customers to chose between providers and/or find a perfect offer catered for their special needs.

Security [26] was, and still is, one of the biggest reasons holding some users from integrating the Cloud. Customers are looking for solutions that would *assure* their security. In general, security assurance refers to the grounds for confidence that the set of intended security policies and controls in an information

system or organization are effective in their application. Many strategies can be implemented to assure that. Pearson [65] gives detailed examples of requirements to accountability approaches (cf. Chapter 3).

These issues have spiked the interest of many researchers and companies, some of these works and approaches will be presented in Chapter 3. One way to diverge from these limitations is by combining two Cloud interoperability scenarios, *Cloud brokerage* and *multi-Cloud*. In fact, having a certain level of interoperability between the different Cloud providers could be equally very beneficial for Cloud customers and providers. Some of the motivations and benefits of such an interoperability, as well as a detailed definition of the aforementioned concepts, will be presented in Section 1.2.

This thesis consists in trying to find a trustworthy solution to assist users in integrating the Cloud. To do so, we propose to integrate two aspects: Cloud security assurance and the variety of Cloud offers. We suggest a Cloud broker, a third party, that will intermediate the relation between Cloud customers and providers. The broker will guide the customer through the full process of integrating the Cloud. Our broker will take into consideration the functional (i.e. description of the resources needed) and non-functional (i.e. security properties) requirements of the customer. We present a *transparent* solution as the users describe their customized security requirements from the early stages. This solution is easily documented due to its foundations relying on an easy to understand formalism that is first-order relational logic.

The organization of the rest of this chapter will be as follows. The definitions of the different concepts related to our Cloud brokerage solution will be elaborated in Section 1.2. Section 1.3 will summarize the motivation of the work conducted throughout this thesis. The main contributions are summarized in Section 1.4 and Section 1.5 will present the overall organization of the upcoming chapters.

## 1.2 Context

This section presents the global context of my thesis. It gives definitions of the Cloud computing concept and its surrounding elements: Cloud Providers, Cloud Brokerage, Cloud Federation and Multi-Cloud.

### 1.2.1 Cloud Computing

The most referenced definition of Cloud computing is that given by the National Institute of Standards and Technology (NIST) [25]:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources ... that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Several essential characteristics distinguish Cloud computing. *On-demand self-service*: the Cloud computing power and the storage capacity can adapt automatically according to the needs of the customers. Without the need for human interaction with service providers, a customer can easily provision computing resources in a flexible manner thanks to Cloud computing. *Broad network access*: Cloud services are available on the network and are accessible through standard mechanisms. They can be accessed by heterogeneous thin or thick client platforms (mobile phones, tablets, laptops, and workstations). *Resource pooling*: Physical and virtual resources can be combined to serve multiple Cloud customers. They are allocated and automatically adapted according to customer demand. Customers are not aware of the exact location of resources but can specify it at a higher level of abstraction (e.g. the country). Some examples of the provided resources are storage, processing, memory, and network bandwidth. *Rapid elasticity*: Resources can be quickly, in some cases even automatically, provisioned and released according to the customer's request. This makes the customer feel that the resources are unlimited and can be accessed at any time. However, providers, generally, tend to define thresholds. *Measured*

*service*: Thanks to its metering capability Cloud systems automatically control and optimize resource use. The latter can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service. In fact, it fuels the billing model adopted by the majority of providers which is "pay only for what you consume".

Cloud is known for its three major service models. Starting with the Software as a Service (SaaS), which offers fully ready applications that are accessible via various client devices, through a lightweight user interface such as a web browser or a dedicated software. The user does not have to worry about software upgrades nor how they are provided. Famous examples for SaaS providers are Google Apps [34] and Microsoft Office365 [56].

Then we find the Platform as a Service (PaaS). This service model consists of provisioning pre-configured environments on which users/customers can develop, test, and run their applications. The users can deploy applications and add their own tools. They control the application and its configuration but not the underlying infrastructure (i.e. network, server, storage). The operating system and infrastructure tools are under the control of the provider. Among the PaaS providers are Google App Engine [1], Microsoft Windows Azure [55] and OpenShift [62].

Last, the Infrastructure as a Service (IaaS) that provides the user with computing, networking and storage resources. It offers virtualized hardware resources based on physical hardware resources. Users have the choice to provision, either virtual machines to deploy and run software including operating systems, or pre-configured virtual machines from the provider with installed operating systems and applications. Users benefiting from this service model do not control the underlying Cloud infrastructure but have control over the operating systems, storage, and applications, as well as if needed, network components such as firewalls. IaaS is considered as the most established Cloud service model [88]. Amazon Web Services (AWS) is among the leading IaaS providers.

*Public Cloud* is the set of Cloud infrastructures accessible via the Inter-

net, open to the public, and made available by the different Cloud service providers. It is only one of the deployment models of Cloud computing. The other existing deployment models consist of the *private Cloud*, a Cloud infrastructure deployed by an organization for restricted internal use only. Access to this Cloud is limited to employees of the organization. It may be held, managed and operated by the organization and/or a third party. *Community Cloud* is an infrastructure shared among several organizations for the needs of a specific community. This type of model can be found in academic circles for example. It may be held, managed and operated by one or many of the organizations in the community, a third party, or some combination of them. And then there is the *hybrid Cloud*, a composition of several different Cloud infrastructures (public, private, community) that share data and applications. This allows organizations to take advantage of the low-cost and performance of the public Cloud and the possibility to retain critical data in the private Cloud.

### 1.2.2 Cloud Provider

A Cloud provider is an entity that makes a service available for the interested parties. It is the one who collects and manages the computing infrastructure needed for providing services, runs the Cloud software that provides the services, and delivers the Cloud services to the customers through network access. There are different providers according to the service model they are offering.

A Software as a Service Cloud provider will be deploying, configuring, maintaining and updating the applications on a Cloud infrastructure, in order to guaranty that the services are provisioned at the expected service levels to the consumers. While the Cloud consumers have limited administrative control of the applications, the SaaS provider assumes most of the responsibilities in managing and controlling the applications and the infrastructure.

For PaaS, the provider is managing the computing infrastructure for the platform. It runs the software that provides the components of the platform (e.g. runtime software execution stack, databases, and other middle-ware components). It provides tools such as integrated development environments

(IDEs), a development version of Cloud software, software development kits (SDKs), deployment and management tools, to support the development, deployment and management process of the consumers. The consumer has control over the applications and possibly some hosting environment settings, while the provider has full control of the infrastructure underlying the platform such as network, servers, operating systems (OS), or storage.

When it comes to the IaaS model, the provider is responsible for acquiring the physical computing resources underlying the service (servers, networks, storage and hosting infrastructure), as well as, running the Cloud software necessary to make computing resources available to the consumer through a set of service interfaces and computing resource abstractions, such as virtual machines and virtual network interfaces. The consumers in turn use these computing resources for their fundamental computing needs. An IaaS Cloud consumer has access to more fundamental forms of computing resources and thus has more control over the software components in an application stack, including the OS and network, unlike the SaaS and PaaS Cloud consumers. The provider, on the other hand, will only have control over the physical hardware and Cloud software that makes the provisioning of these infrastructure services possible (e.g. the physical servers, network equipment, storage devices, host OS and hypervisors for virtualization).

A Cloud provider's activities can be grouped into five major areas: service deployment, service orchestration, Cloud service management, security, and privacy.

### 1.2.3 Cloud Brokerage

As Cloud computing evolves, the integration of Cloud services can be too complex for Cloud consumers to manage. A Cloud consumer may request Cloud services from a Cloud broker, instead of using a Cloud provider directly. Quoted from a definition given by the NIST [25], a Cloud *broker* is "an entity that manages the use, performance and delivery of Cloud services and negotiates relationships between Cloud providers and Cloud consumers."

In general, the services provided by a Cloud broker can categorized as follows:

- **Service intermediation:** A Cloud broker strengthens a certain service, offered by a Cloud service provider, by improving specific capabilities and providing added-value services to the customer. This improvement may include managing access to services, managing identity or improving security.
- **Service aggregation:** A Cloud broker can combine multiple Cloud computing offerings into one or more new services. It provides data and service integration and ensures the security of data transfer between the customer and the various Cloud service providers.
- **Service arbitrage:** The arbitration is similar to the aggregation except that the combined services are not fixed. An arbitrage broker has the flexibility to choose the services to be provided to the customer from several service providers. The objective is to optimize the services provided to the customer. For example, it can use a credit-scoring service to measure and select a provider with the best score.

### 1.2.4 Multi-Cloud and Cloud Federation

Multi-Cloud architecture provides an environment where users can use resources from multiple Cloud providers to build Cloud environments outside the traditional in-house infrastructure. Though it is complicated to toggle between Cloud providers to perform tasks, Cloud service providers are working to make this increasingly efficient. Many works are also being made around multi-Cloud.

A Cloud federation is an explicit collaboration between multiple providers. Despite the many benefits that arise from the aspect of Cloud federation and multi-Cloud, to mention: QoS improvement, reduction of Service Level Agreement (SLA) violations, cost efficiency. These are discussed in detail in [50]. It

is not a common practice in current Cloud solutions due to the many challenges Cloud interoperability is still facing. A spectrum of the obstacles and challenges facing the Inter-Cloud realization is presented in [81].

Adopting multi-Cloud or Cloud federation architecture has many benefits, to mention a few:

- Disaster recovery: there is a big risk when using resources from one Cloud provider, of a cyber-attack taking down all the operations, leaving end-users inaccessible until it resolves. Multi-Cloud architecture, on the other hand, can make services resilient against such cyber-attacks because the customer's infrastructure is scattered on different Cloud providers.
- Avoiding vendor lock-in: the multi-Cloud platform allows organizations to select the best services from different Cloud providers, tailoring them to their organizational goals, rather than having to modify their business processes to fit a specific provider's setup.
- Cloud cost optimization: multiple Cloud providers are offering similar services at different price points. By being able to choose different services from different providers, end-users are able to optimize the cost of their Cloud architecture.
- Low latency: latency is inherent in Cloud services delivered from servers at distant locations. In a multi-Cloud environment, users can deploy data centers to multiple regions according to the needed locations. This is especially useful for global organizations that need to serve data across geographically disparate locations while maintaining a satisfactory end-user experience.

## 1.3 Motivation

Despite the numerous benefits Cloud computing has to offer, its full adoption is hindered by the security and privacy concerns it causes. In [5, 26] is

presented an overview of different Cloud concepts (i.e. virtualization, Cloud platforms, data outsourcing, data storage standardization and trust management) and the security issues related to each of them. Due to the fact that our solution focuses on IaaS architectures, the rest of this section will focus primarily on the security issues related to the IaaS model [20,88].

The brokerage solution that we propose in this thesis aims to cover the security issues that are related to virtualization properties in the Cloud. In [71] many of these security issues have been presented in detail. The authors categorize the threats and attacks against virtualized systems according to the different layers (i.e. hardware, virtualization, OS and application). They also list some of the known attacks against virtualized environments at the different layers. The work presented in this thesis could limit the damages of some of the attacks on the virtualization layer. To mention some of the attacks targeting the virtualization layer:

- Cross-VM Attacks: attacks between virtual machines (VMs),
- VM Escape and VM Hopping: where attackers run a malicious code on the VM that would break the operating system giving them direct access to the hypervisor (i.e. a program that creates and runs virtual machines),
- VM Detection: when attackers detect that the targeted system is running inside a VM and start targeting their attacks on the virtualization layer instead of the application or OS.

In the survey [71], a framework for threat models categorization has been proposed. When trying to propose a new protection solution, this framework can be useful for a more precise definition of the security and trust assumptions. Our work will especially help limit the damage of the *Cross-VM Attacks* and the exploits taking advantage of inter-VM communications. By giving the customers the possibility to define their security properties under the form of communication relations, they will be able to control where they place their most critical data as well as the type of connections between their components.

Thus, with well defined and robust security requirements, a malicious attacker won't be able to access the critical data.

The initial aim of my thesis was to improve upon and carry on the work done during Asma Guesmi's PhD thesis [36]. Our joined motivation is to find a solution to aid customers in integrating the Cloud while taking into consideration their functional and non-functional needs as well as assuring them a personalized level of security. The contributions of Guesmi's work will be presented as part of the related works in Chapter 3.

## 1.4 Contributions

First, improving upon the work conducted in [36] by introducing a dynamic aspect in the sense of we gave customers the possibility to modify their Cloud architecture after its deployment and the broker the ability to handle a multitude of customers. Such work and improvements prove the potential that formal methods have to describe and assure customers' security in the Cloud.

Second, using formal methods and linear modeling to create a Cloud brokering solution that assists customers in migrating their applications to the Cloud, while respecting both their functional and non-functional needs all while being cost efficient. The customer's demand is described in the format of interconnected components. While the functional requirements consist of the characteristics of these components, the non-functional requirements are the communication links connecting them. We use formal methods to describe and verify the consistency of the customer's application architecture, in order to detect any communication conflicts between the different components before their deployment. Then we use linear programming to find an adequate, cost efficient, placement strategy that fulfills the customer's needs.

Finally, having used the Alloy specification language [19] throughout our work we have felt its limitations. Alloy is a declarative language that allows modular descriptions and complex configurations of the systems to be verified. Due to its conception, this verification is only done in finite scopes. Thus, we

thought that the ability to have an additional step using heavier weight provers, such as the Coq proof assistant, could be interesting when handling critical systems. We developed a tool to translate models written in the Alloy language into Coq syntax and help prove properties about them.

## 1.5 Manuscript Organization

The organization of this manuscript will closely mimic the different stages I went through during my PhD thesis. Starting with Chapter 2, giving an overview of the different languages and tools used in the different contributions, i.e. Alloy language, KodKod model finder, Coq Proof Assistant and the GLPK (GNU Linear Programming Kit).

Chapter 3 summarizes the related works that have been conducted regarding Cloud brokerage solutions, multi-Cloud and Cloud Federations. Chapter 4 presents the description and results of our first contribution published in [74], where we present a Cloud brokerage solution that assists customers in the full process of integrating the Cloud while taking into consideration their personalized security requirements from the early stages. This solution has shown interesting results but still had its limitations. Some of these limitations were related to the complexity of finding a placement strategy in the federation of Clouds, as well as the description of the latter model being too simplistic and not reflecting real life scenarios. To solve these limitations we have decided to explore a different path using linear programming, described in detail in Chapter 5. Another limitation is due to the use of the Alloy language to formally describe our model. In fact, trying to remediate this limitation was behind the work described in Chapter 6 and published in [75]. We have decided to use the Coq proof assistant to prove the correctness of Alloy models. In Chapter 6 we present a translation method and tool of models written in Alloy language to Coq syntax in order to be able to prove the needed specifications about them.

We will be closing up the manuscript with an overall discussion and conclusion in Chapter 7.

# Preamble 2

Contents



In order to ease the comprehension and make the experience of reading the next chapters more pleasant, I will be introducing the languages and tools mentioned throughout the document in this chapter. The overviews presented in the latter will be organized as follow: Section 2.1 presents first-order relational logic, which is underlying the Alloy language and the KodKod model finder, detailed respectively in Sections 2.1.1 and 2.1.2. These are the languages and tools that were used in the first contribution of my thesis presented in Chapter 4. Section 2.2 introduces linear programming and presents an overview of the GNU Linear Programming Kit. The latter is the direction we have explored as part of my final contribution presented in Chapter 5. Section 2.3 presents the Coq proof assistant which is the tool we have decided to use for proving the correctness of Alloy models, this work will be detailed in Chapter 6.

## 2.1 First-Order Relational Logic

First-order relational logic is a fusion between first-order logic and relational logic presented by Daniel Jackson [40]. First-order logic, also called pred-

icate logic, quantificational logic, or first-order predicate calculus, uses quantified variables over non-logical objects and allows the definition of sentences that contain variables. Relational logic is based on propositional logic to which it introduced two linguistic features, variables and quantifiers. These features made it possible to express information about multiple objects without enumerating them as well as express the existence of objects that satisfy specified conditions without explicitly specifying them. According to Daniel Jackson, first-order relational logic is more than a definitional extension of first-order logic, because it; not only adds the ability to combine predicates with special operators, such as the navigation operator, it also includes the definition of transitive closure. For a detailed description of this logic refer to [40, Section 2]. In this section will be presented a language and a tool that implement the first-order relational logic.

### 2.1.1 Alloy and the Alloy Analyzer

Alloy [40, 41] is both a language and an analyzer for writing and checking formal models, developed in MIT (Massachusetts Institute of Technology) by the Software Design Research Group under the direction of Daniel Jackson. In the remaining, in order to be unambiguous, *Alloy* will refer to the language and *Alloy Analyzer* to the tool. Alloy has been widely used for modeling systems in order to simulate them and verify their properties. It allows a simplified view of the systems by abstracting implementation details and focusing on their properties and constraints. The language has a simple syntax based on the Z language. It is a structural language. It allows to model complex structures with hierarchies and relations. It offers the possibility to define entities with properties and constraints to describe systems but does not conduct treatments. Alloy is an analyzable language, the properties of models written in the Alloy language can be checked and simulated using the Alloy Analyzer. An Alloy model is a collection of entities and constraints used to describe the structure of a system. Writing an Alloy model consists of defining the following statements:

**Atoms & Relations:** atoms are the basic elementary entities. These are abstract concepts used to model aspects of the real world. Alloy data types are universally based on relations. They represent a concept that serves to define correlations between atoms. Relations and atoms cooperate to represent different aspects of the modeled system. Relations can have a $n$ arity and can be declared as $f : A_1 \rightarrow \ldots \rightarrow A_n$ where $A_i$ is an atom.

**Signatures:** they represent sets of entities of a system. A signature is the only element to represent the types and atoms in an Alloy model. Although it is a non-object-oriented language, Alloy allows inheritance between signatures. A signature can have attributes as explained below.

**Facts:** they are used to describe different constraints about the system being modeled that remain always true. In Alloy, all facts are defined using the keyword *fact*.

**Predicates:** they are an abstraction of logical formulas for reuse purposes. A predicate can be defined with parameters used in the logical formula of its body. Predicates are often used in assertions to be verified in the model.

**Functions:** they are similar to *Predicates* with the difference of returning typed values for reuse and model clarity.

**Assertions:** they are used to specify properties about the model that are either expected to hold true or checked if they hold. Once an assertion is stated it can be checked if it holds in a specific scope, using the keyword **check** and feeding the model to the Alloy Analyzer. The analyzer looks for a counterexample to the assertion within the specified scope, if no counterexample is found, the assertion is said to hold true in the specified scope.

The scope is the cardinality, specified by the user, of the top-level signatures in a model. Although working within limited scopes ensures that the model-finding problem is decidable, it limits the generality of the results produced

by the Alloy Analyzer. Jackson explains this design decision through the *small scope hypothesis*: "most bugs can be found by testing programs for all test inputs within a small scope". For more details refer to [41, Section 5].

The example of Figure 2.1, that is basically the example of [41, page 16], presents a detailed overview of the Alloy syntax for a library management system. Name and Addr are two *signatures* in Alloy terminology. They are sets. Book is also a signature containing an attribute, addr. While addr is given the type Name→Addr, the fact that it is an attribute of Book means it is actually a ternary relation between Book, Name and Addr. In Lines 3 and 4 of Figure 2.1, we can see the definition of the predicates add and del both defining two different states of book, the first by adding a new entry (i.e. addr) and the second by deleting an existing one. In this code, + means union, − set difference, and . is the relational join of Alloy. One specificity of the join operation in Alloy is that in an expression r1.r2, the right-most column of relation r1 and the left-most column of relation r2 are not in the join result. The function lookup returns the Addr associated to the Name n in the book b, n and b given as arguments of the function.

The assertions are defined for this example in Lines 11–21. The assertion delUndoesAdd is stating that by adding an entry to a book then deleting this same entry, the result is the initial state of the book (taking into consideration that these are the only two operations done on the book). By checking this assertion we verify that the state of the book remains coherent. In order to check if this assertion holds, the model is checked using the Alloy Analyzer by executing the **check** stated in Line 22 (the scope, in this case, is 5 atoms, if the scope is not specified it is set to the default value of 3).

### 2.1.2 KodKod model finder

KodKod [24] is a general-purpose relational engine, targeting problems such as design analysis, code analysis, test case generation, scheduling and planning. It was developed by Emina Torlak and Daniel Jackson from MIT Computer Science and Artificial Intelligence Laboratory. It is a model finder (i.e. an engine

```
1  sig Name, Addr { }
2  sig Book { addr: Name → Addr }
3  pred add [b, b': Book, n: Name, a: Addr] { b'.addr = b.addr + n→a }
4  pred del [b, b': Book, n: Name] { b'.addr = b.addr − n→Addr }
5  fun lookup [b: Book, n: Name] : set Addr { n.(b.addr) }
6  assert delUndoesAdd {
7    all b, b', b'': Book, n: Name, a: Addr |
8      no n.(b.addr) and add [b, b', n, a] and del [b', b'', n]
9      implies b.addr = b''.addr
10 }
11 assert addIdempotent {
12   all b, b', b'': Book, n: Name, a: Addr |
13     add [b, b', n, a] and add [b, b'', n, a]
14     implies b'.addr = b''.addr
15 }
16 assert addLocal {
17  all b, b': Book, n, n': Name, a: Addr |
18    add [b, b', n] and n != n'
19    implies
20    lookup [b, n'] = lookup [b', n']
21 }
22 check delUndoesAdd for 5
```

Figure 2.1 – *Alloy Example*

that searches for models of a *formula* in a finite *universe*) provided as a Java API, for a constraint language combining first-order logic, relational algebra, transitive closure.

As of its 4.0 version, the Alloy Analyzer started using the Kodkod model-finder as its background engine. To model a problem in KodKod, similarly to Alloy, we follow these steps:

- **Universe declaration**: the problem's universe groups all its relational constants, called *atoms*.
- **Relations declarations**: in KodKod everything is presented as relations, we start by declaring and describing all the relations of our system using *formulas*. A *formula* in relational logic is a sentence written over an alphabet of relational variables.
- **Global formula definition**: the next step is to write the global formula to be verified. It is a sentence in which the declared relations appear as free variables.
- **Solve problem**: prior to analysis, all relations should be bound. Each relation variable will be bound by a set of tuples drawn from the problem's universe. There are two types of bounds that can be specified, a *lower bound* and an *upper bound*. The *lower bound* contains the tuples that the variable's value **must** include in an instance of the formula, the union of all relations' lower bounds forms a **partial instance** of the problem. The *upper bound* on the other hand contains the tuples which the variable's value **may** include in an instance of the formula. Finding a solution for the problem consists of finding a *model*, which is an *instance* of the global formula. In relational logic, an instance of a formula is a binding of its free variables to relational constants which makes the formula true.

From a syntactic point of view, the formal foundation of KodKod [84] is first-order relational logic. Syntax for such a logic as presented in Alloy [41],

and the notations given below are very close to ALLOY syntax, the only difference is that usual mathematical notations are used here.

First, all the objects used are considered as sets. In particular, an element is identified with the singleton set that contains only this element (identifier). The symbol $\in$, therefore, denotes *set inclusion* as well as set membership.

For sets $A_i$, $1 < i \leq n$, $A_1 \times \ldots \times A_n$ denotes the set of tuples of arity $n$. We write such a tuple $(a_1, \ldots, a_n)$ where for all $i$ with $0 < i \leq n$, $a_i$ belongs to $A_i$.

Subsets of $A_1 \times \ldots \times A_n$ are relations of arity $n$. For example $<$ is a binary relation on natural numbers and can be seen as the subset of $\mathbb{N} \times \mathbb{N}$ defined as $\{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x < y\}$.

For $n = 1$, a unary relation is just a subset. As it may not be a strict subset, any set is a unary relation. Thus, from now on all the objects manipulated are considered as *relations*.

Logical formulas are used to state properties about relations. Such formulas are built using basic predicates such as $\in$ or $=$, and using usual logical connectors: $\wedge$ is logical conjunction, $\vee$ is logical disjunction, $\Longrightarrow$ is logical implication, $\Longleftrightarrow$ is logical equivalence, as well as logical quantifiers: $\forall$ is universal quantification, $\exists$ is existential quantification, $\exists!$ is existential quantification with a unique element. For example the formula in Figure 2.2 states the existence of a unique quotient and reminder for the division of a number $a$ by a non-zero natural number $b$.

There are several operations to manipulate relations. Relation union is denoted by $\cup$, relation intersection is denoted by $\cap$, relation difference is denoted by $-$, relation navigation is denoted by ".". Other operations, union, intersection and difference, have their usual meaning.

Navigation is similar to a relational join, with the difference of the relation join operator joins the relations by column and the matching column is *kept*. The navigation operation joins the last column of the first relation with the first column of the second relation and *drops* the matching column.

For example if *PNCPU* is a relation between physical machine identifiers and *CPU* description and *PhysicalNode* is a set of physical machine identifiers

$$\forall (a,b) \in \mathbb{N} \times \mathbb{N},\ b > 0 \Longrightarrow$$
$$\exists ! (q,r) \in \mathbb{N} \times \mathbb{N},\ a = b \times q + r \wedge r < b$$

Figure 2.2 – *First Order Relation Logic: Formula Example*

*PNCPU*

| | |
|---|---|
| $node_1$ | Intel-1-2.0Ghz |
| $node_2$ | Intel-1-2.0Ghz |
| $node_3$ | Intel-1-2.4Ghz |
| $node_4$ | Intel-1-2.4Ghz |

*PhysicalNode*

| |
|---|
| $node_1$ |
| $node_6$ |

Figure 2.3 – *First Order Relation Logic: Relation and Set Examples*

as shown in Figure 2.3, using navigation, the expression *PhysicalNode.PNCPU* denotes the relation containing only the value *Intel-1-2.0Ghz*.

Finally, the transitive closure of $R$ is denoted by $R^+$ and defined by: if $(x,y) \in R$ then $(x,y) \in R^+$; and if $(x,y) \in R$ and $(y,z) \in R^+$ then $(x,z) \in R^+$.

## 2.2 Linear Programming

Linear Programming is a mathematical technique for generating and selecting the optimal or the best solution for a given objective function. Technically, linear programming may be formally defined as a method for optimizing (i.e. maximizing or minimizing) a linear function for a number of constraints stated in the form of linear inequalities.

Linear programs are problems that can be expressed in the following form:

*Minimize* $\quad a^T x$

subject to :

$Bx \leq c \quad and \quad x \geq 0$

where x represents the vector of unknown variables, a and c are vectors of coefficients, B is a matrix of coefficients, and $a^T$ is the transpose

of a. The expression $a^T x$ is called the objective function. The inequalities $Bx \leq c \quad and \quad x \geq 0$ are the constraints over which the objective function is to be optimized.

Various solvers exist that are able to solve different linear optimization problems. In [53], Meindl et al. present an overview and comparison of existing open-source and commercial solvers. Meindl et al. present an evaluation of the performance of different solvers in the format of a case study of solving multiple attacker problems. Their results show that among the free and open-source solvers they have tested, GNU Linear Programming Kit (GLPK [51]) was the best. Although the latter was significantly slower than the commercial solvers, mainly due to the size of the problems, the *time to solve* for smaller problems was still within reason and thus can still be used.

### GNU Linear Programming Kit: GLPK

The GLPK (GNU Linear Programming Kit) [51] is a free and open-source software, written following the standard ANSI C, that allows solving problems of linear optimization of continuous or mixed variables (discrete and continuous). This kit is composed of a *GNU MathProg* modeling language and a library of C functions (GLPK) using the *Glpsol* solver.

The GNU MathProg modeling language is a subset of the modeling language AMPL [28] (A Mathematical Programming Language). AMPL is a very powerful modeling language that can be coupled with various solvers like CPLEX (IBM ILOG CPLEX Optimization Studio [39], a commercial solver, developed by IBM, designed for larger-scale linear models), XPRESS (Xpress Optimization Suite [27] another powerful proprietary software developed to solve linear models), MOSEK [8] (a proprietary software package used to solve mathematical optimization for linear problems among others)... The limitation to linear problems (LP) of GLPK comes from the solver used. Coupled with the adequate solver, AMPL can be easily used to model more complex problems (i.e. nonlinear optimization, conic optimization... ).

The construction of a model is carried out from elementary bricks which

are called modeling objects. For the construction phase, there are 5 types: *Parameter*, *Set*, *Variable*, *Constraint* and *Objective*. The *Parameter* and *Set* objects allow the definition of all the problem data, *Variable* defines all the variables of the problem, *Constraint* to define the constraints of the problem and finally *Objective* is the objective function of the problem. Each brick is made up of expressions. The set of construction possibilities is quite broad as long as the problem remains linear.

The modeling of an optimization problem is divided into two parts:

- The **Model section** contains all the declarations, the calculable parameters and the denials of the constraints and the objective.
- The **Data section** contains all the fixed data (values of the parameters, the content of the sets).

Let's take for example the Dantzig (1963) optimization problem which aims to minimize the transport cost while respecting the customers' demands and factories supply. The model section of this problem is presented in Figure 2.4, an example corresponding to the data section is presented in Figure 2.5. The next step will be to feed the model to the *Glpsol* solver.

```
/*Set of Factories*/
set I;
/*Set of Customers*/
set J;
/*Factory supply in case number*/
param a{i in I};
/*Customer demand*/
param b{j in J};
/*Distance between customer and factory in thousands of kilometers*/
param d{i in I, j in J};
/*Transport Cost in dollars per case per thousands of kilometers*/
param f;
/*Calculating the transport cost in kilo−dollars per case*/
param c{i in I, j in J} := f * d[i,j] / 1000;
/*Numbers of cases transported from factory i to customer j*/
var x{i in I, j in J} >= 0;
/*Or : var x{i in I, j in J} >= 0, integer;*/
/*Goal : minimize the total transportation cost in kilo−dollars*/
minimize cost: sum{i in I, j in J} c[i,j] * x[i,j];
/*Constraints*/
/*Constraint due to the maximum a factory i can supply*/
s.t. CteSupply{i in I}: sum{j in J} x[i,j] <= a[i];
/*Constraint due to the a customer j 's demand*/
s.t. CteDemand{j in J}: sum{i in I} x[i,j] >= b[j];
```

Figure 2.4 – *GNU MathProg Model Section Example [66]*

```
data;
set I := Seattle San−Diego;
set J := New−York Chicago Topeka;
param a := Seattle  350
 San−Diego  600;
param b := New−York  325
 Chicago  300
 Topeka  275;
param d :  New−York  Chicago  Topeka :=
 Seattle  2.5  1.7  1.8
 San−Diego  2.5  1.8  1.4;
param f:= 90;
end;
```

Figure 2.5 – *GNU MathProg Data Section Example [66]*

## 2.3 Coq proof assistant

The Coq proof assistant developed at the French Institute for Research in Computer Science and Automation (INRIA) and first released in 1989 is an interactive theorem prover. The Coq environment allows the definition of mathematical assertions. The Coq compiler checks and help find proofs of these assertions. Coq is based on the calculus of inductive constructions [63], a higher-order typed $\lambda$-calculus. Coq and the calculus of inductive constructions are based on the Curry-Howard correspondence, meaning, a type corresponds to the statement of a theorem, and a program to the proof of a theorem.

The core of Coq is very small. For example, there is no pre-defined data type. All definitions are typed in Coq. Therefore a user-defined type has a type, named a sort. There are three sorts in Coq: **Set** is the sort of types that correspond to types found in usual programming languages. It is the sort of the "computational" types. **Prop** is the sort of "logical" types. Both **Set** and **Prop**

```
1  Definition id: ∀ (A:Type), A → A :=
2   fun A x ⇒ x.
3
4  Inductive nat : Set :=
5  | O : nat
6  | S : nat → nat.
7
8  Fixpoint add (n1 n2:nat) : nat :=
9   match n1 with
10  | O ⇒ n2
11  | S n1 ⇒ S(add n1 n2)
12  end.
13
14 Lemma add_n_O: ∀ n,
15  add n O = n.
16 Proof.
17  induction n as [ | n IH ].
18  − trivial.
19  − simpl. rewrite IH. trivial.
20 Qed.
```

Figure 2.6 – *Coq Example*

are typed and their type is **Type**. Most of the time when using Coq, the type of **Type** will be displayed as **Type**. Actually, there is a countable infinity of sorts **Type**.

In Gallina, the language of Coq, a definition contains three components: a name, a type, and a term. For example, the polymorphic identity function can be defined as shown in Lines 1–2 of Figure 2.6.

As the core does not contain predefined types (but the sorts **Set**, **Prop** and **Type**), Coq provides a mechanism to define new inductive types. This is done by giving a list of *constructors* for values of the defined type. For example, Peano natural numbers are defined in Lines 4–6 of Figure 2.6. There are two constructors for values of type nat: O and S the latter taking a nat as an argument.

Functions are most often written using pattern matching as in Lines 8–12 of Figure 2.6. For each possible way of constructing a value of the type of the matched expression (in this case n1 of type nat), the pattern matching construct returns (after ⇒) a specific result. The patterns (on the left-hand side of ⇒) may contain variables: in case the matching succeeds, the free variables are bound to the matched values in the right-hand side of ⇒. Note that add is a recursive function (**Fixpoint** keyword). Only terminating functions are allowed in Coq, in this case, the system checks the termination by checking that the recursive call is done on a strict syntactic subterm of n1.

Coq is a proof assistant that makes it possible to define theorems and prove

them. As mentioned at the beginning of this section, a Coq definition contains three elements: a name, a type and a term. In the case of a theorem (or lemma, proposition, *etc.*), the term (i.e. the proof) is usually not written as a program (even though the Curry-Howard correspondence states a program and a proof are the same thing) it is written in the proof script language of Coq instead. In the code of Figure 2.6, add_n_O is the name of the lemma, ∀ n, add n O = n is its type, and the proof script between **Proof** and **Qed** builds a term that is the proof of the lemma.

One important feature of Coq is that computational terms can be embedded into types. For example the module Vector of Coq standard library contains the following inductive type definition:

```
1 Inductive t (A : Type) : nat → Type :=
2 | nil : t A 0
3 | cons : ∀ (h:A) (n:nat), t A n → t A (S n).
```

The size of a value of this type contains the length of the vector. For example, a value of type Vector.t nat 10 is a vector containing ten nat values. Vector.t is called a *dependent type*.

This feature can also be used to define predicates as inductive types. For example, the =< predicate on Peano natural numbers is defined in the Coq standard library as:

```
1 Inductive le (n : nat) : nat → Prop :=
2 | le_n : le n n
3 | le_S : ∀ m : nat, le n m → le n (S m).
```

More generally, Coq functions can take both computational values and types as arguments, and also return them as results. As values of some types (like add_n_O) are proofs, Coq functions can also take proofs as arguments and return proofs as results. We use these features in the Coq code generated from Alloy models.

It is also possible to *declare* values in Coq, in this case we have only a name and a type. In the case of a value that needs a proof, it means an axiom is introduced in Coq's logic. Note that when such declarations can be written inside

a section, in such a way that at the closing of the section, all the elements that depend on these hypotheses are added as additional arguments corresponding to these hypotheses.

Some of the well-known applications of the Coq proof assistant include the study of properties of programming languages, the CompCert [47,48] compiler certification project. Proof of correctness and verification of C programs with Frama-C [16] (a FRamework for Modular Analysis of C programs).

# Literature Review 3

Contents



## 3.1 Introduction

Works have been conducted in the context of Cloud brokerage, Cloud federations and multi-Cloud, all with the aim to offer better value and assistance to Cloud customers in the process of integrating the Cloud. This chapter represents a summary of some of the contributions regarding the different concepts related to Cloud brokering, customers' needs as well as position our proposed solution among the previously proposed ones. The works surveyed range from works presenting effective deployed brokering tools to conceptual models and algorithms that could be implemented into brokering tools. Section 3.2 presents an overview of the different approaches and results. Another project I worked on during this thesis is the translation of models written in Alloy into Coq syntax in order to prove properties about them. Section 3.3 summarizes works conducted to verify and prove Alloy specifications.

## 3.2 Cloud Brokering Related works

The works presented in this section are categorized into brokering tools in Section 3.2.1 and models and algorithms in Section 3.2.2. We suggest a taxonomy of the different works presented here in Section 3.2.3 .

### 3.2.1 Cloud brokering tools

In the following section, we present research works that were portrayed and identified as tools from the research papers here cited. In other words, here are grouped the works that present a fully functioning tool, as well as, academic works where researchers describe implementations of tools for Cloud services brokering.

SMICloud [31] is a framework to systematically measure all the Quality of Service (QoS) attributes proposed by CSMIC (Cloud Service Measurement Consortium), which are: *Accountability, Agility, Cost, Performance, Assurance, Security and Privacy, Usability*, and then rank the Cloud services based on these attributes. The major goal of the framework is to help Cloud customers to find the most suitable Cloud provider and therefore can initiate SLAs (Service Level Agreements). It is a decision-making tool, designed to provide an assessment of Cloud services in terms of KPIs (Key Performance Indicators) and user requirements. Customers provide their application requirements (essential and non-essential) to this framework which gives a list of Cloud services where customers can deploy their applications. Also, by using the techniques given in their work, Cloud providers can identify how they perform compared to the other competitors and therefore assess the market and improve their services.

CloudCmp [49] is a framework to estimate and compare the performance and costs of deploying an application on a Cloud. It characterizes the common set of services a Cloud provides, including computation, storage, and networking services, benchmarks each service's performance and cost, and combines

the benchmarking results with an application's workload to estimate the application's processing time and costs. In other words, this framework helps potential Cloud customers to estimate the performance and costs of running their application with a certain Cloud provider without actually deploying the application.

Cloud Offerings Advisory Tool (COAT) [3] is a tool proposed to filter the variety of Cloud service offers presented to Cloud customers based on some security and privacy attributes (Data preservation, deletion and portability after termination, Data location and transfer, Data disclosure and integrity, Subcontracting, Direct and indirect liability, Ownership of data, Change of terms of service by Cloud providers and Encryption). This tool aims to educate the user on the meaning of these attributes and offer a level of guidance to understand the contractual terms in the providers' offers. To summarize its functioning, COAT is an independent web-based broker that matches user requirements to offers provided by Cloud service providers, compares these offers, explains the terms of these offerings, suggests best offerings that match the user requirements and gives general guidance to users on service offerings. Overall the tool [3] aims to find offers and present users with the necessary information to understand the terms, and it is up to the customer to directly communicate with the provider to get the recommended offer.

CloudSurfer [30] is a Cloud broker that browses through Cloud service offerings that fulfill a set of security requirements. The latter were selected from a repository tailored for Cloud computing based on standards and guidelines from organizations such as NIST, Cloud Security Alliance and Enisa. CloudSurfer was developed as part of a school project. The requirements that could be entered by the customer are grouped in the following categories: Data storage, Data transfer, Access control, Security procedures, Incident management, Privacy, Hybrid Clouds. The broker would then return the possible services, or service providers depending on what the customer asked for. Due to the lack of a standardized machine-readable contract language that can be used

for automatic reasoning and discovery, they have made an assumption that providers would provide an XML file containing their offers under the form of the requirements and so they recover the customer's demand under the same form and do the matching in order to get the offers fulfilling the customer's demand. The limitations of such a model are the fact that it's limited to only finding a solution from one provider, that it only does intermediation and that it does not consider the customer's functional requirements.

Schlouder [54] is a broker for *IaaS* Cloud resources able to provision and schedule online (*i.e.* dynamically as jobs arrive, without knowledge of future submissions) independent jobs or static workflows according to strategies chosen by the customer. Besides its function of job execution management, it allows the user to predict the result of one scheduling strategy or another through simulation. The strategies for scheduling and provisioning resources offered by this broker are heuristics for computing a bi-objective optimization problem based on monetary cost and make the span of the whole workload.

mOSAIC [57] is a project that aims at improving the state of the art in Cloud computing by creating, promoting and exploiting an open-source Cloud application programming interface and a platform targeted for developing multi-Cloud oriented applications. One of the main goals is obtaining transparent and simple access to heterogeneous Cloud computing resources and to avoid locked-in proprietary solutions. Also, a detailed ontology for Cloud systems that can be used to improve interoperability among existing Cloud solutions, platforms and services, was presented in their work.

STRATOS [64] is a broker service to facilitate cross-Cloud, application topology platform construction and runtime modification. It allows the customer, or namely the application deployer, to specify what is important to them in terms of KPIs so that when a request for resource acquisition is made it is able to consider the request against all providers' offerings and select the acquisition that is best aligned with the objectives. It was a step toward developing a broker service to facilitate the use of cross-provider Cloud offerings. Experiments

presented in this paper demonstrate how cross-Cloud distribution of an application could decrease the cost of the topology and create one that is better fits the deployer's objectives.

QBrokerage [6] is a Cloud brokering approach that exploits the information that commercial providers make available to their customers, such as, virtual machines (VM) cost and characteristics, in order to help the broker's customers in finding a deployment solution that meets Quality of Service (QoS) requirements. The latter should be formally expressed by Service Level Agreements (SLAs). The aim of QBrokerage is to find a deployment solution while avoiding scalability and vendor lock-in issues. In [6], Anastasi et al. present a design and implementation of their solution following some software requirements such as meeting the heterogeneous QoS requirements of applications; finding near-optimal solution according to customers preferences while avoiding provider lock-in; supporting providers with various cost models and scaling up with hundreds of providers. A customer submits an application to QBrokerage, containing both functional and non-functional aspects, such as specific security requirements that a provider should respect, following a specification format, to mention the Open Virtualization Format (OVF). Qbrokerage leverages a Genetic Algorithm (GA) approach, for finding near-optimal solutions in large search spaces, to find a suitable deployment solution for the customers' requests. GA approach is considered suitable in the context of Cloud Computing, due to the continuous enrichment of QoS models, as well as its flexibility and ability to support multiple constraints and the injection of additional constraints with minimal interventions on the algorithm.

FCSB (Federated Cloud Service Broker) [38] is a broker designed for Public Administrations (PA). A solution that aims at overcoming existing challenges in the public sector such as Governance, Interoperability and Portability, SLAs compliance and assessment, Intelligent discovery of Cloud services, cross border interoperability and legislation awareness. Ibarra et al. present in [38] an overview vision of the design, implementation, functionalities and features of

the FCSB. The high-level architecture of the latter is envisioned to have seven main components that try to overcome the specific challenges faced by the PAs mentioned earlier. These components are as follow:

- *Service Management:* executes and manages all the operations related to the services offered by the FCSB, starting from customers requests to providers offers;
- *User Management:* controls the different FCSB users' activities
- *Service Contract Management:* executes and manages all operations related to Service Contracts in the FCSB
- *Console:* implements the interface between the different users and the FCSB
- *Interoperability:* performs the portability between the different services and the communication between the FCSB and the CSPs and the other FCSBs
- *Financial Management:* performs the financial operations with the different users of the FCSB
- *Regulatory Framework Assessment:* judges the compliance of the services with the different legislations

For more details about each module and technologies that Ibarra et al. judge fitted for this solution, refer to [38].

In [60], Nair et al. present the concept of Cloud brokerage in relation to the concept of Cloud bursting. The latter is a hybrid Cloud deployment model, where an application deployed and running in a private Cloud, or data center, bursts into a public Cloud once the computing capacity demand spikes. They also propose a possible architectural framework capable of powering the brokerage based Cloud services. Their framework was developed in the scope of OPTIMIS, an EU FP7 project that aims to offer functionality substantially

beyond that of current Cloud infrastructure. In their paper, they describe different ways to model customers needs that a brokerage solution is supposed to serve. The three described models are *enterprise use of multiple Cloud providers*, *brokering multiple providers to provide a SLA-based tiered pricing model* and *Cloud aggregation ecosystem*. They then propose an architecture that implements a broker model involving the *Brokering multiple providers to provide a SLA-based tiered pricing model*. According to their architecture the functional components needed in a Cloud broker are: *Cloud API*, *Deployment Service*, *Staging/ Pooling Service*, *Identity and access management*, *SLA monitoring*, *Capability Management and Matching*, *Audit*, *Risk Management*, *Network/ Platform Security*, *Usage Monitoring* , *SLA Management*. These components work together in order to provide multiple capabilities to the broker's customers. Ensure the confidentiality and integrity of data, match the customer's requirements with the provided service, negotiating SLA, maintain SLA performance check, manage the API, securely transfer data, provide effective staging and pooling services, manage access control, manage risk and handle Cloud burst/spill-over situations. Nair et al. sum up their paper by identifying the necessary steps to provide a brokering service in storage and compute use cases.

**Summary** SMICloud [31], CloudCmp [49] and Cloud Offerings Advisory Tool (COAT) [3] are similar brokering tools, while they are useful when trying to rank the performance of the different providers and their compatibility to the customer's demand, they don't offer additional security measures nor do they fully intermediate the Cloud integration process. Depending on the ranking and simulation results the customer will have to get in direct contact with the chosen service provider. In a similar manner, CloudSurfer [30] is a tool that does not intermediate the relation between the Cloud customers and providers, it ranks different Cloud services or Cloud services providers that respects a set of security requirements described by the customer. Schlouder [54] is an online job scheduling IaaS broker that enables customers to simulate and chose their scheduling strategies.

The common theme about these works is that we have a one to one relationship between a customer and a Cloud service provider. Trying to avoid provider lock-in and benefit from using multiple providers has attracted many works. mOSAIC [57] and STRATOS [64] present tools for assisting and creating applications to be deployed on multi-provider platforms. Brokers presented in QBrokerage [6], [60] and FCSB (Federated Cloud Service Broker) [38] aims at finding deployment strategies in a multi-Cloud environment while respecting QoS / SLA agreements.

### 3.2.2 Cloud brokering models and algorithms

This section will group works presenting algorithms, architectures, models and techniques that can be used in a Cloud brokerage solution. These are works where researches focused more on presenting and introducing innovative algorithms and techniques that were not implemented into a functioning tool.

In [69], Rogers and Cliff show that financial brokering in Cloud Computing has potential as a viable commercial proposition and that all parties participating in the brokering operation, can potentially benefit from it, to mention: Cloud providers by having a consistent forecast of the market, Cloud brokers by making a financial profit and customers by benefiting from lower-cost, can potentially benefit from it. In fact, they have built a brokerage model based on trust and composed of two periods. In the first period, the customers would submit a probability of the amount of units they will need for the next period (i.e. Period 2), and the broker shall then in the same period (i.e. Period 1) do the necessary calculations to see if he has enough resources to cover all his customers', potentially, needed units or reserve the needed units from providers, at a discounted price. In the second period, on one hand, the customer could claim or not the resources he asked for and pays a certain amount in either case, on the other hand, the broker has to deliver the claimed resources. In [69] is presented in more detail how the calculations of the amount paid by the customer to the broker and the broker to the providers, are done in both the cases of claimed and unclaimed resources. They have simulated different types of mar-

kets and found that a broker will always have his share of benefits. A theory that was shut down by Cartlidge and Clamp in [15], where they weren't able to replicate the simulation in [69]. They discovered two errors made in [69]'s code that has caused overestimated results. First, the reservations bug, when there is a bug in the software code that causes the broker to purchase fewer reservations than the model suggests (each new reserved unit purchase is effectively allocated to two units of demand). Second, the payment bug, during months that demand is greater than capacity, the broker does not pay the monthly utilization charge for reserved units. After running the same simulations with the corrected program they have found that the broker's benefits are drastically less than what was presented in [69]. They also present in [15] other innovations in Cloud computing that, for them, closes the window of opportunity to commercially exploit a basic brokerage model.

The work in [89] proposes a Cloud brokerage service that considers both, pricing scheme when it comes to Cloud users and reservation methods when it comes to Cloud providers. They introduce the utilization of two types of priorities in the design of the priority aware pricing and priority-based reservation algorithms. By analyzing the widely used Google cluster trace, Wang et al. found that jobs of Cloud users tend to always have different levels of priorities. Low priority jobs can be preempted by high priority ones. Thus, make the decision to apply priority characteristic into the resource reservation of Cloud broker to reduce the cost. Wang et al. propose a priority aware pricing scheme, *PriorityPricing*. The latter requires that Cloud users trade with Cloud broker by fairly charging job requests based on priorities. They have designed resource reservation algorithms that consider the priority of users' requests to solve the idle resource waste problem for the broker. And then, evaluate the effectiveness of the proposed algorithms by conducting simulations with a 1-month Google trace. Results presented show that the broker's profit can be increased up to 2.5x than that without considering priority for the offline algorithm, and 3.7x for the online algorithm.

Cloud customers' needs vary, some look for cost efficiency some for performance. In [59], Naha and Othman propose three brokering algorithms. The first algorithm is a Cost-Aware (CA) brokering algorithm. Second, a Load-Aware (LA) brokering algorithm. And a brokering algorithm that takes into consideration both cost and load called, Load-Aware over Cost (LAOC). In their simulation, Naha and Othman considered six different Cloud scenarios where they compared the three algorithms they have suggested with an already existing brokering algorithm called, Service Proximity Based Routing (SPBR). They also used two load balancing algorithms: Round Robin (RR) and an algorithm they have proposed, State-Based Load Balancing (SBLB). The latter is an algorithm, they proposed, that dynamically assigns tasks to idle hosts. It also assigns tasks to available hosts from a task queue. Such a technique prevents the host from becoming heavily loaded. It has been shown through the simulations that an effective load balancing algorithm saves operational costs, improves user satisfaction and leads to accelerate overall performance.

In [77], Sundareswaran et al. present a Brokerage-Based decision-making approach for Cloud service selection. This Cloud broker analyzes and indexes the service providers according to the similarity of their properties. Upon receiving the service selection request from an end-user, the Cloud broker will search the index to identify a ranked list of candidate providers based on how well they match the user requirements. This list forms the basis of the end user's final decision. The novelty presented in [77] is the indexing method used for Cloud providers, which plays a big role in the service selection method. A Cloud Service Provider (CSP) index is created in three steps. Starting with *Property Encoding*, *Relationship Encoding* and then *Index Key Generation*. The internal nodes of the CSP-index have a similar format as the B+ -tree and serve as the search directory. Each entry in the leaf node of the CSP-index has the following format:

$$< Key_{sp}, SID, P_1, P_2, ..., P_{10} >$$

with $P_i$ the CSP's properties ($P_1$: Service Type, $P_2$: Security, $P_3$: QoS, $P_4$: Mea-

surement units, $P_5$: Pricing Units, $P_6$: Instance Sizes, $P_7$: OS, $P_8$: Pricing, $P_9$: Pricing sensitivity, $P_{10}$: Subcontracting). A customer would approach the broker with a certain *query* defined as follows:

$$Q =< (QP_1 : D_1), (QP_2 : D_2), ..., (QP_k : D_k) >$$

with $D_i$ the value of the property $QP_i$ demanded by the customer, $QP_i$ are equivalent to the CSP's $P_i$. In order to answer the customer's demand, the broker follows four steps.

- *Step1*: Query Encoding,
- *Step2*: K-nearest neighbor search,
- *Step3*: Refinement,
- *Step4*: Consideration of Special Criteria.

In [18] is presented a different take on Cloud actors, where they consider two types of providers that complement each other in the process of providing a complete Cloud service. These are known as Network Service Providers (NSP) and Cloud Service Providers (CSP). Multiple assumptions are considered about the type of services and offerings handled by this solution. Cucinotta et al. present in [18] a brokering logic for distributed Cloud application deployment with guaranteed end-to-end QoS, achieved as an end-to-end composition of SLAs established between the different actors (i.e. customers, broker, NSP and CSP). Cucinotta et al. have mathematically modeled and formalized this brokering approach as a mixed-integer gemetric programming optimization program. There exist different solvers that specialize in solving such programs, to mention those found in the GAMS suite (`http://www.gams.com/solvers`).

anyBroker [42] is a concept design and architecture of a Cloud brokerage system that supports integrated service concept in service provisioning and management; SLA based service life cycle management; Flexible connection

and interaction with heterogeneous Clouds and customer's service requirement based on best-fit Clouds in respect of price, location, security, availability. The *anyBroker* system is composed of three main parts: *anyBroker portal*, *anyBroker core engine* and *anyBroker Cloud connection proxy*. *anyBroker portal* represents the front-end of the Broker for the communications with the Cloud service consumers and providers and it also includes multiple business support functionalities. On one hand, *anyBroker core engine* is in charge of the main brokerage functionalities, such as to request verification, service provisioning, arbitrage, monitoring, service life cycle management. On the other hand, *anyBroker Cloud connection proxy* is in charge of the interactions between the broker and Cloud service providers or network service providers.

In [37], Guesmi et al. present a methodology and prototype of a Brokering solution that takes into consideration customers' functional and non-functional requirements and finds if exists, a placement in a multi-provider Cloud environment. Customers approach the broker with a description of the model they would want to deploy on the Cloud. The model is composed of functional requirements (i.e. number of virtual machines needed and their properties, CPU speed, OS and Location) and non-functional or security requirements under the form of relations relating the virtual machines, which gives customers the possibility to personalize the interaction and access to their virtual machines. Once the broker receives the customer's request, the first step is to verify its consistency and return a counterexample in case an incoherence is found. If not the broker pursues to find a placement in a multi-Cloud environment, if a placement is found, it is returned to the customer who can validate so that the broker can deploy the placement solution, or chose to browse for another solution. Guesmi et al. use a formal analysis tool named Alloy, to do the verification and placement search.

Tordsson et al. propose in [82] a Cloud brokering approach that aims to optimize virtual machine placements in a multi-Cloud environment. They describe a broker architecture that would abstract the deployment and manage-

ment of the infrastructure components. They present, and evaluate, an algorithm for application placement across multiple Clouds while taking into consideration price and performance with the possibility of additional constraints in terms of hardware configuration and load balancing. In their architecture, a broker has two main actions: First, finding the optimal placement of the virtual resources across a given set of Cloud providers (this is where the Cloud scheduling algorithm comes in handy), second, the management and monitoring of these virtual resources (this is the job of a component named virtual infrastructure manager, where they relied on the OpenNebula virtual infrastructure manager). The scheduling algorithms described, in the paper, are aimed at scheduling applications in a static Cloud brokering scenario. The goal is to deploy a number of virtual machines across a static number of available providers to improve a criterion, chosen by the customer, such as performance, cost or load balance. AMPL and CPLEX were the chosen modeling language and solver they used for their scheduler. They evaluate the described architecture and scheduling algorithms through a case study where they deploy a set of high throughput computing clusters across commercial Cloud providers (details in [82] Sections 4, 5 and 6). Their most important finding was that deployment in a multi-provider environment improves performance.

Subramanian and Savarimuthu present a brokering architecture and algorithm for optimal resource provisioning in a heterogeneous multi-Cloud environment in [76]. The proposed broker provides an optimal cost deployment plan, that takes into consideration attributes defined in the service measurement index (SMI) with additional physical and legal constraints. Customers describe their service requests in the format of number and type of virtual resources, optimization criteria and other constraints. In [76] Cloud providers are ranked according to a service measurement index (SMI) score. SMI, in a nutshell, is a framework of attributes that enables the comparison of the different Cloud services provided by a set of Cloud providers. These attributes are Accountability, agility, assurance, financial, performance, security and privacy, usability. The optimization criteria, described by the customer, are either, in

the format of weights attributed to each of the aforementioned characteristics or a pairwise comparison method. Once the broker receives the request, the first step is to find and rank according to the SMI score the Cloud providers that meet the customer's service request. Then, he implements an algorithm to produce a deployment plan that reduces the total cost of the virtual infrastructure. The latter is obtained by using the mixed integer programming formulation and relying on the benders decomposition algorithm to solve. While the solution proposed in [76] takes into consideration some interesting criteria and allows the customer to define a personalized service request, it does not offer an additional layer of security. It only considers the security measures already given by the Cloud providers when calculating the SMI score.

Cloud Service Broker Demand Response (DR-CSB) is a mechanism that was introduced in [21]. It aims to maximize the profit of CSB under dynamic customer demands with respect to the capacity and availability constraints, to mitigate the insufficient provisioning problem. In this work, Deng et al. formulate an optimization problem of profit maximization for the CSB, as well as, employ economic demand response mechanism to allow Cloud customers to adjust their consumption with the dynamic aspect of Cloud service prices. The provisioning process is divided into two periods. During the reserved period, Cloud service providers (CSPs) provide reserved instances to the CSB according to the customer demands; If the number of instances required by customers exceeds the existing number of reserved instances, the broker goes into the on-demand period where on-demand instances are provisioned from CSPs. In order to maximize the broker's profit, the DR-CSB adjusts the instance price accordingly to incentivize customers to independently regulate their own demands. Simulations to evaluate the DRCSB can be found in [21]. Results presented show that CSB's profit can be increased varying from 8% to 20%. Customers also achieve savings up to 37%.

The work in [2] proposes an architecture for a brokerage model specifically for multi-Cloud resources. This model targets spot markets specifically

and it tries to integrate the resource brokerage function across several Cloud providers. A spot market trades unused computing resources with a 'bid or ask' mechanism. Customers publicly announce the maximum price they are willing to pay for the product or service, and providers the minimum price they are willing to sell for. The architecture presented supports the matching process by finding the best match between customer requirements and providers, offers are matched taking into consideration the lowest possible cost available at the time of the request for the customer in the market. The architectures aim to provide the coordination techniques using tuple space [1] and adapted to the Cloud spot market.

Ding et al. present in [23] a QoS-aware resource recommendation method to support Cloud resource selection considering both functional and QoS attributes. They present a method to help with the recommendation of adequate Cloud resources by integrating a multi-attribute matching metric, group customer evaluation and price utility. The matching mechanism they introduce is a multi-attribute matching mechanism to support QoS-aware resource filtering. For a detailed presentation of this method refer to Section 3 of [23].

In [4], Alsina et al. are interested in Virtual Machine Planning for a Cloud Broker. In their definition, the latter owns a number of Cloud reserved instances that he offers to his customers. The broker presented in [4] has as objective to maximize its revenue by efficiently managing its reserved resources while taking into consideration parameters such as geographical localization of resources and users, different types of applications and data transfer costs. In order to solve this problem while offering appropriate QoS to users, they propose and evaluate six different heuristics.

Although in [9], Aral and Ovatman might not be presenting a "Cloud Broker" I found their work interesting to present. They are interested in optimizing the resource allocation in a Cloud federation. Thus, they propose a

[1] tuple space is a coordination model for parallel processing and data sharing proposed in the Linda model. It has an architecture that is suitable for Cloud service communication

heuristic, implementing subgraph matching, to output an allocation for a set of requested VMs in the Cloud data centers while taking into consideration resource capacities, VM topologies, performance and resource costs. Topology Based Matching (TBM) heuristic, is the title of the heuristic suggested in [9]. TBM considers both the topology of the Cloud federation, VM request and employs an algorithm based on LAD subgraph isomorphism solver to find a match between the request and a subgraph of the federation topologies. As an evaluation of their heuristic, Aral and Ovatman used CloudSim framework to compare it against three different heuristics: *Arbitrary First Fit(AFF)*, where the first available data center is chosen; *Latency First Fit(LFF)*, where data centers are probed in an increasing order of latency to the user and *Load Balancing(LBG)*, where a VM is assigned to the data center with the lowest resource utilization. This evaluation showed that TBM outperforms both AFF and LBG but struggles to get the same results as LFF.

**Summary** Here we have presented a variety of Cloud brokering solutions, some presenting general architectures: anyBroker [42] a concept design and architecture of a Cloud brokerage system, in [82] a Cloud broker architecture for optimizing virtual machine placement in a multi-Cloud environment and in [76] brokering architecture and algorithm for resource provisioning in a multi-Cloud environment. Those presenting algorithms and mechanisms to maximize broker's profit: in [69] is presented a financial Cloud brokering, in [89] different pricing algorithms, multiple algorithms: Cost Aware, Load Aware, Load Aware Over Cost are described in [59] and [21] Cloud Service Broker Demand Response a mechanism to maximize the profit of the broker under dynamic demands. Different methods to help customers in the process of decision making: an indexing method to rank different Cloud providers is presented in [77] and in [23] a QoS aware method to recommend Cloud resources. Then we find works that treat the problem of Cloud brokering in a multi-Cloud environment: in [2] a multi-Cloud brokerage model for spot markets, in [4] are presented different heuristics for efficient virtual machine

planning for a Cloud broker, a QoS brokering logic for distributed Cloud application deployment in [18] and in [9] a heuristic for resource allocation optimization in a Cloud federation.

**Security Concerns** Many of the proposed Cloud brokering solutions in academia, mentioned here above, as well as those commercialized tackle issues related to performance, cost, QoS and respect of SLA contracts. Those that make an effort to integrate security only consider the basic security levels and services proposed by the Cloud providers. As far as my researches went, other than the work proposed by A. Guesmi's in [37], which is the initial solution we carried on in this thesis, I haven't been able to find a work that fully considers the customer's personalized needs from a security standpoint. In this thesis, we propose a Cloud broker that aims to do that. Our solution gives the possibility to customers, wanting to migrate their application to the Cloud, to describe their request in the format of components, precise their functional description as well as the security relations between them. The broker then verifies the consistency of this demand and finds an adequate cost-aware placement strategy that respects both their functional and non-functional requirements.

### 3.2.3 Taxonomy

Here we present a summary of the aforementioned works in regard to different criteria, that we judge important for a well-rounded Cloud broker. In fact, we consider that a broker should be able to intermediate the full process of integrating the Cloud. Starting by assisting the customer through defining, not only his functional needs but also non-functional ones, until deploying his architecture and delivering it. We also believe that taking full advantage of the multi-Cloud architecture is very important in finding the best-suited placement strategy for the customers' demands as well as shielding them from provider lock-in related issues. Finally, assuring customers' security is of high importance and thus a broker should implement the necessary requirements to stay accountable.

This set of criteria is presented in the different columns of Table 3.1. With each of the entries we are trying to answer the following questions:

- Intermediation: *does the work present an intermediate between the customer and provider?* In order to ease the process of integrating the Cloud for potential customers, we judge that intermediating the whole process and making it as seamless as possible, meaning no direct contact between the customer and the service providers to be a very important criterion.

- Functional Requirements: *does it take into consideration the customers' functional requirements into consideration?* This goes hand in hand with the previous criteria. In fact, to intermediate the process of migrating the customer's architecture into the Cloud, the broker has to take into consideration at least the functional requirements in order to find an adequate placement strategy.

- non-Functional Requirements: *what are the non-Functional Requirements that this work considers?* Different works have different takes on non-functional requirements, in this column, we're trying to summarize the list of the non-functional requirements taken into account.

- Multi-Cloud: *does the work implement concepts similar to multi-Cloud to avoid issues like provider lock-in?* There are many benefits of using multi-Cloud, starting with avoiding the provider lock-in issue and finding a more catered Cloud solution to customers' demands. Thus, we include it as a criterion for judging the quality of the brokers.

- Security Assurance: Pearson [65] gives several requirements for accountability approaches. First organizations should establish policies that follow some recognized external criteria and that allow for external enforcement, monitoring and auditing. Such a requirement calls for Cloud services certification [78] that would enhance the trust of customers in Cloud solutions. We translate this, to the need to having a well-documented or easily documented solution. Secondly, organizations should provide

*transparency* and mechanisms for customer participation from the very early stages. Finally, organizations should offer mechanisms for remediation. Techniques to support accountability range from verifiable evidence collection [7] to formal proved mechanism with proof certificates that a security property holds [87]. We refine the security assurance criteria into the three following columns:

  - Documented: *is the approach well, or could be easily, documented?*
  - Transparent: *are the techniques used transparent and include the customers' in the decision process?*
  - Formal Methods: *does it have any properties guaranteed using formal methods?*

- Implementated Tool: *does it have a testable implementation or a fully functional tool?* Last thing to round up the evaluation of the different works, we get interested in seeing if the brokerage solution has been implemented.

From Table 3.1 we show that none of the proposed solutions, presented in this section, fulfill all of the requirements. In this work, we propose to fill the gap and provide a tool that implements our proposal. Chapters 4 and 5 describe formally our approach.

| Brokerage Solution | Intermediation | Functional Requirements | non-Functional Requirements | Multi-Cloud | Security Assurance | | | Implemented Tool |
|---|---|---|---|---|---|---|---|---|
| | | | | | Documentation | Transparent | Formal Methods | |
| CloudCmp [49] | No | Yes | Performance<br>Cost | No | No | Yes | No | Yes |
| SMICloud [31] | No | Yes | QoS attributes proposed by CSMIC:<br>Accountability, Agility, Cost,<br>Performance, Assurance,<br>Security and Privacy offered by the provider,<br>Usability | No | No | Yes | No | Yes |
| COAT [3] | No | Yes | Security and Privacy:<br>Data preservation, Data location and transfer,<br>Data disclosure and integrity, Data Ownership,<br>Encryption | No | No | Yes | No | Yes |
| CloudSurfer [30] | No | No | Access control<br>Provider's security procedures<br>Incident management<br>Privacy | No | Yes | Yes | No | Yes |
| STRATOS [64] | No | Yes | KPIs [Zachos et al.] | Yes | Yes | Yes | No | Yes |
| QBrokerage [6] | Yes | Yes | QoS under OVF format | Yes | No | Yes | No | Yes |
| FCSB [38] | No | No | SLA<br>Legislation awareness | Yes | Yes | Yes | No | Yes |
| [37] | Yes | Yes | Personalized Security Requirements | Yes | Yes | Yes | Yes | No |
| [60] | Yes | Yes | SLA<br>Security offered by the provider | Yes | Yes | No | No | Yes |
| [69] | Yes | Yes | Cost | No | Yes | Yes | No | No |
| [59] | No | Yes | Cost<br>Load balancing | No | Yes | No | No | No |
| [77] | No | Yes | Security offered by the provider<br>QoS<br>Cost<br>Subcontracting | No | Yes | Yes | No | No |
| [18] | Yes | No | End-to-end QoS | Yes | Yes | No | Yes | No |
| anyBroker [42] | Yes | Yes | SLA<br>Security offered by the provider | Yes | Yes | No | Yes | No |
| [82] | Yes | Yes | Performance<br>Cost | Yes | Yes | No | Yes | No |
| [76] | Yes | Yes | QoS attributes proposed by CSMIC:<br>Accountability, Agility, Cost,<br>Performance, Assurance,<br>Security and Privacy offered by the provider,<br>Usability | Yes | Yes | Yes | No | No |
| DR-CSB [21] | Yes | Yes | Cost | Yes | No | Yes | No | No |
| [89] | Yes | No | Cost | No | Yes | No | No | No |
| [23] | No | Yes | QoS attributes | No | No | Yes | No | No |
| **This thesis** | **Yes** | **Yes** | **Personalized security requirements**<br>**Cost** | **Yes** | **Yes** | **Yes** | **Yes** | **Yes** |

Table 3.1 – *SOTA Summary* [2]

## 3.3 Alloy to Coq Related Works

Although theorem provers have proved their effectiveness in proving detailed properties of multiple complex system specifications, they are still considered to be too expensive to use frequently during software development. Lightweight formal methods, on the other hand, are frequently used for checking software during the design and implementation stages. Alloy is a popular language and tool used for checking software systems against their requirements. On one hand, one of Alloy's strong suits is the counterexample returned in case of unfulfilled requirements. On the other hand, lack of counterexample, generally, does not give correctness proof. Thus, for critical systems, a second round of analysis might be crucial. Several previous works have addressed the verification of Alloy specifications.

In [10], Arkoudas et al. present a tool, Prioni, that integrates model checking and theorem proving for relational reasoning. Prioni takes as input formulas written in Alloy. It first uses the Alloy Analyzer to check their validity for a given scope. Once no counterexample is found, Prioni translates these Alloy formulas into Athena (i.e. a denotational proof language) proof obligations and uses the Athena tool for proof discovery and checking.

Another solution that works on infinite domains is presented in [86]. Kelloy [86] is a tool for verifying Alloy specifications with respect to potentially infinite domains. Kelloy is an engine for verifying Alloy specifications aiming to bridge the gap between lightweight formal methods and theorem provers. It provides a fully automatic translation of Alloy language to KFOL (the first-order logic of KeY, the deductive theorem prover used in Kelloy), an Alloy-specific extension to KeY's calculus and a reasoning strategy that improves KeY's capability in finding proofs and generates intermediate proof obligations that are easy to understand.

Unlike Prioni and the transformation tool we are presenting, Kelloy was developed in a way that only takes into consideration translation of Alloy rela-

[2]The answers provided in this table were inferred exclusively from the cited papers

tions up to ternary relations (i.e. arity 3). Such an approach requires to define the Alloy operations for all the different combinations of the arities in KFOL.

Mariano et al. [58] followed an approach closer to ours. They present an extension of PVS (Prototype Verification System), called Dynamite, that embeds Alloy calculus. It automatically adds and analyzes new hypotheses with the aid of the Alloy Analyzer. The generated PVS sequents get cluttered with some unnecessary formulas, thus, Alloy unsat-core extraction feature is used in order to refine proof sequents.

Unlike research previously mentioned, that aim to prove properties about Alloy models as a second step using theorem provers, we find a more recent one presented by in [45] with the aim of translating models from the Alloy specification language to the B specification language. Krings et al. present a tool for automatic translation of Alloy models to the B language that can be used in ProB. Language and Method B provide a means of producing mathematically proven software or systems, which helps ensure that the produced system meets the need, and ProB is a model checker for Method B. In [45], is presented an alternative semantics definition of Alloy, which enables proof and constraint solving tools of B to be applied to Alloy specifications. The aim of the translation is to make ProB's back-ends available to Alloy users. Which will enable them to experiment with technologies other than the ones employed by the Alloy Analyzer.

We present our approach for proving Alloy specifications as well as implement a tool for translating Alloy models to Coq syntax in Chapter 6. Unlike Prioni [10], which only analyzes finite domains due to the fact that Athena cannot handle infinite sets, our proposed solution handles infinite domains. Similarly to the work presented in [58], our work also relies on users conducting proof manually, with the difference that we provide a library with predefined lemmas to provide assistance in the proof process.

# A first Cloud brokerage solution 4

Contents



## 4.1 Introduction

Although Cloud computing has many benefits to offer, to mention a few: cost reduction, scalability and mobility, many potential users are still skeptical around the idea of integrating the Cloud, either because of the overwhelming number of Cloud service providers and offers proposed nowadays, or the

security issues related to the Cloud. We propose a Cloud brokerage solution to assist customers in the process of integrating the Cloud by finding an offer catered to their personalized needs in terms of functional requirements and security.

This chapter summarizes the work conducted as improvement of the Cloud brokerage solution presented in [37]. The thesis [36], defended by Guesmi Asma, has ended leaving a number of questions unanswered. Although her final results presented some obvious limits, we couldn't deny the potentials that this approach has, hence, the decision to take on the work and try to develop it further.

One of the limitations of the approach taken in [36], is that the size of the problems that can be dealt with effectively is limited. A second limitation is the impossibility of addressing the dynamicity of customers' needs, suppliers' offers or deployed architectures. The fact that the project is limited to IaaS-type architectures is not considered as a negative limitation. A decision was taken, to keep the focus on this type of architecture since it allows the resource manipulations we were looking for. Once proven effective for this type of architecture, our approach can easily be extended to PaaS-type architectures.

The aim of the work presented in this chapter was to find a way to introduce more dynamic features to the existing model. These features aim to consider the evolution of customers' needs, suppliers' offers as well as deployed architectures. The following content is based on the paper published in [74] and will be structured as follows: Section 4.2 will describe the models of the main actors of our solution, Section 4.3 contains the description of the general architecture, Section 4.4 presents the implementation of the solution and Section 4.5 will conclude this chapter.

## 4.2 The Initial Model

The main distinctive aspect of our initial proposed solution consists of considering the personalization of the security of the customers' models from its

first description. The security approach consists of formalizing security properties under the form of communication relations, from both the customer and provider sides. In this solution, the actors were modeled as follows:

### 4.2.1 IaaS Provider Model

IaaS is a cloud computing model where the cloud provider manages server hardware, virtualization layers, storage, networks. We simplify the presentation of an IaaS provider's architecture by presenting it as a set of *clusters*, each cluster containing a set of *physical nodes*. A Physical node hosts the customers' virtual machines and is defined by the set of VM templates, i.e. technical specifications that it can host, and its geographic location.

In this initial system, providers describe their offers in two steps. The first step will consist of describing the architecture, which consists in specifying the number of *clusters*, the number and characteristics of *physical nodes* contained in each cluster. The second step will be defining the relations between the clusters, intra-provider (i.e. relations between their own clusters) and inter-provider (i.e. relations between their clusters and other providers' clusters). It is important to note that the relations should be imperatively defined between different clusters and this fact is verified when verifying the consistency of the model. The aforementioned relations are defined as follow:

- *Link*: an information flow, permitting data interchange between two clusters;
- *Guardian*: two clusters having an intrusion protection system or filtering systems, such as network firewalls, relating them;
- *Conflict*: conflicted clusters, are clusters that cannot have any communication flow relating them.

An example of the provider model is presented in Figure 4.1a, it will be studied in Section 4.3.3.

### 4.2.2 Federation Model

A federation is modeled as a set of providers. As mentioned in the previous section these providers describe the relations between their own clusters and the other clusters owned by providers belonging to the same federation. Figure 4.1b presents an example of a federation, this example will be further detailed in Section 4.3.3.

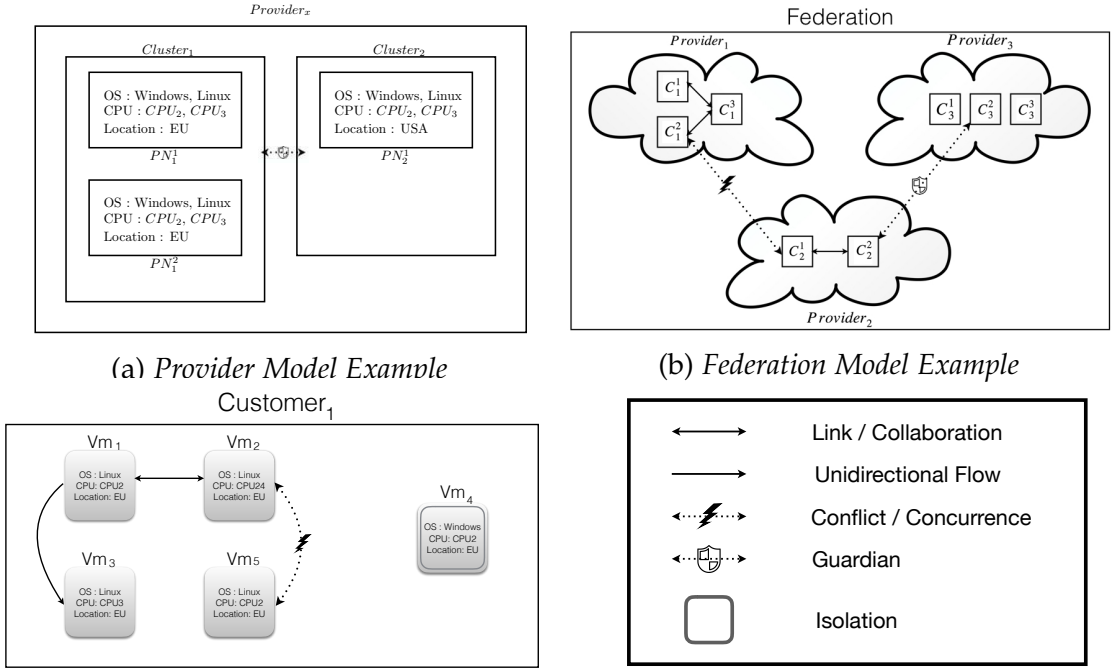
### 4.2.3 Customer Model

Customers describe the architecture to be deployed to the cloud. This architecture includes functional requirements, in the format of a number of virtual machines and their characteristics (e.g. processor, operating system, memory), as well as, non-functional requirements, which are the security requirements defined as relations between these virtual machines. Similar to relations between the provider's clusters, the relations here should be defined between different virtual machines of the same customer, and that also is verified when checking the consistency of the model.

The inter-VM relations that have been implemented in this solution are:

- *Isolation*: an isolated VM cannot communicate with the others;
- *Collaboration*: collaborating VMs are allowed to share data between them;
- *Concurrence*: two concurrent VMs are unauthorized to have a communication flow direct or indirect relating them;
- *Unidirectional Flow*: one-way flow between two VMs, meaning one has the right to send data to the other one but not read any.

An example of the customer's model is presented in Figure 4.1c, which will be explained and used in Section 4.3.4.




(a) *Provider Model Example*

(b) *Federation Model Example*

(c) *Customer Model Example*

(d) *Legend*

Figure 4.1 – *Model Example*

## 4.3 A First Brokerage Solution

### 4.3.1 Broker General Architecture

The general architecture presented in Figure 4.2 of our brokerage solution can be seen as three phases.

First, from a customer's standpoint, he / she describes the desired model through the user interface. In a prior time, the broker receives the descriptions of providers offers, as mentioned in Section 4.2.

Second, we use formal methods to verify the consistency of the customer's demand, the providers' offers and the coherence of the federation. In case of an inconsistency, the broker returns a counterexample, highlighting the reason behind the inconsistency.

Third, once the customer's model is consistent, the broker will try to find a possible placement, satisfying both the functional and non-functional requirements. If a placement is found the broker would forward the suggestion to the

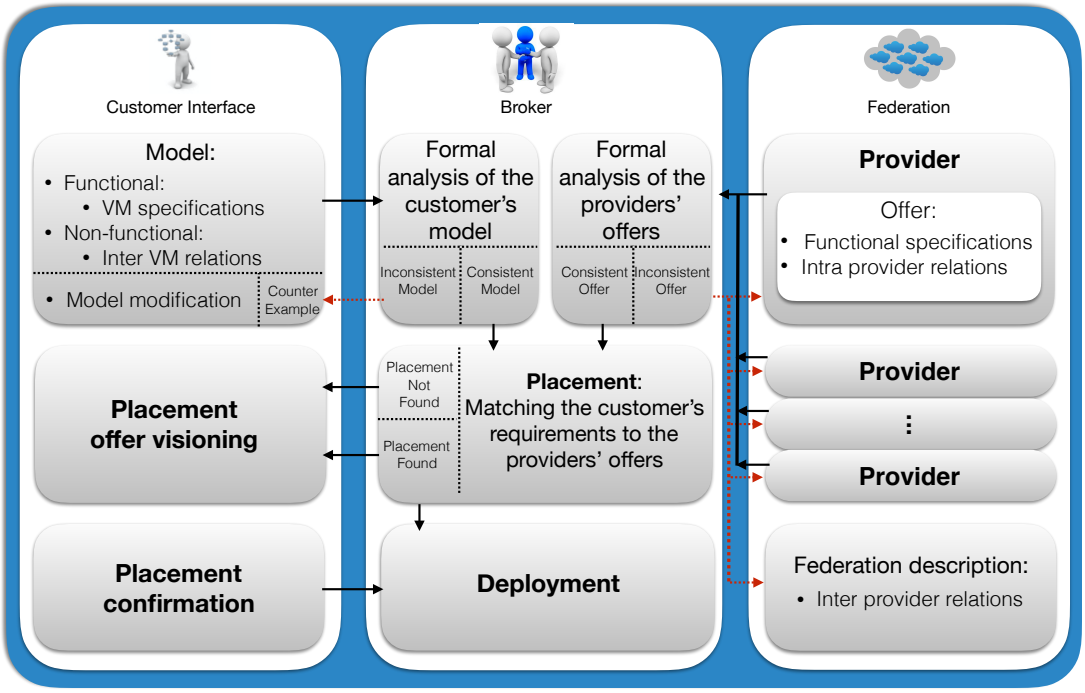



Figure 4.2 – *General Architecture*

customer, and waits for an approval in order to go through with the deployment.

### 4.3.2 Basic Sets

Here we group some of the basic sets and relations that will be used further in this section:

- *CPU*: contains all the values of CPU characteristics (in the current setting only speed) which means it is a unary relation that has the name "CPU" and that will be seen as a set of values,
- *OS*: contains all the possible operating systems,
- *Location*: contains all the possible locations,

| | |
|---|---|
| $CPU = \{ CPU_2, CPU_{24}, CPU_3 \}$<br>$Location = \{ \text{EU}, \text{USA} \}$<br>$Customer = \{ Customer_1 \}$ | $OS = \{ \text{Linux}, \text{Windows} \}$<br>$VM = \{ Vm_1, Vm_2, Vm_3, Vm_4, Vm_5 \}$ |
| $PhysicalNode = \{Pn_1^1, Pn_1^2, Pn_2^1 \}$<br>$Cluster = \{ Cluster_1, Cluster_2, C_1^1, C_1^2, C_1^3, C_2^1, C_1^2, C_3^1, C_3^2, C_3^3 \}$<br>$Provider = \{ Provider_x, Provider_1, Provider_2, Provider_3 \}$ | |

Figure 4.3 – *Example Sets*

- *VM*: contains all the virtual machines identifiers present in the system,
- *Customer*: contains a set of all customer identifiers,
- *PhysicalNode*: contains the identifiers of all the providers' physical nodes,
- *Cluster*: contains the identifiers of all clusters,
- *Provider*: contains a set of identifiers all the providers,

If we combine the examples given in Figures 4.1a, 4.1b and 4.1c, we would get an instance, of the previous unary relations, defined in Figure 4.3.

### 4.3.3 Provider Offer and Federation Verification

#### Provider Offer Description

A provider offers a set of clusters, each cluster is composed of physical nodes. For the sake of conciseness, we do not formally define the relations between *Provider* and *Cluster*, and *Cluster* and *PhysicalNode*.

A physical node is characterized by its location and the different virtual machine characteristics (i.e. virtual machine templates, here we consider the

type operating systems and virtual processor characteristics) that it is able to host:

- *PNOS*: a binary relation between the sets *PhysicalNode* and *OS*,
- *PNCPU*: a binary relation between the sets *PhysicalNode* and *CPU*,
- *PNLocation*: a binary relation between the sets *PhysicalNode* and *OS*.

The placement of virtual machines on physical nodes is modeled as *PNVM*, a binary relation between *PhysicalNode* and *VM*.

The provider describes the relations, *Link*, *Guardian* and *Conflict*, between his own clusters as well as between his clusters and the other providers' clusters within the same federation. *Link*, *Guardian* and *Conflict* are binary relations, subsets of $Cluster \times Cluster$. Our system makes sure that when a provider adds a new pair $(c, c')$ to one of the relations, at least one of $c$ and $c'$ is owned by this provider.

An example of a provider model is presented in Figure 4.1a. In this case only one relation was defined:

$$Guardian = \{\ (Cluster_1,\ Cluster_2)\ \}.$$

**Federation description**

The accumulated provider descriptions form a model of a federation. The aim of the latter is to have more interoperability between cloud providers, and prevent provider lock-in issue (i.e. a customer depending on only one provider). Figure 4.1b presents an example of a small federation containing three providers: $\{\ Provider_1,\ Provider_2,\ Provider_3\ \}$. In this example the relations grouping the different clusters are as follow:

- $Conflict = \{\ (C_1^2, C_2^1)\ \}$,
- $Link = \{\ (C_1^1, C_1^3),\ (C_1^2, C_1^3),\ (C_2^1, C_2^2)\ \}$,
- $Guardian = \{\ (C_2^2, C_3^2)\ \}$.

$$
\begin{aligned}
CustomerVM &= \{(Customer_1, Vm_1), (Customer_1, Vm_2), (Customer_1, Vm_3)\} \\
VMOS &= \{(Vm_1, Linux), (Vm_2, Windows), (Vm_3, Linux)\} \\
VMCPU &= \{(Vm_1, CPU_3), (Vm_2, CPU_2), (Vm_3, CPU_3)\} \\
PNCPU &= \{(Pn_1^1, CPU_2), (Pn_1^1, CPU_{24})\} \\
PNOS &= \{(Pn_1^1, Linux), (Pn_1^1, Windows)\} \\
PNLocation &= \{(Pn_1^1, USA)\} \\
VMLocation &= \{(Vm_1, EU), (Vm_2, USA), (Vm_3, EU)\} \\
PNVM &= \{(Pn_1^1, Vm_2)\}
\end{aligned}
$$

Figure 4.4 – *Example: Relations Values*

**Offer and Federation Consistency Verification**

In order for an offer or a federation to be consistent, one major rule should be respected.

**Rule 1** Two conflicted clusters can not have a link of any sort relating them (i.e. if two clusters are identified as conflicted the virtual machines hosted on them cannot communicate, no communication tunnel, direct or transitive, can connect these two clusters):

$$
\begin{gathered}
\forall c_i, c_j \in Cluster, (c_i, c_j) \in Conflict \Longrightarrow \\
((c_i, c_j) \notin (Link \cup Guardian)^+ \wedge (c_j, c_i) \notin (Link \cup Guardian)^+)
\end{gathered}
$$

A counterexample is sent to the provider in case of an inconsistency. Once the offer is consistent, the system adds it to the federation.

### 4.3.4 Customer Model Description and Verification

#### Customer Model Description

In our system, a customer's model represents the description of the application's architecture to be deployed in the Cloud. This model is identified by a unique instance of *Customer* and a relation *CustomerVM* between *Customer* and *VM* representing the set of virtual machines she owns.

The customer's functional requirements for virtual machines are expressed as their processor speed, operating system and location. This is formalized by three relations: *VMCPU* a binary relation between *VM* and *CPU*, *VMOS* a binary relation between *VM* and *OS*, and *VMLocation* a binary relation between *VM* and *Location*.

To define her/ his applications, a customer then specifies the non-functional requirements: the relations between the different virtual machine, *Isolation*, *Collaboration*, *Concurrence*, and *Flow*. These relations have been informally described in Section 4.2. *Isolation* is a subset (unary relation) of *VM*, while *Collaboration*, *Concurrence*, and *Flow* are binary relations between *VM* and *VM*. The system ensures that a customer only defines or modifies relations regarding her own virtual machines.

An example of a customer model is presented in Figure 4.1c. The customer's non-functional requirements are:

$$\begin{aligned} \mathit{Isolation} &= \{\ Vm_4\ \} \\ \mathit{Flow} &= \{\ (Vm_1, Vm_3)\ \} \\ \mathit{Collaboration} &= \{\ (Vm_1, Vm_2)\ \} \\ \mathit{Concurrence} &= \{\ (Vm_2, Vm_5)\ \}. \end{aligned}$$

#### Requirements Consistency Verification

Verifying the consistency of the customer's requirements is verifying that it respects a set of rules related to the security relations.

**Rule 2** An *isolated* VM cannot have any information flows going into or out of it:

$$\forall v_i \in Customer.CustomerVM, v_i \in Isolation \implies$$
$$\nexists v_j \in Customer.CustomerVM$$
$$(v_i, v_j) \in Flow \vee (v_j, v_i) \in Flow$$

**Rule 3** Two *collaborating* VMs have a bidirectional information flow, they can read and write data from one to another:

$$\forall v_i, v_j \in Customer.CustomerVM, (v_i, v_j) \in Collaboration \implies$$
$$((v_i, v_j) \in Flow \wedge (v_j, v_i) \in Flow)$$

**Rule 4** Two *concurrent* VMs cannot have any direct or transitive information flow between them:

$$\forall v_i, v_j \in Customer.CustomerVM, (v_i, v_j) \in Concurrence \Rightarrow$$
$$(v_i, v_j) \notin Flow^+ \wedge (v_j, v_i) \notin Flow^+$$

Once the brokerage system receives the customer's model, it verifies that the requirements satisfy the rules defined. In case of an inconsistency of the demand, a counter-example is returned to the customer to rectify the model. Otherwise, the broker proceeds to the next step which consists of finding a placement. The placement phase is detailed in Section 4.3.5.

### 4.3.5 Placement Strategy

The brokerage system searches for a placement that satisfies the customer's functional and non-functional requirements, taking into consideration the relations defined between the clusters. It tries to find a placement respecting the following rules, we define two families of rules:

#### Rules related to the functional requirements

Here we present the rules responsible for checking the compatibility from a functional standpoint (i.e. the virtual machines' functional requirements as

specified by the customer: the operating system, the processor speed and the location).

**Rule 5** A VM should be placed in a physical node offering the adequate VM characteristics and that is located in the requested location:

$$\forall v \in VM, \exists p \in PhysicalNode, v \in Customer.CustomerVM \wedge v \in p.PNVM$$
$$\Longleftrightarrow$$
$$v.VMOS \in p.PNOS \wedge v.VMCPU \in p.PNCPU \wedge v.VMLocation = p.PNLocation$$

For a better understanding of the formula, let us take a concrete example. Taking the values given in Figure 4.4, for the relations mentioned in the formula above, along with those given in Figure 4.3.

Considering we have only one customer $Customer_1$ and one physical node $Pn_1^1$, the navigations will contain the following values:

$$Customer_1.CustomerVM = \{\ Vm_1,\ Vm_2,\ Vm_3\ \}$$
$$Pn_1^1.PNVM = \{\ Vm_2\ \}.$$

$Vm_2$ is both in $Customer.CustomerVM$ and $PhysicalNode.PNVM$, which means that the virtual machine $Vm_2$ can be placed in the physical node $Pn_1^1$, if and only if additionally $Vm_2$'s characteristics are offered by this physical node. As we have:

$$\begin{aligned}
Vm_2.VMOS &= \{Windows\}\\
Pn_1^1.VMOS &= \{Linux, Windows\}\\
Vm_2.VMCPU &= \{CPU_2\}\\
Pn_1^1.VMCPU &= \{CPU_2, CPU_{24}\}\\
Vm_2.VMLocation &= \{USA\}\\
Pn_1^1.PNLocation &= \{USA\}
\end{aligned}$$

all $Vm_2$ characteristics are handled by $Pn_1^1$.

**Rule 6** A *VM* should be placed in one and only one *PhysicalNode*, all the customer's *VMs* should be placed:

$$\forall v \in VM, \exists! p \in PhysicalNode, v \in Customer.CustomerVM \wedge v \in p.PNVM$$

**Rules related to the non-functional requirements**

Here we group the rules responsible for checking the compatibility from a non-functional standpoint. We describe rules related to the fore-defined relations between the virtual machines (i.e. collaboration, concurrence, flow and isolation).

**Rule 7** A customer's VMs cannot be placed on *conflicting* clusters:

$$\begin{gathered}\forall v_i, v_j \in VM, \exists c \in Customer, \exists c_i, c_j \in Cluster, \exists p_i, p_j \in PhysicalNode, \\ v_i \in c.CustomerVM \wedge v_j \in c.CustomerVM \wedge v_i \in c_i.p_i.PNVM \wedge v_j \in c_j.p_j.PNVM \Longrightarrow \\ (c_i, c_j) \notin Conflict\end{gathered}$$

**Rule 8** An *isolated* VM should be placed in a separate physical node:

$$\begin{gathered}\forall v_i, v_j \in VM, \exists c \in Customer, \exists p \in PhysicalNode, \\ v_i \in c.CustomerVM \wedge v_j \in c.CustomerVM \wedge v_i \in p.PNVM \wedge v_i \in Isolation \\ \Longleftrightarrow \\ v_j \notin p.PNVM\end{gathered}$$

Currently, a customer's isolated VM will only be separated from his own VMs, but could be placed with other customers' VMs in the same physical node. A possible improvement of the current program could be to give the customers the possibility to specify, for each of their VMs, whether they want to place a VM in the same physical node as other customers or not.

**Rule 9** Two *collaborating* VMs should be on the same cluster:

$$\forall v_i, v_j \in VM, \exists c_i, c_j \in Cluster, \exists p_i, p_j \in PhysicalNode,$$
$$v_i \in c_i.p_i.PNVM \wedge v_j \in c_j.p_j.PNVM \wedge (v_i, v_j) \in Collaboration \Longrightarrow$$
$$c_i = c_j$$

**Rule 10** Two *concurrent* VMs should be placed on non-communicating clusters, or clusters linked by a *guardian*:

$$\forall v_i, v_j \in VM, \exists c_i, c_j \in Cluster, \exists p_i, p_j \in PhysicalNode,$$
$$v_i \in c_i.p_i.PNVM \wedge v_j \in c_j.p_j.PNVM \wedge (v_i, v_j) \in Concurrence \Longrightarrow$$
$$((c_i, c_j) \notin Conflict \wedge (c_i, c_j) \notin Link \wedge (c_i, c_j) \notin Guardian) \vee (c_i, c_j) \in Guardian$$

**Rule 11** Two VMs related by a *unidirectional flow* should be placed on clusters linked by a *guardian*:

$$\forall v_i, v_j \in VM, \exists c_i, c_j \in Cluster, \exists p_i, p_j \in PhysicalNode,$$
$$v_i \in c_i.p_i.PNVM \wedge v_j \in c_j.p_j.PNVM \wedge (v_i, v_j) \in Flow \Longrightarrow$$
$$(c_i, c_j) \in Guardian$$

## 4.4 Implementation & Evaluation

As part of this contribution, we have managed to develop a Cloud broker performing all the described functionalities in one JAVA application, using the KodKod API for formal analysis. The workflow of the application follows the general architecture of the broker described in Section 4.3.1.

KodKod [24] is a finite model finder, that we used for both the consistency verification and the search for placement strategies. These functionalities were transformed into KodKod *problems*, where the *relations* consists of the system's actors and relations previously mentioned and the *universe* groups all the submitted values. The verification and placement rules previously described were all translated into KodKod *formulas*.

**Relational variables** As previously mentioned everything is a relation in KodKod, and so all the actors of our system are defined as relations. In this section is presented an example of the steps followed in order to verify the consistency of the customer's model using the KodKod API. We start by defining the relational variables, using the different methods of the class *Relation* in order to precise the *arity* of the relation and its *name*:

```
1 Relation Customer = Relation.unary("Customer");
2 Relation c_vms = Relation.binary("c_vms");
```

**Universe** In this case the *atoms* would be the different identifiers of the customer, the set of his virtual machines and a VM relation, relating the customer to the relations he have described. We use the constructor of the class *Universe* giving it as argument a list of the values in a string format (i.e. Set<String>):

```
1 Universe U = new Universe(atoms);
```

**Formula** The consistency rules previously described are written in KodKod syntax using the class *Formula*. An example of the **contraposite** of the isolation rule (i.e. Rule 2 that indicate if a VM defined as *isolated* it should be placed in a separate physical node):

$$\forall v_i, v_j \in VM, \exists c \in Customer, \exists p \in PhysicalNode,$$
$$v_i \in c.CustomerVM \wedge v_j \in c.CustomerVM \wedge v_i \in p.PNVM \wedge v_i \in Isolation$$
$$\Longleftrightarrow$$
$$v_j \notin p.PNVM$$

in KodKod syntax is given in Figure 4.6.

Once relational variables are defined, the next step is to bound them with tuples created from the universe, using a tuple factory. An example of bounding the VM relation is shown in Figure 4.5. In this case we use the method *boundExactly* of the class *bound* which means we are specifying its *lower bounds* (i.e. the relation should contain all of the values passed as argument). We could

```
1 TupleFactory f = U.factory(); Bounds b = new Bounds(U);
2 TupleSet vms = f.noneOf(1);
3 for(String id: VMIds)
4  vms.add(f.tuple("Vm"+C+"-"+id));
5 b.boundExactly(vm, vms);
```

Figure 4.5 – *KokKod Example: Relation Bounding*

assign both the *upper* and *lower bounds* using the signature with three arguments of the method "bound" as follows: bound(Rel, low, up) or just the *upper bounds* using the signature with two arguments: bound(Rel, up).

In the case of the model verification we tightly *bound* the *relational variables* (i.e. we specify both the lower and upper bounds) to create a *model* that will then be fed to the KodKod engine to ensure that the created model satisfies the consistency *formulas* wanted. In case of an inconsistency, unlike the Alloy Analyzer that returns a counterexample, KodKod only returns the *unsatisfied formula*. In order to return a clearer *counter example* to our customer (resp. provider), we try to find a *solution model* for the *contraposite* of the returned formula, this way we know the source of the inconsistency and forward it to the customer (resp. provider).

Once the verifications are done and the customer's demand is consistent the broker proceeds to find a placement strategy. The idea is to loosely *bound* the *relational variables* by only precising the upper bounds and then ask KodKod to find a *model* that satisfies *formulas* representing the placement rules.

KodKod's *partial solution* is an interesting feature that has permitted us to add a dynamic aspect to the project. The collection of all the variable *lower bounds* is a *partial solution* of the problem. Our broker allows customers to modify their architecture after deployment. They can either add virtual machines or remove them, modify the virtual machines relations as they desire, and the broker will modify the placement strategy of the concerned virtual machines respecting the old placement of the other ones. The same thing goes for the provider's offer. In order to do so, we use the existing placement strategy as lower bounds of *modified problem*. Finally, we refeed the problem to KodKod to

```
1  public Formula Isolation_rules(){
2    final Variable vm_i = Variable.unary("vmi");
3    final Variable vm_j = Variable.unary("vmj");
4    final Formula F1 = Customer.c_relation
5      .product(vm_i).in(isolation).not()
6      .and(Customer.c_relation.product(vm_j)
7      .in(isolation).not());
8    final Formula iso = Customer.c_relation
9      .product(vm_i).product(vm_j)
10     .in(vm_flow).implies(F1).forAll(vm_i
11     .oneOf(VM.vm).and(vm_j.oneOf(VM.vm)));
12   return iso;
13 }
```

Figure 4.6 – *KokKod Example: Formula*

solve the conjunction of all the formulas and find the adequate changes to be executed on the deployed architecture.

## Evaluation

In order to have a concrete idea of the time spent verifying the consistency of a customer's architecture and find an appropriate placement in a federation, we have used the following test as a proof of concept. We started by using a federation grouping four providers, respectively, composed of {11, 12, 32, 308} physical node. We first tried to find placements for three different customers in this federation. The customers are $C_1$ with 12 virtual machines and 10 security properties, $C_2$ with 30 virtual machines and 15 security properties and $C_3$ with 100 virtual machines and 40 security properties. The consistency of $C_1$'s architecture was verified in 128 ms and the placement problem was solved in 20 ms. The consistency of $C_2$'s architecture was verified in 188 ms and the placement problem was solved in 175 ms. The consistency of $C_3$'s architecture was verified in 682 ms and the placement problem was solved in 22.5 s. Then we added two providers composed respectively of 535 and 957 physical nodes. We found the placement for a customer with 100 VMs in 2.5 min and another with 200 VMs in 5.6 min. This shows the limitation of using the KodKod engine for the

placement finding step. The translation of KodKod problem, relational logic, to CNF (Conjunctive Normal Form), boolean logic, in the background takes the most amount of time.

**Summary**

First test scenario:

- A federation of four providers:
  - $P_1$: 11 physical nodes
  - $P_2$: 12 physical nodes
  - $P_3$: 32 physical nodes
  - $P_4$: 308 physical nodes
- Customers:
  - $C_1$: 12 VMs + 10 security properties
  - $C_2$: 30 VMs + 15 security properties
  - $C_3$: 100 VMs + 40 security properties
- Timings:

| Customer | Consistency | Placement |
|---|---|---|
| $C_1$ | 128 ms | 20 ms |
| $C_2$ | 188 ms | 175 ms |
| $C_3$ | 682 ms | 22.5 s |

Second test scenario:

- A federation of six providers:
  - $P_1$: 11 physical nodes
  - $P_2$: 12 physical nodes
  - $P_3$: 32 physical nodes

  - $P_4$: 308 physical nodes
  - $P_5$: 535 physical nodes
  - $P_6$: 957 physical nodes

- Customers:
  - $C_1$: 100 VMs
  - $C_2$: 200 VMs

| Customer | Total time |
|---|---|
| $C_1$ | 2.5 mins |
| $C_2$ | 5.6 ms |

## 4.5 Conclusion

Our system takes into account the functional and non-functional requirements of a Cloud customer. It verifies the consistency of both the customers' demand and providers' offers, and finds, if exists, an adequate placement of the customer's model in a Cloud federation. We have managed to introduce a dynamic aspect to the solution by using a finite model finder called KodKod.

However, even though the solution we proposed in [74], is more dynamic than its predecessor presented in [37], in the way that we can create multiple customers and providers with the possibility of modifying their architectures when wanted, we had more paths to explore in order to improve the solution. A recurring feedback that our solution received was related to the simplicity of the federation and providers' models descriptions. We have worked on an architecture closer to real-world scenarios with a more realistic hypothesis. The next chapter will cover the updated solution and model description. Another limitation related to our solution was tightly linked with the use of the KodKod model finder and Alloy language. In fact, the limitation of the latter is that when no counterexample found in the given scope the global formula is considered *true*, but never proven *true*. In Chapter 6, will be presented a potential

solution to this limitation, not just for our project but for any project using the Alloy modeling language.

# A cost efficient Cloud brokerage solution 5

## Contents

## 5.1 Introduction

Although the proposed solution, presented in the previous chapter, had shown great potential in assisting customers in integrating the Cloud while taking into consideration their personalized functional and security requirements, it has limitations. The main limiting factors were the simplistic models, more precisely those of the providers and the federation. The way we have presented our solution relies on the existence of an already standardized way to describe Cloud providers' infrastructures as well as the federation grouping them. Unfortunately, these expectations are unrealistic due to the interoperability issues between providers and the inexistent standardization of the providers' infrastructure descriptions. Our model heavily relied on the fact that the providers' offers will be given in a very specific manner, in the format of different clusters and specifications of each of the provided physical nodes as well as predefined network relation between their clusters. This is a counter-intuitive vision of the Cloud concept. For our solution, to be more relative to nowadays issues, we decided to modify the providers and federation descriptions. Another limitation we had when finding an appropriate placement of the customers' architecture, is that we returned the first placement strategy found without taking into consideration any specific sorting criteria. Cost being one of the most important criteria for potential customers, we propose in our updated solution a placement strategy that optimizes the cost.

So far in our work, we have focused on the theoretical aspect of the brokerage solution. Taking into consideration reviewers' feedback and our own willingness to explore other paths, we have decided to upgrade our current broker model. Section 5.2 presents the modifications introduced to the overall architecture of our brokering solution. Section 5.3 describes the new broker model and defines its different components. The implementation of this model into the overall solution is presented in Section 5.4. A case study example is

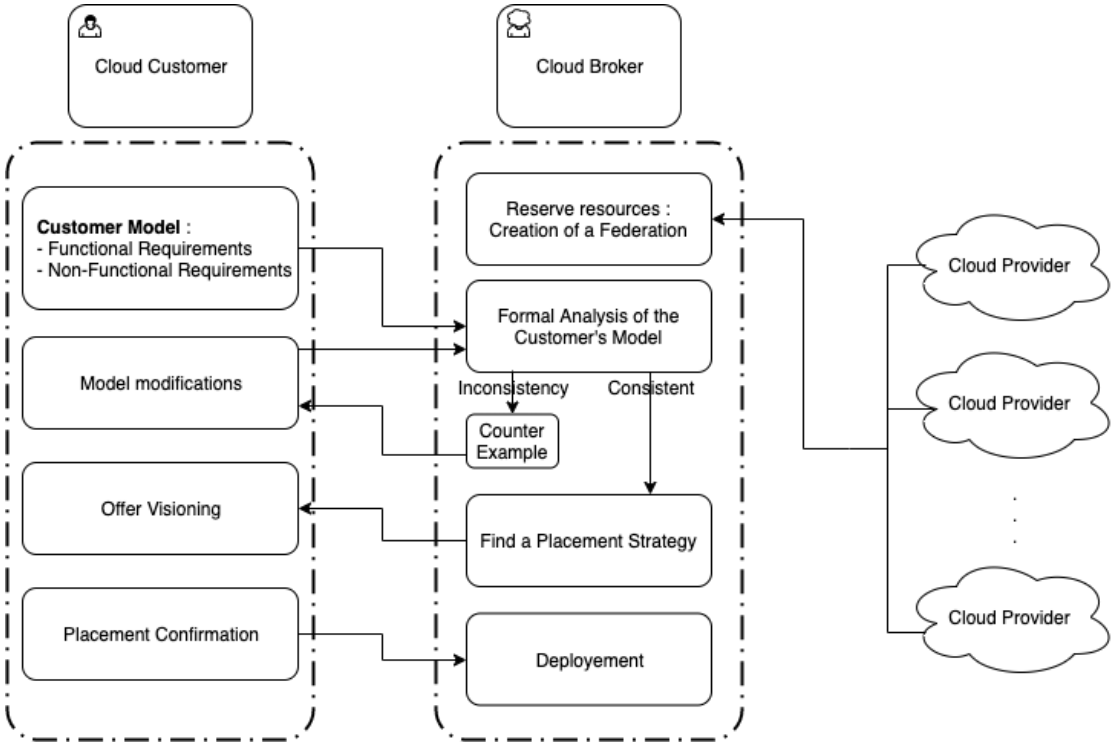



Figure 5.1 – *Broker Solution Updated General Architecture*

detailed in Section 5.5. Tests and evaluation of our solution are summarized in Section 5.6.

## 5.2 Updated Architecture

The general architecture of our solution can be divided into three major parts depending on the interactions between the different actors. The first is the relation between the customer and the broker. The second is the one between the broker and the different providers. Then there is the full procedure of finding a placement strategy. A simplified overview is presented in Figure 5.1 and detailed in the following.

**Customer-Broker relation** The customer's demand is now presented as an application, instead of an infrastructure architecture. The request is in the format of a black box component-based architecture. First, the customer should specify the functional requirements consisting of the size of the set of components and the configuration for each component (number of virtual processors, memory size, operating system and location). Second, he gives the non-functional requirements under the format of inter-component relations (alone, flow and concurrence). Once this request is communicated, the broker uses formal methods to verify the consistency of the customer's model as described in Section 5.3.1.

**Broker-Provider relation** The broker pre-reserves a certain amount of resources, with different characteristics, from different providers. We can base this decision on prior studies about the minimum needed resources to assure a satisfactory service delivery by the broker. After that, the provisioning over time could be done taking into consideration the number of customers and resources used, or not, over a period of time. An adequate period could be chosen and an assessment of resources would be done prior to the beginning of every period.

**Placement strategy** The broker looks for a placement that matches the customer's functional requirements while minimizing the set-up cost. Once the customer approves of the returned placement strategy, the broker then takes care of the deployment and adequate network configurations to satisfy the customer's security requirements.

## 5.3 The new broker model

In this section will be presented the modifications brought to the overall brokerage solution model presented in the previous chapter. Leaning towards a more realistic scenario, the major changes brought to the model are related to

the federation definition and the representation of the providers. The two major tasks to be executed by our broker consist of the verification of the customer's demand and the search for an adequate cost-efficient placement strategy that respects the customer's functional and non-functional requirements. Prior to that, the broker has to provision the necessary resources, that consist of different types of instances reserved from different providers in order to form his own federation. In our solution, we consider that the reserved resources, at any time, are enough to answer customers' demands.

### 5.3.1 Customer Model Description and Verification

The major changes that were brought to the customer's model description are that we consider the customer's demand describes a component-based application in the format of a set of components with a set of functional requirements and non-functional requirements. The functional requirements we consider are:

- vCPU: The number of virtual processors needed by this component
- Memory: The memory size in GiB
- OS: The operating system
- Location: where the component should be located

The non-functional requirements are still considered as relations between the different components that the customer defines. We have modified the definitions of some of the previous relations. We decided to loosen the definition of the "Isolation" unary relation because we can not guarantee that a component is fully isolated in a Cloud set up. Cloud providers can not guarantee that an instance is fully isolated, nor that we can prove the absence of hidden network tunnels. We now have an "Alone" unary relation which indicates that the concerned component is not to be hosted with the other components of the same application. We suggest that this component is to be hosted by a provider different from all the other components of the same application, due to the lack of

guarantees that this level of isolation is possible at the provider level. We have also merged the definitions of the "Collaboration" and "Unidirectional Flow" relations to have one relation called "Flow" that indicates a flow of information between two components that is by default unidirectional and in the case of needing to express a bidirectional flow the customer can simply specify two flows between the chosen components. We kept the "concurrence" relation, which limits the communication between components, as is. To verify the consistency of the customer's demand, we continue using the KodKod API. The customer's demand model is identified by a unique instance of *Customer* and now a relation *CustomerComp* between *Customer* and *Comp* representing the set of the application's components. The relations between these components are *Alone* a unary relation subset of *Comp*, *Concurrence* and *Flow* are binary relations between *Comp* and *Comp*. The demand consistency verification rules we now have are as follows:

- Concurrence rule : No flow of information, neither direct nor transitive, between two concurrent components.

$$\begin{gathered}\forall cmp_i, cmp_j \in Customer.CustomerComp, \\ (cmp_i, cmp_j) \in Concurrence \wedge \\ (cmp_i, cmp_j) \notin Flow \wedge \\ (cmp_j, cmp_i) \notin Flow \\ \Rightarrow (cmp_i, cmp_j) \notin Flow^+ \wedge \\ (cmp_j, cmp_i) \notin Flow^+\end{gathered}$$

- Flow rule : If two components have a unidirectional flow in one direction and are in concurrence, then there should be no transitive flow of information in the direction where there is no direct flow.

$$\begin{gathered}\forall cmp_i, cmp_j \in Customer.CustomerComp, \\ (cmp_i, cmp_j) \in Concurrence \wedge \\ (cmp_i, cmp_j) \in Flow \\ \Rightarrow (cmp_j, cmp_i) \notin Flow^+\end{gathered}$$

### 5.3.2 Placement Strategy

The main feature in our new brokerage model and the biggest change in comparison with the first model consists of the interaction between the broker and the different Cloud services providers and finding the placement strategy step. Using KodKod and the Alloy language have limitations, mainly from a size point of view, in finding a placement strategy. It was the most appropriate solution in respect to how we decided to describe the federation and providers models. However, with the modifications we brought to their description, using KodKod is no longer justified, it will be more of a hurdle than a tool. From another hand, one of the main advantages of coupling Cloud brokering with multi-Cloud is to maximize performance and minimize the overall cost of migrating to the Cloud. Thus, we wanted our brokerage solution to be able to offer a cost-efficient placement strategy all while respecting the functional and non-functional requirements of the customer. In order to do so, we have decided to use linear modeling. In retrospect, although we have chosen the cost as our optimization decision factor to sort through the different possible placement strategies, the way we have written the linear model allows the possibility to change the decision factor to cater to different customers' priorities. We present the different steps of the cost optimization model in Section 5.4.1. By solving this problem, we get a placement strategy for the customer's components that respects both the functional and non-functional needs while minimizing the set-up cost. This offer will have to be approved by the customer before going to the next step. The hypothesis of our solution is that at the time of searching for a placement strategy, the broker has enough resources reserved and ready. We assume that all the pre-reserved instances are pre-configured to have no communications with other instances. Thus, once the placement strategy is approved, adequate network configurations need to be conducted on the affected instances in order to allow communication flows between the hosted components.

**Virtual Infrastructure configuration** In the updated solution, instead of assuming that we have an all ready Cloud federation with certain standards put in place for inter-Cloud providers communications, we propose that the Cloud broker pre-provisions resources from different Cloud providers. This set of resources will be considered as the broker's federation. With such a solution, the communication issue between the different resources from different providers still remains. Here we introduce the notion of virtual network configurations. These are the network configurations needed to be put in place to respect the communication constraints described by the customer in the security requirements. When it comes to inter-provider communication one of the ways to handle this problematic is through using some of the services proposed by the different providers. The virtual private network (VPN) services offered by the different Cloud providers can be used to create encrypted channels between virtual private Clouds (VPCs) hosted by different providers to transfer data using private IP addresses. Many Cloud service providers have their own managed VPN products, which allow IPsec VPN tunnels to be created between the different environments. For example, Google offers Cloud VPN [35], AWS offers AWS Site-to-Site VPN [11] and Azure the Azure VPN gateway [12]. The setup cost of these services differs from a provider to another. Thus, we have included these intermediary costs in our optimization problem as will be presented in Section 5.4.1.

## 5.4 Implementation

### 5.4.1 Linear Model

We decided to describe the cost optimization problem as a linear model. Here we present how the model for only one customer is written using the GNU MathProg modeling language from the GNU Linear Programming Kit [51].

We start describing our model by first listing the different elements of the customer's request and the providers' offers. These are given as parameters.

In fact, the consistency of the customer's demand is verified using KodKod as presented in Section 4.3.4. As for the data concerning the Cloud providers, it is the Broker's federation created from the pre-reserved resources. The declaration of the parameters:

```
/* Components for the custumer */

param nbcomp, integer, >=1;
set Comp, default {1 .. nbcomp};

/* Relations between customers */
/* We assume the consistency has been verified */
set Alone, within Comp;
set Flow, within Comp cross Comp;
set Concur, within Comp cross Comp;

/* Providers */
param nbp, integer, >= 2;
set P, default {1 .. nbp};

/* Instances */
# number of type of instances per provider
param nbti{i in P};
# Identifiers of types of instances
set idti{i in P, j in 1..nbti[i]};
#number of instances per type of instances assumed here constant
param nbinstances, integer;
```

In order to know which instance types are compatible with the customer's components, we introduce a parameter Alpha. The characteristics of a component being: the number of virtual processors, the memory size, the operating system and location. An instance is eligible to host a component if it has the same operating system and location, at least and at most two times the same number of virtual processors and size. This decision is based on the fact that

if the broker doesn't have the exact instance to answer the customer's request when it comes to virtual processors number and memory size, it will still be beneficial for both the broker and customer to use an already reserved instance with up to two times the request. In fact, once we go over this threshold, getting an on-demand instance will be more beneficial. Thus, the compatibility parameter is defined as follows:

$$Alpha[i,j,k] = \begin{cases} 1, & \text{if Component } i \text{ can be mapped on the } k^{th} \text{ instance type} \\ & \text{of Provider } j \\ 0, & \text{otherwise} \end{cases}$$

We add it into our linear model as follows:

*/* Adequation of type of instance to a given component */*
**param** Alpha{i **in** Comp, j **in** P, k **in** 1..nbti[j]};

The goal of this model is to optimize the overall set up cost of the customer's architecture. In our solution we only consider the initial set up cost and not the usage cost, since the latter is relative to the customer's activity. The different costs that we consider are the following:

- Set Up costs of the different instances

  */* Instances Set Up Cost */*
  **param** InstancesPrices {i **in** P, j **in** 1..nbti[i]};

- Costs of using a virtual private Cloud

  */* Virtual Private Cloud */*
  *# Cost of using a Virtual Private Cloud from Provider i*
  **param** VPC {i **in** P};

- Costs of using a virtual private network tunnel

  */* VPN tunnel service */*
  **param** VPT {i **in** P};

- Costs of the network configurations. The cost for internal link configuration, when two instances are hosted by the same provider, and for external links, when they are hosted by different providers. This cost being relative to the network configuration needed to be done to allow the communication, we consider it to be a constant value to be precised by the broker.

  /* *Internal Link Configuration Cost* */
  **param** CIL;
  /* *External Link Configuration Cost* */
  **param** CEL;

Then we define the different system variables. Their value is going to be affected according to the model constraints that will be presented later on in this section.

- First we define the variable $X$ that indicates which component is assigned to which instance.

$$X[i,j,k,l] = \begin{cases} 1, & \text{if Component } i \text{ is mapped on the } l^{th} \text{ instance} \\ & \text{on the } k^{th} \text{ instance type of Provider } j \\ 0, & \text{otherwise} \end{cases}$$

  **var** X{i **in** Comp, j **in** P, k **in** 1..nbti[j], l **in** 1 .. nbinstances}, **binary**;

- Variable $Y$ indicates if the customer is using a virtual private Cloud from a certain provide. This is similar to saying that one of the customer's components is mapped on an instance hosted by this provider.

$$Y[i] = \begin{cases} 1, & \text{if the customer is using a VPC hosted by provider } i \\ 0, & \text{otherwise} \end{cases}$$

  **var** Y{i **in** P}, **binary**;

- Variable $Z$ indicates if the customer needs to use a virtual private network between two different providers. This is needed when a customer's components, that are hosted on these different providers, need to communicate.

$$Z[i,j] = \begin{cases} 1, & \text{if there is a need of a VPN Tunnel between providers } i \text{ and } j \\ 0, & \text{otherwise} \end{cases}$$

**var** Z{i **in** P, j **in** P: i!=j}, **binary**;

- In order to be able to compute the cost of internal and external link configurations between instances hosted by different providers we need to know how many links exist, intra-provider represented by $NIL$ and inter provider represented by $NEL$. To simplify the computation, we use the $ZIL$ and $ZEL$ binary variables, where $NIL$ is the sum of $ZIL$ and $NEL$ that of $ZEL$.

$$ZIL[i1,i2,j] = \begin{cases} 1, & \text{if components } i1 \text{ and } i2, \text{ related by a flow,} \\ & \text{are hosted by the same provider } j \\ 0, & \text{otherwise} \end{cases}$$

$$ZEL[i1,i2,j1,j2] = \begin{cases} 1, & \text{if components } i1 \text{ and } i2, \text{ related by a flow,} \\ & \text{are hosted by providers } j1 \text{ and } j2, \text{ respectively} \\ 0, & \text{otherwise} \end{cases}$$

They are added to the model as follows:

*# Internal*
**var** ZIL{(i1, i2) **in** Flow, j **in** P}, **binary**;
**var** NIL;
*# External*
**var** ZEL {(i1, i2) **in** Flow, j1 **in** P, j2 **in** P:j1!=j2}, **binary**;

**var** NEL ;

- Then we find the cost variables, that we define as follows in order to make it easier to check the prices of each phase independently.

```
# Use of instances
var CostInstances ;
# use of VPC
var CostVPC;
# use of VPN Tunnel
var CostVPT;
# use of Internal Links
var CostIL;
# use of External Links
var CostEL;
# Global Cost
var GlobalCost;
```

Once all parameters and variables are defined, we need to write the constraints of our model. Most of these constraints are equivalent to the placement rules already described in the previous chapter. We first start with the constraints related to the functional requirements.

- The chosen instance provides enough resources for mapping the component:

```
s.t. Suitable{i in Comp, j in P, k in 1..nbti[j], l in 1 .. nbinstances}:
 X[i, j, k, l] <= Alpha[i,j,k];
```

- Each component has to be placed only once:

```
s.t. Placed{i in Comp}:
sum{j in P, k in 1..nbti[j], l in 1 .. nbinstances} X [i,j,k,l] = 1;
```

- Each instance can accept at most one component:

```
s.t. OneCompPerInstance{j in P, k in 1..nbti[j], l in 1 .. nbinstances}:
 sum {i in Comp} X [i,j,k,l] <= 1;
```

Then the constraints related to the non-functional requirements.

- Each Component Alone must be isolated from the others and no other component should be mapped in the same Provider:

```
s.t. AloneComponent{i1 in Alone, i2 in Comp, j in P: i1 !=i2}:
 sum{k in 1..nbti[j],l in 1 .. nbinstances}(X[i1,j,k,l] + X[i2,j,k,l]) <= 1;
```

- If two components are in concurrence, they shouldn't be hosted by the same provider:

```
s.t. ConcurComponent{(i1,i2) in Concur, j in P}:
 sum{k in 1..nbti[j],l in 1 .. nbinstances}(X[i1,j,k,l] + X[i2,j,k,l]) <= 1;
```

Then additional constraints on the variables that will be used in calculating the costs.

- We consider that once one of the customer's components is placed on an instance hosted by a provider $x$ we are automatically using a virtual private Cloud service from this provider. Thus, the constraint on the variable $Y$ is defined as follows:

```
s.t. UseVPC{i in Comp, j in P, k in 1..nbti[j], l in 1 .. nbinstances}:
Y[j] >= X[i,j,k,l];
```

- We also consider that a tunnel is needed between two providers $j_1$ and $j_2$ (i.e. $Z[j_1, j_2] = 1$) if two of the customer's components $i_1$, hosted by the provider $j_1$, and $i_2$, hosted by the provider $j_2$ (i.e. $\sum_{k,l} X[i_1, j_1, k, l] + \sum_{k,l} X[i_2, j_2, k, l] = 2$) have a flow relation between them. We translate that into the following constraint:

```
s.t. UseVPNT{(i1,i2) in Flow, j1 in P, j2 in P: j1!=j2}:
 sum{k in 1..nbti[j1], l in 1 .. nbinstances} X[i1, j1, k, l]
 + sum{k in 1..nbti[j2], l in 1 .. nbinstances} X[i2, j2, k, l]
 <= Z[j1, j2] + 1;
```

- In a similar fashion to the previous constraint, we define the constraints to calculate the numbers of internal and external links between the different components. There exists an internal link between two of the customer's components $i_1$ and $i_2$ (i.e. $ZIL[i_1, i_2, j] = 1$) if they are hosted by the same provider $j$ and if these two components have a flow relation between them. There exists an external link between two of the customer's components $i_1$ and $i_2$ (i.e. $ZEL[i_1, i_2, j_1, j_2] = 1$) if they are hosted by two different providers $j_1$ and $j_2$ and if these two components have a flow relation between them. The constraints are written as follow:

```
s.t. Zint{(i1,i2) in Flow, j in P }: ZIL[i1, i2, j] + 1 >=
 sum{k in 1..nbti[j], l in 1 .. nbinstances}(X[i1, j,k,l] + X[i2, j, k, l]);
s.t. NIn: NIL = sum{(i1,i2) in Flow, j in P } ZIL[i1, i2, j];
s.t. Zext{(i1, i2) in Flow, j1 in P, j2 in P: j1!=j2}: ZEL[i1, i2, j1, j2] + 1 >=
 sum{k in 1..nbti[j1], l in 1 .. nbinstances}X[i1, j1,k,l] +
 sum{k in 1..nbti[j2], l in 1 .. nbinstances}X[i2, j2, k, l];
s.t. Next: NEL = sum{(i1,i2) in Flow, j1 in P, j2 in P:j1!=j2 }
    ZEL[i1, i2, j1, j2];
```

Calculating the different intermediary costs and the global cost to be minimized:

- The total cost of instances used by the customer:

```
CostInstances_C: CostInstances =
 sum{i in Comp, j in P, k in 1..nbti[j], l in 1.. nbinstances}
  X[i,j,k,l] * InstancesPrices [j,k];
```

- The total cost of the customer's virtual private Cloud is the summation of the virtual private Cloud set up cost from the different used providers:

```
CostVPC_C: CostVPC = sum{i in P} Y[i] * VPC[i];
```

- The cost of the virtual private tunnels used is equivalent to the total of the VPT service cost used from each of the providers:

CostVPNT_C: CostVPT =
**sum**{i1 **in** P, i2 **in** P: i1!=i2} Z[i1, i2]* (VPT[i1] + VPT[i2]);

- Total cost of the configuration of both internal and external links:

  # *internal Links*
  CostIL_C: CostIL = NIL * CIL;
  # *external links*
  CostEL_C: CostEL = NEL * CEL;

- The global cost of the customer's architecture:

  # *Global Cost*
  GlobalCost_C: GlobalCost = CostInstances + CostVPC + CostIL + CostVPT +
  ↪ CostEL;

Finally, the cost optimization goal is simply written as follows:

/* *Goal* */
**minimize** z: GlobalCost;

**Example** As working example to test our linear model we take a customer application of five components as presented in Figure 5.2. The relations between the five components are:

$$\begin{aligned} Alone &= \{ Comp_4 \} \\ Flow &= \{ (Comp_1, Comp_2) , (Comp_2, Comp_1) , (Comp_1, Comp_3) \} \\ Concurrence &= \{ (Comp_2, Comp_5) \}. \end{aligned}$$

The consistency of this customer's model was first verified by KodKod. We run our model with the broker's federation data consisting of:

# *Number of Providers*
**param** nbp := 4;
# *The providers*
**set** P :=

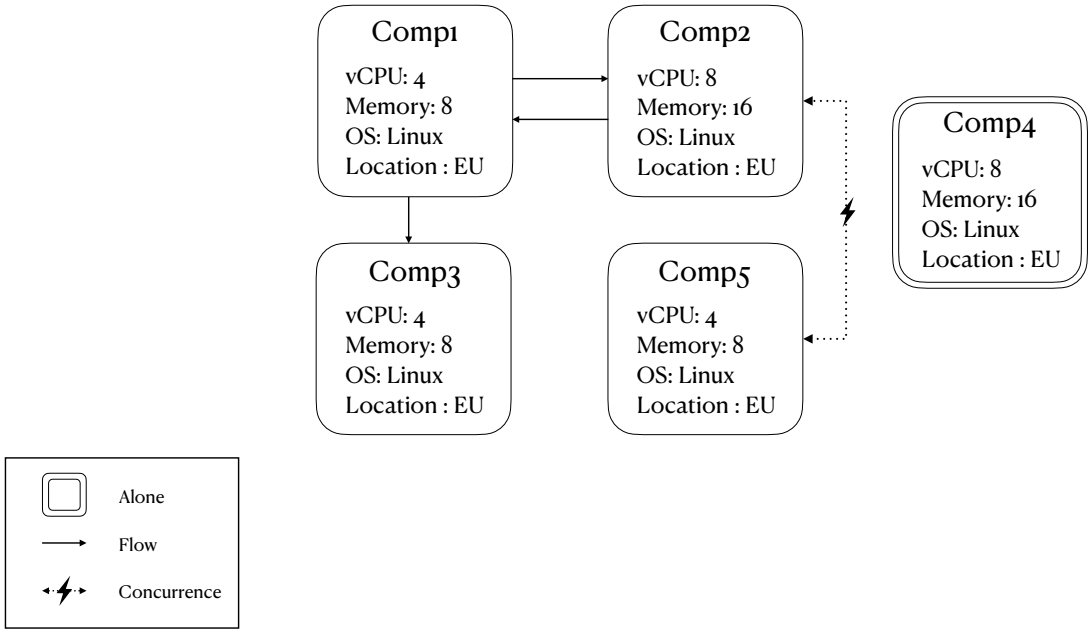



Figure 5.2 – *Customer Model Example*

Alibaba
Azure
Google
Amz
;
# *The number of instances per provider*
**param** nbti :=
Azure 16
Alibaba 84
Amz 11
Google 46
;
# *The number of instances per instance type*
**param** nbinstances := 5 ;

The problem is solved in *0.4 secs* and the components are affected to the appropriate instances hosted by the different providers as presented in Figure 5.3.

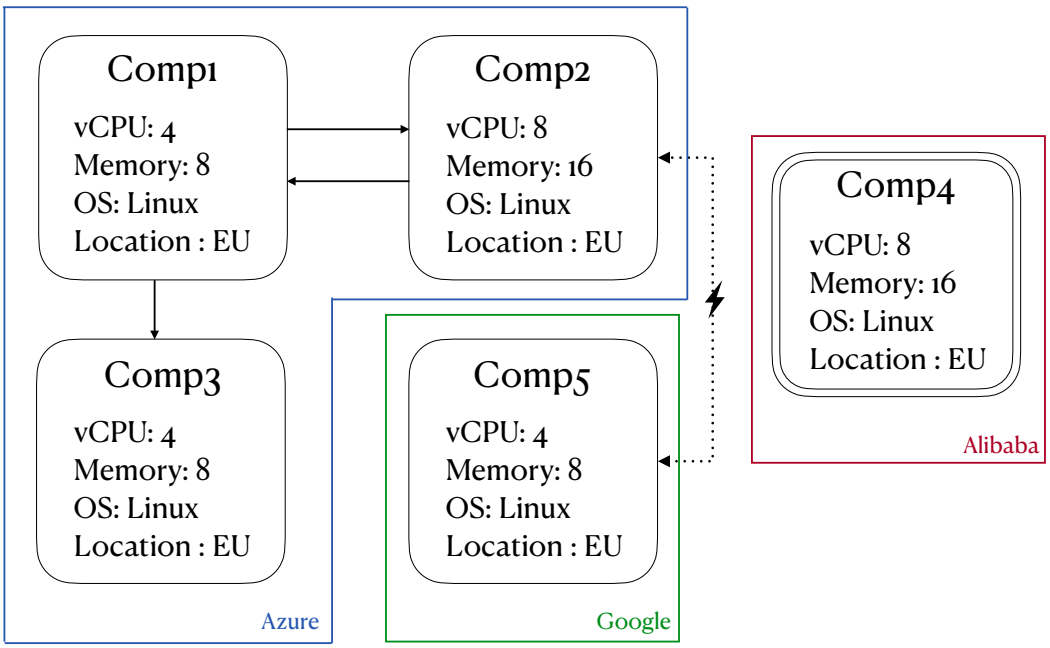



Figure 5.3 – *Customer Model Placement Example*

### 5.4.2 Brokerage Tool

The brokerage tool is written in Java and has three main parts.

**Consistency verification** The verification of the consistency of the customer's demand is done using the KodKod API. This step has stayed the same as the previous version of the tool, with only minor changes brought to the definition of verified formulas (i.e. in the sense of changing some of the characteristics of the components, modifying the isolation related formulas and removing those concerning the collaboration relation). Only once the demand is consistent we move to the next step. We generate part of the data file consisting of the values of the parameters, related to the different elements of the customer's request, of the linear model.

**Compatible Resources** In order to find the compatible resources we match the customer's functional requirements with the resources available in the broker's federation. The federation data is stored in a SQL database. We retrieve

the information about the different providers as well as the matching instances in order to create the data file for our linear model. In order to retrieve the matching instance types, and calculate the *Alpha* parameter presented in Section 5.4.1, for a component with *nv* virtual CPUs, *ms* memory size, *os* operation system and location *l*, we use the following SQL query:

```
1 select * from InstanceType where vCPU < 2*nv and vCPU >=
    nv and Memory < 2*ms and Memory >= ms and OS like os
    and Location like ''l\%''
```

As a reminder we fix the maximum of the matched type instance to twice the number of virtual CPUs and Memory size, due to the fact that once we exceed that value the cost of the reserved instance becomes higher than the on the demand one. Thus, it would be more beneficial for the broker to provision an on-demand instance to answer than customer's demand than use an already reserved one.

**Cost Optimization** The data retrieved from the second step are formated in a **GNU MathProg** data file format. Solving the optimization problem with as input this data file returns a placement strategy that minimizes the overall step-up cost of the customer's application model.

## 5.5 Case Study

### 5.5.1 Customer Model

As a case study for our brokerage solution, we choose a health care application. Let us imagine an international collaborative project working on a pandemic study. A project where different research teams from different laboratories, situated in different countries are working on the same project and need to collaborate in order to study, analyze and compare critical patient data collected by the different laboratories. All while respecting the federal regulations regarding data storage set by the different countries.

Let the following component types be the building bricks of the international pandemic study application:

- Data Input (DI): the component responsible for the data treatment and inputting it
- Data Storage (DS): the component responsible for the local data storage respecting the local regulations. We can imagine that in this component the anonymization work needed to protect the patients privacy is conducted before transmitting the data
- Centralized Data (CD): the component centralizing the data sets to analyze
- Data Analysis (DA): the component containing the program responsible for analyzing the data and inferring results

In our initial scenario, we can imagine the collaboration happening between two laboratories in two different European countries and a laboratory in the United States of America (USA). Due to the European standard regulations regarding data storage and treatment, we can imagine that the two European laboratories will be able to use the same data storage component localized anywhere in Europe. These regulations being different from those existing in the USA, the latter based laboratory cannot store their data nor analyze it in the same components as the European laboratories and will need to have different, data storage and data analysis components of his own located preferably in the USA. Let us imagine as an example the application presented in Figure 5.4. The functional characteristics of the different components of this application are summarized in table 5.1. The relations are the following:

$$\begin{aligned} \textit{Flow} &= \{ (DI_1, DS_1) , (DI_2, DS_2) , (DI_3, DS_2) , (DS_2, DA_2) , \\ &\quad (DS_1, DA_1) , (DA_1, CD) , (DA_2, CD)\} \\ \textit{Concurrence} &= \{ (DI_1, DI_2) , (DI_2, DI_3) , (DI_1, DI_3)\}. \end{aligned}$$

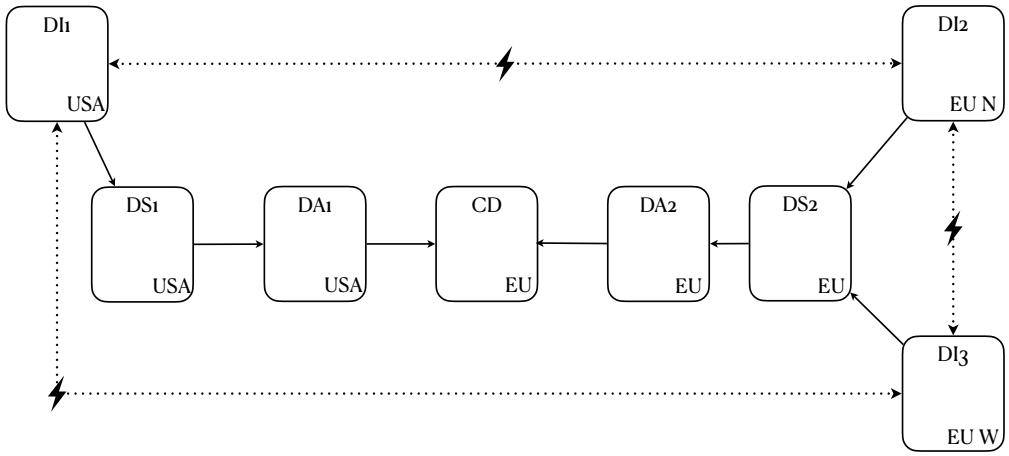



Figure 5.4 – *Case Study: A collaborative health-care application*

| Characteristics / Components | vCPU | Memory | OS | Location |
|---|---|---|---|---|
| DI1 | 4 | 8 | windows | USA |
| DI2 | 4 | 8 | windows | EU North |
| DI3 | 4 | 8 | windows | EU West |
| DA1 | 8 | 16 | linux | USA |
| DA2 | 8 | 16 | linux | EU |
| DS1 | 16 | 32 | linux | USA |
| DS2 | 16 | 32 | linux | EU |
| CD | 16 | 32 | linux | EU |

Table 5.1 – *Case Study: A collaborative health-care application - Components characteristics*

### 5.5.2 Federation Model

The federation model used, to find an adequate placement for the described health-care application, is the $Federation_1$ scenario presented in Table 5.2. In order to have a realistic set of metrics and results, we gathered data published on the different Cloud providers' official websites. We tried to get a set of heterogeneous types of instances from different locations.

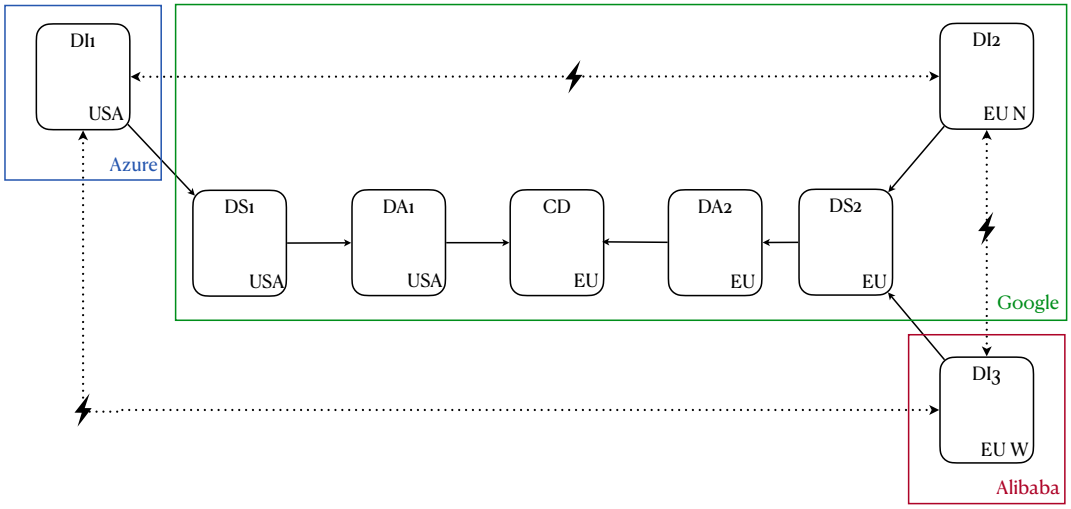



Figure 5.5 – *Case Study: A collaborative health-care application placement strategy*

### 5.5.3 Result

The consistency of the demand verified using the KodKod is done in *45 ms*. Using the same federation data as the one used in the example presented in Section 5.4.1, the optimization problem for this case study was solved in *10 minutes*, the components were placed in adequate instances hosted by the different providers as presented in Figure 5.5.

## 5.6 Evaluation

### 5.6.1 Introduction

Here we present some of the tests we have conducted to evaluate the performance of our tool and general model. The tests conducted here were with the aim of answering the following questions:

- $Q_1$: How does the time to solve the optimization problem change with the incrementation of the number of occurrences of types of instances?

- $Q_2$: How does this solve time change with different customers' demand scenarios, in regard to just adding more components with / without adding more relations?
- $Q_3$: How does this solve time change with the incrementation of the number of providers?

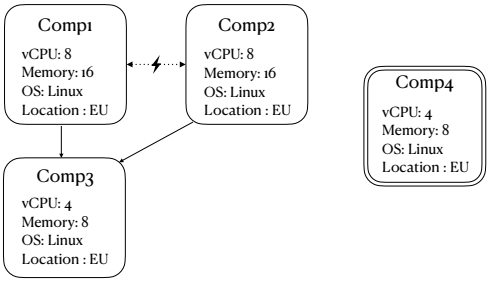
We tried to find an appropriate cost-efficient placement strategy for different customer scenarios, these will be presented in Section 5.6.2, in different federation scenarios that will be presented in Section 5.6.3. A summary of our results and answers to the previous question is presented in Section 5.6.4.

### 5.6.2 Customer Scenarios

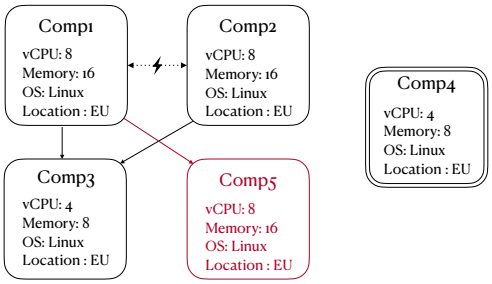
In order to answer the previous questions, we decided to test different incrementations of a simple customer application model. We increment the number of components and the relations in waves in order to see how the solving time reacts to these modifications. We start with a simple customer application model presented in Figure 5.6a, then we add different combinations of components and relations. The customer scenarios tested, with the highlighted differences between each scenario and the one prior to it, are presented in Figure 5.6.

### 5.6.3 Federation Scenarios

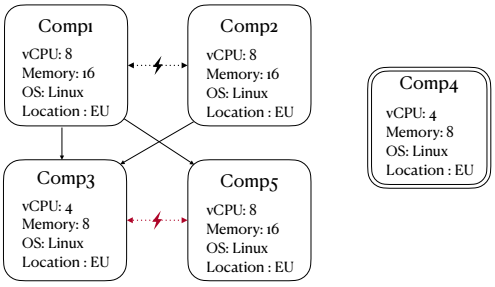
In order to have a better idea of how our model acts under different circumstances, we wanted to test out the different customer scenarios with different sets of resources (i.e. federations). To create different federation scenarios we copied the data we already have of one provider, in this case, Google, into two new providers (a.k.a ProvX and ProvY) then added each one at a time to make our federations. The federations scenarios are summarized in Table 5.2.



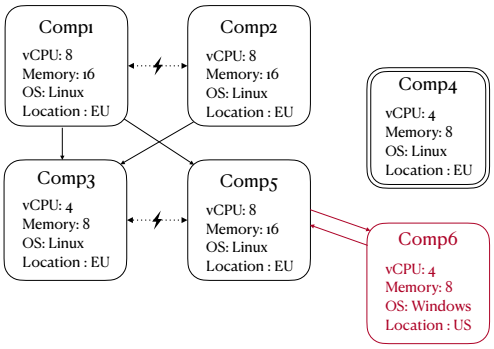

(a) *Scenario of a customer 1*



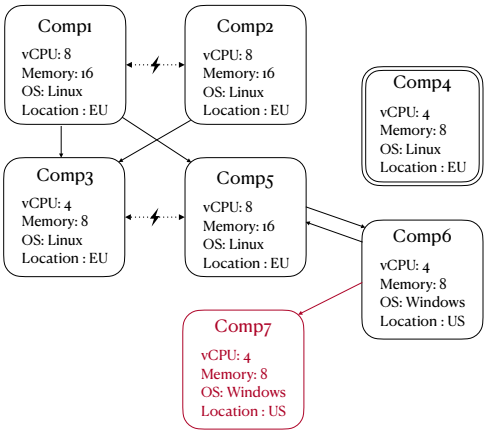

(b) *Scenario of a customer 2*



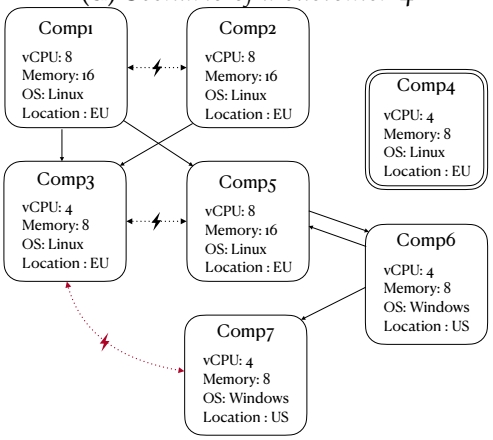

(c) *Scenario of a customer 3*



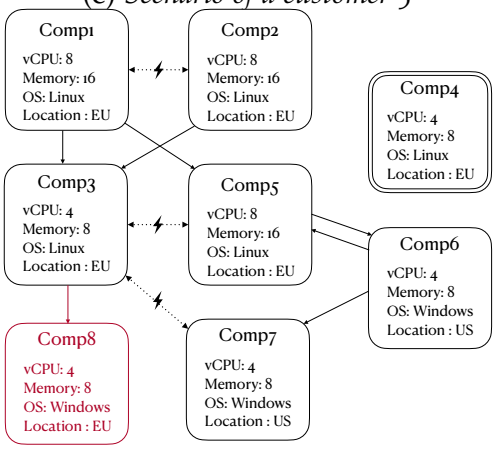

(d) *Scenario of a customer 4*



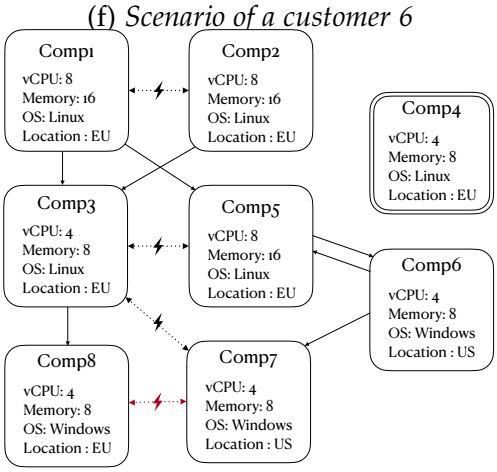

(e) *Scenario of a customer 5*



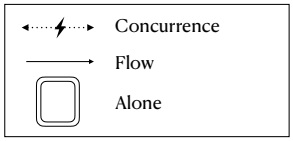

(f) *Scenario of a customer 6*




(g) *Scenario of a customer 7*




(h) *Scenario of a customer 8*




(i) *Legend*

Figure 5.6 – *The different Customers Scenarios*

| Federation Scenario | Providers | Number of Instance Types |
|---|---|---|
| Federation 1 | Alibaba | 85 |
| | Amazon | 12 |
| | Azure | 17 |
| | Google | 47 |
| Federation 2 | Alibaba | 85 |
| | Amazon | 12 |
| | Azure | 17 |
| | Google | 47 |
| | ProvX | 47 |
| Federation 3 | Alibaba | 85 |
| | Amazon | 12 |
| | Azure | 17 |
| | Google | 47 |
| | ProvX | 47 |
| | ProvY | 47 |

Table 5.2 – *The different Federations Scenarios*

### 5.6.4 Results

The results of these tests are summarized in Table 5.3.

The answers to the previous questions, relying on the results found from the different test scenarios are as follow:

- $A_1$: This was the answer that was very consistent throughout all of our tests. Increasing the number of occurrences of each type of instance increases the time needed to solve the problem. This could be considered as waisted time, especially in our solution, due to the size of the demand models and the fact that we're trying to find the placement for only one customer at a time. Thus, there is no need to have a large number of instance types occurrences. This number can be fixed to the maximum of the number of components with the same specifications that coexist in the customer's demand, which in the case of no similarities, it would be set to one, thus, that would decrease the time to solve significantly.

- $A_2$: Although as expected with more components, the more variables

| Customer Scenario | N of Components | N of Relations | Federation Scenario | N of Providers | N of Type Instances | N of Instances of each Type Instance | Time to Solve |
|---|---|---|---|---|---|---|---|
| Customer1 | 4 | 4 | Federation1 | 4 | 161 | 20 | 30.2 s |
| | | | | | | 10 | 1.9 s |
| | | | | | | 5 | 0.5 s |
| Customer2 | 5 | 5 | Federation1 | 4 | 161 | 20 | 378.4 s |
| | | | | | | 10 | 62.3 s |
| | | | | | | 5 | 2 s |
| Customer3 | 5 | 6 | Federation1 | 4 | 161 | 10 | 5.6 s |
| | | | | | | 5 | 0.7 s |
| Customer4 | 6 | 8 | Federation1 | 4 | 161 | 10 | 39.2 s |
| | | | | | | 5 | 5.5 s |
| Customer5 | 7 | 9 | Federation1 | 4 | 161 | 10 | 253 s |
| | | | | | | 5 | 5.6 s |
| Customer6 | 7 | 10 | Federation1 | 4 | 161 | 10 | 40.6 s |
| | | | | | | 5 | 3.8 s |
| Customer7 | 8 | 11 | Federation1 | 4 | 161 | 10 | 764.2 s |
| | | | | | | 5 | 6.8 s |
| Customer8 | 8 | 12 | Federation1 | 4 | 161 | 10 | 222.1 s |
| | | | | | | 5 | 6.3 s |
| Customer1 | 4 | 4 | Federation2 | 5 | 208 | 10 | 3.1 s |
| | | | | | | 5 | 0.5 s |
| Customer2 | 5 | 5 | Federation2 | 5 | 208 | 10 | 70.3 s |
| | | | | | | 5 | 3.5 s |
| Customer3 | 5 | 6 | Federation2 | 5 | 208 | 10 | 5.7 s |
| | | | | | | 5 | 0.7 s |
| Customer8 | 8 | 12 | Federation2 | 5 | 208 | 10 | 76.9 s |
| | | | | | | 5 | 5.3 s |
| Customer1 | 4 | 4 | Federation3 | 6 | 255 | 10 | 1.7 s |
| | | | | | | 5 | 0.6 s |
| Customer2 | 5 | 5 | Federation3 | 6 | 255 | 10 | 67.5 s |
| | | | | | | 5 | 6.7 s |
| Customer3 | 5 | 6 | Federation3 | 6 | 255 | 10 | 5.6 s |
| | | | | | | 5 | 0.6 s |
| Customer8 | 8 | 12 | Federation3 | 6 | 255 | 10 | 256.7 s |
| | | | | | | 5 | 9.5 s |

Table 5.3 – *Evaluation Summary*

generated and thus the longer it takes to find and prove the optimal solution of the linear problem. But, it is not fully true in this case. As we can see from the difference between the values of solving time of $Customer_2$ and $Customer_3$ scenarios (similarly between $Customer_5$ and $Customer_6$, as well as, $Customer_7$ and $Customer_8$), having the same number of components but more relations decreases the time needed to find the best solution. In fact, having a well-defined application model with explicit and well-specified relations, decreases the pull of possibilities and the

solver spends less time finding and proving the optimal solution. Having a more constrained demand with more relations explicitly defined between the components does decrease the time to solve.

- $A_3$: Here would be expected that with the increased number of possibilities the time to solve will accordingly be longer. As we can see in Table 5.2, the difference between the $Federation_1$ and $Federation_2$ scenarios was adding 1 provider *ProvX* and the difference between the $Federation_2$ and $Federation_3$ scenarios was adding provider *ProvY* which is an exact copy of the *ProvX*. To our surprise, although for all the customer scenarios the time to solve has indeed gotten longer due to the increased number of providers and instance types, for the $Customer_8$ scenario, the time to solve decreased when looking for a solution in the $Federation_2$ scenario then increased back up in the $Federation_3$ scenario. The optimization problem seems to be NP-hard which causes its behavior to be unexpected. Unfortunately, this issue is out of the scope of this thesis but the idea could be to try and find an adequate number of components and instances that would optimize the solving time.

## 5.7 Conclusion

In this chapter, we present an improvement of our initial Cloud brokerage solution. The high efficiency of the KodKod engine in proving the consistency of customers' demand in regard to the security rules we have defined, has motivated us to keep using it for that purpose. We then relied on linear programming to optimize the cost and find the placement strategy that best fits the customer's functional and non-functional requirements.

The consistency verification at early stages, before deploying the architecture into the Cloud, is very important, it spears the customer information leakage and attacks that could be revealed in the future. If we take as an example the application modeled in Figure 5.7, KodKod returns this model as inconsistent due to the fact that we have a concurrence between the components $Comp_1$

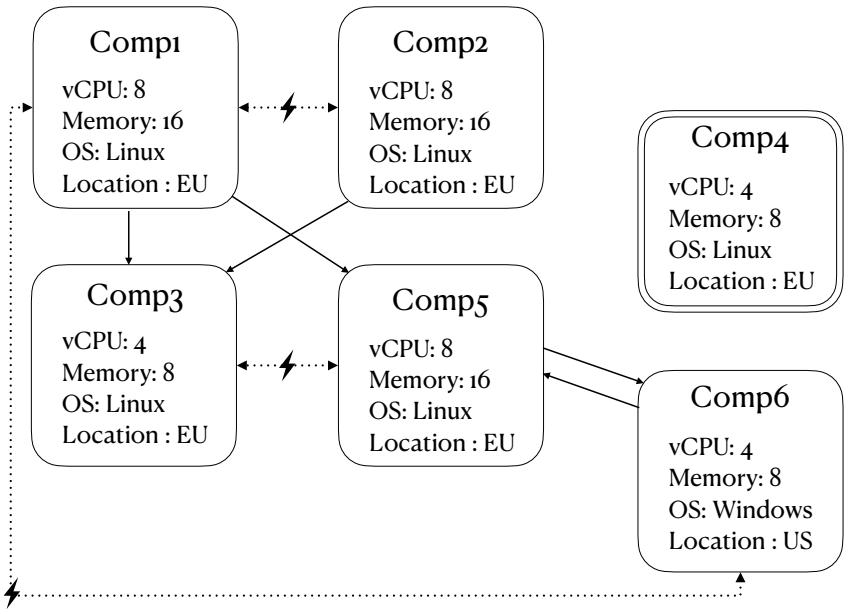



Figure 5.7 – *Inconsistant Customer demand example*

and $Comp_6$, all while having a transitive communication flow going through the component $Comp_5$. If we try to find a placement strategy without verifying its consistency, a possible placement strategy would be the one presented in Figure 5.8. In fact, the linear problem we try to solve does not include constraints about the consistency of the application model, if we were to include this, the problem would have become more complicated and thus even harder to solve. This example demonstrates the importance and efficiency of using both KodKod and linear programming in our solution. Each part of the latter uses the best and appropriate skill for its aim. More work can be conducted on both these parts of the broker in order to reach their full potential. We present some possible perspectives in the conclusion chapter.

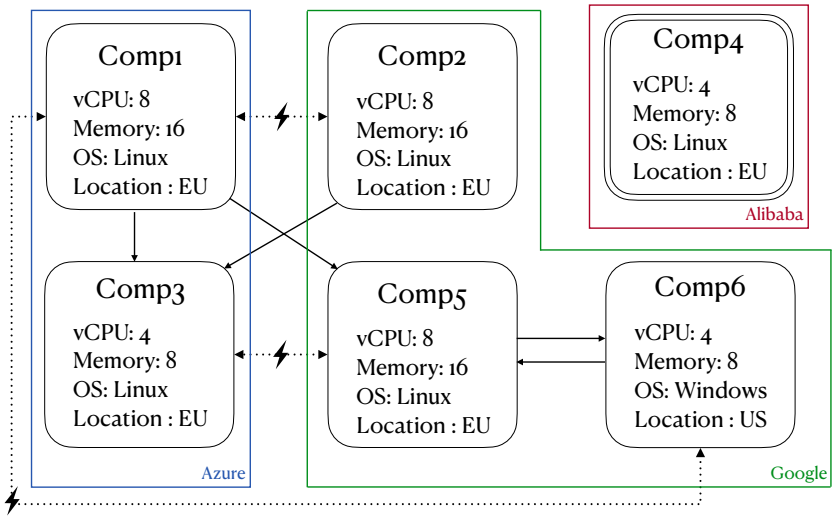



Figure 5.8 – *Inconsistant Customer demand placement*

# 6 Transformation from Alloy to Coq

Contents



## 6.1 Introduction

While using the Alloy language and exploring its potentials in describing and verifying the consistency of personalized security requirements in our customers' demands, we have come across some limitations related to the conception of the language as a whole.

In fact, Alloy [41] is a lightweight formal method as it relies on the small scope hypothesis: "examining all small cases is likely to produce interesting counterexamples". However, the Alloy analyzer cannot prove the absence of errors. Other formal tools such as the interactive theorem provers Coq [80] and

Isabelle [61] have been used to provide very strong guarantees on verified software, including a C compiler [46] and the kernel of an operating system [44]. Although in this work we focus on the use of an interactive theorem prover, it is worth noting the efforts that have been conducted in the automation of theorem proving, to mention Sledgehammer [14], a powerful interface from Isabelle to automated provers.

We think it is very valuable to use lightweight formal methods when modeling critical systems. In practice, if one is to use a tool such as Alloy as a first step, then wants to use more heavyweight tools such as Coq as a second step, the formalization done first is lost. To support the transition from Alloy to Coq, we propose a translator from Alloy models to Coq code. However, in addition to a "raw" translation from Alloy to Coq, we wanted our tool to provide some support to ease the proof in Coq of the assertions of an Alloy model. Such a support includes general lemmas about the properties of the set and relational operations of Alloy. The overviews of both Alloy and Coq have been previously presented in Chapter2.

The content of this chapter is based on the published article [75] and will be organized as follows. The basic principles of the transformation of Alloy models to Coq syntax we propose are described in Section 6.2. A step by step translation of the different elements of an Alloy model is presented in Section 6.3. Section 6.4 contains full example of the translation. Then, we discuss the current limitations of our tool in Section 6.6, and conclude in Section 6.7.

## 6.2 Basic Principles of the Transformation

**Logical Quantifiers and Connectives** Logical elements present in the Alloy language, are also present in Gallina (i.e. Coq language), either as primitives (universal quantification) or defined in the standard library (existential quantification, negation, conjunction, disjunction). For example, in the Alloy language we use the operators **{and, &&}** (resp. **{or, ||}**) to express conjunction (resp. disjunction) between two expressions. In Coq syntax conjunction (resp.

disjunction) between two propositions A and B is written: A /\B (resp. A \/ B ).

**Sets, Relations and Elements** In the Coq standard library, sets and binary relations are formalized using predicates. Given a type A, a subset of A is formalized as a predicate on A, i.e. a value of type A→**Prop**, and a binary relation on types A and B as a value of type A→B→**Prop**. We could use directly such a formalization, and consider higher arities: the simple example of Figure 2.1 indeed contains a relation of arity 3. Some other translation tools from Alloy to provers (discussed in Section 3.3) have explicit different translations for sets, binary relations, ternary translations, *etc,* which limits some of them to a given arity. While of course possible in Coq, we chose to avoid using the already existing formalism as it means we would have to generate as many versions of the operations as there is a combination of the arities, and as many supporting lemmas as there are combinations of these operations. Also in Alloy, relational operations can be applied to elements that are seen as singleton sets. Just for example sake, even if we choose to only consider up to binary relations and use the existing definition of *relation* from the library *Relations*. The *join* operator will need to be translated to three different versions: one to join a set and a relation, one to join a relation and a set and one to join two relations. In fact, we won't have only multiple versions of the different operators' definitions but the lemmas describing properties over these operators as well. If we define a different version of the inclusion operator for each of the considered arities, that means we will need to re-write and prove the lemmas concerning the properties of transitivity and reflection, just to mention a few, for each of these versions.

Therefore we chose to generalize the approach present in the Coq standard library: considering a type U (the universe of Alloy), a relation of arity $n$ (with $0 < n$) is formalized as a value of type U→...→U→**Prop** that contains $n$ U.

To be able to define operations on arbitrary relations, we first need to express the arity of a relation. This is done by the following definition:

```
1 Fixpoint arity (n : nat): Type :=
2  match n with
3    | 0 ⇒ Prop
4    | S n' ⇒ U → arity n'
5  end.
```

Therefore arity 1 simplifies to U→**Prop**, arity 2 to U→U→**Prop**, *etc.* With this definition we are able to translate any Alloy signature into a set of declarations of Coq values whose types are declared using arity.

To model an element as a singleton set, we define a Singleton predicate:

```
1 Fixpoint Singleton n (R: arity (S n)) : Prop :=
2 match n with
3 | 0 ⇒ ∃! (x:U), R x
4 | S n' ⇒ ∃! (x:U), Singleton n' (R x)
5 end.
```

Basically what this predicate does is that for a relation R of arity $n$ greater than 1, it indicates there exists a unique element x of U such that the partial application R x is also a singleton relation. For a relation of arity 1, it just states that there exists a unique x such that R x.

Unfortunately, the code above is not accepted by Coq. The problem is that Coq cannot determine without additional information that R x can be considered as a value of type arity n. To help the system we need "cast" functions (Figure 6.1). Note that both these functions are defined using the proof script language of Coq. In Coq, we need to be very explicit about the returned type. In Figure 6.1, we can see the two different cast functions: cast for the cast of arity 1 relations and cast' for relations of arity n greater than 1. However, these cast functions are not enough: we need to provide them a proof as their last argument. This is needed to prove that the relation provided as an argument is indeed an arity 1 relation for the cast function and arity n greater than 1 relation for the cast' function. This proof is simple, that is actually a proof by reflexivity, and we can use what Chlipala calls the "convoy pattern" [17, page 172] to get these proofs in the right-hand sides of the pattern matching construction.

```
1  Definition cast n1 (R1 : arity n1) (H: n1 = 0) : Prop.
2   subst. simpl in *. trivial.
3  Defined.
4  Definition cast' n1 n1' (R1 : arity n1) (H: n1 = S n1') : arity (S n1').
5   subst. simpl in *. trivial.
6  Defined.
7  Fixpoint Singleton n (R: arity (S n)) : Prop:=
8   match n as m return n = m → Prop with
9   | 0 ⇒ fun H ⇒ ∃! x, cast _ (R x) H
10  | S n' ⇒ fun H ⇒ ∃! y, Singleton n' (cast' _ _ (R y) H)
11  end eq_refl.
```

Figure 6.1 – *Actual Definition of Singleton*

And so, by using this pattern and applying the cast functions the new functional definition of the "Singleton" function is the one defined in lines 7-11 of Figure 6.1.

This small example shows that while having generic arity relations is indeed very generic, it makes the formalization more technically challenging. However, by providing general theorems on the Coq formalization of Alloy operations, we think the user of our tool will not have to deal with such technicalities most of the time.

**Operations** All the basic relational operations have the same shape as Singleton. For example, the inclusion operator **in** of Alloy is translated as (the cast and convoy pattern are omitted for a clearer definition):

```
1  Fixpoint IN n (R1: arity n)(R2: arity n): Prop :=
2  match n with
3  | 0 ⇒ R1 → R2
4  | S n' ⇒ ∀ (x:U), IN n' (R1 x) (R2 x)
5  end.
```

Basically it means that for all $n$-tuple t, if R1 t then R2 t.

The Alloy equality is not translated as the default syntactic equality (up to reduction) of Coq, but as:

```
1  Definition EQUAL n (R1: arity n)(R2: arity n): Prop :=
```

```
2 (IN R1 R2) ∧ (IN R2 R1).
```

Note that all the first nat arguments of these definitions are made implicit. It is therefore not necessary to give them explicitly when using these definitions: Coq infers them. Also instead of writing EQUAL a b, we use Coq's notations a == b.

Slightly more challenging operations are the join and the product. Again omitting the casts and the convoy pattern, the Alloy join operation is defined as shown in Figure 6.2. The actual detailed definition of the join operation with the definitions of different cast functions needed is shown in Figure 6.3.

```
1  Fixpoint JOIN_R n2 (R1: arity 1)(R2: arity (S n2)) : arity n2 :=
2   match n2 with
3   | 0 ⇒ ∃ x:U, (R1 x) ∧ (R2 x)
4   | S n2' ⇒ fun (y:U) ⇒ JOIN_R n2' R1 (fun (x:U) ⇒ R2 x y)
5   end.
6  Fixpoint JOIN n1 n2 (R1: arity (S n1)) (R2: arity(S n2)) : arity(n1+n2) :=
7   match n1 with
8   | 0 ⇒ JOIN_R n2 R1 R2
9   | S n1' ⇒ fun (y:U) ⇒ JOIN n1' n2 (R1 y) R2
10  end.
```

Figure 6.2 – *Definition of Join (Details Omitted)*

**Operation Properties** As mentioned before, in addition to translating the definitions, operations, formulas of Alloy, we also provide properties of Alloy operations. The first set of properties concerns the Alloy equality ==: we proved it is an equivalence relation and also that it is compatible with the Alloy operations, i.e. for an operation f, if for all a, b such that a == b, then f a == f b. This allows us to use the rewriting tactics of Coq while writing proofs. These are very important as most of the other properties are stated as equalities using ==.

```
1  Definition cast_0 n1 (R1 : arity n1) (H:n1 = 0) : Prop.
2    subst. simpl in *. trivial.
3   Defined.
4  Definition cast_1 n1 (R1 : arity (S n1)) (H:n1 = 0) : arity 1.
5    subst. simpl in *. trivial.
6  Defined.
7  Definition cast_0n n (R1 : Prop) (H:n = 0) : arity n.
8    rewrite H. simpl. trivial.
9   Defined.
10 Definition cast_0n2 n1 n2 (R: arity n2)(H:n1 = 0) : arity(n1+n2).
11   subst. simpl in *. exact R.
12 Defined.
13 Definition cast_n1S n1 n1' (R1 : arity n1) (H:n1 = S n1') : arity (S n1').
14   subst. simpl in *. trivial.
15 Defined.
16 Definition cast_Sn1 n1 n1' (R1 : arity (S n1')) (H:n1 = S n1') : arity n1.
17   subst. simpl in *. trivial.
18 Defined.
19 Definition cast_Sn1n2 n1 n2 n1' (R1 : arity(S(n1'+n2))) (H1:n1=S n1') : arity (n1+n2).
20   subst.  simpl. apply R1.
21  Defined.
22
23 Fixpoint JOIN_R n2 (R1: arity 1)(R2: arity (S n2)) : arity n2 :=
24   match n2 as m return n2 = m → arity n2 with
25   | 0 ⇒ fun H2 ⇒ cast_0n _
26                  (∃ x:U, (R1 x) ∧  (cast_0 _ (R2 x) H2))
27                  H2
28   | S n2' ⇒ fun H2 ⇒ cast_Sn1 _ _ (fun y ⇒ JOIN_R n2' R1 (fun x ⇒ (cast_n1S _ _ (R2 x) H2) y)) H2
29   end eq_refl.
30
31  Fixpoint JOIN n1 n2 (R1: arity (S n1)) (R2: arity(S n2)) : arity(n1+n2) :=
32   match n1 as n return n1 = n → arity (n1+n2) with
33   | 0 ⇒ fun H' ⇒  cast_0n2 _ _ (JOIN_R n2 (cast_1 _ R1 H') R2) H'
34   | S n1' ⇒ fun H1 ⇒ cast_Sn1n2 _ _ _ (fun y ⇒ JOIN n1' n2 (cast_n1S n1 n1' (R1 y) H1) R2) H1
35   end eq_refl.
36
```

Figure 6.3 – *Detailed Definition of Join*

The second set of properties are mostly algebraic properties. For example we have:

```
1  Lemma UNION_idem:
2    ∀ n (R: arity n), UNION R R == R.
3
4  Lemma UNION_assoc:
5    ∀ n (R1 R2 R3: arity n),
6      UNION R1 (UNION R2 R3) ==
7      UNION (UNION R1 R2) R3.
8
9  Lemma UNION_comm:
10   ∀ n (R1 R2: arity n),
11     UNION R1 R2 == UNION R2 R1.
12
13 Lemma INTERSECT_idem:
14   ∀ n (R: arity n),
15     INTERSECT R R == R.
16
17 Lemma INTERSECT_assoc:
18   ∀ n (R1 R2 R3: arity n),
19     INTERSECT R1 (INTERSECT R2 R3) ==
20     INTERSECT (INTERSECT R1 R2) R3.
21
22 Lemma INTERSECT_comm:
23   ∀ n (R1 R2: arity n),
24     INTERSECT R1 R2 == INTERSECT R2 R1.
```

We developed a tactic that is able to prove all of these properties (and most of the other properties defined in the library), the proof script in this case for each of the above mentioned lemmas is **Proof**. solve_alloy. **Qed**. The code of the solve_alloy tactic can be found in the Appendix A.

Other properties are more specific to Alloy operations. Here are some of the lemmas we provide:

- A lemma that states that if the join of a binary relation with itself contains the relation, then this relation is transitive:

```
1 Lemma JOIN_IN_transitive : ∀ R: arity 2,
2  IN (JOIN R R) R ↔ (∀ x y z, R x y → R y z → R x z).
```

- A lemma to state that the join operator is right distributive over the union operator:

```
1 Lemma JOIN_UNION_distr_r:
2   ∀ n1 n2 (R: arity(S n1))(R1 R2: arity (S n2)),
3    JOIN R (UNION R1 R2) ==
4    UNION (JOIN R R1) (JOIN R R2).
```

- A lemma to state that the union operator is left distributive over the difference operator:

```
1 Lemma UNION_DIFFERENCE_distr_l:
2   ∀ n (R1 R2 R3: arity (S n)),
3    EQUAL
4     (DIFFERENCE (UNION R1 R2) R3)
5     (UNION
6       (DIFFERENCE R1 R3)
7       (DIFFERENCE R2 R3)).
```

The proofs for these lemmas among others are all detailed in the Coq library present with the tool.

## 6.3 Alloy Models Translation

Now that we have translated the basic elements of the Alloy language, let us use them to translate Alloy models. Here we present how each of the components of Alloy models is translated into Coq syntax and the reasoning behind it. We continue using the Alloy model given in Figure 2.1, presented in Chapter 2, as example for the translation.

**Signatures** As we presented so far, everything that is going to be in our Coq translation of the Alloy models should be of type arity n. In order to follow this reasoning and to be able to manipulate Alloy signatures, we have decided to represent them in the format of Coq Variables (declarations) by specifying first their arity. Top-level signatures like Name, Addr and Book are sets and thus unary (i.e. arity 1) relations. Signature attributes are declared as relations (arity greater than 1) then a Hypothesis is added to the Coq code for their types, lines 2 and 3 in the following Coq translation shows the example of attribute addr:

```
1 Variable Name Addr Book: arity 1.
2 Variable addr: arity 3.
3 Hypothesis addr_sig: IN addr (PRODUCT Book (PRODUCT Name Addr)).
```

**Facts** A way of declaring facts about a system in Coq is by stating Hypothesis. Thus, Alloy model facts are translated in our tool to Hypothesis and the syntax is as follows:

```
1 Hypothesis Model_fact: translated_fact_formula.
```

**Functions and Predicates** both are transformed in the same way to Coq syntax. For reasons of re-usability and ease of application, we have decided to transform them into Coq inductive type definitions. This type is closed with respect to its introduction rules which explain the canonical ways of constructing an element of the type. In this sense, the inductive type characterizes the recursive type. The following examples are the transformation of the del predicate and lookup function presented in Figure 2.1. When writing the constructor for the inductive type, we start by modeling the "types" of the arguments as inclusions, possibly with additional expressions for modeling the cardinality. In the example of del, the argument b has type Book thus In b Book, but also b is an element, thus ONE b. We formalize functions as predicates, but with an additional argument that models the result returned by the function. In the case of lookup, the result is the value r_lookup:

```
1 Inductive del: arity 1 → arity 1 → arity 1 → Prop:=
```

```
| del_def: ∀ (b: arity  1) (b': arity  1)  (n: arity  1),
   IN b Book ∧  (ONE b) →
   IN b' Book ∧  (ONE b') →
   IN n Name ∧  (ONE n) →
   JOIN b' addr == DIFFERENCE (JOIN b addr) (PRODUCT n Addr) →
   del b b' n.

Inductive lookup: arity 1 → arity 1 → arity 1 → Prop:=
| lookup_def: ∀ (r_lookup: arity 1) (b: arity  1)  (n: arity  1),
   IN r_lookup Addr →
   IN b Book ∧  (ONE b) →
   IN n Name ∧  (ONE n) →
   r_lookup == JOIN n (JOIN b addr) →
   lookup b n  r_lookup .
```

**Assertions** are defined using the key word **Definition** in Coq syntax and then stated as Lemmas when called in an Alloy **check** block. Thus, the assertion delUndoesAdd is transformed as follows:

```
Definition delUndoesAdd:=
 ∀ (b: arity  1) (b': arity  1) (b'': arity  1) (n: arity  1)(a: arity  1),
 ( NO (JOIN n (JOIN b addr)) ∧  add b b' n a ∧  del b' b'' n ) →
 JOIN b addr == JOIN b'' addr.

Lemma delUndoesAdd_Lemma: delUndoesAdd.
```

## 6.4 Example: The Address Book

In the previous subsections, we presented most of the translation of the Alloy example of Figure 2.1. Figures 6.4–6.6 present the automatic translation using our tool of the two other assertions addIdempotent and addLocal, as well as the proof scripts we wrote to prove two of the corresponding lemmas.

```
Definition addIdempotent:=
 ∀ (b b' b'' a n: arity  1),
 (add_ b b' n a  ∧  add_ b' b'' n a ) →
 JOIN b' addr == JOIN b'' addr.
```

```
Definition addLocal:=
 ∀ (b b b' a n n': arity  1) r_1 r_2,
 lookup b n' r_1 →
 lookup b' n' r_2 →
 (add_ b b' n a  ∧  not(n == n')) →
 r_1 == r_2 .
```

Figure 6.4 – *Translation of the Assertions* addIdempotent *and* addLocal

```
Lemma delUndoesAdd_Lemma : delUndoesAdd.
Proof.
 unfold delUndoesAdd.
 intros b b' b'' n a H. destruct_and.
 assert(Hadd: add_ b b' n a) by trivial.
 assert(Hdel: del b' b'' n) by trivial.
 inversion Hadd; inversion Hdel; subst.
 destruct_and.
 (* We are ready to prove: JOIN b addr == JOIN b'' addr *)
 assert(Hr1: JOIN b'' addr ==  DIFFERENCE (JOIN b' addr) (PRODUCT n Addr)) by trivial.
 assert(Hr2: JOIN b' addr == UNION (JOIN b addr) (PRODUCT n a)) by trivial.
 rewrite Hr1, Hr2.
 rewrite UNION_DIFFERENCE_distr_l with (R1:=JOIN b addr).
 rewrite UNION_NO_l by
    (apply DIFFERENCE_IN_NO;
     apply PRODUCT_IN_compat with (R1:=n);
     auto using IN_refl).
 rewrite DIFFERENCE_NO_INTERSECT by
    (assert(HH: NO (JOIN n (JOIN b addr))) by trivial;
     castsimpl; intros;
     specialize(HH x);
     contradict HH;
     intuition eauto).
 reflexivity.
Qed.
```

Figure 6.5 – *Proof of Lemma* delUndoesAdd

```
1  Lemma addIdempotent_Lemma: addIdempotent.
2  Proof.
3    unfold addIdempotent.
4    intros b b' b'' n a H. destruct_and.
5    assert(Hadd1: add_ b b' n a) by trivial.
6    assert(Hadd2: add_ b' b'' n a) by trivial.
7    inversion Hadd1; inversion Hadd2; subst.
8    assert(Hr1: JOIN b'' addr == UNION (JOIN b' addr)(PRODUCT n a)) by trivial.
9    assert(Hr2: JOIN b' addr == UNION (JOIN b addr)(PRODUCT n a)) by trivial.
10   rewrite Hr1, Hr2.
11   rewrite ← UNION_assoc, UNION_idem.
12   reflexivity.
13 Qed.
```

Figure 6.6 – *Proof of Lemma addIdempotent*

A recommended style in Coq, is to avoid using explicitly automatically generated names by tactics. Our destruct_and tactics, that basically systematically replaces hypotheses of the form A∧ B by two hypotheses A and B, and automatically generates names for these new hypotheses. The inversion tactic also automatically generates names. To explicitly give names to the hypotheses we want to manipulate, we use the assert tactic of Coq that is used to prove an intermediate result. In our case, we just state and give an explicit name for already existing hypotheses, hence the use of the trivial tactic used to prove the assertion (for e.g. lines 10–11 of Figure 6.5). In order to get the corresponding formulas to the definition of an Alloy predicate, or an Alloy function, the inversion tactic of Coq is needed (e.g. line 7 of Figure 6.5 and line 7 of Figure 6.6). Using the assert tactic, we give explicit names to the hypotheses generated by inversion (for e.g., lines 8–9 of Figure 6.6).

The two other main characteristics of these proof scripts are:

- The use of the rewrite tactic, that relies on the proofs of == is an equivalence relation, and the Alloy operations are compatible with this equivalence relation (e.g. line 13 of Figure 6.5 and line 10 of Figure 6.6).
- The systematic use of properties proved on Alloy operations: for example

the distributivity of the union over the difference (line 15 of Figure 6.5) and the associativity and idempotence of the union (line 11 of Figure 6.6).

Most of the proof scripts are based on the elements described above. The exception are lines 20–23 of Figure 6.5. The proof of the condition of the lemma DIFFERENCE_NO_INTERSECT is in a way more "low-level" than the other parts of the proof scripts as it directly makes use of the definitions of some Alloy operations. One non standard Coq tactic is castsimpl: it is a tactic we provide, and that simplifies the application of the Alloy operations and also removes all the casts in the hypotheses and the goal. In the example the goal before calling castsimpl is:

```
1  NO (INTERSECT (JOIN b addr) (PRODUCT n Addr))
```

meaning we have to prove that the intersection of JOIN b addr and PRODUCT n Addr is empty, while after it is:

```
1  ∀ y x : U, ∼ ((∃ x0 : U, b x0 ∧  addr x0 y x) ∧  n y ∧  Addr  x)
```

As castsimpl simplifies the hypothesis HH in a similar way, it is quite easy to finish the proof.

These two proof scripts show that while most of the time the user can rely on proofs by rewrite and application of operation properties, when it is not possible, the proof writing remains accessible. With these two proofs, we guarantee that the Alloy assertions hold for arbitrary sets and relations Book, Name, Addr and addr.

## 6.5 The Tool

We developed a tool to automatically translate Alloy models to Coq syntax and that comes with a supporting library to assist users in the proof process. The tool is written in Java and relies on ANTLR [79] for parsing. ANTLR (ANother Tool for Language Recognition) is a parser generator widely used for building languages, tools and frameworks. It reads a grammar and generates a parser that recognizes the language defined by this grammar, which allows to

build and walk parse trees. We used ANTLR in the background of our tool to parse Alloy models and generate an abstract syntax tree. Then we generate as output a file containing the translated Alloy model into Coq syntax according to the translation rules described in Section 6.3. The version of Coq used in this implementation is the **8.7.2** version. There are about 2 KLoC of non-generated Java code, and the Coq supporting library Alloy is about 600 LoC.

## 6.6 Discussion

The tool presented in this chapter shows the potential of translating and proving the correctness of critical Alloy models, but it still has some limitations in its current state.

The first limitation is the subset of the Alloy language that is supported. There is one aspect that the current translation does not handle which is the cardinality of sets and relations. The design choice we made is not incompatible with dealing with cardinalities. It however requires additional hypotheses. A way to remediate from this limitation would be to, first, make the universe U countable, this is actually in line with what is considered in Alloy, but it is not set as a hypothesis in our current Coq modeling. Then to compute the cardinality (the # operator in Alloy), the argument should be a finite relation, this can be added as a hypothesis each time the operator is used. Other Alloy features that could be integrated into our tool are integer support, Coq can handle integer definition and thus, adding this to our solution will only require some formalization efforts. The other feature to improve further is the arrow operation. For now, our arrow operation is by default a many to many arrow operation, while Alloy's arrow operation handles different multiplicities.

The second limitation is not related to the translation itself, but rather to the support provided to the user in the translated Coq code. Although we do provide a few Coq tactics to ease the work to prove what are assertions in Alloy, currently the proofs are written mostly manually by the users. More powerful tactics are needed to enrich the Coq Alloy library.

In translating one formal language to another one, the question of the correctness of the translation arises. One possibility would be to have a Coq representation of Alloy's abstract syntax and then give a Coq semantics to this syntax, this would be a formalization of Alloy in Coq. Then we could implement in Coq what is currently the back-end of our translation in Java, the generation of Coq code from Alloy's syntax. Proving the correctness of the translation would then mean check that the semantics and the translation are equivalent. However, it is very likely that the semantics could be given using the same basic constructs we use for our translation, they would be essentially no difference between the Coq *semantics* of Alloy, and the Coq *translation* of Alloy. Another possibility would be to have a deep embedding of both Alloy and Coq in Coq and check that the translation (from syntax to syntax) preserves the semantics. However, our formalization of Alloy in Coq uses features that formalizations of Coq in Coq (for e.g. [33]) do not currently handle.

## 6.7 Conclusion

This chapter presented a tool for translating Alloy models into Coq code. Alloy's main objects are relations. In fact, sets are unary relations, elements are considered as singleton sets. We chose to keep this view in Coq and to consider, as in the module Relation_Definitions of Coq's standard library, that a relation is a function to **Prop**. This module, however, only considers binary relations, meaning they have a type U→U→**Prop** where U is the type of the universe.

We decided to generalize this approach. This choice required us to use dependent types everywhere in the Coq library that provides the primitive relational operations of Alloy and supports the translation. One of the major challenges of the translation and proofs afterwards was due to this choise. In fact working with dependent types is quite complex because the types can contain arbitrary terms. For example, consider the types arity (2 + 1) and arity (1 + 2). The two types represent, intuitively, the same proposition: namely, that

a relation of arity 3. The two types can be transformed, by calculation, into the same type arity (3). Then, any proof M of arity (2 + 1) should also be a proof of arity (1 + 2) or of arity (3). Therefore it is very important to find ways to guide through the calculation (i.e. reduction) process. We were able to use our tool on examples and prove with Coq the lemmas generated by the translation. Thus, this choice of Coq formalization seems appropriate.

# Conclusion & Perspectives 7

Contents



## 7.1 Conclusion

In this thesis, we present a fully functional brokerage tool to assist potential Cloud customers in the process of integrating the Cloud. Users specify a component-based application they want to be deployed in the Cloud by giving first the functional requirements by specifying the number of components and their functional description, then the non-functional requirements, which represent their personalized security requirements in the format of relations between the different components. The Kodkod model finder verify the consistency of this demand, then the broker matches the available types of instances to each of the components. Finally, another version of the broker uses linear programming to find the placement strategy that optimizes the overall cost. Thus, we present a placement strategy that does not only respects the functional and non-functional requirements of the customer, but also minimizes the overall cost of integrating the Cloud.

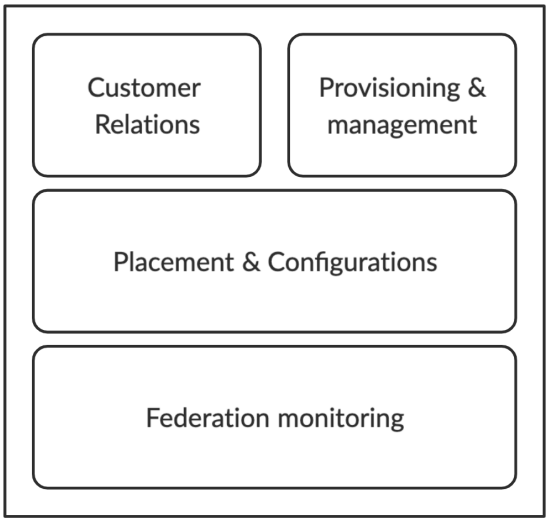



Figure 7.1 – *Broker Components*

## 7.2 Perspectives

In this section, we present some of the ideas that we couldn't pursue fully due to either lack of time or resources, that could improve the work presented in this thesis even further.

**Broker Components** The implementation of the new broker model can be parallelized and executed in different steps. The broker may be divided into different components as presented in Figure 7.1, each having different functionalities. These components can be developed as part of a collaborative work between multiple experts in the needed domains or even as part of a bigger school project. The functionalities of the different components can be summarized as follows:

- Customer relations: this component will be responsible of all actions related to the customer, starting from describing the demand, verifying the consistency, returning a counterexample in case of anomaly, to presenting a possible placement solution.
- Provisioning and management: this component will be the interface between the broker and the Cloud resource providers. It will be responsible

for the initial resource provisioning and subsequent provisioning. An algorithm, or an already implemented solution, can be integrated here to do the calculations of the amounts of resources needed as a start point, resources needing to be provisioned or released at a given time taking into consideration the number of customers, their needs, and the overall tendency of the market.

- Placement and configuration: this component will be finding the matching Cloud resources that satisfy the customer's functional requirements and that will do the adequate network configuration to satisfy the non-functional requirements. In case of insufficient resources, it alerts the *Provisioning and management* component.

- Federation monitoring: this component will allow the broker to monitor its federated resource both from a functional and security standpoints.

**Linear Model** First, the current linear model aims to find a placement strategy that minimizes the overall cost. We can imagine refining the linear model in order to have it take into consideration one or multiple other optimization criteria chosen by the customer.

Second, the evaluation tests we have conducted have shown the complexity of the optimization problem. Other tests varying the number of occurrences of the instances, sizes of customers' architectures as well as varying the types of resources provisioned could give us more insight on how to further improve on our linear problem in order to reach it's full potential.

In a theoretical point of view, it would be worth showing that the optimization problem is NP-hard and try to refine the complexity. Further, it may be worthwhile to define some heuristics to find adequate placements, especially when the linear model takes a long time to solve.

**Alloy2Coq** As mentioned before, one of the main motivations for this tool is our project around a broker for the Cloud that takes into account user security requirements that can be expressed as first-order relational logic formulas and

that we checked using Alloy/Kodkod [74]. In order to increase the trust in this broker, we aim at formalizing all the hypothesis made on the system and make sure that if the formal requirements given by the user contain no error and are added to the system, then conclusions about the security of the new state of the system can be drawn. This case study requires a significantly larger translation and Coq proofs than the examples we considered so far.

# Appendix

# Tactic solve_alloy

# A

```
1  Definition cast_0 n1 (R1 : arity n1) (H:n1 = 0) : Prop.
2    subst. simpl in *. trivial.
3  Defined.
4
5  Definition cast_1 n1 (R1 : arity (S n1)) (H:n1 = 0) : arity 1.
6    subst. simpl in *. trivial.
7  Defined.
8
9  Definition cast_0n2 n1 n2 (R: arity n2)(H:n1 = 0) : arity(n1+n2).
10   subst. simpl in *. exact R.
11 Defined.
12
13 Definition cast_n1S n1 n1' (R1 : arity n1) (H:n1 = S n1') : arity (S n1').
14   subst. simpl in *. trivial.
15 Defined.
16
17 Definition cast_Sn1 n1 n1' (R1 : arity (S n1')) (H:n1 = S n1') : arity n1.
18   subst. simpl in *. trivial.
19 Defined.
20
21 Definition cast_nS n n' (R1: U → arity n)(H:n'=S n): arity n'.
22   rewrite H. simpl. trivial.
23 Defined.
24
```

```
25 Definition cast_0n n (R1 : Prop) (H:n = 0) : arity n.
26   rewrite H. simpl. trivial.
27 Defined.
28
29 Definition cast_Sn1n2 n1 n2 n1' (R1 : arity(S(n1'+n2))) (H1:n1=S n1') : arity (n1+n2).
30   subst.  simpl. apply R1.
31 Defined.
32
33 Definition cast_plus0S n1 n2 n2' (R1: arity(S(n1+n2'))) (H1: n1=0)(H:n2=S n2') : arity (n1+n2).
34   subst. simpl. apply R1.
35 Defined.
36
37 Ltac castsimpl :=
38  repeat (
39     repeat(unfold cast_0, cast_1, cast_0n2, cast_n1S, cast_Sn1, cast_Sn1n2,
40           cast_0n, cast_plus0S, cast_nS, eq_rect_r in * );
41     repeat rewrite ← UIP.Nat.eq_rect_eq in *;
42     simpl in * ).
43
44 Ltac destruct_and :=
45  repeat
46   match goal with
47   | [ HH: _ ∧ _ ⊢ _ ] ⇒ destruct HH
48   end.
49
50 Ltac solve_alloy :=
51  match goal with
52  | [ ⊢ ∀ (n1:nat)(n2:nat), _ ] ⇒
53   unfold EQUAL;
54   destruct_and;
55   induction n1 as [ | n1 IH1]; [ induction n2 as [ | n2 IH2] | idtac] ;
56   castsimpl; try split; intros; castsimpl; firstorder
57  | [ ⊢ ∀ (n:nat), _ ] ⇒
```

```
58    unfold EQUAL;
59    destruct_and;
60    induction n as [ | n IH];
61    castsimpl; try split; intros; castsimpl; try eapply IH; firstorder
62  end.
```

# Bibliography


[1] Google App Engine: Platform as a service. `https://cloud.google.com/appengine/`. 4

[2] S. Aldawood, F. Fowley, C. Pahl, D. Taibi, and X. Liu. A Coordination-Based Brokerage Architecture for Multi-cloud Resource Markets. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 7–14, 2016. 42, 44

[3] R. Alnemr, S. Pearson, R. Leenes, and R. Mhungu. COAT: Cloud Offerings Advisory Tool. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 95–100, 2014. 31, 35, 48

[4] J. Alsina, S. Iturriaga, S. Nesmachnow, A. Tchernykh, and B. Dorronsoro. Virtual Machine Planning for Cloud Brokering Considering Geolocation and Data Transfer. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 352–359, Dec 2016. 43, 44

[5] Verma Amandeep and Kaushal Sakshi. Cloud Computing Security Issues and Challenges: A Survey. In Ajith Abraham, Jaime Lloret Mauri, John F. Buford, Junichi Suzuki, and Sabu M. Thampi, editors, *ACC (4)*, volume 193 of *Communications in Computer and Information Science*, pages 445–454. Springer, 2011. 8

[6] G. F. Anastasi, E. Carlini, M. Coppola, and P. Dazzi. Qbrokage: A genetic approach for qos cloud brokering. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 304–311, June 2014. 33, 36, 48

[7] Marco Anisetti, Claudio Agostino Ardagna, Filippo Gaudenzi, and Ernesto Damiani. A certification framework for cloud-based services. In Sascha Ossowski, editor, *ACM Symposium on Applied Computing (SAC)*, pages 440–447. ACM, 2016. 47

[8] Mosek ApS. MOSEK. `https://www.mosek.com/`, Software, 2000. 21

[9] A. Aral and T. Ovatman. Subgraph Matching for Resource Allocation in the Federated Cloud Environment. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 1033–1036, June 2015. 43, 44, 45

[10] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating model checking and theorem proving for relational reasoning. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *Relational and Kleene-Algebraic Methods in Computer Science (RAMICS)*, pages 21–33, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 49, 50

[11] AWS. AWS Site-to-Site VPN. `https://docs.aws.amazon.com/vpn/latest/s2svpn/VPC_VPN.html`. 78

[12] Azure. Azure VPN gateway. `https://azure.microsoft.com/fr-fr/services/vpn-gateway/#overview`. 78

[13] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010. x

[14] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 107–121, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 102

[15] John Cartlidge and Philip Clamp. Correcting a financial brokerage model for cloud computing: closing the window of opportunity for commercialisation. *Journal of Cloud Computing*, 3(1):2, Apr 2014. 37

[16] CEA-List & INRIA. Frama-C. `https://frama-c.com/`. 27

[17] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2014. 104

[18] Tommaso Cucinotta, Diego Lugones, Davide Cherubini, and Karsten Oberle. Brokering slas for end-to-end qos in cloud computing. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 610–615. INSTICC, SciTePress, 2014. 39, 45, 48

[19] Jackson Daniel. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006. 10

[20] W. Dawoud, I. Takouna, and C. Meinel. Infrastructure as a service security: Challenges and solutions. In *2010 The 7th International Conference on Informatics and Systems (INFOS)*, pages 1–8, March 2010. 9

[21] T. Deng, J. Yao, and H. Guan. Maximizing Profit of Cloud Service Brokerage with Economic Demand Response. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 1907–1915, 2018. 42, 44, 48

[22] Gregory D. Dennis. *A relational framework for bounded program verification*. PhD thesis, Massachusetts Institute of Technology, 2009. ix

[23] Shuai Ding, Chengyi Xia, Qiong Cai, Kaile Zhou, and Shanlin Yang. QoS-aware resource matching and recommendation for cloud computing systems. *Applied Mathematics and Computation*, 247:941 – 950, 2014. 43, 44, 48

[24] Torlak Emina and Jackson Daniel. Kodkod: A Relational Model Finder. In Grumberg Orna and Huth Michael, editors, *Tools and Algorithms for the Construction and Analysis of Systems: (TACAS)*, pages 632–647, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. x, 16, 64

[25] Jin Tong Fang Liu, Robert B. Bohn Jian Mao, Mark L. Badger John V. Messina, and Dawn M. Leaf. SP 500-292. NIST cloud computing reference architecture. Technical report, Gaithersburg, MD, United States, 2011. 3, 6

[26] Diogo A. B. Fernandes, Liliana F. B. Soares, João V. Gomes, Mário M. Freire, and Pedro R. M. Inácio. Security issues in cloud environments: a survey. *International Journal of Information Security*, 13(2):113–170, Apr 2014. v, 1, 8

[27] Fico. Xpress Optimization Suite. `https://www.fico.com/en/products/fico-xpress-solver`, Software, 2012. 21

[28] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press series. Thomson/Brooks/Cole, 2003. 21

[29] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In Jin Song Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, pages 581–596, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ix

[30] Milena Frtunic, Filip Jovanovic, Mladen Gligorijvic, Lazar Dordevic, Srecko Janicijevic, Per Håkon Meland, Karin Bernsmed, and Humberto Castejon. CloudSurfer - A Cloud Broker Application for Security Concerns. In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 199–206. INSTICC, SciTePress, 2013. 31, 35, 48

[31] Saurabh Kumar Garg, Steve Versteeg, and Rajkumar Buyya. SMICloud: A Framework for Comparing and Ranking Cloud Services. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 210–218, Washington, DC, USA, 2011. IEEE Computer Society. 30, 35, 48

[32] Georg Geri, Ray Indrakshi, Anastasakis Kyriakos, Bordbar Behzad, Toahchoodee Manachai, and Hilde Houmb Siv. An aspect-oriented methodology for designing secure applications. *Information and Software Technology*, 51(5):846 – 864, 2009. ix

[33] Stéphane Glondu. *Towards certification of the extraction of Coq*. Theses, Université Paris Diderot, June 2012. 116

[34] Google. Google Apps for work. `https://www.google.fr/intx/fr/work/apps/business/`. 4

[35] Google. Google Cloud VPN. `https://cloud.google.com/solutions/patterns-for-connecting-other-csps-with-gcp#managed_vpn_between_cloud_providers`. 78

[36] Asma Guesmi. *Formal specification and analysis of security policies in a cloud brokerage process*. PhD thesis, Université d'Orléans, July 2016. vii, 10, 52

[37] Asma Guesmi, Patrice Clemente, Frédéric Loulergue, and Pascal Berthomé. Cloud Resources Placement Based on Functional and Non-Functional Requirements. In *International Conference on Security and Cryptography (SECRYPT)*, pages 335–324. ScitePress, 2015. vii, xi, 40, 45, 48, 52, 69

[38] Juncal Alonso Ibarra, Leire Orue-Echevarria, Marisa Escalante, Gorka Benguria, and Gorka Echevarria. Federated cloud service broker (FCSB): an advanced cloud service intermediator for public administrations. In *CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science, Porto, Portugal, April 24-26, 2017.*, pages 356–363, 2017. 33, 34, 36, 48

[39] IBM. IBM ILOG CPLEX Optimization Studio. `https://www.ibm.com/analytics/cplex-optimizer`, Software, 2012. 21

[40] Daniel Jackson. Automating First-Order Relational Logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software*

*Engineering: Twenty-First Century Applications*, SIGSOFT '00/FSE-8, page 130–139, New York, NY, USA, 2000. Association for Computing Machinery. ix, 13, 14

[41] Daniel Jackson. *Software Abstractions*. MIT Press, revised edition, 2012. ix, 14, 16, 18, 101

[42] Dongjae Kang, Sokho Son, and Jinmee Kim. Design of Cloud Service Brokerage System Intermediating Integrated Services in Multiple Cloud Environment. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 8(12):2134 – 2139, 2014. 39, 44, 48

[43] Eunsuk Kang and Daniel O Jackson. Designing and Analyzing a Flash File System with Alloy. *Int. J. Software and Informatics*, 3:129–148, 2009. ix

[44] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. x, 102

[45] Sebastian Krings, Michael Leuschel, Joshua Schmidt, David Schneider, and Marc Frappier. Translating Alloy and extensions to classical B. *Science of Computer Programming*, 188:102378, 2020. 50

[46] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. x, 102

[47] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009. 27

[48] Leroy Xavier, INRIA. CompCert. `http://compcert.inria.fr/`. 27

[49] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 10th ACM SIG-*

*COMM Conference on Internet Measurement*, IMC '10, pages 1–14, New York, NY, USA, 2010. ACM. 30, 35, 48

[50] Misbah Liaqat, Victor Chang, Abdullah Gani, Siti Hafizah Ab Hamid, Muhammad Toseef, Umar Shoaib, and Rana Liaqat Ali. Federated cloud resource management: Review and discussion . *Journal of Network and Computer Applications*, 77:87 – 105, 2017. 7

[51] Andrew Makhorin. GLPK (GNU Linear Programming Kit). `https://www.gnu.org/software/glpk/glpk.html`, 2000. 21, 78

[52] D. Marinov and S. Khurshid. TestEra: a novel framework for automated testing of Java programs. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 22–31, Nov 2001. x

[53] Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for linear optimization problems. 2012. 21

[54] Étienne Michon, Julien Gossa, Stéphane Genaud, Léo Unbekandt, and Vincent Kherbache. Schlouder: A broker for IaaS clouds. *Future Generation Computer Systems*, 69:11–23, 4 2017. 32, 35

[55] Microsoft. Microsoft Azure: Cloud computing platform and services. `https://azure.microsoft.com/`. 4

[56] Microsoft. Office365. `https://azure.microsoft.com/`. 4

[57] Francesco Moscato, Rocco Aversa, Beniamino Di Martino, Teodor-Florin Fortis, and Victor Ion Munteanu. An Analysis of mOSAIC ontology for Cloud Resources annotation. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *FedCSIS*, pages 973–980, 2011. 32, 36

[58] Mariano M. Moscato, Carlos López Pombo, and Marcelo F. Frias. Dynamite: A tool for the verification of alloy models based on PVS. *ACM Trans. Softw. Eng. Methodol.*, 23:20:1–20:37, 2014. 50

[59] Ranesh Kumar Naha and Mohamed Othman. Cost-aware service brokering and performance sentient load balancing algorithms in the cloud. *Journal of Network and Computer Applications*, 75:47 – 57, 2016. 38, 44, 48

[60] S. K. Nair, S. Porwal, T. Dimitrakos, A. J. Ferrer, J. Tordsson, T. Sharif, C. Sheridan, M. Rajarajan, and A. U. Khan. Towards Secure Cloud Bursting, Brokerage and Aggregation. In *2010 Eighth IEEE European Conference on Web Services*, pages 189–196, Dec 2010. 34, 36, 48

[61] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002. x, 102

[62] OpenShift. OpenShift: The Open Hybrid Cloud Application Platform by Red Hat. `https://www.openshift.com/`. 4

[63] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, 2015. 24

[64] Przemyslaw Pawluk, Bradley Simmons, Michael Smit, Marin Litoiu, and Serge Mankovski. Introducing STRATOS: A Cloud Broker Service. In Rong Chang, editor, *IEEE CLOUD*, pages 891–898. IEEE, 2012. 32, 36, 48

[65] S. Pearson. Toward Accountability in the Cloud. *IEEE Internet Computing*, 15(4):64–69, July 2011. v, 2, 46

[66] Pierre Fouilhoux. Le programme GLPSOL du kit GLPK. `http://www-desir.lip6.fr/~fouilhoux/documents/glpk_glpsol.pdf`. xv, 23, 24

[67] David Power, Mark Slaymaker, and Andrew Simpson. Automatic conformance checking of role-based access control policies via Alloy. In *Engineering Secure Software and Systems (ESSOS)*, pages 15–28. Springer, 2011. ix

[68] Tahina Ramananandro. Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method. *Formal Aspects of Computing*, 20(1):21–39, Jan 2008. ix

[69] Owen Rogers and Dave Cliff. A financial brokerage model for cloud computing. *Journal of Cloud Computing: Advances, Systems and Applications*, 1(1):2, Apr 2012. 36, 37, 44, 48

[70] Robert Seater, Daniel Jackson, and Rohit Gheyi. Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering*, 12(2):77–102, Apr 2007. ix

[71] Daniele Sgandurra and Emil Lupu. Evolution of Attacks, Threat Models, and Solutions for Virtualized Systems. *ACM Comput. Surv.*, 48(3):46:1–46:38, February 2016. 9

[72] Danhua Shao, Sarfraz Khurshid, and Dewayne E. Perry. Whispec: White-box Testing of Libraries Using Declarative Specifications. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 11–20, New York, NY, USA, 2007. ACM. x

[73] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for Security. *CoRR*, abs/1806.04355, 2018. ix

[74] Salwa Souaf, Pascal Berthomé, and Frédéric Loulergue. A Cloud Brokerage Solution: Formal Methods Meet Security in Cloud Federations. In *2018 International Conference on High Performance Computing & Simulation, HPCS 2018, Orleans, France, July 16-20, 2018*, pages 691–699. IEEE, 2018. vi, vii, 11, 52, 69, 122

[75] Salwa Souaf and Frédéric Loulergue. A First Step in the Translation of Alloy to Coq. In Yamine Aït Ameur and Shengchao Qin, editors, *Formal Methods and Software Engineering - 21st International Conference on Formal*

*Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings*, volume 11852 of *Lecture Notes in Computer Science*, pages 455–469. Springer, 2019. vi, x, 11, 102

[76] Thiruselvan Subramanian and Nickolas Savarimuthu. Application based brokering algorithm for optimal resource provisioning in multiple heterogeneous clouds. *Vietnam Journal of Computer Science*, 3(1):57–70, Feb 2016. 41, 42, 44, 48

[77] S. Sundareswaran, A. Squicciarini, and D. Lin. A Brokerage-Based Approach for Cloud Service Selection. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 558–565, June 2012. 38, 44, 48

[78] Ali Sunyaev and Stephan Schneider. Cloud Services Certification. *Commun. ACM*, 56(2):33–36, February 2013. 46

[79] Terence Parr. ANTLR (ANother Tool for Language Recognition). `https://www.antlr.org/`. 114

[80] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`. x, 101

[81] Adel Nadjaran Toosi, Rodrigo N. Calheiros, and Rajkumar Buyya. Interconnected Cloud Computing Environments: Challenges, Taxonomy, and Survey. *ACM Comput. Surv.*, 47(1):7:1–7:47, May 2014. 8

[82] Johan Tordsson, Rubén S. Montero, Rafael Moreno-Vozmediano, and Ignacio M. Llorente. Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. *Future Generation Computer Systems*, 28(2):358 – 367, 2012. 40, 41, 44, 48

[83] Emina Torlak. *A constraint solver for software engineering : finding models and cores of large relational specifications*. PhD thesis, Massachusetts Institute of Technology, 2009. x

[84] Emina Torlak and Greg Dennis. Kodkod for Alloy Users. In *First Alloy Workshop*, 2006. 18

[85] Emina Torlak, Mana Taghdiri, Greg Dennis, and Joseph Near. Applications and extensions of Alloy: Past, present, and future. *Mathematical Structures in Computer Science*, 23:915–933, 2013. ix

[86] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. A proof assistant for Alloy specifications. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 422–436, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 49

[87] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L. Rivest, Emil Stefanov, and Nikos Triandopoulos. Hourglass Schemes: How to Prove That Cloud Files Are Encrypted. In *ACM Conference on Computer and Communications Security (CCS)*, pages 265–280, 2012. 47

[88] Luis M. Vaquero, Luis Rodero-Merino, and Daniel Morán. Locking the sky: a survey on IaaS cloud security. *Computing*, 91(1):93–118, Jan 2011. 4, 9

[89] X. Wang, S. Wu, K. Wang, S. Di, H. Jin, K. Yang, and S. Ou. Maximizing the Profit of Cloud Broker with Priority Aware Pricing. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 511–518, 2017. 37, 44, 48

Ce document a été écrit à l'aide l'éditeur de texte GNU Emacs et du logiciel de composition typographique LaTeX 2ε.

INSA CENTRE VAL DE LOIRE

Salwa SOUAF

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS

# Formal Methods Meet Security in a Cost Aware Cloud Brokerage Solution

## Résumé

Avec la demande croissante sur les ressources Cloud, vient l'augmentation du nombre de fournisseurs de services Cloud. Ce qui rend le processus de choix entre les différents fournisseurs et offres difficile pour les clients potentiels. Le courtier Cloud proposé, est une entité tierce qui intermédiaire la relation entre les clients et les fournisseurs Cloud. Le courtier guidera le client tout au long du processus d'intégration du Cloud. Notre courtier prend en considération les exigences fonctionnelles (c'est-à-dire la quantité et la description des ressources) et non fonctionnelles (c'est-à-dire les propriétés de sécurité) du client dés le premier contact. Après avoir reçu la description de la demande du client, notre courtier commence par vérifier sa cohérence et renvoie un contre-exemple en cas d'incohérences. Nous utilisons des méthodes formelles couplés avec de la programmation linéaire pour vérifier la consistance et trouver le placement approprié dans une fédération de Clouds. Après avoir communiqué l'emplacement trouvé, si il existe un, au client, ce dernier décidera soit d'accepter cette offre ou pas. Une fois que le courtier reçoit la confirmation du client le modèle sera prêt pour le déploiement.

Mots clés: informatique en nuages, sécurité, méthodes formelles, courtage infonuagique, programmation linéaire.

## Résumé en anglais

With the growth of Cloud Computing, comes the growth of the number of companies offering different cloud services, which day after another causes the overwhelming of the consumers. Many researches have been conducted in order to assist consumers in the process of choosing the right provider based on several properties, but not many of them have succeeded in integrating the security aspect. We suggest a third party, namely Cloud broker, that will intermediate the relation between cloud customers and providers. The broker will guide the customer through the whole process of integrating the Cloud. Our broker find a cost efficient placement strategy while taking into consideration the functional and non-functional requirements of the customer. We use formal methods to verify the consistency of the customer's demand, matching techniques and linear modeling to find the appropriate placement in a federation of Clouds. In this thesis will be presented the full creation and thought process behind this brokerage solution, limitations encountered and solutions proposed.

Key words: Cloud computing, Security, Formal Methods, Cloud brokering, Linear Programming.