Static and Automatic Resource Composition in Web-based Environments : An Application for Buildings Energy Management

Lara Kallab

DOCTORAL THESIS

Static and Automatic Resource Composition in Web-based Environments: An Application for Buildings Energy Management

Lara KALLAB

Advisors:	Pr. Richard CHBEIR	Univ Pau & Pays Adour, France
	Pr. Michael Mrissa	University of Primorska, Slovenia
Reviewers:	Pr. Mike PAPAZOGLOU	University of Tilburg, The Netherlands
	Pr. Walid GAALOUL	Télécom SudParis, France
Examiners:	Pr. Ernesto EXPOSITO	Univ Pau & Pays Adour, France
	Dr. Sana SELLAMI	Aix-Marseille University, France
Collaborator:	Dr. Pierre BOURREAU	NOBATEK/INEF4, France

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science

December 6, 2019

To mom and dad...

Acknowledgements

I would like to take the opportunity to pay tribute and express my heartfelt thanks to all those who have contributed to the realization of my doctoral project, which would not have been possible without their support and guidance.

At the outset, I would like to express my greatest gratitude to my advisors: Pr. Richard Chbeir and Pr. Michael Mrissa, who supported me all the way and allowed me to grow as a scientist researcher. I am deeply thankful to Pr. Richard Chbeir for his constant support, constructive comments, strong belief in my abilities whenever I started doubting myself, and inspiring wealth of knowledge, which helped me to establish the overall direction of my research. I would also like to thank Pr. Michael Mrissa for his valuable guidance, helpful advices, trust, and continuous encouragement that kept me motivated throughout my thesis. It has been a wonderful opportunity and a privilege to have worked with them.

I extend my sincere thanks to Dr. Pierre Bourreau for his useful suggestions, support, patient, and valuable inputs particularly on the technical aspect of my work.

My sincere gratitude goes to Pr. Mike Papazoglou and Pr. Walid Gaaloul for dedicating their time to read my thesis report and giving me helpful comments. I am also thankful to the rest of my thesis committee Pr. Ernesto Exposito, Dr. Sana Sellami, and Dr. Pierre Bourreau for their presence in the jury and their valuable comments.

I gratefully acknowledge the funding received towards my thesis project: SIBEX (Solutions Intelligentes pour le Bâtiment en EXploitation), from the "Association Nationale de la Recherche et de la Technologie" (ANRT), France. Without their financial support, my work would not have been possible.

I am highly thankful and honored to be a member of NOBATEK/INEF4, where I have been working during my PhD. I am very much grateful to Dr. Pascale Brassier and Dr. Christophe Cantau for their constant aid and support, which helped me in keeping things into perspective and promoting my work in useful industrial projects.

I would like to thank Dr. Gilbert Tekli for his assistance, support and inspiration during the first year of my thesis. I greatly appreciate his motivations and efforts to pursue my PhD in France. Also, I am grateful to Dr. Joe Tekli for his confidence, encouragement, and valuable advices.

I would also like to thank all the technical and administrative staff of the IUT de Bayonne for the convivial atmosphere, their sympathy, and allowing me to use the materials and facilities. A special thanks to Philippe ANIORTE for his kindness, support and positive vibes.

I am so grateful to my colleagues in NOBATEK/INEF4 and LIUPPA who were supportive and caring, each in their own unique way. Special thanks to my friend Elio Mansour for his availability in the hard moments, continuous listening, and the good times that kept flowing. I also thank Karam Bou Chaaya for his empathy, kindness and support. To Joelle Céméli, Emlyne Tessari Rossi, Christelle Martinez, and Igor Perevoschikov, thank you for your encouragement that kept my moral high.

I would never forget the unconditional support and care of my friends scattered around the world: Lebanon, France and abroad, especially Khouloud Salameh (Kouki), Rima Akiki (Rimzi), Eliana Raad (Louna), and Nadine Abi Aad (Nado). Thank you for not making distance a big deal, and for your kind well-wishes, prayers, phone calls, and texts.

To the music, coffee machines, chocolate, fitness rooms, beers, delicious food, and my piano, thank you for lifting me up and giving me moments of pleasure. I had a blast!

I would like to thank also my adorable cousins Myriam El Gharby and Tracy Klaiany, for being there for me. I surely will not forget the comfort they provided me, with the crazy moments and laughs we had.

To my best and dearest friend Nathalie Charbel, with whom I shared all this journey from the very beginning to the very end, I am truly blessed for having you by my side. From the bottom of my heart, thank you for the care, support, and advices you have given me, and above all, for making ordinary moments extraordinary ones.

To my beloved family, whom without I would not have lasted, I cannot thank you enough. Mom and Dad you are my heroes, and all that I am today or I ever hoped to be, I owe to you. I am profoundly grateful for your sacrifices, your unconditional love, and consistent support to reach my dreams. I will be forever indebted to you for your limitless encouragements to explore new directions in life and seek my own destiny. As for my brothers Dory and David, thank you for believing in me and for always showing me how proud you are of who I am. I love you all...

Lastly, and most importantly, I bow my head to God, who blessed me with health, faith, and strength to undertake and complete my work. I never prayed for an easy life, but to be a strong person, and He kept me strong enough all the way.

Abstract

Nowadays, a plethora of Web-based environments (Web applications, Web platforms, etc.), publish their functions as RESTful services, i.e., self-contained and self-describing resources that follow the REpresentational State Transfer (REST) architectural style principles. As the Web has become a major medium of communication, integrating objects (e.g., smart devices) into the Web and taking advantage of its open popular standards has created an emerging trend: the Web of Things (WoT). In the WoT, objects expose their functions also as resources respecting the REST principles. Each resource provides well defined functions that meet specific users' requests. However, there are cases in which a single resource is not sufficient to answer users' requests, and often, combining two or more resources forming a resource composition, achieves the desired output. Nevertheless, several challenges are to be addressed when composing resources.

In this thesis, we address three challenges. The first one consists on verifying the behavior of static resource compositions built manually by the user, as several design errors may occur (e.g., end-loops preventing other resources to run, and datatype mismatch between the Inputs/Outputs of the linked resources). For the other two challenges, the targeted Web environments are hybrid providing: (i) dynamic resources (connected to/removed from the environment at different instances), and (ii) static resources (established to be always available). The challenges focus respectively on the automatic resource discovery, while considering resource location (whenever exposed by objects), and the automatic selection of the appropriate resources to form suitable compositions satisfying users' requests.

To cope with these challenges, we first propose a formal model based on Colored Petri Nets (CPN) that maps resources behavior with their composition to CPN. This allows to use CPN behavioral properties to verify the correctness of static compositions behavior. Then, we propose a formal graph representation linking static resource to dynamic ones, allowing adapted graph algorithms to explore the semantically annotated descriptions of the traversed graph resources, in order to identify automatically the required resources. The resource discovery process uses an original defined indexing schema that allows identifying the resources based on their location (if exposed by objects), and enhancing resource search in large Web environments connecting many resources. As for automatic resource selection, we present a Selection Strategy Adaptor that selects the suitable resources to form several compositions with different implementation alternatives, taking into account Quality of Resource (QoR), Inputs/Outputs matching of related resources, as well as resource availability.

Our proposal is generic as it can be applicable in Web environments belonging to different application domains. However, in this thesis, it has been illustrated in the smart buildings domain, more particularly, in projects for managing buildings' energy behavior.

Résumé

Chapitre 1

Introduction

Aujourd'hui, de nombreux environnements basés sur le Web, tels que les applications Web et les plateformes Web, fournissent leurs fonctions en tant que services RESTful, qui sont des ressources autonomes et auto-descriptives qui suivent le style architectural du REST (REpresentational State Transfer). Comme le Web est devenu un moyen de communication principal, intégrer des objets dans le Web (tels que les appareils intelligents) et exploiter ses technologies populaires, ont abouti au développement d'un nouveau concept : le Web des objets (ou Web of Things (WoT) en Anglais). Dans le WoT, les objets exposent leurs fonctions en tant que ressources respectant également les principes de REST. Chaque ressource fournit des fonctions qui répondent à des demandes spécifiques des utilisateurs. Mais, parfois, une seule ressource ne suffit pas pour répondre à certaines demandes et, souvent, la combinaison de plusieurs ressources, formant une composition de ressources, permet d'obtenir le résultat souhaité. Néanmoins, il existe plusieurs défis à relever lors de la composition de ressources.

Dans cette thèse, nous faisons face à trois défis. Le premier consiste à vérifier le bon comportement des compositions statiques, où les ressources sont sélectionnées et liées manuellement par l'utilisateur, vu que plusieurs erreurs de conception peuvent survenir, telles que des boucles produites empêchant d'autres ressources de s'exécuter et la non correspondance entre les types de données des entrées/sorties des ressources liées. Les deux autres défis sont liés aux environnements Web hybrides fournissant des ressources : (i) dynamiques (connectées/déconnectées de l'environnement à différents instants) et (ii) statiques (établies pour être toujours disponibles). Dans ce cas, les défis portent respectivement sur la découverte automatique des ressources, en tenant compte de la position des ressources (exposées par des objets), et sur la sélection automatique des ressources appropriées pour former des compositions répondant aux demandes de l'utilisateur.

Pour faire face à ces différents défis, nous proposons une plateforme générique, applicable dans plusieurs domaines, intitulée StARC (a Framework for Static and Automatic Resource Composition) qui a pour but de composer statiquement et automatiquement des ressources qui respectent les principes de REST. Dans cette plateforme, nous présentons plusieurs contributions liées aux deux aspects de la composition de ressources : statique et automatique. Ces contributions se résument comme suit :

- Un modèle formel basé sur les réseaux de Petri colorés (Colored Petri Nets (CPN) en Anglais), un langage graphique pour la conception, la spécification, la simulation et la vérification des systèmes, qui exprime le comportement des ressources et leur composition en CPN. Cela permet d'utiliser les propriétés de CPN pour vérifier le bon comportement des compositions statiques.
- Une représentation graphique formelle liant les ressources statiques à celles dynamiques, permettant à des algorithmes de graphe, adaptés à explorer les descriptions sémantiques des ressources parcourues, de découvrir automatiquement les ressources requises. Le processus de la découverte utilise un schéma d'indexation défini pour identifier les ressources en fonction de leur position (si elles sont exposées par des objets) et améliorer la recherche dans des environnements Web connectant de nombreuses ressources.
- Un adaptateur de stratégie de sélection qui permet de sélectionner les ressources les plus appropriées parmi celles qui sont candidates, pour former plusieurs compositions ayant différentes alternatives d'implémentation, tenant compte de la qualité des ressources, du matching entre les entrées/sorties des ressources liées, ainsi que de leurs disponibilités.

Nos contributions proposées sont génériques qui peuvent être appliquées dans des environnements Web appartenant à différents domaines d'applications métiers. Cependant, dans cette thèse, nos solutions sont illustrées dans le domaine des bâtiments intelligents, en s'appuyant sur des projets de gestion du comportement énergétique des bâtiments.

Chapitre 2

Connaissances de Base

Le chapitre 2 présente des concepts technologiques importants afin de bien comprendre les solutions proposées. D'abord, nous expliquons le concept des services Web, les principaux protocoles/principes supportés pour leur implémentation en se focalisant sur le principe de REST (celui adopté dans notre travail), ainsi que les langages Web sémantiques utilisés pour que les propriétés des services Web, telles que les fonctions fournies et les entrées/sorties, soient compréhensibles par les machines. Ensuite, nous présentons une synthèse concernant les langages existants pour décrire les services RESTful (type des services ciblés dans cette thèse), conçu comme des ressources, en mettant l'accent sur les langages basés sur l'hypermédia tel que le vocabulaire Hydra (celui utilisé dans notre travail).

Chapitre 3

Utilisation des Réseaux de Petri Colorés pour Vérifier les Compositions de Ressources

Dans le chapitre 3, nous proposons une solution pour la vérification des compositions de ressources statiques (construites manuellement par l'utilisateur)

avant leurs exécutions. À cette fin, nous définissons un modèle formel utilisé pour mapper le comportement des ressources et leur composition aux réseaux de Petri colorés (Colored Petri Nets (CPN) en Anglais), un langage graphique pour la conception, la spécification, la simulation et la vérification de systèmes. Sur la base du modèle défini, nous pouvons utiliser les propriétés de CPN (par exemple, Accessibilité (Reachability), pour garantir que l'état final souhaité est accessible, et Vivacité (Liveness), pour garantir que toutes les ressources puissent être exécutées lors de l'exécution de la composition), pour vérifier le bon comportement des ressources avant leur exécution. Cette approche est expérimentée dans un outil basé sur les CPN (CPN Tools) pour vérifier une composition de ressources illustrée dans le domaine de la gestion de l'énergie des bâtiments. Dans le chapitre, nous présentons aussi un prototype développé dans le cadre d'un projet de recherche et développement lancé par NOBATEK/INEF4[1], nommé SIBEX (Solutions Intelligentes pour le Bâtiment en EXploitation), afin de vérifier les compositions de ressources orientées bâtiment avant exécution. Le prototype permet de modéliser, valider, convertir et exécuter des ressources composées vérifiées à travers différents moteurs développés. Plusieurs tests sont effectués pour tester les différents moteurs implémentés, y compris le moteur de validation basé sur notre modèle CPN défini.

Chapitre 4

Découverte Automatique des Ressources Tenant Compte de leurs Localisation dans des Environnements Web Hybrides

Dans le chapitre 4, nous présentons une approche pour la découverte automatique de ressources, tenant compte de leurs localisations (si elles sont exposées par des objets), dans des environnements Web hybrides connectant des ressources: (1) statiques (établies pour être toujours disponibles) et liées entre elles selon le principe HATEOAS (Hypermedia As The Engine of Application State en Anglais) du REST, et (2) dynamiques pouvant être connectées et déconnectées de l'environnement à différents instants. Pour ce faire, nous proposons une représentation formelle liant les ressources (dynamiques et statiques) dans un seul graphe de ressources. Ceci en définissant des ressources virtuelles qui se lient aux ressources statiques et qui contiennent des ressources dynamiques. Le graphe de ressources peut être parcouru par plusieurs algorithmes de graphe (BFS et DFS dans notre travail) qui sont adaptés pour découvrir k nombre de ressources ($k \in \mathbb{N}^*$) pour chaque fonction requise pour réaliser la demande de l'utilisateur. La découverte des ressources se base sur des annotations sémantiques intégrées dans les descriptions de ressources (exprimées avec le vocabulaire Hydra dans cette thèse). Dans le chapitre, nous définissons également un schéma d'indexation en 3 dimensions qui lie les ressources à leurs fonctions fournies et à leurs localisations (si elles sont exposées par des objets). Le schéma d'indexation sert à identifier les ressources de collecte de données en fonction de leurs positions, et à améliorer la recherche de ressources dans des environnements

[1] https://www.nobatek.inef4.com

Web connectant un grand nombre de ressources. Plusieurs tests sont menés pour évaluer la performance de notre solution proposée dans différentes configurations d'environnement Web (par exemple, variation du nombre de ressources, et variation du nombre de fonctions requises pour réaliser la demande de l'utilisateur), et selon 4 aspects: dynamicité, multiplicité, efficacité et évolutivité. Les résultats soulignent l'importance d'utiliser le schéma d'indexation, en particulier dans les environnements Web comprenant un très grand nombre de ressources, pour améliorer le temps de réponse de la découverte de ressources.

Chapitre 5

Sélection Automatique des ressources basée sur leurs Qualités dans des Environnements Web Hybrides

Dans le chapitre 5, nous présentons une approche de sélection automatique de ressources appliquée dans des environnements Web hybrides connectant des ressources statiques et dynamiques. Dans cette approche, nous définissons d'abord un modèle formel qui relie les ressources identifiées au cours du processus de découverte de ressources dans un graphe acyclique dirigé, en fonction de leurs fonctions fournies. Ensuite, nous proposons un adaptateur de stratégie de sélection qui permet de sélectionner, parmi les ressources candidates qui ont été découvertes, les ressources plus appropriées pour former plusieurs compositions ayant différentes alternatives d'implémentation, répondant aux besoins de l'utilisateur (par exemple, compositions optimales et sans coût: ayant les scores les plus élevés mais sans aucune charge, compositions optimistes: ayant des scores acceptables mais obtenues dans des délais plus satisfaisants). Le processus de sélection prend en compte les contraintes de qualité de ressource (Quality of Resource (QoR) en Anglais) données par l'utilisateur, le matching entre les entrées/sorties des ressources liées, ainsi que l'aspect dynamique des ressources. Plusieurs tests sont réalisés pour étudier et évaluer la performance de notre solution de sélection de ressources automatique proposée, dans différentes configurations d'environnement Web, telles que la variation du nombre de ressources candidates (réalisant la même fonction requise) et la variation du nombre de fonctions nécessaires pour répondre à la demande de l'utilisateur. En outre, des analyses sont effectuées pour comparer notre modèle de qualité de ressource proposé aux travaux existants, montrant l'efficacité de notre solution dans des environnements hybrides où l'aspect de la dynamicité des ressources est considéré.

Chapitre 6

Conclusion

Le chapitre 6 conclut cette étude et présente plusieurs axes de développement et de recherche futurs que nous envisageons d'explorer par la suite,

vis à vis des limitations identifiées. Par exemple, comme axes de développement, nous visons intégrer les solutions proposées pour la composition de ressources statique/automatique dans des environnements Web réels. En fait, pour l'aspect statique, le prototype développé dans le cadre du projet SIBEX, utilise actuellement des ressources hébergées en dehors de la plateforme implémentée dans le projet (une plateforme qui fournit des ressources de collecte de données, de prétraitement, et de traitement avancé). Dans le futur, nous allons intégrer le prototype dans la plateforme pour que la composition de ressources se base sur des ressources proposées par la plateforme. De même, nous cherchons à tester les solutions que nous avons présentées concernant l'aspect automatique de la composition (découverte et sélection), qui aujourd'hui sont testées dans des environnements Web simulés orientés bâtiments, dans un contexte réel.

Ensuite, comme axes de recherche, notre objectif est d'étendre l'approche de découverte automatique de ressources en intégrant/adaptant d'autres algorithmes de graphe (outre que BFS et DFS dans ce travail) et proposer dynamiquement celui qui convient le mieux en fonction de la topologie actuelle du graphe de fonctions (le graphe qui définit les dépendances entre les fonctions fournies par les ressources), contrairement à l'approche courante où l'algorithme est choisi manuellement. De plus, nous visons à considérer la correspondance sémantique entre les fonctions requises pour réaliser la demande de l'utilisateur et les fonctions des ressources traversées dans le graphe de ressources, plutôt que de tester leur correspondance exacte, comme c'est le cas dans l'approche actuelle. Aussi, dans la solution de découverte de ressources, les nouvelles fonctions fournies par les ressources dynamiques sont liées de manière aléatoire au graphe de fonctions existant. Dans l'avenir, nous cherchons à étudier les mesures qui définissent les dépendances entre les nouvelles fonctions et les fonctions existantes. Sans oublier de mettre à jour le schéma d'indexation de manière dynamique, en évitant de le régénérer à partir de zéro à chaque fois.

Pour l'approche de la sélection, nous visons à améliorer sa performance en intégrant la vérification de l'éligibilité des ressources (s'ils respectent les contraintes de l'utilisateur) lors du processus de la découverte, et non pas durant la sélection comme dans l'état actuel de la solution. Ainsi, aucune ressource non éligible ne peut passer au processus de sélection. Nous cherchons également à lancer le processus de sélection en parallèle avec la découverte de ressources dans certains cas. Cela peut améliorer davantage les performances de la sélection en termes de temps de réponse.

Enfin, nous cherchons à assurer automatiquement la bonne interaction entre les ressources sélectionnées, constituant une composition appropriée réalisant la demande de l'utilisateur. Ainsi, le processus d'orchestration des ressources prendra rôle, dans lequel les ressources impliquées seront contrôlées par un processus central (une autre ressource).

Contents

List of Figures

List of Tables

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
BMS	**B**uilding **M**anagement **S**ystem
CPN	**C**olored **P**etri **N**et
CPU	**C**entral **P**rocessing **U**nit
CRUD	**C**reate, **R**ead, **U**pdate, **D**elete
DB	**D**ata**B**ase
EPBD	**E**nergy **P**erformance of **B**uildings **D**irective
FDD	**F**ault **D**etection and **D**iagnosis
FSM	**F**inite **S**tate **M**achine
GUI	**G**raphical **U**ser **I**nterface
HAL	**H**ypertext **A**pplication **L**anguage
HATEOAS	**H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate
HDF	**H**ierarchical **D**ata **F**ormat
HIT2GAP	**H**ighly **I**nnovative building control **T**ools **T**ackling the energy performance **GAP**
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
HVAC	**H**eating, **V**entilation and **A**ir **C**onditioning
IBM	**I**nternational **B**usiness **M**achines
ICT	**I**nformation and **C**ommunication **T**echnology
ID	**ID**entifier
IEA	**I**nternational **E**nergy **A**gency
IoT	**I**nternet of **T**hings
IT	**I**nformation and **T**echnology
JSON	**J**ava**S**cript **O**bject **N**otation
LD	**L**inking **D**ata
OWL	**W**eb **O**ntology **L**anguage
PNML	**P**etri **N**ets **M**odeling **L**anguage
RDF	**R**esource **D**escription **F**ramework
RDFa	**R**esource **D**escription **F**ramework in **A**ttributes
RDF-S	**R**esource **D**escription **F**ramework **S**chema
REST	**RE**presentational **S**tate **T**ransfer
URI	**U**niform **R**esource **I**dentifier
QoR	**Q**uality of **R**esource
QoS	**Q**uality of **S**ervice
RAM	**R**andom **A**ccess **M**emory
SBA	**S**mart **B**uilding **A**lliance
SIBEX	**S**olutions **I**ntelligentes pour le **B**âtiment en **EX**ploitation
SOA	**S**ervice **O**riented **A**rchitecture
SOAP	**S**imple **O**bject **A**ccess **P**rotocol
SA-REST	**S**emantic **A**nnotations of Web Resources
UDDI	**U**niversal **D**escription, **D**iscovery, and **I**ntegration
W3C	**W**orld **W**ide **W**eb **C**onsortium

WADL	**W**eb **A**pplication **D**escription **L**anguage
WoT	**W**eb **o**f **T**hings
WSDL	**W**eb **S**ervices **D**escription **L**anguage
XML	e**X**tensible **M**arkup **L**anguage

Chapter 1

Introduction

"If the challenge exists, so must the solution"

Rona Mlnarik

1.1 Context

1.1.1 Insight on the Web Environment: Developments and Technologies

The World Wide Web (commonly known as the Web) [28], is the leading communication model that, through HTTP[1], enables the exchange of information over the internet (the worldwide computer network). Originally designed in 1991 [26], the Web is conceived to allow users to access information, that are connected to each other by means of hypertext or hypermedia links, from any source, in a consistent and simple way. It is built around the client-server paradigm; a client is a software program that sends requests to a server, whereas a server is a software program that processes clients' requests.

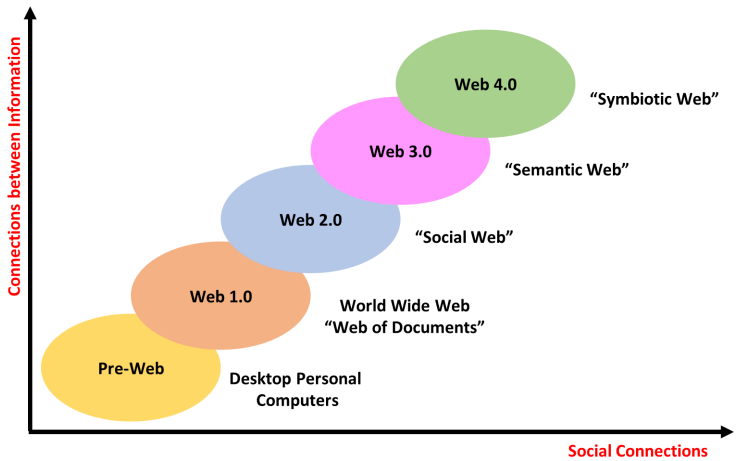


Figure 1.1 – The Web evolution

The Web has been through various phases of development, and has been marked by several generations with very short cycles. These generations, discussed more in [7], are depicted in Figure 1.1. The first implementation of the Web, representing the Web 1.0, is considered as the "read-only Web", where users were allowed to search for information and read it. There was

[1]Hypertext Transfer Protocol is a set of rules defined to transfer data between servers and clients

no option given for users to communicate back the information to the content providers. Then, the Web 2.0, referred to as the "Social Web", was built around the users, allowing them to communicate with other users, and share their perspectives, opinions, thoughts and experiences. Some of the famous Web 2.0 applications are Facebook, YouTube, and Twitter. The Web 3.0, which is known as the "Semantic Web", is the third generation of the Web in which information is given a meaning, to enable computers and people to work in cooperation [27]. This is done by representing Web content, its properties, and its relations, through machine-readable data that can be effectively used across various Web applications [35]. The Web 4.0 is the newest evolution of the Web paradigm (currently it is in its early stage), in which further sophistication and higher levels of intelligence will be added. It is known as the "Symbiotic Web", which considers that humans and machines are mutually dependent. The Web 4.0 is also characterised as the Web Operating System (OS), where information flows from any point to any other. It is associated to the Web of Things (WoT) concept [17], where users, real and virtual objects are integrated together in the Web environment forming an "always on" connected world.

In the Web' different generations, Web functions are provided via "Web services", which are software components published, found, and used on the Web. They are built on top of open standards and protocols (e.g., HTTP and XML[2]), for exchanging data between Web applications or systems. During the Web development, Web service technologies and paradigms (e.g., REST[3] and JSON[4]) have been also evolving to facilitate service implementation, description, discovery, and integration. Next, we present the most interesting ones to our work.

1.1.1.1 Web Services

Published on the Web, a Web service can be searched over the Web and invoked by a client (i.e., end-users, a Web application, a Web browser, etc.) to realize a certain set of functions. Web services encapsulate Web application functionalities to make them available through programmatic interfaces [123]. The main advantage of using Web services relies in providing a common platform that allows heterogeneous applications from different sources to have the ability to communicate with each other on the Web, realizing thus, interoperable application-to-application interactions [43].

Web services' implementation is mainly based on the SOAP[5] protocol [117] or the REST principles [45]. SOAP is a standard messaging protocol used by Web services to exchange data, whereas REST is an architectural style used to design and develop Web services. In both cases, HTTP is used to transport the data on the Web. Nevertheless, as shown in Figure 1.2[6], REST has recently become the most adopted solution for implementing Web services. This is due to several factors as: (i) its simplicity and ease of use that make

[2]Extensible Markup Language

[3]REpresentational State Transfer

[4]JavaScript Object Notation

[5]Simple Object Access Protocol

[6]Source: `https://trends.google.com/trends/explore?date=2004-01-01%202019-02-01&q=%2Fm%2F077dn,%2Fm%2F03nsxd/`

services' integration cost-effective, (ii) its support for different data formats (e.g., XML and JSON), and (iii) its ability to support caching for better performance and scalability.

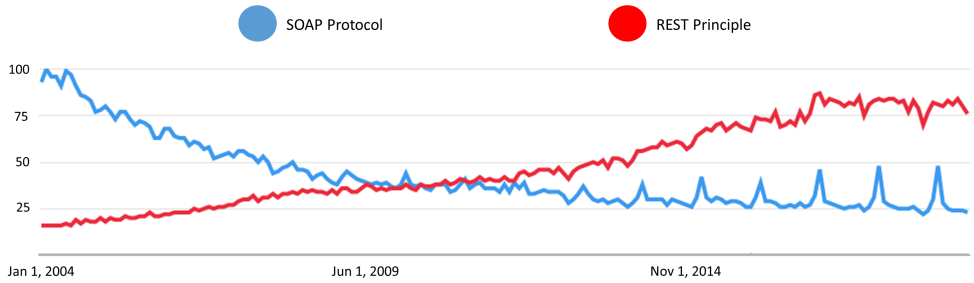


Figure 1.2 – The use of SOAP and REST from 2004 till present

Hence, more and more Web applications provide functionalities as Web services that follow the REST principles [45]. These services are referred to as "RESTful services". In the REST architecture, everything is designed as a resource that is addressable by Uniform Resource Identifier (URI). Resources take advantage of existing Web open standards, typically HTTP. As such, they can be invoked directly using HTTP methods (e.g., GET, POST, PUT, and DELETE) to realize specific requested functions, without adding overhead on the request and response messages as in SOAP. REST allows to use various data formats to represent a resource like text, JSON, and XML. JSON is the most popular one, as it allows an easy parsing and faster execution of the data. Another main advantage of using REST, is the possibility to link resources together, with support to the "Hypermedia As The Engine Of Application State" (HATEOAS) principle [108], one of the main REST constraints that we consider in this thesis. HATEOAS allows to use hypermedia links in the response contents so that the client (typically Web browsers) can dynamically navigate to the appropriate resource by traversing the hypermedia links. This is conceptually the same as a Web user navigating through Web pages by clicking on the appropriate hyperlinks in order to achieve a final goal. In this way, all future actions the client may take are discovered within resource representations returned from the server. However, and in order to facilitate resource discovery and allow an automatic resource usage, the semantic Web is used. In the semantic Web era, resources properties, capabilities, interfaces, behavior, etc., are encoded in an unambiguous and machine understandable form. This is done to create semantic resources [76], whose descriptions are annotated by machine-readable languages so that other software agents can use them without having any prior "built-in" knowledge about how to invoke them.

1.1.1.2 Web of Things

As the Web has become a major medium of communication, integrating objects (e.g., smart devices) into the Web and taking advantage of its open standards, such as HTTP and REST, has created an emerging trend: the Web of Things (WoT) [17]. The WoT was designed to enable interoperability across Internet of Things (IoT) platforms and application domains. IoT

mainly refers to the networked interconnection of smart devices, which embeds electronics, software, sensors, and communication components, enabling them to collect and exchange data [7].

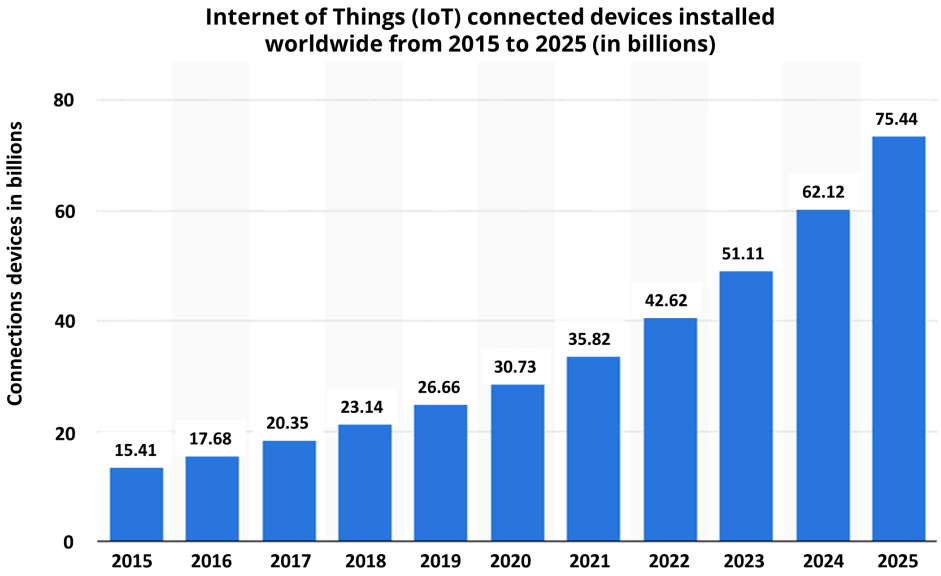


Figure 1.3 – Statistics on the number of Internet of Things (IoT) connected devices from 2015 to 2025

Recent statistics, as depicted in Figure 1.3[7], show that the total installed Internet of Things (IoT) connected devices is projected to an amount of 75.44 billion worldwide by 2025, making the world a connected place. The WoT extends the IoT by integrating devices with the existing Web infrastructure, and exposing them as Web resources. This allows IoT devices to communicate with each other, independently from their underlying implementation, and across multiple networking protocols. Thus, in the WoT, objects resources are identifiable by URIs and can be callable using HTTP methods, similar to RESTful services.

1.1.1.2.1 Smart Buildings: An Example of Web Connected Environments

Nowadays' smart buildings are a good example of Web connected environments, as they are equipped with sensors, domestic appliances, and other electronic and electric devices that can be monitored, accessed and controlled using the Web. Ensuring the communication between these Web connected devices, e.g., cameras, air filters, cooling coils, smoke detectors, heating units, pumps, fire alarm panels, chillers and boilers, etc., allows: (1) collecting data related to building light, temperature, occupancy, asset usage, etc., and (2) managing buildings' installed services, to provide as much as possible a flexible, comfortable and secure environment for the building occupants. As such, several type of systems within the building can be monitored and controlled:

- Heating, Ventilation, and Air Conditioning (HVAC) systems

[7]Source: `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-/worldwide/`

- Lighting systems
- Security and access control systems
- Fire systems
- Electric power systems
- Plumbing systems
- Piping and pumping systems
- Renewable energy sources systems used for self-consumption

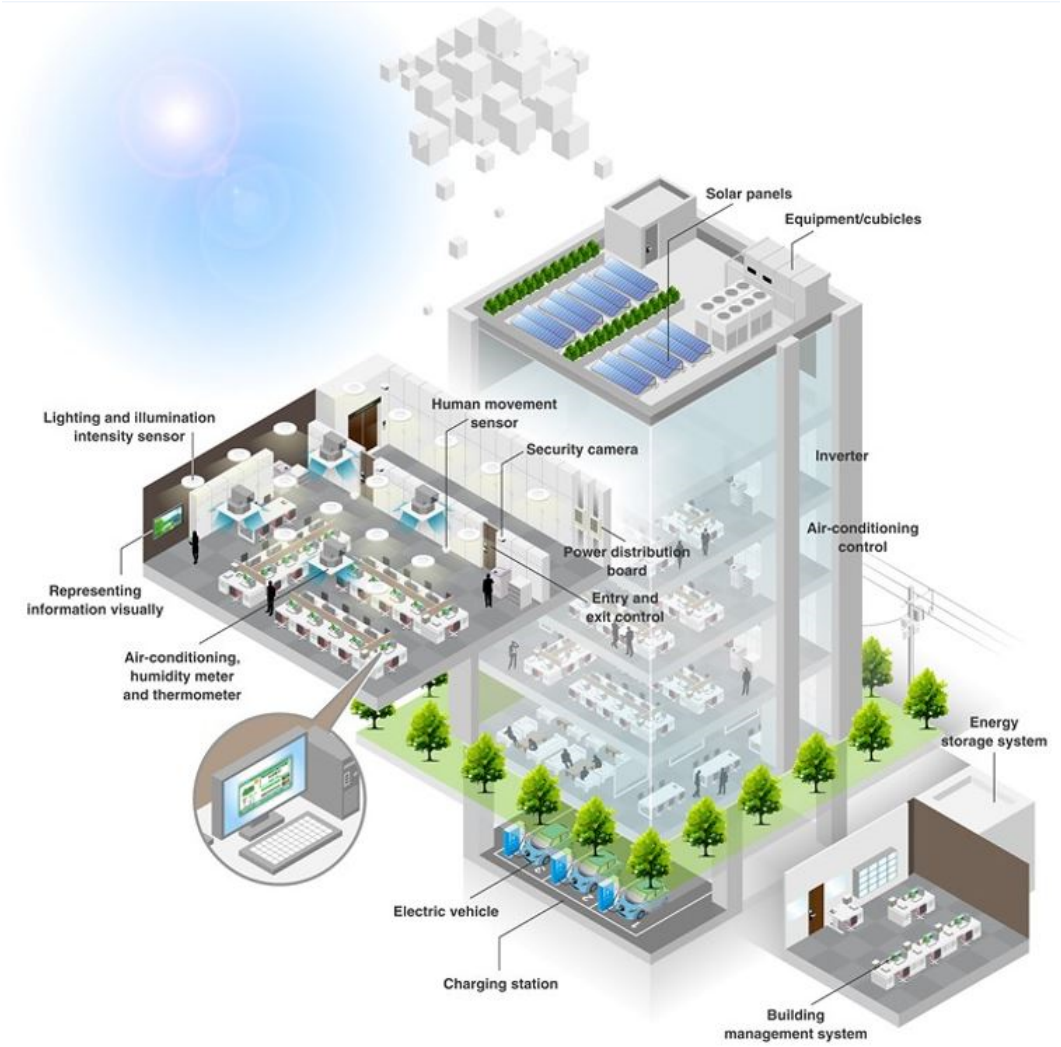


Figure 1.4 – Smart building example

Figure 1.4[8] shows an example of a smart building, in which various devices are connected to monitor and control different systems.

[8]Source: `https://tinyurl.com/yxuoqdeq/`

1.1.1.3 Service Oriented Architecture (SOA)

The emergence of Web services' developments and standards has driven major technological advances in the Web, most notably, the Service Oriented Architecture [89] (SOA). By definition, SOA is a software development model for distributed application components that are designed in the form of interoperable loosely coupled services. Services are components that can be reused for different purposes than originally intended, by a variety of applications and across different platforms.

Adopting SOA has many benefits, among them: (i) services reusability as they are loosely coupled and self-contained programs, (ii) improved manageability and scalability as services are separate components, which makes it much simpler to scale up the architecture, and (iii) increased productivity since Web application developers do not need to create every application from scratch, instead, they can adapt and reuse services to evolve. Figure 1.5[9] shows how each required business function is dependent from the application using it before SOA, and how these functions are reusable business services that can be used independently by multiple applications across different platforms with SOA.

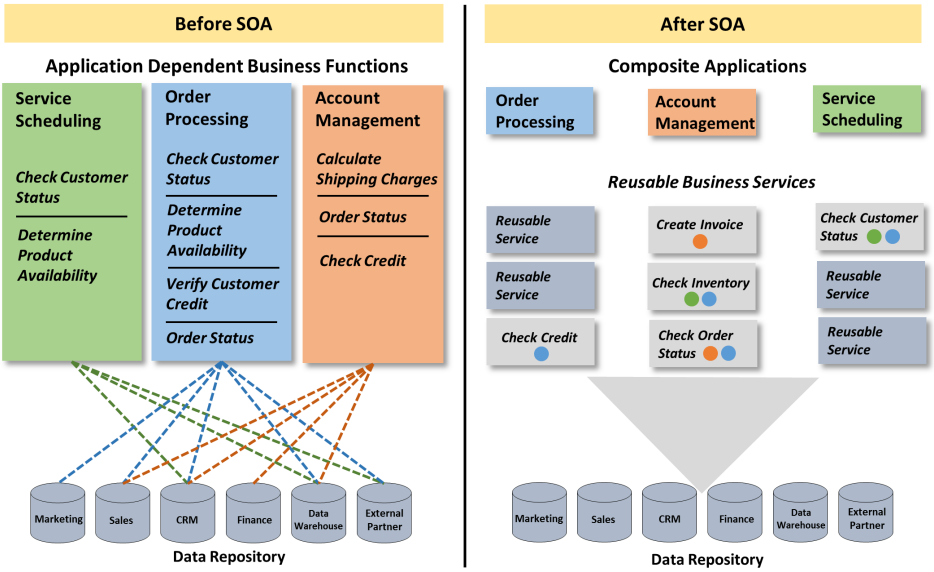


Figure 1.5 – Before SOA VS After SOA

1.1.1.3.1 SOA-based Project Examples in the Smart Buildings Domain

According to recent studies of the International Energy Agency (IEA), the building sector is responsible for 30% of global final energy consumption and 28% of total direct and indirect CO_2 emissions, as presented in Figure 1.6[10]. It is also responsible of 55% of global electricity demand[11].

Over the building life cycle phases (i.e., Design, Construction, Commissioning, and Operation), more than 80% of the energy is consumed during

[9]Source: `http://fr.logimethods.ca/enterprise-soa-soe_fr.php/`
[10]Source: `https://www.iea.org/statistics/`
[11]Source: `https://www.iea.org/buildings/`

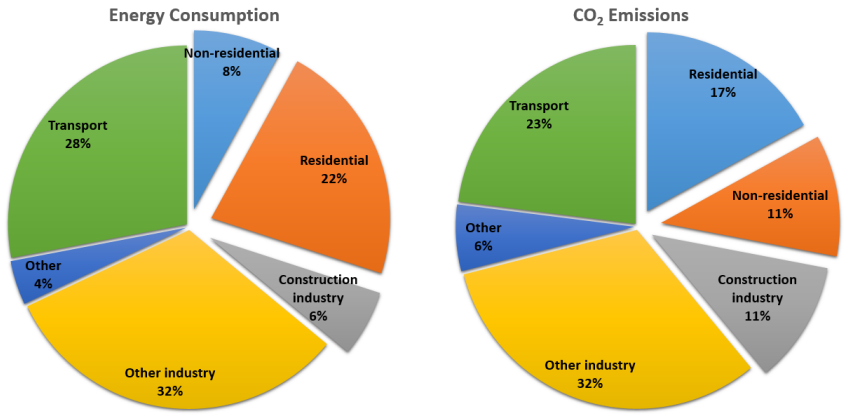


Figure 1.6 – Global final energy consumption and global energy-related CO_2 emissions by sector

the operation phase [3], while the remaining energy mainly goes to the construction and commissioning phases. This is mainly due to the HVAC, lighting and appliances' energy use. Therefore, energy and cost saving strategies addressing the building' operation phase are highly being implemented to achieve long-term energy savings. However, although the efforts taken, operational buildings are still in a need to be efficiently managed. This appears with the existence of a huge energy performance gap, i.e., the difference between the estimated and the actual energy consumption of a building. Recent measurement campaigns reveal that the actual energy use of buildings is 5 up to 10 times higher than calculations carried out during their design phase[12], creating a huge energy performance gap. This gap arises from various sources related to the building life cycle phases:

- In the design phase: design specifications are not always reliable and simulation tools can be inaccurate
- In the construction phase: equipment and materials may lack of quality, and construction methods can be inadequate
- In the commissioning phase: the installed systems are not well verified and thus, they do not operate as intended by the building owner and as designed by the building engineers
- In the operation phase: energy systems do not run properly and occupants' behavior highly impacts buildings energy consumption

With the evolution of the Information and Technology (IT) sector, a large number of solutions are being adopted to manage buildings energy behavior during their operation phase. Some are directly related to the smart building domain and act as building management solutions (BMSs) [79, 29, 78] that monitor and operate on the electrical and mechanical systems of a building: heating, ventilation, air conditioning (HVAC), lighting, etc. Other solutions are related to the business analytics domain [16, 87] and can be used to analyze the collected building related data (i.e., internal temperature/humidity, energy consumptions, energy costs, etc.). Nevertheless, these solutions face

[12] `https://www.designingbuildings.co.uk/wiki/Performance_gap_between_building_design_and\_operation/`

major obstacles when being implemented due to various factors, amongst others: (1) the non-interoperability of data between the existing building devices and the management applications used, (2) the limited analysis of data collected from the installed equipment (e.g., sensors, meters, etc.), and (3) the non-adaptability to new building requirements (i.e., adding new sensors, implementing new management applications, etc.). To cope with these limitations, NOBATEK/INEF4[13], French Institute for the Energy Transition of Buildings, has launched several projects that focus on the development of tools to support the energy optimisation for the operational phase of buildings:

1. SIBEX[14], is a Research and Development (R&D) project that aims at managing buildings energy behavior through the development of a Web-oriented platform offering several Web services used for: (1) collecting heterogeneous on-site building data (e.g., internal temperature and energy consumption), (2) pre-processing collected data (e.g., outliers correction), and (3) analyzing data (e.g., energy prediction).

2. HIT2GAP[15], is an EU-funded H2020 project that aims to control and reduce buildings energy consumption during the exploitation phase by extending the SIBEX Web energy management platform to allow the integration of several advanced building-oriented services (i.e., developed by the HIT2GAP consortium). The developed Web platform is known today as BEMServer.

Being a service-oriented Web platform, BEMServer architecture design follows the SOA model, as shown in Figure 1.7. The architecture, described more in details in [60], has been elaborated on the basis of a survey conducted on the existing solution architectures and tools used to manage energy buildings, while considering both SIBEX and HIT2GAP projects requirements.

Mainly, BEMServer architecture consists of:

- The Field level, which includes heterogeneous data collected from the building (e.g., internal temperature, energy consumptions, and other building related information such as doors and building levels), and from other sources (e.g., weather forecasts and occupants).

- The Core platform level, that contains mainly (i) the data repository to store time-serie collected data, (ii) an ontology-based data model [98], to store semantically linked descriptive information, (iii) a set of basic services (i.e., data pre-processing services used to prepare and correct the data collected from the Field level), and (iv) Web APIs (Application Programming Interface) implemented as RESTful services, through which advanced services access the data collection and basic services.

- The Management level, that contains the advanced services used to analyze and process building data in advanced manner (e.g., using data

[13] `https://www.nobatek.inef4.com/`

[14] Solutions Intelligentes pour le Bâtiment en EXploitation: http://spider.sigappfr.org/research-projects/sibex/

[15] Highly Innovative Building Control Tools Tackling the Energy Performance Gap: http://www.hit2gap.eu/

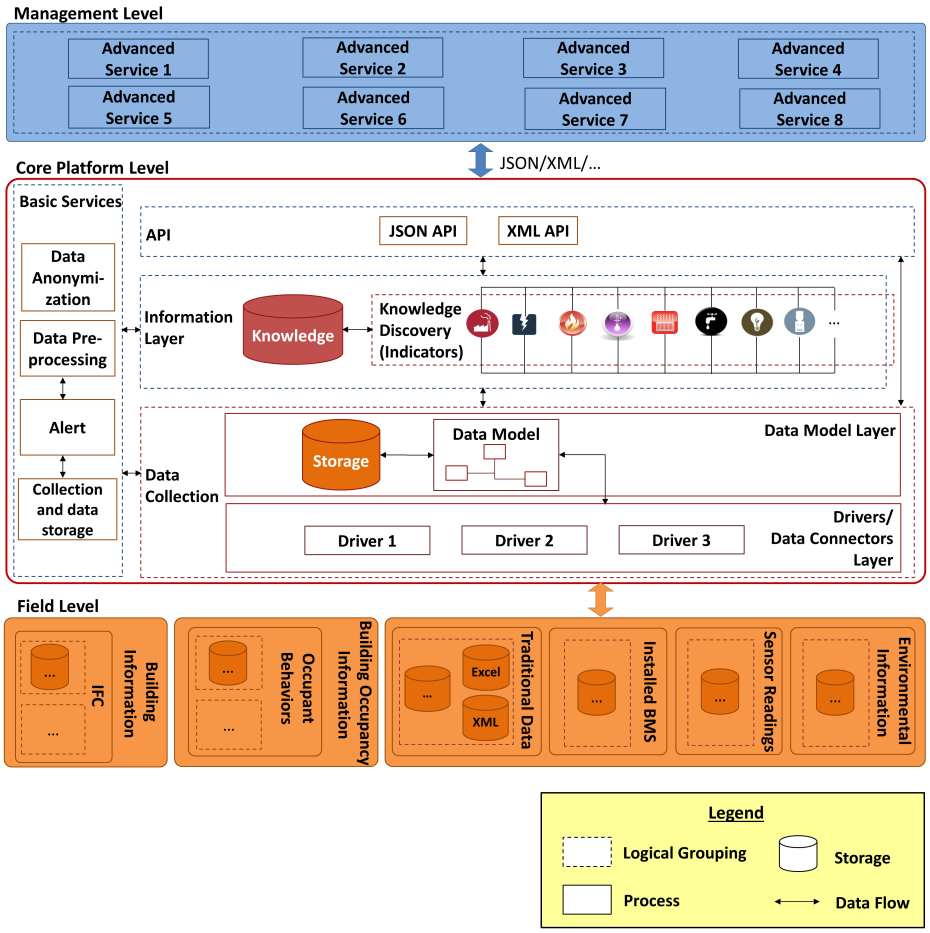


Figure 1.7 – BEMServer Architecture

mining and machine learning algorithms [118]). Using the Web APIs, the advanced services can access the collected data stored in the platform and the basic services to prepare the data before being processed. Examples of such advanced services in the HIT2GAP project are:

- Forecasting, used to predict building energy consumption and anticipate reactive behavior.
- Fault Detection and Diagnosis (FDD), used for detecting faulty system operations based on advanced monitoring capabilities. It generates warnings and recommendations for the facility management.

The services provided by the BEMServer platform mainly follow the REST architecture style, thus, they are designed as resources identified through URIs and callable using HTTP methods. A resource can offer a set of functions that answer specific requests of building actors (e.g., building manager and the building energy manager). However, buildings actors may have complex demands that require the interaction of two or more resources (data collection, data pre-processing, and/or advanced data processing), forming a composition of resources [48]. In order to fulfil such demands, we propose in this thesis, solutions for RESTful service composition, that can be applicable in the smart buildings environments, to help building actors in creating

new resources for managing and controlling more efficiently buildings energy behavior.

1.1.2 Thesis Scope

1.1.2.1 Collaboration

This thesis was held under a French CIFRE convention (Industrial Convention of Formation by Research) between: (1) NOBATEK/INEF4, an Institute for Energy and Environmental Transition in the Construction industry (France), and (2) the Computer Science Laboratory of the University of Pau & Pays Adour (France). The work has received funding from the National Association of Research and Technology in France (ANRT[16]), for the SIBEX project (tightly related to this thesis), previously described in Section 1.1.1.3.1.

1.1.2.2 Objectives

Web Service composition has become the most promising way to support business-to-business application integration [48]. It refers to the combination of two or more services to offer new and value-added services responding to complex user' requests. The aim of this thesis is to propose generic solutions for RESTful service (resource) composition, that can be applicable in the smart buildings Web connected environments (see Section 1.1.1.2.1), providing resources exposed by Web applications and connected objects. The resource composition allows buildings actors (e.g., the building manager and the building energy manager) to create new resources by combining several resources together, in order to answer specific complex demands for managing buildings energy behavior. In our work, we consider two aspects of the composition [48]: (1) static, in which the resources to be used are manually chosen and linked together by the user before being deployed, and (2) automatic, where the composed resources are automatically created on the fly (i.e., resources are automatically discovered and selected to form a suitable composition for user's request) based on some user inputs. More specifically, the main objectives of this thesis can be presented as follows:

1. **Definition of a SOA-based architecture for buildings energy management:** Elaborating the architecture of the Web-based energy management platform, BEMServer (see Section 1.1.1.3.1), which provides multiple building-oriented services (resources), was part of our preliminary tasks in this thesis. In the proposed contributions related to resource composition (static and automatic), we used several case studies identified within the context of BEMServer, since it is a resource-oriented platform. Thus, the motivation behind our solutions have been illustrated in scenarios related to the building energy management domain.

2. **Verification of static resource composition:** When the composed resources are manually built, i.e., the involved resources are selected and linked together by the user, several errors can occur during the composition design (e.g., non-interoperability of data types between the linked

[16] Association Nationale de la Recherche et de la Technologie

resources, and end-loops occurred preventing other resources to run). This leads to an erroneous composition behavior. To avoid such behavior that provokes inaccurate results, and to prevent unnecessary execution of erroneous composition, we aim in this thesis to verify the composition behavior before being executed, while considering the REST principles.

3. **Automatic resource composition:** In order to facilitate the composition process for the user, especially with the existence of numerous and overlapping resources providing similar functions, and allow composing resources in hybrid Web environments providing static resources (i.e, always available on the Web) and dynamic resources (i.e., connected to and removed from the Web at different instances), an automatic composition approach becomes necessary. In this context, the purpose of this thesis is to:

   - Discover automatically the resources providing the functions required to answer user's requests;
   - Select automatically the suitable resources from the identified ones during resource discovery, to form the required compositions.

   To allow such automatic approaches, it is important that resources properties (provided functions, inputs/outputs, etc.), are machine-readable data. This is done through the use of a descriptive language (Hydra vocabulary in this work) that allows annotating resource properties with semantic data processable by machines.

1.2 Motivating Scenario and Research Challenges

In this section, we motivate our work focused on RESTful service composition, with a scenario illustrated in the BEMServer Web platform (see Section 1.1.1.3.1). Mainly, the platform provides static resources (i.e., established to be always available) for: collecting heterogeneous on-site building data, pre-processing the collected data, and analyzing the data. We suppose that the static resources follow the HATEOAS principle (one of the main constraints of the REST architecture style that we seek to adopt in this thesis), thus, they are linked together based on their provided functions defined in a function graph. The links between the static resources are included in each resource description, which is expressed in Hydra [64] and registered in a triplestore-based repository. Each resource provides a specific function that can be called using HTTP methods (e.g., GET, POST, PUT, and DELETE) to satisfy a building actor request. However, some requests require the combination of several resources forming a composition. We consider the case of a building manager (with little skills in both technical and energy domain aspects) who needs to predict the heating energy consumption of a specific zone of his building (e.g., his office), for the upcoming week. The resulted consumption can help him to anticipate the building' energy resource needs. In order to satisfy the building manager demand, several resources are required, as depicted in Figure 1.8:

- Data collection resources to collect the required data: Air Temperature (i.e., internal temperature) extracted from the BEMServer repository, and climate temperature (i.e., external temperature) provided by an external weather forecast resource).
- Data pre-processing resources to clean the collected data from the external weather forecast resource:
  - Resources for detecting and correcting outliers values, which are data values outside the range of most of the other values
  - Resources for detecting and correcting empty or missing values retrieved during certain timestamps
- A resource for predicting energy demand using the collected and prepared data. This resource relies on a prediction model that is considered already implemented in our work.

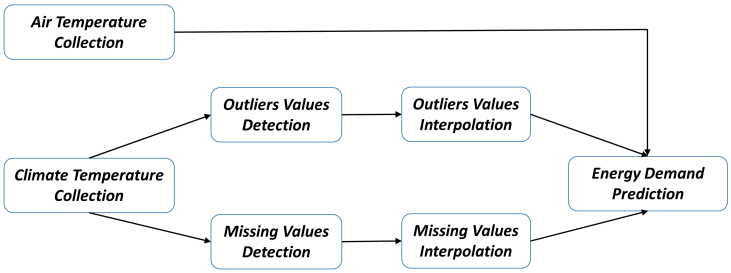


Figure 1.8 – Resource composition scenario

To realize the necessary composition scenario involving several resources, two cases are possible:

- **Case 1: Static resource composition** - In this case, the building manager is responsible to choose and select the required resources from a list of the provided static resources and link them manually.
- **Case 2: Automatic resource composition** - In this case, we tackle the resource composition process, in more complex Web environments by considering the dynamicity and mobility aspect of resources (whenever they are exposed by connected objects). As such, we assume that BEMServer is extended to be an open and dynamic environment allowing ad-hoc connection of external resources exposed by stationary/mobile objects (e.g., mobile phones and tablets) at runtime, as shown in Figure 1.9. These resources are dynamic (i.e., can be connected to and removed from the environment at different instances). In order to facilitate the composition task for the building manager in such hybrid environment providing both static and dynamic resources, an automatic composition process, in which resource properties are semantically described, is required. In the process, the building manager is only required to provide some inputs (e.g., required function (energy demand prediction), startdate/enddate, and the necessary location).

However, several challenges arise when dealing with each of the above two cases:

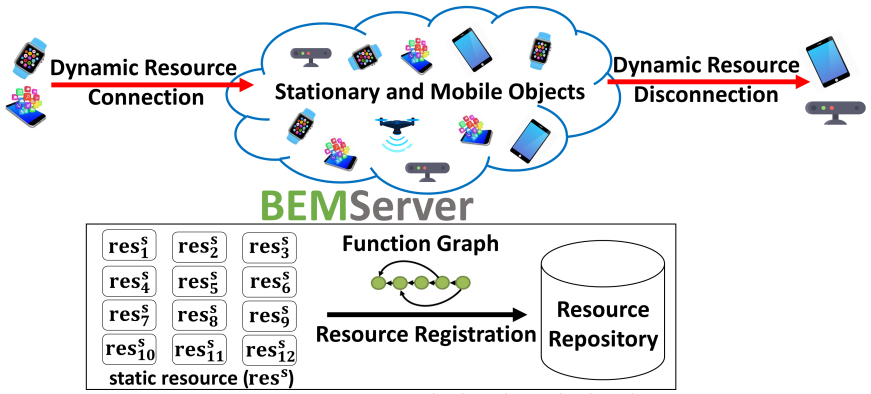


Figure 1.9 – BEMServer extended to be a hybrid environment

1. **For the static resource composition:**

   - **Verification of the composition behavior:** As the composition is built manually by the building manager, several problems may occur during the composition execution. For example, he may link the "Missing Values Interpolation" resource illustrated in Figure 1.8, to the "Outliers Values Interpolation" resource, causing an end-loop in the composition and preventing other resources (i.e., the "Energy Demand Prediction" resource) to run. In order to cope with such design problems, it is important to verify the correctness of a composition behavior before being executed.

2. **For the automatic resource composition:**

   - **Automatically identify resources in hybrid Web environments:** The existence of numerous published resources in a Web-based environment as the BEMServer platform, and the dynamic nature of part of the connected resources, which are not linked to the existing static resources, make the automatic discovery of resources realizing building manager request a challenging task. Also, allowing an efficient resource discovery with an acceptable response time in an environment connecting large number of resources, makes resource identification even more complex. Additionally, and in order to ensure accurate prediction results, it is essential to consider the location of the data collection resources exposed by objects (as the connected tablets and mobile-phones in our scenario) during resource identification. For instance, a resource that collects temperature data whose location is different than the required prediction zone (e.g., the building manager office) will not be efficient to the building manager demand. However, considering object locations is a critical task that requires processing spatial queries, e.g., Range type [18] to identify objects in a specific region and KNN [67] to locate the K nearest neighbors (objects).
   - **Automatically select the resources forming suitable compositions in hybrid Web environments:** The discovery of several resources

providing similar functions required for the building manager demand, makes it difficult for him to select the appropriate ones forming suitable composition matching his needs. As such, a resource may be better than others since it may have: (i) continuous connectivity to the environment (if it is static), (ii) cost free when using it, and (iii) high usage rate (i.e., it has been invoked many times). Moreover, the selection becomes more complex when dealing with dynamic resources that are unavailable for execution (after being selected). Therefore, and in order to prevent missing dynamic resource in a composition, other compositions with different implementation alternatives realizing user's request are necessary. In this context, the building manager may need to have different types of compositions (e.g., optimal compositions having the highest scores in terms of Cost, Availability, etc., and optimistic compositions with acceptable scores but obtained in more satisfactory delays). Therefore, considering user's needs is also important to answer more efficiently his demands.

To tackle the aforementioned challenges, we propose in this thesis three main contributions for: (1) verifying the behavior of statically composed resources, (2) discovering automatically the required resources in hybrid Web environments, and (3) selecting automatically the appropriate resources identified in resource discovery to form the necessary compositions. Each of the proposed solutions is detailed next in Section 1.3.

1.3 Proposal: StARC Framework

In this thesis, we present StARC, a framework for Static and Automatic Resource Composition. The framework is generic, as it can be applicable in different Web-based environment domains. It allows to compose statically and automatically resources that follow the REST principles, including HATEOAS. As shown in Figure 1.10, StARC covers 2 aspects of resource composition:

- **Static Resource Composition -** In this case, the resource composition is manually created by the user, based on a resource' descriptions guide that provides data on the available static resources properties (e.g., providing functions, inputs, outputs, etc.). In order to verify the composition behavior, it is passed to the Verification process for validation. In the latter, the composition model (i.e., expressed in JSON in this work) is transformed by the Modeling Engine into a formal language based on Colored Petri Nets (CPN)[17], i.e., PNML (XML-based syntax for high-level Petri nets). The transformation is done using the Resource CPN Mapper, which includes a defined model that maps the resources and their composition to CPN. Based on CPN, the Validation Engine allows to verify the resource composition behavior using CPN formal properties (i.e., Reachability, Liveness, and Interoperability). After the composition is validated, the Conversion process is launched. In the process, the composition is converted by the Conversion Engine into a suitable

[17] A graphical oriented language for design, specification, simulation and verification of systems

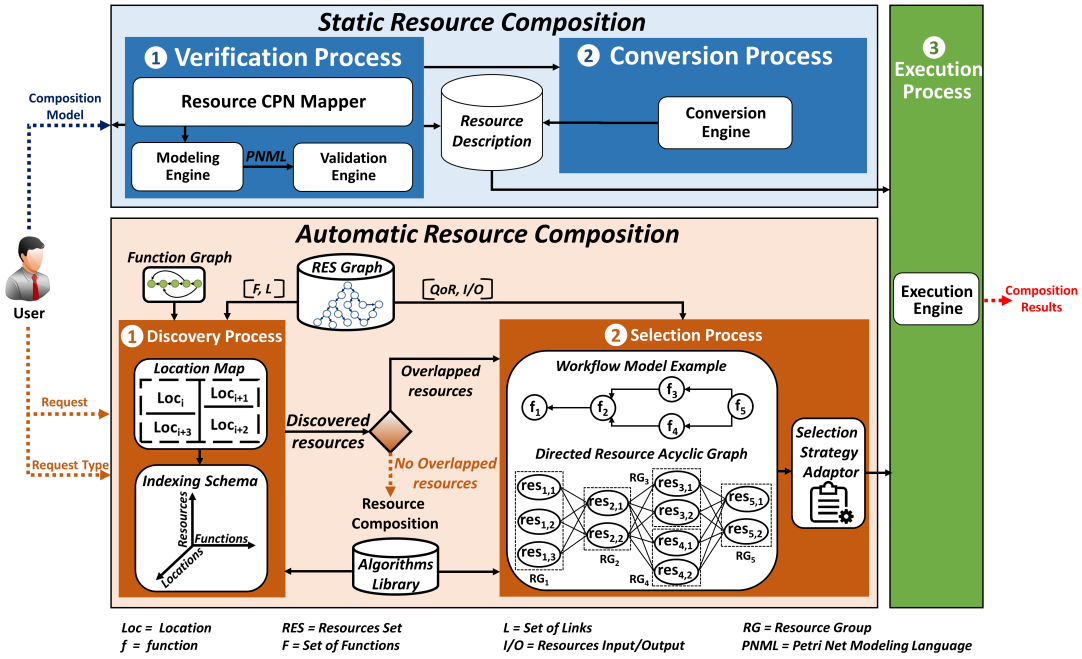


Figure 1.10 – An overview of the proposed framework: StARC

format that can be stored into the Resource Description repository (i.e., expressed in the Hydra Vocabulary [64] in this thesis), so it can be invoked and executed later by the Execution Engine of the Execution process.

- **Automatic Resource Composition -** It includes 3 processes: Automatic Discovery, Automatic Selection, and Execution. The discovery process aims at identifying the resources (static and dynamic) satisfying the functions needed to answer the user's requested function included within its request. These functions are presented in a directed acyclic Function Graph (FG) that defines their dependencies. To identity the necessary resources, the discovery process uses RES Graph, a resource graph where static and dynamic resources are linked together, based on their providing functions defined in FG. An Indexing Schema, that maps resources to their provided functions and location (i.e., defined in a Location Map and assigned to resources that are exposed by objects as smart devices) is defined. The schema allows to identify the resources from which the discovery process (using a graph-based algorithm stored in the Algorithms Library) starts crawling the RES graph to identify the resources providing the functions required for user's request. If no overlapped resources are discovered for the same required functions, one resource composition is returned. However, when there are several candidate resources identified during resource discovery, for at least one function necessary to realize user's request, the selection process is launched. In the latter, a directed acyclic graph that links the discovered resources according to the Workflow Model required for user's request is formed. Based on the built graph, and using a graph-based algorithm, the selection process consists on selecting the suitable resources to form different composition alternatives, i.e., i-compositions (with i $\in \mathbb{N}^*$), answering user's request. With the existence of many possible

compositions, the selection process uses a Selection Strategy Adaptor that forms different compositions types responding to different user's needs expressed in user's request type (e.g., optimal compositions having the highest scores, optimistic compositions having acceptable scores but obtained in satisfactory delays, etc.). This is done while considering user Quality of Resource (QoR) constraints (e.g., Cost and Availability), resources Input/Output (I/O) semantic matching, as well as resource dynamicity. Once the compositions are formed, they can be executed by the Execution process to obtain the required results. However, in this thesis, the execution of the automatically composed resources is not covered. It will be held in a subsequent work.

1.3.1 Contributions and Publications

In this section, we present the key contributions previously identified in the StARC framework. The contributions concern the verification process in the static resource composition, and both resource discovery and selection processes, in the automatic resource composition applicable in hybrid Web environments (i.e., connecting static and dynamic resources).

1.3.1.1 Verification of Static Resource Composition

For the static resource composition in the StARC framework, our contribution is focused on the composition verification process. As such, and in order to allow the verification of a static resource composition behavior before execution, the key contribution is the formal CPN-based Model.

- **Formal CPN-based Model:** It is defined to represent formally the behavior of static resources and their composition using Colored Petri Nets (CPN). By mapping resources to CPN, the model allows to use CPN verification tools properties to verify relevant composition behavior: (1) Reachability, to make sure that the desired results can be reached from the initial composition state, (2) Liveness, to verify that all involved resources can be invoked during composition execution, and (3) Interoperability, to check if the linked resources are compatible according the their related Input/Output data types.

The composition verification contribution is published in the proceedings of the 25^{th} International Conference on Cooperative Information Systems (CoopIS):

- *KALLAB, Lara, MRISSA, Michael, CHBEIR, Richard, et al. Using colored petri nets for verifying restful service composition. In : OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". Springer, Cham, 2017. p. 505-523.*

1.3.1.2 Automatic Location-aware Resource Discovery

As part of the automatic resource composition in the StARC framework, we propose an automatic location-aware resource discovery that can be applicable in hybrid Web environments connecting linked static resources supporting the HATEOAS principle, and dynamic resources. Within the solution, several key contributions have been presented:

- **Resource Graph Linking Static and Dynamic Resources:** It is one of the major key contributions related to the automatic resource discovery process. It allows to link dynamic resources (i.e., connected to and removed from the Web environment at different instances) to existing static linked resources (i.e., established to be always available on the Web environment) that follow the HATEOAS principle. This is done by using defined virtual resources; one for each provided function in the environment defined in a directed acylic function graph. The virtual resources hold the connected dynamic resources answering the same corresponding function, and are linked to static resources providing that same function.

- **Indexing Schema:** It is defined to map the available resources in the Web environment, to their providing functions and location (whenever they are exposed by objects as smart devices). The indexing schema is used to identify the resources required for user's request while considering their location. This is important mostly for the discovery of data collection resources, as it enables to have accurate data necessary for user's request. Also, the indexing schema allows to make resource discovery faster, especially in large Web environments. This is done by pointing to the necessary resources (providing the required functions for user's demand) from which the resource discovery process will start its search, instead of crawling resource graph from the root resources.

- **Resource Discovery Process:** It uses several graph-based algorithms, as Breadth First Search (BFS) and Depth First Search (DFS) [97] in this thesis, to traverse resource graph (including static and dynamic resources) and identify the resources required to answer user's request. The different implemented algorithms, are adapted to explore resource descriptions (expressed in Hydra vocabulary in our work) enriched by semantic annotations.

The contribution related to the automatic resource discovery is published in the proceedings of the IEEE International Conference on Web Services (ICWS):

- *KALLAB, Lara, CHBEIR, Richard, et MRISSA, Michael. Automatic K-Resources Discovery for Hybrid Web Connected Environments. In : 2019 IEEE International Conference on Web Services (ICWS). IEEE, 2019. p. 146-153.*

1.3.1.3 Automatic QoR-based Resource Selection

For the selection process of the automatic resource composition in the StARC framework, we present an approach for selecting the appropriate resources (static and/or dynamic) among other candidates, to form the required resource compositions responding to user's needs. The key contribution behind our selection approach is the Selection Strategy Adaptor.

- **Selection Strategy Adaptor:** Using a formal model graph that relates the identified resources during resource discovery, and based on several defined rules, the Selection Strategy Adaptor allows forming different

compositions alternatives answering user's request (e.g., Optimal compositions having the highest scores, Optimistic compositions having acceptable scores, i.e., $\geq$ a defined threshold, but obtained in satisfactory delays, etc.). This is done while considering QoR constraints, I/O semantic matching of related resources, and resource dynamicity aspect.

This is an ongoing work that will be submitted soon.

In addition to the aforementioned scientific contributions, we defined the SOA-based architecture of the BEMServer platform used for managing building energy behavior in the context of SIBEX and HIT2GAP projects. The architecture is published in the Energy Procedia proceedings of the International Scientific Conference related to Climate Resilient Cities, Energy Efficiency, and Renewables in the Digital Era (CISBAT):

- *KALLAB, Lara, CHBEIR, Richard, BOURREAU, Pierre, et al. HIT2GAP: Towards a better building energy management. Energy Procedia, 2017, vol. 122, p. 895 - 900.*

Also, we mention below other publications in which we included part of our work related to Web service composition in the buildings energy domain:

- *CHBEIR, Richard, CARDINALE, Yudith, CORCHERO, Aitor, et al. OntoH2G: A Semantic Model to Represent Building Infrastructure and Occupant Interactions. In : International Conference on Sustainability in Energy and Buildings. Springer, Cham, 2018. p. 148 - 158.*
- *BOURREAU, Pierre, CHBEIR, Richard, CARDINALE, Yudith, et al., "BEMServer: An Open Source Platform for Building Energy Performance Management", presented at the 2019 European Conference on Computing in Construction, 2019, p. 256 - 264.*

1.4 Report Organization

The rest of this report is organized as follows.

Chapter 2 gives some background information for the full understanding of the different approaches proposed in this thesis. It presents: (i) the concept of Web services, (ii) the main technologies used to implement them with emphasis on the REST architecture style (the one adopted in our work), and (iii) the different semantic Web languages used to make Web services properties machine-readable. Then, the chapter provides a review on the existing languages that allow resource descriptions, with a focus on hypermedia-driven approaches as Hydra vocabulary, the one used to describe the resources in this thesis.

Chapter 3 tackles the challenge of verifying a resource composition behavior before execution. For this aim, we propose a formal model that maps the behavior of resources with their composition to Colored Petri Nets (CPN). Using the defined CPN-based model, we show how the verification of a resource composition behavior can be done by applying several CPN behavioral properties (i.e., Interoperability, to check whether the linked resources

are compatible according the their related Input/Output datatypes, Reachability, to ensure that the final desired state is reachable, and Liveness, to ensure that all resources can be executed during composition execution). The work has been tested using CPN tools to verify the applicability of our approach. The chapter presents also a standalone prototype that has been developed within the context of the SIBEX project to verify building-oriented resource compositions. The prototype includes different engines for modeling, validating, converting, storing and executing new composed resources. Several tests have been conducted to test the different functionalities of the developed prototype, including the validation process that is based on our defined CPN model.

Chapter 4 tackles the challenge of the automatic resource discovery in hybrid Web environments connecting: (1) static resources that are established to be always available and follow the HATEOAS principle, and (2) dynamic resources, which can be connected to and removed from the environment at different instances. In the chapter, we propose a formal model representation that links resources (i.e., dynamic and static) in one single resource graph. The resource graph can be traversed by several graph-based algorithms (i.e., BFS and DFS in this thesis) to discover the resources realizing the required functions for user's request. The graph-based algorithms are adapted to follow the semantic annotations integrated in the resource descriptions (expressed with Hydra vocabulary in our work) of the traversed resources, for resource discovery. In this chapter, we also define an original 3-dimensional indexing schema that maps the resources to their provided functions and location (whenever they are exposed by objects as smart devices). Such indexing schema allows the identification of data collection resources based on their location. It also enhances resource search in large Web environments by pointing to the resources from which the crawling of the resource graph will start, instead of starting from the root graph resources. Several tests have been conducted to evaluate our solution performance in different environment setups (e.g., varying the number of resources and the number of functions required for user's request), and on 4 aspects: dynamicity, multiplicity, efficiency, and scalability. The results show the utility of using the indexing schema to enhance resource discovery response time, especially in large Web environments.

Chapter 5 tackles the challenge of the automatic resource selection in hybrid Web environments. In the solution, we first propose a formal model that links the identified resources during the discovery process in a directed acyclic graph, based on their providing functions. Then, we define a Selection Strategy Adaptor that allows resource selection to form several alternative compositions having different types answering user different needs (e.g., optimal compositions having the highest scores, optimistic compositions having acceptable scores but obtained in more satisfactory delays, etc.). During resource selection, user QoR constraints, resource I/O semantic matching and dynamicity (whenever it is required) are considered. Several tests are conducted to study the performance of our proposed solution in different environment setups (e.g., varying the number of resource candidates and

the number of required functions answering user's request), and analysis are made to compare our QoR model with existing works.

Chapter 6 concludes this study and presents several future directions that we are planning to explore afterwards, on the basis of the limitations identified in our work.

Chapter 2

Background

"The building is only as tall as the foundation is strong enough to build on"

Paula White

Before elaborating on our main contributions related to both static and automatic resource composition presented in the StARC framework: (1) verification of static resource compositions, (2) automatic resource discovery, and (3) automatic resource selection, we present in this chapter several important technological concepts in order to fully understand the proposed solutions. The chapter gives preliminaries on Web services, i.e., their definition, the most supported protocol/principles used to implement them with emphasis on the REST architecture style (the one adopted in this thesis), and the known semantic Web languages used to make their properties (provided functions, inputs/outputs, etc.) machine-readable. Also, the chapter provides a review on the existing languages used for describing RESTful services (resources) by focusing on the hypermedia-driven languages as the Hydra vocabulary (the one used to describe the resources in our work).

2.1 Web Services: Technology and Semantics

According to the World Wide Web Consortium (W3C)[1]: "A Web service is a software system identified by a URI and designed to support interoperable machine-to-machine interaction over a network. It has an interface defined and described in a machine-processable format. Its definition can be discovered by other software systems. Other systems may then interact with the Web service in a manner prescribed by its description". In essence, Web services are self-describing and loosely coupled application components developed using any programming language, and on any platform. They expose business logic through Application Programming Interfaces (APIs), which can be published, discovered, and invoked over the Web. Mainly, there are two types of Web services: (1) SOAP-based services, that use the SOAP protocol, and (2) REST-based services, that follow the REST architecture style principles. Both types are described in the following sections.

2.1.1 SOAP-based Services

SOAP (Simple Object Access Protocol) is an XML-based messaging protocol for exchanging structured information in the implementation of Web services in computer networks [117]. It has been introduced two decades ago and has been popular for about ten years, before beginning to be less used with the huge evolution of REST. Briefly, a SOAP message is an ordinary XML document containing the following main elements:

- An Envelope element that identifies the XML document as a SOAP message. It defines the start and the end of the message.
- A Header element that contains any optional attributes used in the processing of the message.
- A Body element that contains the XML data comprising the message being sent.
- A Fault element providing information about errors that occur while processing the message.

In the example below, a "GetNumberFloors" request is sent to a SOAP server over HTTP. The request, shown in Listing 2.1, has a building name parameter, and a number of floors parameter that is returned in the response, presented in Listing 2.2.
The namespace of the function is defined in "http://www.example.org/building".

```
1 POST /building HTTP/1.1
2 Host: www.example.org
3 Content-Type: application/soap+xml; charset=utf-8
4 Content-Length: nnn
5
6 <?xml version="1.0"?>
7 <soap:Envelope
8 xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
9 soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
```

[1] https://www.w3.org/TR/ws-arch/

```
<soap:Body xmlns:m="http://www.example.org/building">
  <m:GetNumberFloors>
    <m:BuildingName>buildingA</m:BuildingName>
  </m:GetNumberFloors >
</soap:Body>
</soap:Envelope>
```

Listing 2.1 – Example of a SOAP request

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
<soap:Body xmlns:m="http://www.example.org/building">
  <m:GetNumberFloorsResponse>
    <m:NumberF>7</m:NumberF>
  </m:GetNumberFloorsResponse>
</soap:Body>
</soap:Envelope>
```

Listing 2.2 – Example of a SOAP response

SOAP offers basic communication for Web services, but it does not provide information about what messages must be exchanged to successfully interact with a service. That role is filled by WSDL (Web Services Description Language) [36], an XML format developed by IBM[2] (International Business Machines) and Microsoft to describe Web services as collections of communication end points that can exchange certain messages. Mainly, a WSDL document describes a Web service's interface: what the Web service does (what operations it offers and what messages need to be exchanged), how to use it (what protocols and data encoding systems it uses), and where it is located (its access point).

WSDL service descriptions are registered in the Universal Description, Discovery, and Integration (UDDI) registry [31][3]. UDDI provides a mechanism that can be used to find a Web service that meets user requirements and to find information about how to use the service, usually specified in a WSDL document. Thus, these three technologies (SOAP, UDDI, and WSDL) are seen as the core of what most people view as the standard Web services infrastructure, as it is shown in Figure 2.1.

SOAP has several advantages [117], among them:

- It is not tied to any transfer protocol. It can be transferred via SMTP[4], FTP[5], HTTP (which is the most popular transfer protocol that SOAP uses), etc.
- It supports WS-Security, which adds enhancements to SOAP messaging to provide quality of protection through message integrity, message

[2] `https://www.ibm.com/`

[3] Source: `http://www.wst.univie.ac.at/workgroups/sem-nessi/index.php?t=semanticweb/`

[4] Simple Mail Transfer Protocol

[5] File Transfer Protocol

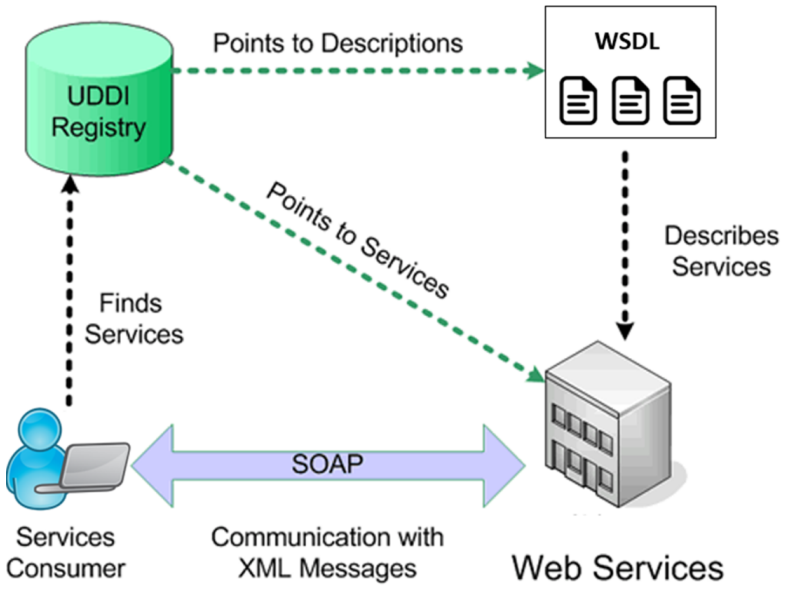


Figure 2.1 – SOAP, UDDI, and WSDL interactions

confidentiality, and single message authentication. WS-Security mechanisms can be used to accommodate a wide variety of security models and encryption technologies.

- It supports WS-ReliableMessaging: SOAP has successful/retry logic built in and provides end-to-end reliability even through SOAP intermediaries.
- It has a built error-handling. As such, if a problem occurs in the request, the response will contain error information that can be used to fix the problem.

Despite these advantages, many developers found SOAP complex and hard to use. In addition to being dependent from XML, and other XML-based standards as WSDL, working with SOAP requires writing a lot of code to perform sometimes simple tasks because it is necessary to have the required XML structure in every task. As an alternative, REST architecture style has emerged [91], and has been gaining popularity since the last decade when designing and implementing Web services.

2.1.2 REST-based Services

REST architecture style [45] has recently become a popular choice for implementing Web Services. A Web service that conforms to all REST principles is referred to as a "RESTful service". Mainly, a RESTful service provides a functionality through an abstract resource-oriented view identified by a Unique Resource Identifier (URI), and invoked via HTTP-based methods (e.g., GET, POST, PUT and DELETE). Several architectural principles [46] are defined to design and implement RESTful services:

- **Resource-oriented and addressability**: Resources are central elements addressable through their URI. They can be defined as objects with different format representation (e.g., JSON, XML, etc.), associated data

(e.g., text files, images, etc.), relationships to other resources, and a set of methods that operate on it (e.g., GET, POST, PUT and DELETE).

- **Uniform interface**: The interactions with the exposed resources are made via a uniform interface, which provides a set of standard operations supported by the HTTP protocol. The consequence of the execution of these methods is the change of the resource state that can be transferred from/to clients, and represented in various types (e.g., JSON, XML, etc.). Each method has a well-defined semantics in terms of its effect on the state of the resource. It can be: (1) idempotent, i.e., it produces the same results when executed once or multiple times, or (2) safe, i.e., it does not modify the resource on the server side. The main methods are:
  - GET: The GET verb is idempotent and safe. It is used to retrieve information. For example, retrieving a building with an ID of "buildingA" would be: GET /buildings/buildingA
  - POST: The POST verb is used to send data to the server to create new resources. In particular, it is used to create subordinate resources, i.e., subordinate to some other resource (e.g., parent). In other words, when creating a new resource, the server associates the new resource to the parent and assigns to it an ID (new URI). POST is not idempotent and not safe. As such, making two identical POST requests will result in two resources containing the same information, thus, producing different resource states on the server side. As an example of POST, in order to add a floor to the building A, the request would be: POST /buildingA/floors.
  - PUT: A PUT request is idempotent, but not safe. It is mainly used to update existing resources, but in some cases, when the client knows the URI of the resource to create, it can also be used to create new resources. As an update example, to modify a building with an id of "buildingA", the request would be: PUT /buildings/buildingA
  - DELETE: It is idempotent but not safe. It is used to delete a resource having a specific URI. If a resource is deleted, it is removed. To delete a resource with an id of "buildingA", the request would be: DELETE /buildings/buildingA
- **Client-Server model**: This essentially means that client applications and server applications are able to evolve separately, and independently. As such, a client is not concerned with the data storage and business logic, which remain internal to each server, and a server is not concerned with the user interface or user state.
- **Stateless communication**: Every interaction with a resource is stateless. This means that the server does not store any state about the client session on the server side. In fact, each request is handled independently from the other, and request messages must contain all the necessary information that the server needs to process it. From another side, the response messages received from the server should also contain data

related to the response state (e.g., HTTP response code[6] and Content-Type). Stateless communication saves energy on the server side, as the state of the interaction with any client does not need to be stored in memory.

- **Layered architecture**: REST promotes to use a layered system architecture where each layer does not know any thing about any layer other than that of immediate layer. In such model, there can be lot of intermediate servers between the client and the end server. This allows improving system availability by enabling load-balancing and by providing shared caches.
- **Cacheable**: A cacheable response is an HTTP response which can be cached, i.e., stored to be retrieved and used later, thus, saving a new request to the server. In this context, client will return the data from its cache for any subsequent request and there would be no need to send the request again to the server. This can eliminate some client-server interactions, and further improve availability and server performance because the load has reduced.
- **Code on demand**: It is an optional feature. According to it, servers can provide downloadable and executable code to the client in the form of applets or scripts. This helps clients in reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility.
- **HATEOAS**: Known as Hypermedia as the Engine of Application State, is the latest constraint of the REST paradigm used to provide directions to the client/agent regarding the next possible operations to be triggered. The principle is to include within returned server responses, the possible next resources URIs to follow, based on the current resource state. The methods used to invoke such resources can also be included. The main advantage of HATEOAS is to enable runtime drive of the application without the need to pre-design the workflow.

A comparison between SOAP and REST is presented in Table 2.1, which reveals the advantages of REST on different aspects (e.g., responses formats, message contents, bandwidth, etc.). Therefore, in this thesis, Web services are designed as RESTful services (resources) that follow the REST principles. During the elaboration of the contributions, we focused on the following main REST constraints [91]: (1) resource-oriented and addressability, (2) uniform interface, and (3) HATEOAS. The remaining constraints (excepting the "Code on demand" feature) are supported during resource implementation.

2.1.3 Semantic Web Languages

In the semantic Web [27], information is given a meaning by representing it through a machine-readable markup language with a well-defined semantics. This is done to allow data to be shared and reused across applications

[6]Examples: "200 OK", means that the request is succeeded, and "201 Created", means that the request has been fulfilled and a new resource is created (see `https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)/`

Table 2.1 – SOAP vs REST

Category	SOAP	REST
Transfer	It was designed to be independent of any transfer protocol. Often, it missuses HTTP and adds overheads in the request/response messages	It is typically implemented with HTTP, and takes advantage of all of it features
Responses Formats	Only permits XML data format	Permits different data format such as Plain text, HTML, XML, JSON etc.
Message Contents	SOAP-based Web services send additional information apart from the message (e.g., the header) which makes the size of message heavy	RESTful services only send the message to be passed, which makes it lighter than SOAP. The message contains the information about the internet resource to be accessed (URI)
Bandwidth	Requires more bandwidth	Requires less bandwidth
Security	Has more control over security using WS-Security for example	It lacks security compared to SOAP. It inherits security measures from the underlying protocol (HTTP)
Reliability	It has inbuilt retry logic in case of transaction failure and provides end-to-end reliability	It does not have a standard messaging system and expects clients to deal with communication failures by retrying the operation
Transaction Management	SOAP has more control for transactions management	In REST, it is necessary to write the logic for managing transactions
Operations State	It is stateful	It is designed to be stateless
Caching Mechanisms	Cannot be cached	Can be cached
Development and Implementation	It requires writing a lot of code to perform simple tasks	It uses HTTP and basic CRUD operations (Create, Read, Update, Delete), so it is simple to write

or systems, and provide knowledge understandable to all (machines and humans).

From this perspective, and in order to allow generic clients (typically Web browse-rs) to discover and invoke resources automatically, facilitating thus the composition process for end-users, especially in complex environments connecting mobile/stationary objects and providing static/dynamic resources, it is important that resource properties are described using a machine readable language. In this context, semantic Web markup languages are used to represent data related to Web services, i.e., known as semantic Web services, by making their properties (e.g., provided functions, inputs/outputs, etc.), encoded in an unambiguous and machine understandable form. Each semantic Web language has a well-defined syntax and semantics to enable unambiguous computer interpretation when describing Web services. It is based on a specific formalism. A number of languages have been proposed for representing semantic Web meta-data, in particular RDF [37] and OWL [20].

2.1.3.1 RDF, RDF-S

RDF [74], which stands for Resource Description Framework, is a framework that describes Web resources, identified by Web identifiers (URIs), with properties and property values. The combination of a resource, a property, and a property value forms a statement known as the subject, predicate and object of a statement.
Below an RDF example describing the resource "https://www.buildings.com/bldgA":

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:ex= "http://www.semanticweb.org/myontology">
<rdf:Description rdf:about="https://www.buildings.com/bldgA">
  <ex:NumberFloors>8</ex:NumberFloors>
  <ex:Type>Residential</ex:Type>
</rdf:Description>
```

```
7 </rdf:RDF>
```

Listing 2.3 – RDF Example

RDF documents, which are designed to be read and understood by computers, are written in XML. By using XML, RDF information can easily be exchanged between different types of computers using various types of operating systems and application languages.

RDF Schema (RDF-S) [30] is an extension to RDF. It provides mechanisms for describing groups of related resources and the relationships between them. Besides defining triples, it allows to define class and property hierarchies. The RDF-S class and property system is much like classes in object oriented programming languages, allowing resources to be defined as instances of one or more classes, and subclasses of classes. For example, a class "Sensor" might be defined as a subclass of "Device". This means that any resource that is in class "Sensor" is also implicitly in class "Device" as well. RDF-S constructs are the RDF-S classes, associated properties (i.e., used to describe a relation between subject resources and object resources) and utility properties, built on the limited vocabulary of RDF. Below an RDF-S document showing a subclass example:

```
1 <?xml version="1.0"?>
2
3 <rdf:RDF
4 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6 xml:base="http://www.example.com/Buildings/devices#">
7
8 <rdf:Description rdf:ID="Device">
9   <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
10 </rdf:Description>
11
12 <rdf:Description rdf:ID="Sensor">
13   <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
14   <rdfs:subClassOf rdf:resource="#Device"/>
15 </rdf:Description>
16
17 </rdf:RDF>
```

Listing 2.4 – RDF-S Example

2.1.3.2 OWL

The Web Ontology Language (OWL) [90] facilitates greater machine interpretability of Web content than that supported by RDF and RDF-S, by providing additional vocabulary along with a formal semantics, and thus OWL goes beyond these languages in its ability to represent machine interpretable content on the Web. Apart from defining classes and properties as in RDF-S, it mainly provides constructs to create new class descriptions as logical combinations (intersections, unions, or complements) of other classes, and defines cardinality restrictions on properties. OWL provides mainly three increasingly expressive sub-languages [75]:

- OWL-Lite: is the least expressive sub-language. It is used in situations where only a simple class hierarchy and simple constraints are needed.

- OWL-DL: is more expressive than OWL-Lite and is based on Description Logics [12] (hence the suffix DL). It is used when more expressiveness is required, while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time).
- OWL-Full: is the most expressive OWL sub-language. It is used in situations where a very high expressiveness is required. OWL Full is undecidable, so no reasoning software is able to perform complete reasoning for it.

In the following, we show an OWL example where the building occupant class is equal to the person class defined in another ontology.

```
<?xml version="1.0"?>

 <!DOCTYPE rdf:RDF [
  <!ENTITY foaf "http://xmlns.com/foaf/0.1/#" >
  <!ENTITY exemple "http://www.semanticweb.org/ontology#">
  ]>

<rdf:RDF
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<owl:Class rdf:about="&exemple;occupant">
<owl:equivalentClass rdf:resource="&foaf;person"/>
</owl:Class>
</rdf:RDF>
```

Listing 2.5 – OWL Example

2.1.3.3 JSON-LD Format

JSON-LD [102], which stands for JavaScript Object Notation for Linked Data, is a JSON-based representation of RDF. As such, a JSON-LD document is both an RDF document and a JSON document, representing an instance of an RDF data model. However, JSON-LD also extends the RDF data model to optionally allow JSON-LD to serialize generalized RDF Datasets[7]. JSON-LD provides a way to help JSON data interoperate at Web-scale, by adding semantics to existing JSON documents. It is easy for humans to read and write. Also, little effort is required from developers to transform their plain old JSON to semantically rich JSON-LD. The basic idea of JSON-LD is to link objects and their properties in a JSON document to concepts defined in data models (i.e., ontologies). These ontologies are defined in the form of a so called context. A context can either be directly embedded in a JSON-LD document or put into a separate file and referenced from different documents. An example of a JSON-LD is given in Listing 2.6:

```
{
   "@context": {
      "h2g": "http://www.h2g.eu/ontoh2g/",
      "description": "h2g:description",
      "nbfloor": "h2g:nbfloor"
   },
```

[7] https://www.w3.org/2018/jsonld-cg-reports/json-ld/#relationship-to-rdf/

```
7     "@id": "http://building.example.com",
8     "@type": "h2g:building",
9     "description": "This building consist of 5 floors",
10    "nbfloor": 5
11 }
```

Listing 2.6 – JSON-LD Example

Listing 2.6 describes a building using the BEMServer ontology [98] related to the smart buildings domain. In the example, the context is embedded in the JSON-LD document, which contains a prefix referring to the ontology concepts, instead of using their long Internationalized Resource Identifier (IRI). As seen in lines 4-5, the two JSON properties: description and nbfloor, are mapped to concepts in the BEMServer ontology, allowing them to be machine-readable and understandable properties.

Since automatic RESTful service (resource) composition is one of the objectives of this thesis, more specifically, the resource discovery and resource selection processes, it is important to describe the resources using a semantic-based machine-processable language. In our work, we use Hydra vocabulary expressed in JSON-LD, as it is a lightweight format that is understandable and less complex to use by both machines and humans. Hydra is presented in the next section, with other existing resource descriptions languages.

2.2 Resource Description

Resource descriptions allow to give information about resources to minimize the amount of shared knowledge and customized programming that is needed to ensure communication between the resource provider and the resource requester. Several description languages have been proposed in order to describe REST services. In this section, we review the most known ones. During the review, we analyze the descriptive languages according to the following criteria:

- Resource-oriented: the ability of the language to describe the services, as resources by associating to each, a URI, an HTTP verb, links to other resources, etc.
- Hypermedia-based: the ability to define links between resources. This is important to cover the HATEOAS principle, one of the main REST principles that we aim to consider in this thesis. It consists in including within resource response message, the set of resources URIs that can be called next, based on the current resource state. Although HATEOAS is still rarely used [83], our work seeks to support it, as it allows generic clients (typically Web browsers) to dynamically navigate to the next appropriate resources.
- Support for semantic annotations: the ability to integrate semantic annotations allowing the properties of the resources to be machine processable.
- Supported data format: defines the data format used to express the descriptive language (e.g., XML, JSON, HTML, etc.).

The above criteria match the principles of the Linked Data[8], which refers to a set of best practices for publishing structured data on the Web. Linked Data allows to use standards for the representation and the access of data on the Web, and enables the propagation of set of hyperlinks between data from different sources.

2.2.1 The Web Services Description Language (WSDL)

The WSDL [36] is an XML-based language that is mainly used to describe SOAP-based Web services. A WSDL description contains the necessary details that clients can use to interact with a service. It mainly consists of:

- The service's URL
- The communication mechanisms it understands
- What operations it can perform
- The structure of its messages

The first version introduced was WSDL 1.1. However, the WSDL 1.1 was inadequate to describe communications with HTTP and XML, and thus REST services, which rely only on HTTP, were not compatible with such language.

WSDL 2.0 [34] was declared a W3C recommendation, and introduced new elements and attributes to allow REST services endpoints definition. In fact, this second version of WSDL was created to mainly (i) address issues with WSDL 1.1, many of which had been identified by the Web Services Interoperability organization (WS-I), and to (ii) support HTTP bindings. Briefly, the root element of a WSDL 2.0 document is the **description** element which has four child elements:

- **Types**: describes the data types used by the Web service. Most often a Web service will have an input type, an output type, and a fault type. The data types can be declared in any language, as long as the Web service supports it. Data types are often specified using XML Schema though, since XML Schema is a natural fit for XML structures.
- **Interface**: defines the Web service operations, including the specific input, output, and fault messages that are passed along with their order.
- **Binding**: defines how a client can communicate with the Web service. In the case of REST services, a binding specifies that clients can communicate using HTTP.
- **Service**: associates an address, referring to a URI Web service, with a specific interface and binding.

An example of a WSDL 2.0 is given in Appendix A.

[8] `https://www.w3.org/wiki/LinkedData/`

2.2.2 The Web Application Description Language (WADL)

The Web Application Description Language (WADL) [50] is a machine readable XML description of HTTP-based Web services that models the resources provided by a service and the relationships between them. It is intended to simplify the reuse of Web services that are based on the existing HTTP architecture of the Web. WADL is considered to be the REST equivalent of SOAP's Web Service Description Language (WSDL). The service in WADL is described using a set of **resource** elements. Each **resource** contains **method** elements, which describe the **request** and **response** of a resource. The **request** element specifies through **param** elements the input, the necessary type, and any specific HTTP headers that are required. The **response** describes the representation of the service's response, as well as any fault information to deal with errors. An example of a WADL is given in Appendix B.

SWSAL [42] is an XML-based annotation language that allows to link data expressed in WADL, with concepts taken from a domain ontology [72] that is used to define a domain, and to reason about the properties of that domain. By annotating resources with concepts of an ontology, semantics meanings are given to the resource. This allows service users as well as machines to understand the behavior of the resources.

2.2.3 Web Page Annotations-based Languages

SA-REST [65], which refers to Semantic Annotations of Web Resources, is an annotation scheme that is used to add additional meta-data to Web resource descriptions expressed in HTML or XHTML[9] pages. The meta-data contained in various models, e.g., OWL and RDF, can be embedded into the documents, making them human and machine readable. This allows various enhancements, such as improving resource search, facilitating data mediation, and simplification of resources integration. Figure 2.2, shows an exam-

```
<html xmlns:sarest="http://lsdis.cs.uga.edu/SAREST#">
...
<p about="http://craigslist.org/search/">
The logical input of this service is an object
<span property="sarest:input">
http://lsdis.cs.uga.edu/ont.owl#Location_Query
</span>
The logical output of this service is a list of objects
<span property="sarest:output">
http://lsdis.cs.uga.edu/ont.owl#Location
</span>
This service should be invoked using an HTTP Get
<span property="sarest:action">
HTTP Get
</span>
<meta property="sarest:operation" content=
"http://lsdis.cs.uga.edu/ont.owl#Location_Search/">
</p>
</html>
```

Figure 2.2 – SA-REST example

ple of a RESTful service described using SA-REST [101]. In the description, semantic annotations are inside the <meta> tag as well as inside formatting

[9] Extensible HyperText Markup Language

tags such as . Annotations in the <meta> tags are not visible to the user, contrary to the tags.

hRESTS [62], or HTML for RESTful Services, is a microformat used to structure existing RESTful Web service documentation expressed in the form of Hypertext Markup Language (HTML) Web pages. The microformat serves as the basis for extensions that introduce additional information (in the form of annotations) to the services HTML-based Web pages allowing them to be machine-processable. It is made up of a number of HTML classes, such as service class, operation class, address class, input and output classes, etc., that correspond directly to the various parts of services models. An example of a hRESTS-based service description is given in Figure 2.3. Alternatively to

```
<div class="service" id="svc">
  <h1><span class="label">ACME Hotels</span> service API</h1>
  <div class="operation" id="op1">
    <h2>Operation <code class="label">getHotelDetails</code></h2>
    <p> Invoked using the <span class="method">GET</span>
    at <code class="address">http://example.com/h/{id}</code><br/>
      <span class="input">
        <strong>Parameters:</strong>
        <code>id</code> -- the identifier of the particular hotel
      </span> <br/>
      <span class="output">
        <strong>Output value:</strong> hotel details in an
        <code>ex:hotelInformation</code> document
      </span>
    </p>
</div></div>
```

Figure 2.3 – hrest example

using the microformat to capture the service model structure in the HTML documentation of RESTful Web services, RDFa[10] [2] can also be deployed pointing directly to the RDF model. RDFa specifies a collection of XML attributes to express RDF data in any markup language, including HTML.

2.2.4 Hypermedia-based Languages

HAL[11], referring to Hypertext Application Language, is an open specification describing a generic structure for RESTful resources. It can be expressed through two hypermedia types, XML and JSON. HAL is designed for implementing RESTful services in which clients navigate around the resources by following links. It revolves around representing two concepts: Resources and Links. Resources include:

- Links to URIs
- Embedded Resources (i.e., other resources contained within them)
- State (JSON or XML related data properties)

Links include:

- A target that is a URI
- A relation, which is the name of the link

[10] It stands for "Resource Description Framework in Attributes"

[11] `https://apigility.org/documentation/api-primer/halprimer/`

- A few other optional properties related to content negotiation

An example of HAL description expressed in JSON is given in Appendix C.

SIREN[12] is a hypermedia specification for representing entities, i.e., URI-addressable resources. SIREN offers (i) structures to communicate information about entities, (ii) actions for executing state transitions, and (iii) links for client navigation. The initial implementation of SIREN is expressed through JSON. However, XML may also be used. SIREN specification includes:

- Entities that are URI-addressable resource having properties and actions associated with them. They may contain sub-entities and navigational links
- Actions showing available behaviors that an entity exposes
- Fields that represent controls inside the actions
- Links representing navigational transitions

An example of SIREN description is given in Appendix D.

Mason[13] is a JSON format that introduces hypermedia elements to classic JSON data representations. Through Mason several data can be acquired:

- Hypermedia elements for linking and modifying data
- Useful information to client developers
- Standardized error handling

Mainly, Mason consists of the following features:

- The "@meta" element: which conveys information to the client developers. It contains a "@title" and a "@description" property, to describe the response for the client developers. The ‘"@meta" element may even contain links for the client developers, as an online documentation.
- The "@controls" element: which includes the links representing the relations between the current resource and another resource, and other hypermedia control elements (e.g., HTTP verb). The relation between each two resources has a name (i.e., relation type) that is used as an index by the client to locate the link.

An example of MASON description is given in Appendix E.

Hydra [64], represented by the model in Figure 2.4, is a lightweight vocabulary used to describe RESTful services and publish valid state transitions to clients. This published information can help clients to construct HTTP requests and access exposed resources. It can be expressed with different formats, but essentially via JSON-LD, described in Section 2.1.3.3, which simplifies the mapping of resources properties, e.g., their provided functions and

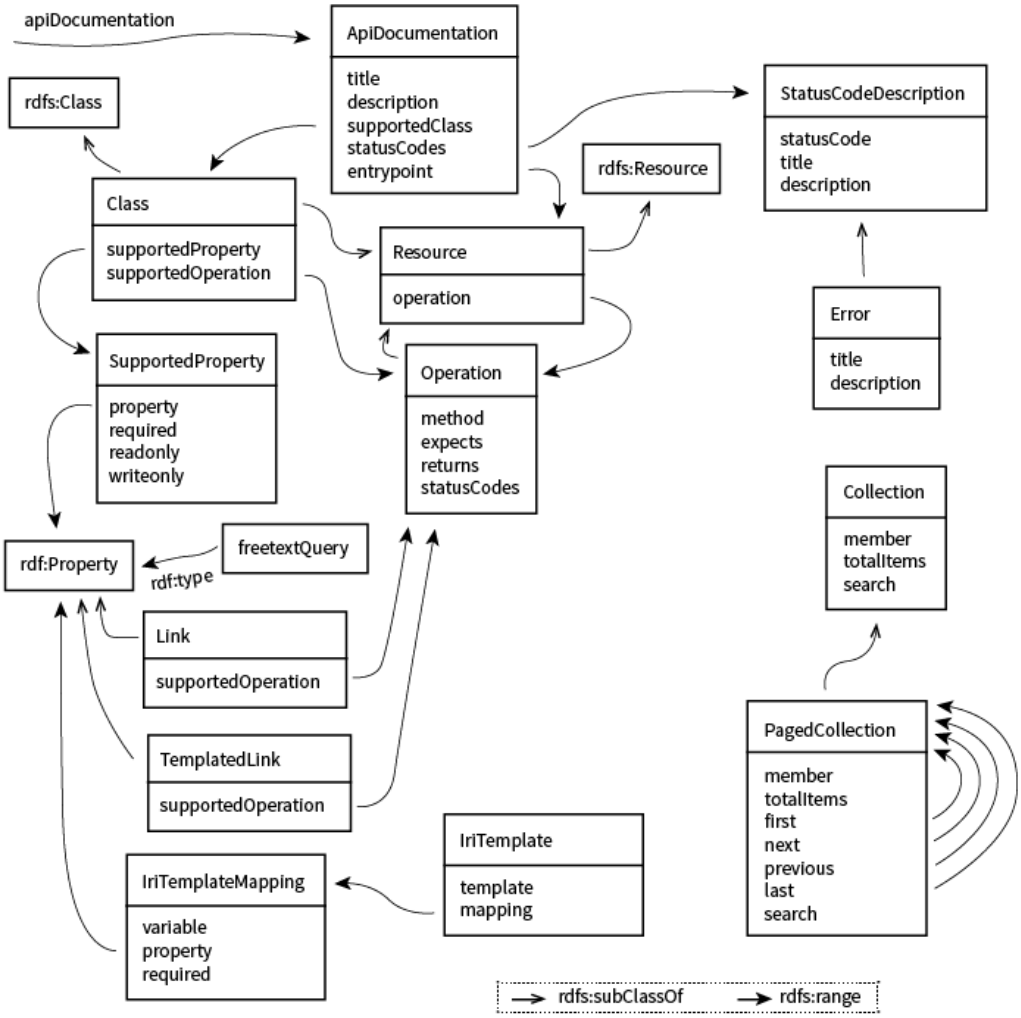


Figure 2.4 – The Hydra core vocabulary model

linked resources, to concepts defined in existing data models such as ontologies. This allows resources to be effectively exploited by automated tools. An example of Hydra is shown in Listing 2.7.

```
{
    "@context": {
        "@vocab": "http://www.w3.org/ns/hydra/core#",
        "schema": "http://schema.org/",
        "h2g": "http://www.h2g.eu/ontoh2g/",
        "function": "h2g:servicefunction"
    },
    "@id": "http://www.h2g.eu/h2g/resourcedescription/getairtemperature.
        md",
    "Operation": [{
        "method": "GET",
        "expects": ["h2g:startdate", "h2g:enddate"],
        "returns": ["schema:DateTime", "schema:Float"],
        "function": "ATC"
    }],
    "Link": [{
        "entrypoint": "http://www.h2g.eu/h2g/resource/predheatengcons",
        "method": "GET"
```

[12] https://github.com/kevinswiber/siren/
[13] https://github.com/JornWildt/Mason/

```
18      }]
19  }
```

Listing 2.7 – Resource description example using Hydra in JSON-LD

2.2.5 Evaluation Summary

Table 2.2 shows the evaluation summary of existing languages used to describe REST-based services according to the identified criteria. With the exception of the "Supported Data Format", where we give clearly the data format supported by each language, we used for the rest of criteria the "+" symbol to express a positive criterion coverage, and "-" symbol to express a lack of criterion coverage.

Table 2.2 – Evaluation of existing languages used to describe REST-based Web services w.r.t. the identified criteria

		Resource-oriented	Hypermedia-based	Support for Semantic Annotations	Supported Data Format
WSDL-based	WSDL 2.0	-	-	+	XML
WADL-based	WADL	+	-	-	XML
	SWSAL	+	-	+	XML
Web Page Annotations-based	SA-REST	+	+	+	HTML
	hREST	+	+	+	HTML
Hypermedia-based	HAL	+	+	-	XML and JSON
	SIREN	+	+	-	XML and JSON
	MASON	+	+	-	JSON
	Hydra	+	+	+	XML, JSON-LD, and Turtle

As seen in the table, WSDL 2.0 is an XML-based language that is not resource oriented, nor a hypermedia-driven language, i.e., it does not define links to other related resources, despite its ability to be linked to several semantic models concepts (e.g., ontologies). Also, and among the two WADL-based languages supporting XML, only SWSAL can add semantic annotations. However, both do not allow the definition of links to other related resources (they are not hypermedia-based). In addition, and although the Web page annotations-based languages (SA-REST and hRESTS) are resource-oriented, hypermedia-based and support semantic annotations, they are description based on HTML integrated directly into RESTful services HTML pages. This makes service descriptions unclear and hard to follow for many developers, as they are combined within the services HTML pages. As for the Hypermedia-based languages, HAL, SIREN, and MASON are resource-oriented and hypermedia-based languages that allow the definition of links to other related resources. While HAL and SIREN support each XML and JSON as data formats, MASON is only expressed with JSON. These three languages, however, lack in adding semantic annotations, which is an important criteria to consider, to provide knowledge about resources properties and links that are understandable to machines. Expressed with different data formats, but mostly with JSON-LD, Hydra is a resource oriented and hypermedia-based vocabulary that allows the integration of semantic annotations by linking resources properties to concepts in existing data models. As such, in the description shown in Listing 2.7, and based on JSON-LD, resource properties are linked to concepts existing in ontologies, as the ontology [32] developed in BEMServer (see Section 1.1.1.3), to give a semantic meaning of the shared information to both humans and machines. The "@context" element contains the ontologies prefixes mapped to their URL.

These prefixes are used to link the JSON-based resource properties to their corresponding ontologies concepts (e.g., the "expects" values: startdate and enddate, which refer to the inputs parameters of the resource, are defined in the HIT2GAP ontology specified by the prefix "ontoh2g"). As for the properties that are not linked to a data model, they are directly defined in the Hydra vocabulary, e.g., Operation and method. For these reasons, we used Hydra expressed with JSON-LD to describe the resources in this thesis. Such Hydra-based descriptions allow the automatic discovery and selection of resources, which will be explained respectively in details in both Chapter 4 and Chapter 5.

2.3 Summary

This chapter gives some background information related to several main technical concepts for the full understanding of the context of this thesis. It starts by presenting the concept of Web services, the main protocol/principles used to implement them, with emphasis on the REST architecture style (the one supported in our work due to its various advantages), and the semantic Web languages defined to make their properties machine-readable, facilitating thus their usability. Then, it presents a review on the existing languages used to describe RESTful services (resources) with a focus on hypermedia-driven languages, including Hydra vocabulary expressed in JSON-LD (the one adopted in this thesis).

In the next chapter, we present our first contribution related to the verification of static resource compositions before their execution.

Chapter 3

Verification of Static Resource Compositions Behavior

"Trust, but verify"

Ronald Reagan

RESTful service composition, which consists on combining several services (resources) in one single process, has received much interest to satisfy complex user requirements. However, verifying the correctness of a composition remains a tedious task. In this chapter, we present our approach for verifying the behavior of static resource compositions involving several resources manually selected and linked by the user, which is part of the verification process in the static resource composition of the StARC framework. To do so, we propose a formal model based on Colored Petri Nets (CPNs) for modeling the behavior of resources and their composition. Using the proposed model, we show how it can be used to verify relevant composition behavior properties: (1) Interoperability, to check if the linked resources are compatible according the their related Input/Output data types, (2) Reachability, to make sure that the desired results can be reached from the initial composition state, and (3) Liveness, to verify that all involved resources can be invoked during composition execution. A standalone prototype has been developed, to model, verify, store, and execute a resource composition. The prototype concerns two categories of resources: data collection and data preprocessing.

3.1 Introduction

The construction of new resources by combining two or more existing resources in the same composition scenario, raises several challenges, among them, the guarantee of the correct interaction between the resources, leading to a proper behavior of the overall composition [105]. Within this context, there is a growing interest for the verification techniques to ensure the correctness of the composition and enable its designers to detect erroneous behavior before actual composition run [121]. As such, several problems may occur during composition execution due to an erroneous composition design, e.g., I/O datatype mismatching between the linked resource, and endloops occurred during composition execution preventing other resources to run.

Verifying a composition usually relies on the verification of its behavioral properties [121] (e.g., Reachability, and Liveness). Such verification typically depends on the formal modeling of the composition behavior using a modeling language with clear semantics. Several works have been carried out in this scope. Some RESTful composition approaches are based on formal languages (e.g., Petri Nets [5, 40], Finite State Machine [128, 113], and Process Algebra [119, 122]), and others rely on services descriptions with embedded semantics such as in [109]. Although these approaches respect the majority of REST principles, inlcuding HATEOAS, they mainly contribute in modeling and constructing RESTful services composition without verifying its correct behavior. In this chapter, we propose a formal language based on Colored Petri Nets [69], known as CPNs, to model and verify RESTful service composition. The main contribution of this work is the mapping between CPNs model and RESTful services, to allow the use of CPNs behavioral properties for verifying RESTful service composition. Based on CPNs, several properties can be checked, among them:

- Reachability, is used to verify that the desired final composition state is reachable from the initial state.
- Liveness, is used to ensure that all resources participating in the composition will be invoked (i.e., not dead) during composition execution.
- Interoperability, is used to check if the resources involved into the composition can be linked together. This is related to data type compatibility between the input and output of the linked resources where the output of a resource should be of the same type of the input of another resource.
- Persistence, is used when parallel resources accessing the same information are executed simultaneously. It ensures that there is no conflict between them.
- Boundness, is used to check the maximal allowed number of input/output data (i.e., tokens) of each resource in a composition.

Nevertheless, in this work, we focus on the first three properties, namely: Reachability, Liveness, and Interoperability, as they are considered to be fundamental in the literature [94, 107], and cope with the main design problems

that an end-user may occur when linking manually the resources of a composition: (i) erroneous resource linking causing an unreachable desired state, and preventing other resources to run (i.e., dead resources), and (ii) datatype mismatch of the linked I/O resources.

The remainder of this chapter is organized as follows. Section 3.2 presents a scenario illustrated in the BEMServer Web platform to motivate our work, and highlights the research problem we tackle. Section 3.3 presents the related work and highlights the originality of our approach. Section 3.4 gives a description on the basics and formal model of Colored Petri Nets (CPNs). Section 3.5 details our CPN-based approach for verifying RESTful service composition. Section 3.6 illustrates the proposed solution within our motivating scenario. Section 3.7 presents the prototype implementing our approach in the context of SIBEX. Finally, Section 3.8 concludes the chapter.

3.2 Motivation and Problem Statement

Our motivating scenario is illustrated in the BEMServer Web platform presented in Chapter 1. Technically, the platform provides: (1) resources for collecting heterogeneous on-site data (e.g., internal temperature and energy consumption) contained in the Field layer, (2) basic resources, presented in the Core layer, for preparing the collected data (e.g., outliers correction and data alignment), and (3) advanced resources, integrated in the Management layer, to process the prepared data (e.g., energy prediction and energy model calibration). In the scenario, we assume that a building manager wants to estimate the upcoming week heating energy consumption of his building. The prediction output will help him to anticipate building energy resource needs required for the resulted consumption, and analyze building energy behavior. To do so, the building manager has to invoke several services simultaneously, embedded as resources, through the corresponding Web platform instance deployed in the building (as depicted in Figure 3.1), and link them together correctly to reach the desired composition behavior. Figure 3.2 de-

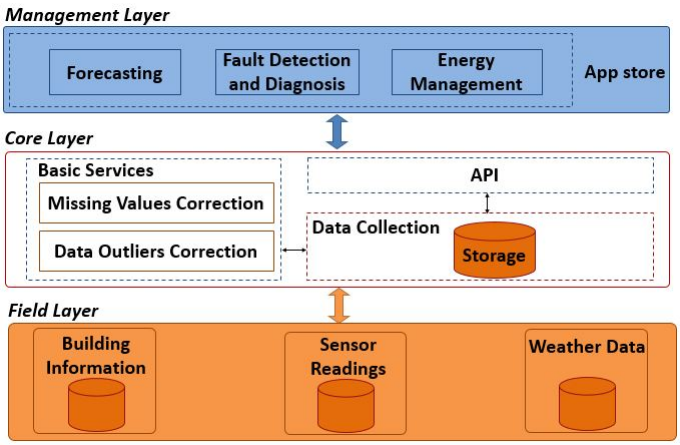


Figure 3.1 – Instance of the BEMServer Web platform

picts the overall process to be executed for answering the building manager's request. It mainly requires the interaction with the following resources:

1. Data collection resources to collect data required for the prediction process:

(a) Upcoming week predicted internal temperature, which is extracted directly from the platform storage

(b) Upcoming week predicted external temperature, which is provided by an external weather forecast RESTful service

We consider that the collected data are aligned to the same required frequency (per 15 minutes).

2. Data pre-processing resources that clean the collected data from the external weather forecast service:
   (a) Resource that manages and corrects outliers values, which are data values outside the range of most of the other values
   (b) Resource that manages and corrects empty or missing values retrieved during certain timestamps
3. Resource responsible for the prediction of the heat energy consumption. This resource, embedded into the Forecasting module in the BEMServer platform, uses a prediction model considered already implemented in the module.

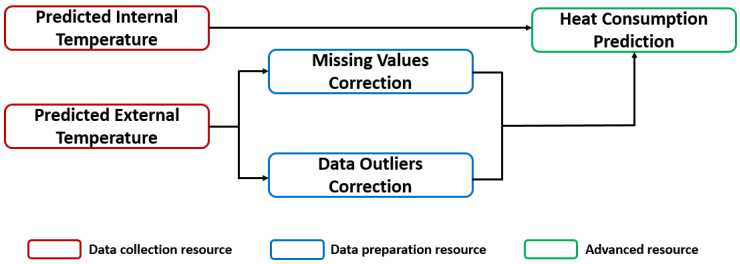


Figure 3.2 – The resources involved in the prediction process

The scenario shows the resources' composition needed to satisfy the request at hand. However, building the composition properly and ensuring its correct behavior is a difficult task for the building manager. In fact, several problems may occur when building the composition:

- **Non-interoperability:** Links between the output of a resource and the input of another may be invalid. This is due to the difference of data types that each resource handles. For example, as shown in Figure 3.3, the resource responsible for correcting outliers' values returns an array of temperature values whose data type is "Reak". However, as the heat consumption prediction resource receives values with a different data type (e.g., "Integer"), the composition will be erroneous by losing precision in the data.

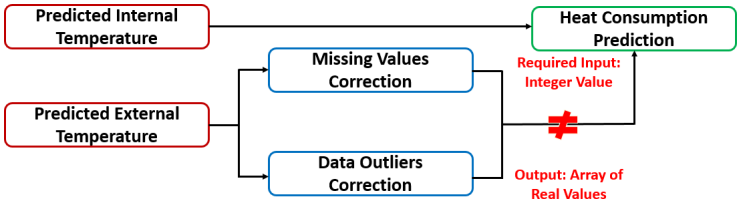


Figure 3.3 – Non-interoperability of data types between the linked resources

- **Looping:** Starting the composition by collecting the required data (the internal and external predicted temperature), the process may not reach the final expected result, which is acquiring the predicted energy heat consumption of the building. This can be due to an end-loop occurred at one of the data preparation resources, as depicted in Figure 3.4.

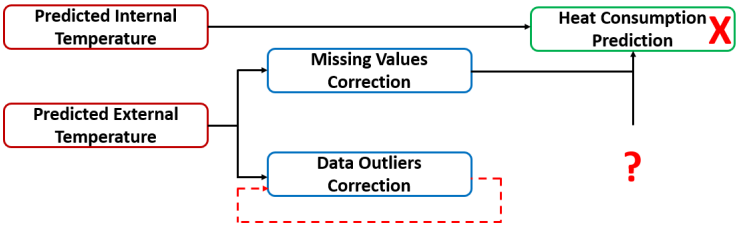


Figure 3.4 – Linking error causing a loop in the execution of a resource

- **Dead resources:** A resource, as the resource responsible for correcting missing' values, may not respond due to some missing linking (see Figure 3.5). This can prevent the next related resources involved in the composition process to run or cause an erroneous results (the missing values will not be handled).

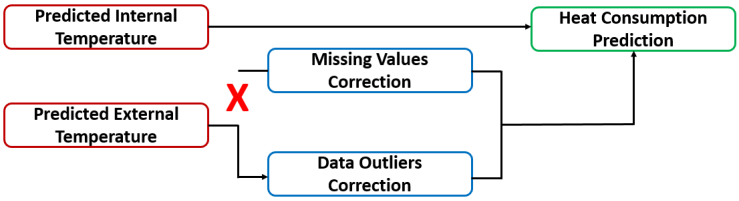


Figure 3.5 – Linking error causing a dead resource

In order to overcome the aforementioned problems, we propose in this chapter a formal language based on Colored Petri Nets (CPNs) to model resources and their composition, while considering the REST principles. By using CPNs, our approach is able to handle data types, and thus, to check resources datatype matching. Moreover, with the formal syntax and semantics of CPNs, we are able to validate the behavior of the built composition models through several verification properties (i.e., Reachability, Liveness, and Interoperability), embedded in open source and well known tools such as the CPN tools. The advantage of CPNs are detailed more in Section 3.4.

3.3 Related Work

In this section, we review existing approaches that modeled REST service compositions using formal languages (e.g., Petri nets, Process algebra, etc.). As REST is a recent emerging technology, in the literature, there exist little work related to this research area. Therefore, we also show some of the works related to SOAP oriented services. During the review, we mainly compare existing modeling approaches according to the following criteria:

- **RESTful principles support:** This allows modeling RESTful services and compositions behavior. As such, the solution should be aligned

with the resources properties w.r.t. the REST architecture style principles (see Section 2.1.2), including HATEOAS.

- **Data types handling:** The ability to handle the types of data flowing between resources (i.e., String, Integer, etc.) allows composition syntax checking and thus better management of the links between them.

- **Composition behavior verification:** In order to verify the correctness behavior of the composition, the solution should be able to verify several behavior properties (i.e., Reachability, Liveness, and Interoperability).

3.3.1 Petri Net-based Approaches

Petri nets are graphical and mathematical modeling language used to model distributed systems. They are designed to describe and study information processing systems, with concurrent, asynchronous, distributed, parallel, nondeterministic and stochastic behaviors [82]. A Petri net is graphically represented by a number of places (represented by circles) occupied by tokens, and transitions (represented by bars). Transitions and places are connected via arcs. A transition may fire when each of its input places has the required tokens. When it fires, all tokens from its input places are removed, and a token is placed into every output place.

In [40], a formal language for RESTful Web services is defined based on high level Petri nets, i.e., a class of Petri nets that includes some extensions as colored tokens (having a data type). Although this approach integrates the hypermedia aspect with external services, the links between the internal resources of a single Web service are not defined. Moreover, there are no verification properties to evaluate the composition correctness behavior, and all tokens are expressed only in XML data, without supporting explicitly defined data types (i.e., standard types and other defined types required for the resources), which implies several advantages:

- The ability to differentiate places by assigning a type (color) for each. This allows us to define the set of possible values that a place can have and the set of operations/ transitions that can be applied on these values.

- The ability to express directly the outputs type of a resource and the inputs type of another, and ensure their correspondence so that their composition can take place. This is easier than handling XML data type presented in the XML document, which requires additional parsing operations (XML parser) to ensure their consistency. As such, interpreting the 'prediction time' parameter type (time) related to the heat consumption prediction resource, is simpler from parsing an XML document and analyze the embedded data types.

- Similar to programming languages, data types can be used to apply specific conditions, known as guards, can be implemented to the transitions. While guards expressed in XML requires additional analysis, guards expressed using data types are faster and easier to evaluate.

- Better checking and verification in the composition process. In fact, with strong typing we will be steered away from error during resources compositions.
- From a graphical point of view, it is easier to analyze workflows where data types are shown visibly, without the need to do additional investigation efforts.

The same Petri nets formalism was also used in [5], where an XML-based language, Resource Linking Language, was defined for describing REST services. Although the proposed formalism models several REST concepts (e.g., resources, media types, and links to resources), internal resources linking is not expressed, and there are no verification properties applied to the Web services composition.

Authors in [68] propose REST Chart, a model and a definition language for REST APIs. Based on Colored Petri Nets (where data types are assigned to tokens), REST Chart integrates REST constraints to guide REST API designs. It basically models a REST API as a set of hypermedia representations and transitions between them. The model is then transformed into a special Colored Petri Net whose token markings define the representational state space of user agents using that API. Although respecting RESTful principles during the modeling, and handling data types, the work does not consider the verification of a REST API behavior by taking advantage of the CPNs properties such as deadlock detection.

3.3.2 FSM-based Approaches

Finite State Machine (FSM) [111], also called finite state automaton, is a mathematical model of computation used to simulate sequential logic and control execution flow of computer programs. It can be represented as directed graph, in which there are finite numbers of states, and each state has transition into next state. The FSM can change from one state to another in response to some inputs that determine which transitions is to be executed.

In [128], RESTful services are modeled through a non-deterministic[1] FSM approach with epsilon transitions (ϵ-NFA) that do not need to read an input symbol in order to modify the system's state. The proposed model follows RESTful design principles, i.e., uniform interface, stateless client-server operation, and code-on-demand execution, and supports hypermedia links between internal resources belonging to the same system. However, there are no composition verification properties used to ensure the correct behavior of the composition execution, and no explicit handling of data types between the related resources.

Authors in [113] propose an approach for verifying hypermedia characteristics of a meta-model, Domain Specific Language (DSL), that is used to define RESTful API components, as resources with attributes and multiple

[1] It can exhibit different behaviors on different runs, as opposed to a deterministic

application states described by a resource and an HTTP verb. Transitions between application states are done using hyperlinks. The verification process of the model is done before transforming it into a ready source code. During model verification, authors first check whether it is ϵ-NFA (Nondeterministic Finite Automaton) compliance, i.e., every state within an application is accessible. And then, they make sure that there are no inappropriate state-to-state transitions within the model. Despite respecting all REST principles, including HATEOAS, and handling data types, there are several properties that are not considered during verification, that we considered important to verify, as Reachability and Interoperability.

3.3.3 Linear Logic-based Approaches

Linear logic is a non-classical logic of actions and resources describing processes dynamics and resource handling [77]. It is expressed in the sequent calculus format: assumptions $\vdash$ conclusions, where the conclusions on the right side are achieved by consuming the assumptions on the left side. Linear Logic has been applied to several areas in computer science, including functional programming, logic programming, general theories of concurrency, syntactic and semantic theories of natural language, artificial intelligence and planning.

In [124], Intuitionistic Linear Logic (ILL) is used to model formally RESTful Web services. The main contribution of this approach is Web services composition modeling, and the ability to ensure composition completeness and correctness, through theorems based on propositional Linear Logic and π-calculus. However, although it respects the main principles of REST architectural style principles, and handles data types, additional verification(e.g. no deadlock) are needed to check the correct behavior of the composition. Moreover, Linear Logic is a complicated formal language that requires extra efforts from Web engineers to put it in practice.

In [125], authors present a formal definition of REST Web services, and provide a method for Web service composition based on Linear Logic. The approach is a two-stage method used to find a composition of existing services that applies the business constraints and satisfies the composition requirements. However, the proposed approach neglects the hypermedia concept of REST (HATEOAS principle), and there is no explicit handling of data types.

3.3.4 Process Algebra-based Approaches

Process algebras (or process calculi) [47] are mathematical languages with well-defined semantics used to formally model systems behavior, and describe their interactions, communications and synchronizations. They provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalences between processes. Several process algebras-based languages are proposed in the literature, such as CSP, CSS, and LOTOS [112].

In [119], REST services are described through the combination of process calculi format with tuple space computing, which is a model for managing a distributed object system. Based on this work, a semantic resource is formalized as a process associated with a triple space and a URI used for handling remote requests. However, the proposed approach does not support HATEOAS principle, nor even verify properties to check the correctness of the composition.

The work in [122] models a RESTful system using CSP (Communicating Sequential Processes), a member of the process algebras mathematical theories. In the proposed model, the client, server and resources are modeled as processes. Formal descriptions are given to check whether the model fulfills the requirements of stateless and hypertext-driven properties of a RESTful system, and the safe and idempotent properties of standard HTTP methods. These constraints are verified using a model checker called PAT (Process Analysis Toolkit) [103]. Despite supporting HATEOAS, the work doe not verify the correctness behavior of a composition of resources.

3.3.5 Semantic-based Approaches

The semantic approaches define the meaning of resource descriptions, by adding metadata that are readable to machines, to enable them to understand and reason on resources properties, their relations to other resources, etc [110].

In [109], a semantic description model, called RESTdoc, is proposed to semantically describe RESTful resources. RESTdoc describes resources' functionalities using the Notation3 syntax with embedded semantics. Though the integration of semantics, RESTdoc is used for resource discovery and interaction, rather than dealing with compositions behavior. Moreover, Notation3 is a format that cannot be interpreted easily by Web engineers during the development, and thus it requires extra analysis skills to be understood.

In [54], a definition of REST semantic Web Services using process calculus formalism is proposed. This model can only describe specific type of REST services that are characterized by several points (e.g., a semantic Web resource that consists of a set of triples stored in a certain shared memory accessible by processes taking part in a computation). The work does not consider the hypermedia property of REST services, nor does it describe a methodology for checking the proper behavior of the composed services.

3.3.6 Verification of SOAP-based Services

Before the emergence of REST technology, many works were conducted to verify the behavior of Web service compositions, involving SOAP oriented services implemented using the SOAP protocol [117] (see Chapter 2).

In [44] a framework for the design and the verification of Web Services based on process algebras (e.g., CCS, π calculus, LOTOS) is proposed. Focusing on LOTOS, the work presents a two-way mapping between BPEL/WSDL

and LOTOS, and general guidelines for translations between BPEL/WSDL and a process algebra. The approach allows reasoning tools to verify and ensure some properties (as liveness), and handles data types.

In [53] an approach that extends π-calculus to allow modeling and verifying dynamic Web service compositions, in which services are selected at run time, is proposed. The work defines the syntax and the semantic of the defined model, however there is no verification applied on the compositions.

In [33], a Web service composition approach modelled by Objects-Oriented Petri nets, called G-nets, is presented. G-nets offer efficient mechanisms for modeling complex systems, based on a defined algebra. Besides defining a model that describes Web services and allowing the verification of properties (as Reachability), the work provides an operators representative set with clear syntax and semantics.

The work in [51], defines a Petri net-based algebra for modeling Web services control flows. The proposed algebra allows capturing the semantics of complex Web service combinations. As such, formal semantics of the composition operators (e.g., Sequence, Parallelism, and Alternative) are expressed using Petri nets by providing a direct mapping from each operator to a Petri net construction.

In [39], an approach for model abstraction and verification of composite Web service application described using BPEL, is proposed. The work consists mainly on transforming BPEL model into Colored Petri Net (CPN) model based on defined transformation rules, and on generating dummy Web services to cope with their defined WSDL. The BPEL transformed model (i.e., the orchestration abstraction model) and the dummy Web services are then composted as one CPN. The resulting CPN model is drawn by a CPN tool that is used to check the desired properties (e.g., deadlock free and an unreachable path).

3.3.7 Evaluation Summary

Table 3.1 shows the evaluation summary of existing formal models used to describe REST services according to the identified criteria. We used "+" symbol to express a positive coverage for a criterion, and "-" symbol to express a lack of a criterion coverage. As seen in the table, most models do not support the HATEOAS principle, which is important to consider when designing a RESTful service (see Section 2.1.2). Second, many models do not handle data types, and are not conceived to verify service compositions behavior. Third, the majority of the approaches do not offer an explicit mapping from REST principles to the proposed formalism. As for SOAP approaches, and though many of them allow the verification of several composition properties, and handle services data types, they are SOAP oriented.

Table 3.1 – Evaluation of existing approaches used for the formal modeling of REST services w.r.t. the identified criteria

			RESTful Principles Support	Date Type Handling	Composition Behavior Verification
REST	Petri Nets Approaches	[40]	+	-	-
		[5]	+	-	-
		[68]	+	+	-
	FSM Approaches	[128]	+	-	-
		[113]	+	+	-
	Linear Logic Approaches	[124]	+	+	-
		[125]	-	-	+
	Process Algebra Approaches	[119]	-	-	-
		[122]	+	-	-
	Semantic Approaches	[109]	+	+	-
		[54]	-	+	-
SOAP Approaches		[44]	-	+	+
		[53]	-	-	-
		[33]	-	+	+
		[51]	-	-	-
		[39]	-	+	+

3.4 Preliminaries: Colored Petri Nets

A Petri net consists of a number of places (circles), transitions (rectangles), and arcs. Arcs link a place (i.e., an input place) to a transition, and a transition to a place (i.e., an output place) [104]. Places in a Petri net may contain a discrete number of marks called tokens. A transition, t, may fire when each of its input places has at least one token (Figure 3.6 (a)). When it fires, a token from each of its input places is removed, and a token is placed into every output place (Figure 3.6 (b)). The number and position of tokens may change during the execution of the Petri net transitions. The assignment of tokens to places designates a state or a marking of the net.

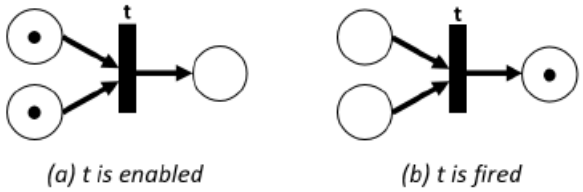


Figure 3.6 – Example of a single Petri net

In ordinary Petri nets, tokens cannot be distinguished and they are all identically represented as black dots. However, in more complex applications it is useful to allow the distinction between tokens and assign them some information. For these reasons, Colored Petri Nets (CPNs) combine the strengths of ordinary Petri nets with the strengths of high-level programming languages [49], to allow handling data types and manipulating data values. As such, within CPNs, each token can have a data type called a token color. Each color can be of a simple type (i.e., String, Integer, Boolean, etc.) or a complex type (i.e., array of String and Integer values). In addition, tokens with assigned colors can contain values. Normally, places in CPNs contain tokens of one type.

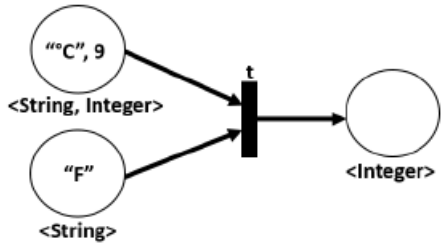


Figure 3.7 – Example of a Colored Petri Net

An example of a CPN illustrating a temperature unit conversion is shown in Figure 3.7. As it is illustrated, the first input place holds a record datatype containing two variables of String and Integer type respectively (one denoting the unit of measurement of the current temperature value, and the other the actual measured temperature value). The other input place holds a String type data representing the desired temperature unit of the Integer output place value.
Formally, a CPN is defined as follows [106]:

Definition 1. ***CPN*** *= (Σ, P, T, A, C, G, E, I), where:*

- Σ *is a finite set of non-empty types, called color sets*
- $(P \cup T, A)$ *forms a directed graph, where:*
  - *P (the set of places) and T (the set of transitions) are disjoint sets, such that* $P \cap T = \emptyset$
  - $A \subseteq (P \times T) \cup (T \times P)$ *is the set of arcs, such that places are only connected to transitions, and vice versa*
- $C : P \rightarrow \Sigma$ *is the color function that maps places to elements of* Σ
- $G : T \rightarrow \mathbb{B}$ *associates a precondition g (a boolean expression) to each transition. g should be evaluated to true for T execution.*
- $E : A \rightarrow Expr$ *associates an expression* $E(a)$ *to each arc a.* $E(a)$ *is used to define input-output behavior of arcs, and may include variables such that:*
  - *Each variable in* $E(a)$ *has a type in* Σ
  - $\forall a \in A, C(E(a)) = C(p)$*, with p is the place connected to a*
- *I is the initialization function that maps each place* $p \in P$ *with an expression such that* $I(p)$ *is associated to the type* $C(p)$

By using CPNs, we are able to analyze composition behavior and verify the properties that we considered important, in a flexible and powerful way. In addition, CPNs have the potential to explicitly show data in the composition, and there are several tools and libraries supporting CPNs that can be used. However, an extension of this model is necessary to be able to model resources while following REST principles. The extensions are explained in details in Section 3.5.

3.5 CPN-based Approach for RESTful Service Composition Verification

3.5.1 General Overview

Before elaborating on our solution for verifying RESTful service compositions, we show in Figure 3.8 the complete overview of the static composition process presented in the proposed StARC framework (see Figure 1.10). As such, in order to define the required composition, and based on a resource description guide that provides data on the available resources' properties (stored in the Resource Description repository), the user selects the desired resources, and link them in a single composition model (expressed in this work in JSON as explained in Section 3.7.1.1).

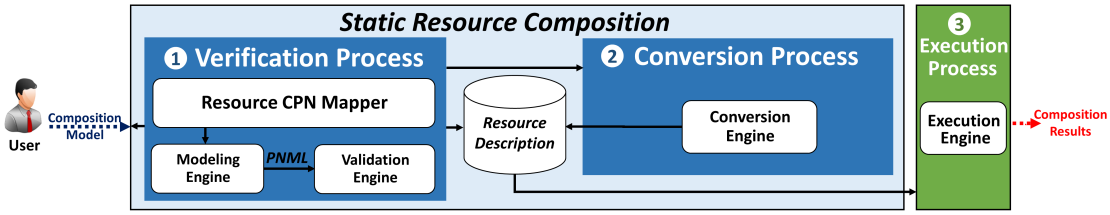


Figure 3.8 – Overview of the static RESTful service composition process

The composition model is syntactically checked by the Modeling Engine. If it is valid, it is modeled into a CPN-based format, i.e., PNML[2]. If not, a message error is returned to the user. The transformation of the composition model to PNML is based on a formal model that we defined to align resources to CPNs. Such model is embedded within the Resource CPN Mapper. The obtained PNML format is then passed to the Validation Engine to verify the necessary properties that we identified important to consider (i.e., Reachability, Liveness, and Interoperability). When the composition is verified, it is sent to the Conversion Process; if not, a report of design error(s) is sent back to the user. During conversion, the Conversion Engine converts the composition related PNML into a description format that can be stored in the Resource Description repository, which includes the resource descriptions. In our work, a verified composition is converted to JSON-LD (explained later in Section 3.7.2). Once the composition description is stored in the Resource Description repository, it can be executed by the Execution Engine of the Execution Process.

Our contribution in this chapter is essentially related to the Verification process, and more specifically to the Resource CPN Mapper, which includes our formal model proposed to align resources properties and their compositions to CPNs, allowing therefore the verification of resource compositions behavior using CPNs-based tools. The overall composition process has been implemented in a standalone prototype within the context of SIBEX (see Section 3.7)

[2]PNML is an XML-based syntax for high-level Petri nets, which is designed as a standard interchange format for Petri net tools.

3.5.2 Resource Generic Interface

One of the key requirements of our modeling approach is to be aligned with REST principles. Therefore, it is essential to define the generic REST interface of a resource before presenting the interfaces of the resources involved in our motivating scenario. Generally, REST interface describes the required URI, HTTP method, query parameters[3], and responses of a resource. Responses includes:

- A list of the next resources to follow with the method used to invoke them. In our work, the list can be empty when there are no resources to call.
- HTTP status code to indicate the query result. Such as, HTTP '200 OK' code denoting that the request has succeeded, and HTTP '201 Created' code designating that the request has been fulfilled and a new resource is created.
- The information provided by the resource, when it is available.

Illustration

Here, we restricted ourselves to the required interfaces in the motivating scenario (see Section 3.2) to ease the illustration of our approach. Table 3.2 lists the URIs of the composition scenario resources[4], and Table 3.3 defines their required interfaces.

Table 3.2 – URIs of the prediction process resources

Id	URI
1	http://www.h2g.eu/h2g/resource/pred-internal-temp
2	http://www.weatherforecast.com/forecast/external-temp
3	http://www.h2g.eu/h2g/resource/missing-data-manager
4	http://www.h2g.eu/h2g/resource/outliers-data-manager
5	http://www.h2g.eu/h2g/resource/missing-data-corrected
6	http://www.h2g.eu/h2g/resource/outliers-data-corrected
7	http://www.h2g.eu/h2g/resource/pred-heat-consumption
8	http://www.h2g.eu/h2g/resource/heat-consumption

URI_1 is called with GET to collect the predicted internal temperature according to 2 parameters: startdate and enddate, denoting the prediction time range requested by the building manager. The required data is retrieved directly from the BEMServer database, and considered as already pre-processed data. The array 'PrInTemp' in the responses represents the predicted internal temperatures. URI_2 is invoked using GET to collect the predicted external temperature according to the same 2 parameters of the previous URI. The required data is retrieved from an external weather forecast resource. The array 'PrExTemp' represents the predicted external temperature. After data retrieval, URI_3 is invoked through POST to correct the missing values presented in the array 'PrExTemp', and URI_4 is called to correct the outliers values presented in the array 'PrExTemp'. Through GET, URI_5 is called to retrieve the modifications applied on the predicted external temperature values obtained from the URI_3 (missing data manager). The

[3]Parameters are to be encoded in the URI or in the message body according to the HTTP format

[4]It is to note that the resources URIs (and later their descriptions) illustrated in this chapter are not yet available online

'#dataset' represents the id of the pre-processed data, modified by URI_3. The array 'CorrMPrExTemp' contains the modifications applied on the predicted external temperatures. Using GET, URI_6 is called to retrieve the modifications applied on the predicted external temperature values obtained from the URI_4 (outliers data manager). Similar to the previous step, the '#dataset' represents the id of the pre-processed data, modified by URI_4. The array 'CorrOPrExTemp' contains the modifications applied on the predicted external temperatures.
In our composition scenario, we considered that the merging of both URI_5 and URI_6 outputs, is being held on the client side to obtain the pre-processed external predicted temperatures array: [CorrExTemp{date, temp}]. URI_7 is invoked with POST to predict the energy heat consumption based on (i) the startdate and the enddate, representing the prediction period range, and (ii) the predicted internal temperatures with the pre-processed external temperature values previously collected. And finally URI_8 is called using GET to retrieve the predicted heat energy consumptions obtained from the URI_7 and represented by '#dataset'. The array 'PrHeatEngCons' in the responses contains the predicted values of the heat energy consumption.

Table 3.3 – Interfaces of the resources involved in the prediction process

URI	HTTP Verb	Parameters	Responses
1	GET	startdate=dd/mm/yyyy enddate=dd/mm/yyyy	200 OK [PrInTemp{date, temp}] {(POST, URI_7)}
2	GET	startdate=dd/mm/yyyy enddate=dd/mm/yyyy	200 OK [PrExTemp{date, temp}] {(POST, URI_3), (POST, URI_4)}
3	POST	[PrExTemp{date, temp}]	201 Created {(GET, URI_5)}
4	POST	[PrExTemp{date, temp}]	201 Created {(GET, URI_6)}
5	GET	#dataset	200 OK CorrMPrExTemp{date, temp}] {(POST, URI_7)}
6	GET	#dataset	200 OK [CorrOPrExTemp{date, temp}] {(POST, URI_7)}
7	POST	startdate=dd/mm/yyyy enddate=dd/mm/yyyy [PrInTemp{date, temp}] [CorrExTemp{date, temp}]	201 Created {(GET, URI_8)}
8	GET	#dataset	200 OK [PrHeatEngCons{date, HeatEngCons}] { }

3.5.3 Colored Petri Nets-based Formal Composition Model

As mentioned in Section 2.1.2, a resource can be invoked through HTTP methods to provide a specified functionality. It is published by a service provider, and located on a specific server. An exposed resource, res, has a set of inputs, a set of outputs, and a function assigned to it. In our modeling approach, a resource can be atomic or composed. In the CPN model, we define (i) an atomic resource as a single CPN with a single transition, input and output places, and (ii) a composed resource as a set of linked CPN representing linked resources. Before we formally define a resource, we define below the following sets:

- $\texttt{DataType} = \{\texttt{BasicT} \cup \texttt{ExtendedT}\}$ refers to the date types supported by a resource, such that:
  - $\texttt{BasicT} = \{\texttt{String}, \texttt{Integer}, \texttt{Real}, \texttt{Boolean}, \texttt{Date}\}$, denotes the basic data types known in programming languages
  - $\texttt{ExtendedT} = \{\texttt{Req}, \texttt{Status}\}$ denotes the extended data types defined to meet resources requirements
- $\texttt{Req} = (\texttt{HTTP x U})$ is the HTTP request sent to the resource URI, where:
  - $\texttt{HTTP} = \texttt{POST}|\texttt{PUT}|\texttt{DELETE}|\texttt{GET}|\texttt{HEAD}|\texttt{PATCH}|\texttt{CONNECT}|\texttt{OPTIONS}|\texttt{TRACE}$ is the HTTP method used to invoke the resource URI
  - $\texttt{U}$ is the resource URI based on the standard RFC3986[5]
- $\texttt{Status}$ = (Code, Desc) is the status of the resource response, where:
  - Code $\subseteq \mathbb{N}^*$ denotes the HTTP response status code
  - Desc represents the description of the HTTP code (e.g., 'Created', 'OK')

Definition 2. *A RESTful resource, res, is defined as res = (URI, N), where URI is the URI associated to res, and $N = (\Sigma, P, T, A, C, G, E, I)$ is a CPN such that:*

- $\Sigma \subseteq \texttt{DataType}$, *denoting the set of data types that the resource can process*
- *P is a finite set of input and output places of the resource, where:*
  - $P = P_{In} \cup P_{Out}$
  - $P_{In} = \bigcup_{i=1}^{m} \{p_{in_i}\} \mid \bigcup_{i=1}^{\mathbb{N}^*} res_i.P_{In}$, *such that:*
    - $\bigcup_{i=1}^{m} \{p_{in_i}\}$, *denotes the set of input places of an atomic resource, with $m \in \mathbb{N}^*$. As such, each resource requires at least one input place, representing the request sent to it. Other input places can be defined according to the resources needs, as the resources' parameters.*
    - $\bigcup_{i=1}^{\mathbb{N}^*} res_i.P_{In}$, *denotes the set of input places of a composite resource*
  - $P_{Out} = \bigcup_{i=1}^{n} \{p_{out_i}\} \mid \bigcup_{i=1}^{\mathbb{N}^*} res_i.P_{Out}$, *such that:*
    - $\bigcup_{i=1}^{n} \{p_{out_i}\}$, *represents the set of output places of an atomic resource, with $n \in \mathbb{N}^*$ and $n \geq 2$. As such, a resource requires two output places, denoting respectively the status response code and the set of the HTTP requests that can be sent to the next possible URI resources. Other places can be defined according to the resources needs, such as resources output results.*
    - $\bigcup_{i=1}^{\mathbb{N}^*} res_i.P_{Out}$, *denotes the set of output places of a composite resource*
- $T = t \mid \bigcup_{i=1}^{\mathbb{N}^*} res_i.T$. *t represents the functionality of an atomic r, whereas the union of T sub-resources represents the functionality of a composite resource.*
- *A is a finite set of arcs linking input places to transitions and transitions to output places, such that: $P \cap T = P \cap A = T \cap A = \phi$*
- *C is a color function. It associates a type from Σ to each place, where:*

[5] https://www.ietf.org/rfc/rfc3986.txt/

  - $\exists p \in P_{In}$, *such that* $C(p) \in$ `Req`
  - $\exists p_1, p_2 \in P_{Out}$, *such that* $C(p_1) \in$ `Status`, *and* $C(p_2) \in$ `Req`
- *G is a guard function. It maps the transition $t \in T$ to a boolean guard expression g. The resource can only be executed if g is evaluated to true.*
- *E is an arc expression function. It maps each arc $a \in A$ into an expression that may include variables.*
- *I is an initialization function that associates places to initial values*

Illustration

Based on our defined CPN formal model for RESTful service composition, we represent formally the composed resource of the prediction scenario, involving the resources defined in Table 3.3. Such a formal language can be directly applicable to other scenarios and represents the corresponding Web services as long as they are RESTful.
EnergyHeatPrediction = (URI, N), where URI is the address associated to the composed resource, and $N = (\Sigma, P, T, A, C, G, E, I)$ is a CPN such that:

- $\Sigma = \bigcup_{i=1}^{8} res_i.\Sigma$ with res_i denoting the resources involved in the prediction process, and where:
  - $res_i.\Sigma \subseteq$ `DataType`, designates the data types handled by the resources participating into the composition
- $P = P_{In} \cup P_{Out}$, where:
  - $P_{In} = \bigcup_{i=1}^{8} res_i.P_{In}$, such that:
    - $res_1.P_{In} = res_2.P_{In} = res_3.P_{In} = res_4.P_{In} = \{p_{in_1}, p_{in_2}\}$, denoting that each of these resources has 2 inputs.
    - $res_5.P_{In} = res_6.P_{In} = res_8.P_{In} = \{p_{in_1}\}$, denoting that each of these resources has 1 input.
    - $res_7.P_{In} = \{p_{in_1}, p_{in_2}, p_{in_3}\}$, denoting that this resource has 3 inputs.
  - $P_{Out} = \bigcup_{i=1}^{8} res_i.P_{Out}$, such that:
    - $res_1.P_{Out} = res_2.P_{Out} = res_5.P_{Out} = res_6.P_{Out} = res_8.P_{Out} = \{p_{out_1}, p_{out_2}, p_{out_3}\}$, denoting that each of these resources has 3 outputs.
    - $res_3.P_{Out} = res_4.P_{Out} = res_7.P_{Out} = \{p_{out_1}, p_{out_2}\}$, denoting that each of these resources has 2 outputs.
- $T = \bigcup_{i=1}^{8} res_i.T$, with $res_i.T$ = t denoting the specific functionality provided by each resource.
- A is a finite set of arcs linking input places to transitions and transitions to output places, such that: $P \cap T = P \cap A = T \cap A = \phi$.
- C is a color function. It associates a type from Σ to each place, where:
  - $\exists p \in P_{In}$, such that $C(p) \in Req$

- $\exists p_1, p_2 \in P_{Out}$, such that $p_1 \neq p_2$, $C(p_1) \in Status$, and $C(p_2) \in Req$

- G is a guard function. It maps the transition $t \in T$ to a Boolean guard expression g.
- E is an arc expression function. It maps each arc $a \in A$ into an expression that may include variables.
- I is an initialization function that associates places to initial values

Another resource composition example is shown in Figure 3.9, where the collected air temperature values are converted to "celsius" unit and aligned to a frequency of 600 seconds (10 minutes).

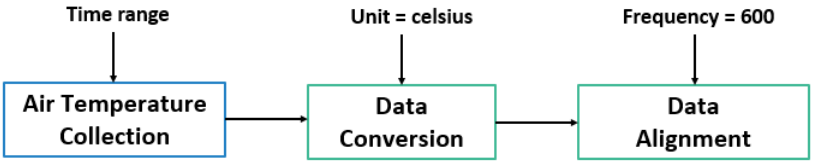


Figure 3.9 – Resource composition to convert and align the collected air temperature

Based on our CPN-based formal model, the resulted composition is represented as: **AirTempConvAlign** = (URI, N), where URI is the address associated to the composed resource, and $N = (\Sigma, P, T, A, C, G, E, I)$ is a CPN such that:

- $\Sigma = \bigcup_{i=1}^{3} res_i.\Sigma$, with res_1, res_2, res_3 refers respectively to the Air Temperature Collection, Data Conversion, and Data Alignment resource, and where:
  - $res_i.\Sigma \subseteq$ `DataType`, designates the data types handled by the resources participating into the composition (e.g., "String" type related to the Unit input of the Data Conversion resource, and "Integer" type related to the Frequency input of the Data Alignment resource).
- $P = P_{In} \cup P_{Out}$, where:
  - $P_{In} = \bigcup_{i=1}^{3} res_i.P_{In}$, such that:
    - $res_1.P_{In} = \{p_{in_1}, p_{in_2}\}$, denoting that the resource res_1 has 2 inputs, i.e., the Time range and the Req type input.
    - $res_2.P_{In} = res_3.P_{In} = \{p_{in_1}, p_{in_2}, p_{in_3}\}$, denoting that each of the resources res_2 and res_3 have 3 inputs, i.e., the necessary data (data array), the required parameter (the Unit or the Frequency), and the Req type input.
  - $P_{Out} = \bigcup_{i=1}^{3} res_i.P_{Out}$, such that:
    - $res_1.P_{Out} = res_2.P_{Out} = res_3.P_{Out} = \{p_{out_1}, p_{out_2}, p_{out_3}\}$, denoting that each of the resources res_1, res_2 and res_3, has 3 outputs, i.e., the obtained data (data array), the Status type output, and the Req type output.

- $T = \bigcup_{i=1}^{3} res_i.T$, with $res_i.T$ = t denoting the specific functionality provided by each of the resources: res_1, res_2, and res_3.
- A is a finite set of arcs linking input places to transitions and transitions to output places.
- C is a color function. It associates a type from Σ to each place.
- G is a guard function. It maps the transition $t \in T$ to a Boolean guard expression g.
- E is an arc expression function. It maps each arc $a \in A$ into an expression that may include variables.
- I is an initialization function that associates places to initial values

In Figure 3.10, we show the graphical CPN-based model of the **AirTemp-ConvAlign** resource composition.

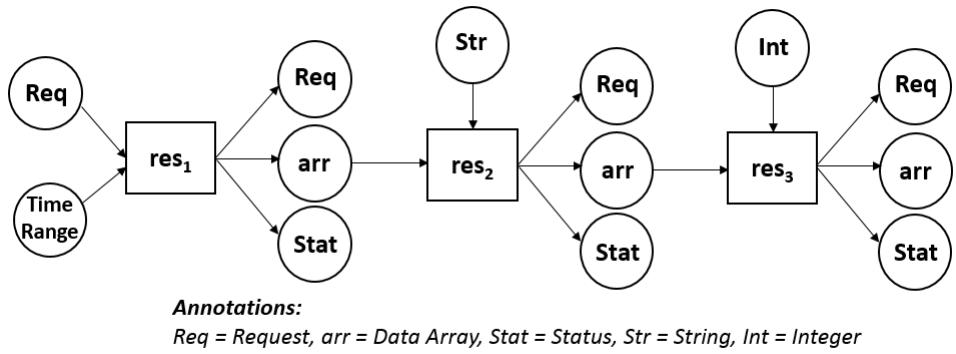


Figure 3.10 – CPN graphical model of the "AirTempConvAlign" composed resource

We note that a URI is assigned to each composed resource after the verification of the composition behavior that will be discussed in the following section.

3.5.4 Composition Behavioral Properties in CPN

Modeling RESTful services with CPNs format allows to analyze several CPN-based behavioral properties of the composition. As such, mapping RESTful services and composition to CPNs with some extensions, enables the execution of the algorithms related to CPNs properties. These algorithms still apply in our extended formal model, as it will be shown in this section.

As it is presented in Section 3.2, three properties have been considered important to verify in this thesis: Reachability, Liveness, and Interoperability. In the Reachability and Liveness definitions below, we use (N, M_0) to denote a Petri net, N, with its initial Marking, M_0. The Petri net marking, M, designates the state of the net, which corresponds to the assignment of tokens to places. The initial marking M_0 designates the availability of some data (tokens) in the input places of one or more resources involved in the composition, before launching the composition execution.

Definition 3. ***Reachability*** *- A marking M_n is reachable from M_0 in a Petri net N, if there exists a sequence of transitions that can be fired from M_0 to M_n.*

One of the challenges in the composition process is to make sure that the final desired state is reachable from the initial state. To verify that the result is reached, we use the Reachability graph.

Definition 4. ***Reachability Graph (RG)*** *- It is a set of all the reachable markings of a Petri net represented as nodes. The nodes are connected with arcs designating the firing of a transition.*

The Reachability graph algorithm, inspired from [82], is described as:

Algorithm 1 : Reachability Graph

```
1 label the initial marking M0 as the root and tag it "new"
2 while "new" markings exist do
3    select a new marking M
4    if no transitions are enabled at M, tag M "dead-end"
5 while there exist enabled transitions t at M do
6       obtain the marking M' that results from firing t at M
7       if M' does not appear in the graph, add M' and tag it "new"
8       draw an arc with label t from M to M' (if not already present)
```

In our scenario, the Reachability property is true when: $\exists\ M_0$ and $\exists\ M \in$ RG as the end node, with M designating the final desired state.

Definition 5. ***Liveness*** *- A Petri net (N, M_0) is considered to be L_k-live if every transition t in the net is L_k-live. t is said to be:*

- *L_0-live, if it can never be fired in any firing sequence. In this case, the transition, t, is deadlocked.*
- *L_1-live, if it can be fired at least once in some firing sequence*
- *L_2-live, if it can fire arbitrarily often*
- *L_3-live, if it can fire infinitely often*
- *L_4-live, if it always fire*

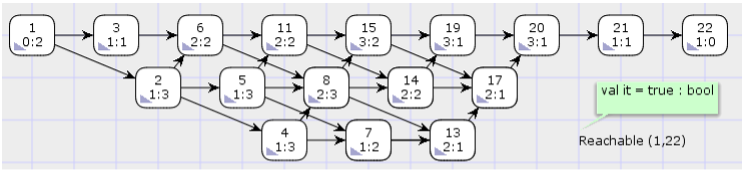
Another challenge in our composition is to verify that the resources involved in the composition process will eventually be executed at least once. If, for example, the pre-processing resources responsible of correcting erroneous data are not executed, the prediction resource will predict the building heat energy consumption based on inaccurate data. This will affect negatively the prediction results quality. Therefore L_1-live was our main focus, to ensure that all CPN transitions will eventually be fired by progressing through further allowed firing sequences. We note that transition firing depends on the availability of tokens (data) in all its input places. Thus, a resource can be executed only if its input places contain the required data which are: HTTP_VERB, URI, and some parameters (when it is necessary). In our composition, a transition t is L_1-live when: $\exists\ M_0$ and $\forall\ t \in T\ in\ N, t \in$ RG. If not, t is considered dead.

By verifying both the Reachability and Liveness properties, we can ensure that the composition contains no loop and all resources receive the required input in order to be executed.

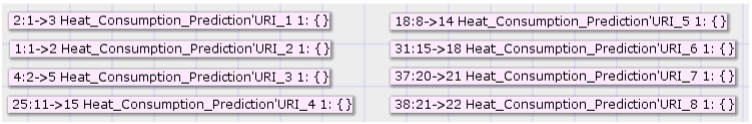
As for the Interoperability, by definition the CPN formalism put the following as a constraint: $\forall a \in A : [C(E(a)) = C(p)]$. This means that the CPN workflow execution will not be possible unless data flowing to and from a place are of the same type. We note that data flowing to a place corresponds to a transition output, while data flowing from a place denotes the input of the next transition.

3.6 Experimental Illustration

We illustrate our proposed CPNs-based formal composition approach within the composition scenario presented in Figure 3.2, using the CPN tools[6], one of the most known tools for editing, simulating, and analyzing Colored Petri Nets models. CPN tools provides a graphical user interface (GUI) with tool palettes and marking menus, to build the CPNs models. Moreover, it features syntax checking while the workflow is being constructed, and generates a standard state space report that contains information about behavioral properties of the modeled system. Figure 3.12 represents the CPN-based model of the prediction scenario implemented according to our formalism. As it is illustrated, the model includes all the resources involved in the energy prediction process. During the tests, we extended the input places related to URI_7 to respect the HATEOAS principle. In fact, due to the existence of several resources (URI_1, URI_5 and URI_6) that point out to URI_7 in their next resources to follow, we linked each of these resources to URI_7 transition. Using the state space tool of the CPN tools, and by implementing queries via ML code (the functional programming language of the CPN tools), we were able to verify Reachability and Liveness properties. As for Interoperability prop-



(a) Reachability graph of the prediction process



(b) Examples of the Reachability graph arcs labels

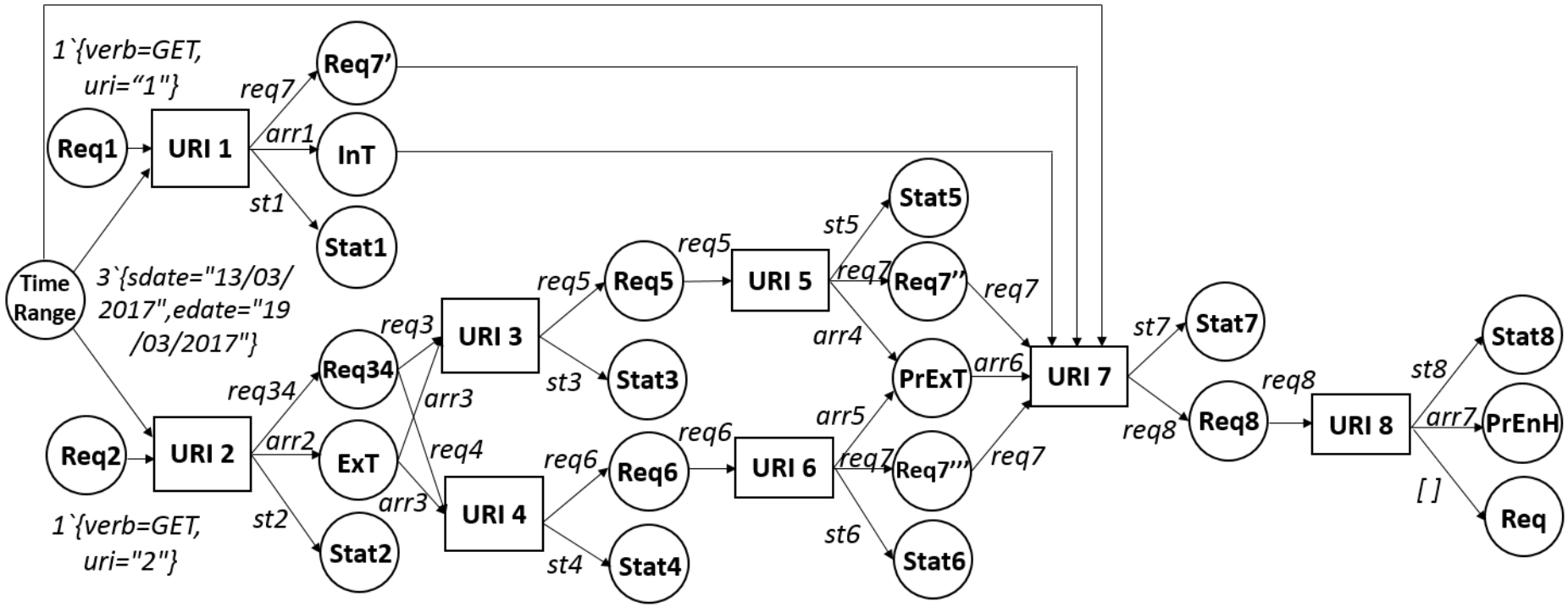
Figure 3.11 – Reachability graph

erties, it is verified automatically during composition construction. Below are the tests applied to the composition scenario to verify the corresponding properties:

- **Reachability Test:** Figure 3.11(a) shows the Reachability graph of our scenario, containing a node for each reachable state. In total we have 22 states, with node 22 representing the final state that is the prediction output results (PrEnH). Moreover, we used the 'Reachable (1,22)' boolean function (written in ML code) to test if state 22 is reached from state 1. The returned boolean value equal to "true" verifies the Reachability property.

- **Liveness Test:** Liveness property is verified through analyzing the Reachability graph arcs, which are labeled by the resources responsible of the state changing. Figure 3.11(b) shows examples of some Reachability graph arcs labels, appeared when clicking on the arcs. It proves

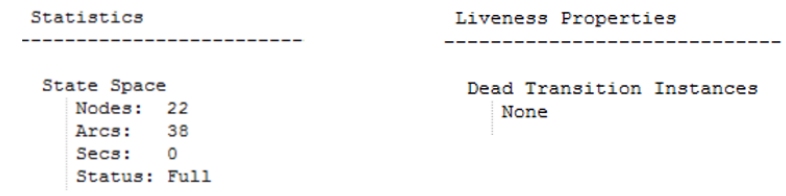
[6] `http://cpntools.org`



Annotations:

Req = Request, InT = Internal Temperature, ExT = External Temperature, Stat = Status, arr = Data Array, PrExtT = Processed External Temperature, PrEnH = Predicted Energy Heat, [ ]= Empty List

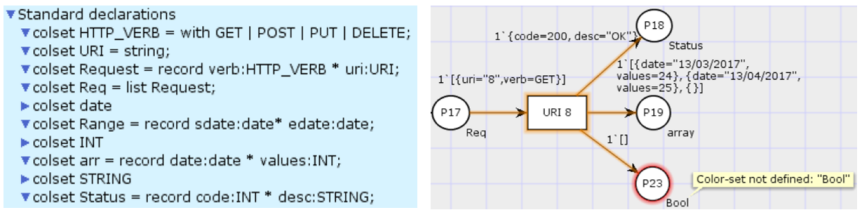
Figure 3.12 – CPN model for the resources composition relative to the prediction scenario



(a) State space report statistics *(b) Dead transitions*

Figure 3.13 – Information retrieved from the state space report

that all URIs (from 1 to 8) are executed at least once. Moreover, and when generating the state space report through the state space tool of the CPN tools, several information can be retrieved including Liveness property results. Figure 3.13(a) for example shows the statistics representing the number of nodes and arcs of the composition scenario, and Figure 3.13(b) proves that there are no dead transition instances, denoting that all the resources will be eventually executed starting from the initial state.



(a) Color sets used to model the required data types *(b) Interoperability Issue*

Figure 3.14 – Defined color sets and interoperability issue

- **Interoperability Test:** In order to represent the flowing data types between resources, we defined the color sets as shown in Figure 3.14(a). Using CPNs, our approach allows to verify the Interoperability property between the linked resources. CPN tools check the syntax of the nets during their construction where errors, such as in data types, can be visually seen through specific color indications, as shown in Figure 3.14(b).

3.7 Developed Prototype

In this section, we present the technical details of our prototype developed as a standalone module for the Web management platform in the context of SIBEX project, to verify resource compositions. The prototype, implemented in Python[7], allows the modeling, verification, storage, and execution of composed resources, involving resources belonging to two categories: Data collection and pre-processing (resources related to the advanced data processing were not included in the prototype). The prototype consists of several components:

- Four engines:

[7] `https://www.python.org/`

  - Modeling engine, used to model the required resource composition into a CPN format, i.e., Petri Net Markup Language (PNML), based on our proposed CPN formalism (see Section 3.5), which is integrated in the Resource CPN Mapper (see Figure 3.8).
  - Validation engine, responsible of verifying the composition correct behavior according to several CPNs properties (i.e., Reachability, Liveness, and Interoperability).
  - Conversion engine, used to transform the verified composition PNML model into a suitable format (JSON-LD in this work) that can be stored in the platform service data model, i.e., ontology.
  - Execution engine, used to execute the verified and stored resource composition.

- REST APIs[8]: Some are used to access the prototype functionalities (e.g., modeling engine and execution engine), and others to access the core platform resource descriptions.

- A service ontology, expressed using Hydra vocabulary in JSON-LD, which is mainly used to describe the properties of the available resources (e.g., provided functions, inputs, outputs, etc.) used for collecting and pre-processing building data, and store the properties of the new composed resources.

In order to understand the different functionalities of the prototype components, we describe in Figure 3.15 the related sequence diagram that covers the set of interactions applied between them.

3.7.1 Engines Specifications

Before elaborating on the specifications of each engine, we will present a composition use case identified in SIBEX, and explain how it can be modeled before it is sent to the prototype for verification, storage, and execution. As shown in Figure 3.16, the composition mainly allows one of SIBEX advanced modules providers (i.e., relative to the Energy Consumption Prediction module) to prepare the required data, before it is used by the module. The module requires: (1) collecting several data: Air Temperature, Power Energy, and Light Radiation, (2) converting the data to the required unit (for instance, convert the air temperature values from Fahrenheit to Celsius), and (3) aligning the data to the same frequency (expressed in seconds), before predicting the energy consumption of a particular zone of a building (zoneId='A') according to a specific date (i.e., startDate−endDate).

In order to represent the required composition, we use a model based on JSON, which is understandable by both machines and humans, and can easily be sent to and from a server. In JSON, data is expressed through keyword/value pairs, separated by a comma. Curly braces "{}" hold objects, while square brackets "[]" hold arrays. The resource model that we propose, includes the keyword "composition" containing the "resources" element. "resources" is an array of objects designating several resources (in parallel or in

[8]They use HTTP requests to GET, PUT, POST and DELETE data

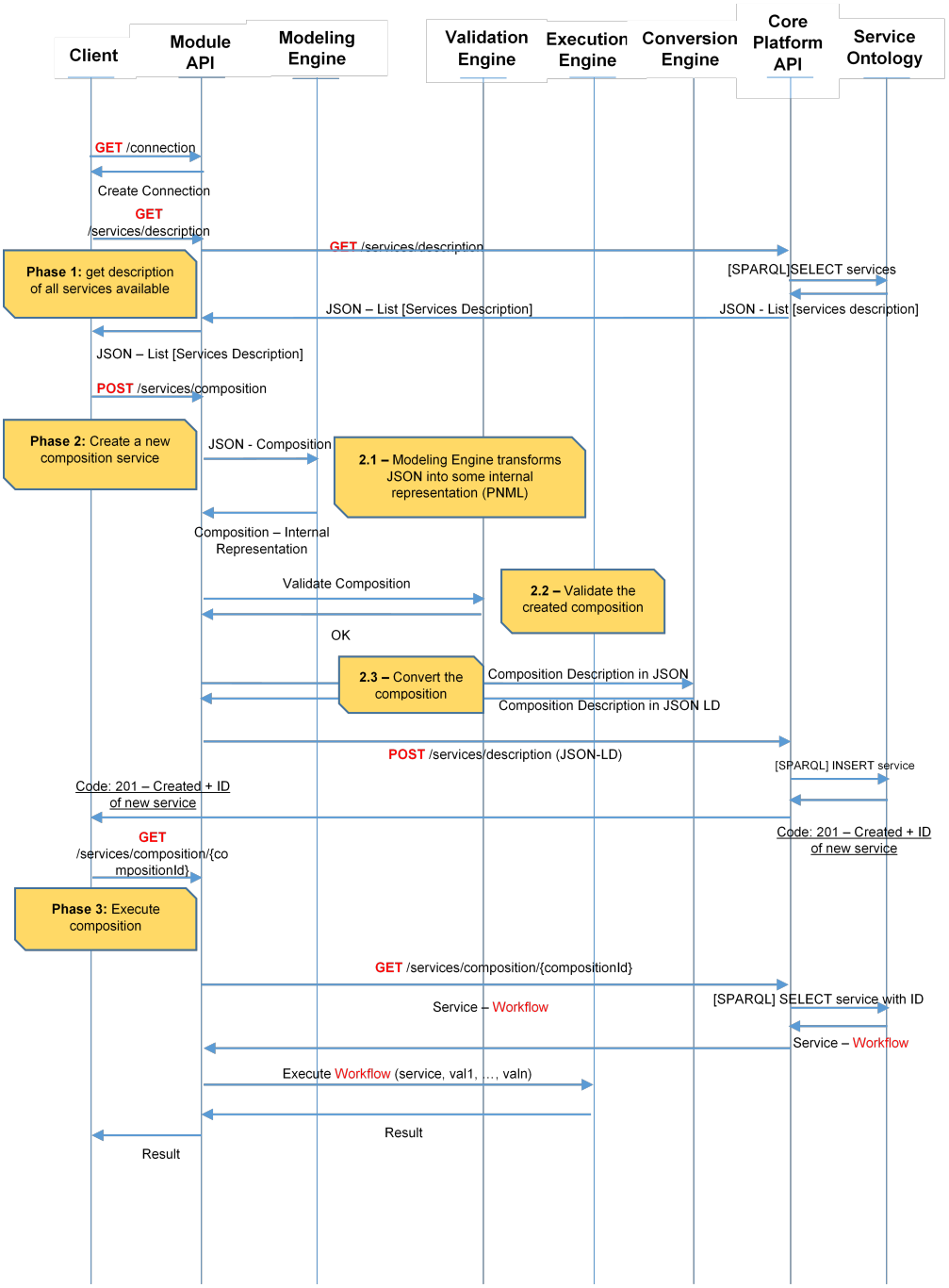


Figure 3.15 – The sequence diagram showing the interaction of the resource composition prototype components

sequence). Each resource involved in the composition process is represented by the following properties:

- "URL": refers to the public address of the resource
- "Method": refers to the HTTP verb by which the resource is called
- "Name": refers to the name of the resource
- "Param": includes the input parameters required to run the resource. "Param" is an array that can contain multiple objects.

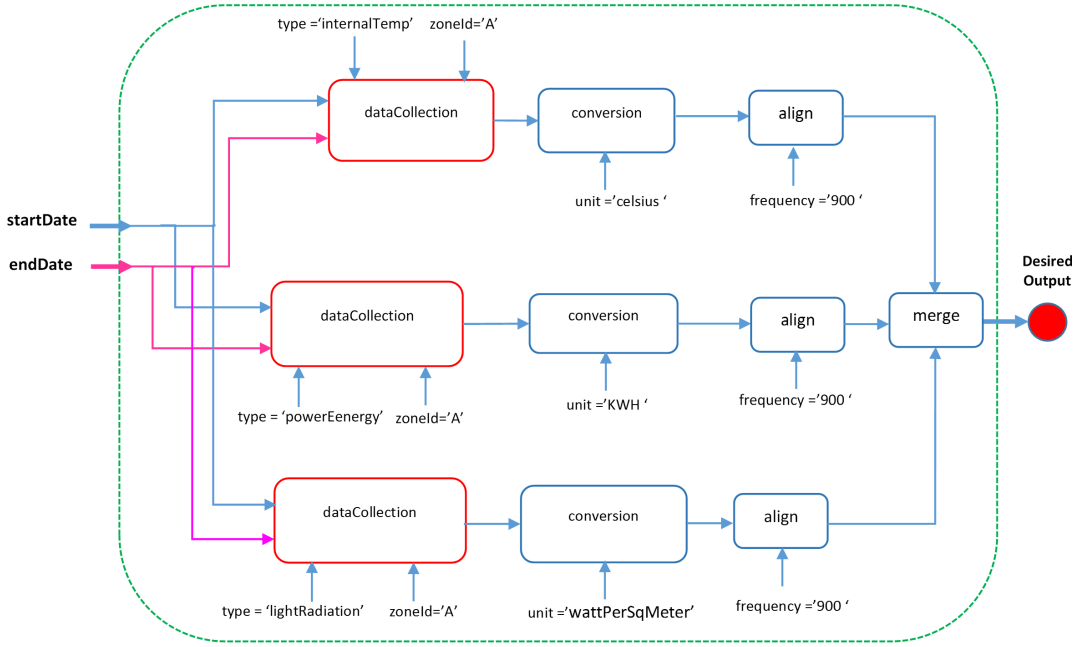


Figure 3.16 – Example of a composition use case in SIBEX

- "Output": refers to the name of the variable where the result of the resource will be stored.

  In the composition modeling, we distinguish between fixed and non-fixed parameters:

  1. Fixed parameter values cannot be changed after creating the composed resource. Their values are given when modeling the composition.
  2. The non-fixed parameters, defined in the "variables" keyword of the JSON model, are not provided during the modeling of the composition. Their values will be affected when executing the composition, after validation and storage.

The link between the resources is done by assigning the value of the keyword "output" of a resource to a value of a parameter, presented in "param", of another one, as illustrated in Figure 3.17.

Figure 3.17 – Link between resources

During the modeling of the composition, the user (as the module provider) must define the name and functionality of the composition using the following two keywords: "name" and "description". We note that each resource in the prototype has only one output, and the composition is supposed to generate a single result designated by the element "goal". "goal" is the same output of the last resource involved in the composition. As an example, we represent in Appendix F, the resource composition modeling of the composition scenario of Figure 3.16, while respecting the defined JSON model.

3.7.1.1 Modeling Engine

The modeling engine receives the composition's JSON format, checks if the JSON model is syntactically correct, and verifies if it contains one single goal. If the JSON format is valid, it is translated into Petri Nets Modeling Language (PNML), using the Snakes toolkit[9] [93], which is used to build Petri nets models in Python. The PNML is then passed to the validation engine. If the JSON format is not valid, a report error is sent to the user (i.e., the composition designer).

3.7.1.2 Validation Engine

Since the user (as the module provider in Figure 3.16) builds his composition manually, he is susceptible to make mistakes (e.g., create infinite loops that prevent the execution of the rest of the resources, link resources that are not compatible with the same data type, etc.), which affect negatively the behavior of the composition. To cope with these problems, the validation engine verifies the behavior of the composition before its execution, according to several CPN properties: Reachability, Liveness, and Interoperability. In order to verify these properties, the verification engine relies on neco[10] library in Python, which allows compiling the Petri nets models (as PNML) and verifying the desired formal properties. Once the composition is verified, it is passed to the Conversion Engine. If not, a report guiding the user on the composition design errors(s) is sent.

3.7.1.3 Conversion Engine

After being verified, the composition is converted to a specific format, JSON-LD (explained later in Section 3.7.2), that can be stored in the service ontology, with:

- An assigned URL
- The composed resource category (i.e. composed)
- The necessary HTTP-Verb: GET
- The composed resource name and description which are given by the user
- The required inputs (matching the non-fixed parameters)
- The resulted output (matching the composition "goal")
- The internal workflow of the composed resource

The stored composition description can help in using the new created composed resource in other composition scenarios.

Appendix G presents the composed resource description, expressed using Hydra in JSON-LD, related to the composition in Figure 3.16.

[9] http://code.google.com/p/python-snakes/

[10] `https://github.com/Lvyn/neco-net-compiler/`

3.7.1.4 Execution Engine

The role of the execution engine is to call the resources involved to create the new composed resource. These resources are presented in the internal workflow of the composed resource description. The execution is done through forming a URL-based list related to all of the resources used in the composition (HTTP VERB + Corresponding URL) with their required inputs.

3.7.2 Data Model for RESTful Services

In order to capture the functional and non-functional properties of a resource, it is important to describe resources using a description language that can be processed by a machine. Such description can guide the user in the selection of the appropriate resources involved in the composition, and helps in retrieving the behavior (i.e., set of resources to invoke) of a composed resource during its execution.

As explained in Section 2.2.4, the Hydra vocabulary has been chosen to describe the resources in this thesis, in JSON-LD format. The later is used to express resource descriptions thanks to its ability to be easily understood by both humans and machines, its easy processing, and most importantly its simple linking to existing data models. In the service data model that contains resource descriptions, each resource (elementary or composed) is represented by the following properties, inspired from Hydra:

- "@id": designates the Unique Resource Locator of the resource;
- "@type": refers to the resource category (i.e., dataCollection, pre-processing, or composed);
- "description": describes the main purpose of the resource;
- "title": refers to the name of the resource;
- "operation": designates an array of all the operations provided by the resource:
  - "method": refers to the HTTP verb used to call the operation, e.g., GET, POST, PUT, and DELETE;
  - "expects": denotes the set of input parameters used to run the operation;
  - "returns": denotes the output parameters resulted from the operation execution;
  - "acronym": refers to the functionality abbreviation (for example: "DCV" for Data Conversion, and "DC" for Data Collection). This field will essentially be used in the automatic composition approach, and not in this chapter where the composition is built statically by the user.
  - "Workflow": designates the internal workflow of a composed resource.

The "Workflow" element will be empty for elementary resources. As for the composed ones, it contains an array named "member" that describes all

the resources used to build the required composition, and a "goal" referring to the composition output that represents the same output of the last resource involved in the composition. The "data" element used in the inputs (expects) and output (returns) of the related members (resources) in the "Workflow", serves in linking the resources to each other.

For each member in a composition "Workflow", there are:

- "id": referring to the resource identifier. As such, each involved resource into the composition should have a unique identifier. In this way, even if in the composition process there are three resources invoked having the same "url", they can be distinguished by their unique id;
- "url": designating the URL address of the resource;
- "method": referring to the HTTP verb used to call the resource operation;
- "expects": designating the set of input parameters used to run the resource operation;
- "returns": containing the name of the variable where the resource result is stored. This variable is used to link resource between each other.

Figure 3.18 shows the structure of the resource description JSON-LD document.

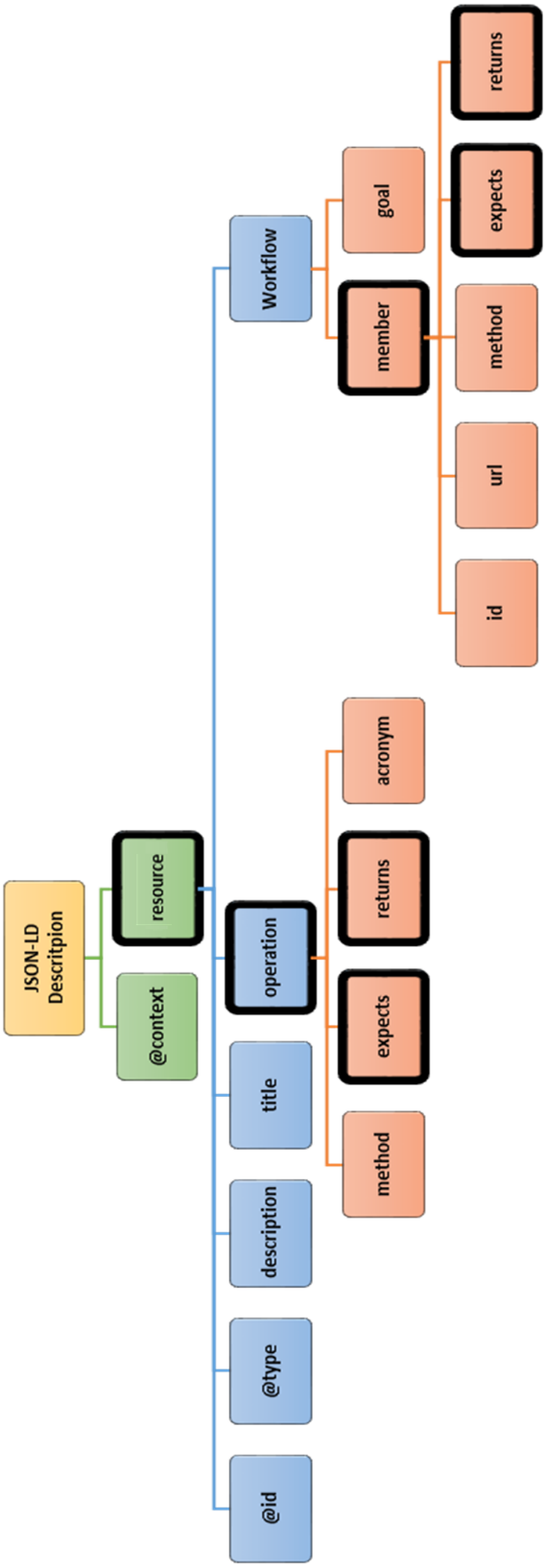


Figure 3.18 – Structure of the resource description JSON-LD document

Appendix H lists the description of all the elementary resources related to data collection and data pre-processing in the SIBEX Web platform, using Hydra in JSON-LD.

3.7.3 Implemented APIs

In order to access the functionalities of the developed resource composition prototype components, we implemented several APIs. The later are used by the user in order to:

- List the description of all resources available in the SIBEX Web platform. This aids the user in knowing the resources that can be selected during his composition modeling.
- Create a new resource composition
- Store the composed resource
- Execute the composed resource

For example, Table 3.4 and Table 3.5 presents respectively the request for creating a resource composition, and the correspondent response.

Table 3.4 – Request to create a resource composition

Method	URL
POST	/resources/composition

Type	Parameters	Values
HEAD	UserID	string
HEAD	SourceID	string
HEAD	ProxyToken	string
HEAD	Options	List[string]

POST	{
	"composition":{
	"name",
	"description",
	"variables":[],
	"resources":[{
	"url": "",
	"method": "",
	"name": "",
	"param": [{"key": "value"},
	{etc.}],
	"output": ""
	}],
	"goal": ""
	}
	}
		
		string
		string
		List[string]
		
		string
		string
		string
		List[string]
		
		string
		
		string

The developed APIs are listed in Appendix I.

3.7.4 Tests

We test our resource composition prototype implemented using Python, on a Linux Debian (64 bits) virtual machine, with 1 dedicated Intel® Core™ i7-46000 CPU @ 2.10GHz 2.70GHz processor, and 1 GB RAM. The source code of the prototype can be accessible online[11]. We first implemented the resources exposed by the SIBEX platform using Flask[12], which is a Web framework that provides tools, libraries and technologies allowing to build Web

[11] https://github.com/larakallab/RESTfulComposition/

[12] https://pymbook.readthedocs.io/en/latest/flask.html/

Table 3.5 – Response for creating a resource composition

Status	Response	
201	{"compID": compID}	
400	{" bad request ":"Invalid options."}	
400	{" bad request ":"Invalid type."}	
401	{" unauthorized ":"Invalid proxy token."}	
401	{" unauthorized ":"Unknown user."}	

403	{
	ValidationResults:{
	Reachability,
	Liveness,
	Interoperability
	}
	}
		
		boolean
		boolean
		boolean

500	{" error ":"Internal error."}	500

services. Then, we uploaded resource Hydra-based descriptions expressed in JSON-LD, as shown in Section 3.7.2, into an RDF graph in Apache Jena Fuseki[13], which is a SPARQL server that provides REST-style SPARQL HTTP Update, SPARQL Query, and SPARQL Update using the SPARQL protocol over HTTP.

After developing each required engine (i.e., modeling engine, validation engine, conversion engine, and execution engine), we conducted several tests on different composition JSON-based models, to: (1) check their correct JSON syntax, and (2) verify the different behavioral properties (i.e., Reachability, Liveness, and Interoperability).

3.7.4.1 Syntax Checking

Here, the purpose is to ensure the correct syntax of the composition model expressed with JSON. To do so, we verify whether the composition model contains a single key "goal", and check the validity of the JSON format (e.g., the use of double quotes when defining a key and preventing to have duplicated keys).

A. **Single "goal"**

Considering the composition model defined in Listing 3.1, in which there is no single key "goal" expressed.

```
{
   "composition": {
      "name": "Conversion of collected data",
      "description": "This composition collects internal
         temperature data from bldg A and converts it to the
         desired unit",
      "variables": ["startDate", "endDate"],
      "resources": [{
            "url": "http://sibex/measures",
            "method": "GET",
            "name": "measures",
            "param": [{
                  "type": "internalTemp"
               },
               {
                  "zoneid": "BldgA"
               },
               {
```

[13] https://jena.apache.org/documentation/serving_data/

```
                    "startdate": "startDate"
                },
                {
                    "enddate": "endDate"
                }
            ],
            "output": "A"
        },
        {
            "url": "http://sibex/dataconversion",
            "method": "GET",
            "name": "Data conversion",
            "param": [{
                "data": "A",
                "unit": "celsius"
            }],
            "output": "B"
        }
    ]
  }
}
```

Listing 3.1 – Composition model defined without a single key "goal"

When the modeling engine receives such composition model example, an error message is returned:

*** Please specify a "goal" for the composition ***

B. **Correct syntax**

For the correct syntax checking, we defined the composition model as shown in Listing 3.2, where the "param" key of the internal temperature collection resource is single quoted and not double quoted.

```
{
  "composition": {
    "name": "Conversion of collected data",
    "description": "This composition collects internal
        temperature data from bldg A and converts it to the
        desired unit",
    "variables": ["startDate", "endDate"],
    "resources": [{
          "url": "http://sibex/measures",
          "method": "GET",
          "name": "measures",
          'param': [{
                "type": "internalTemp"
            },
            {
                "zoneid": "BldgA"
            },
            {
                "startdate": "startDate"
            },
            {
                "enddate": "endDate"
            }
        ],
        "output": "A"
    },
    {
```

```
26                "url": "http://sibex/dataconversion",
27                "method": "GET",
28                "name": "Data conversion",
29                "param": [{
30                    "data": "A",
31                    "unit": "celsius"
32                }],
33                "output": "B"
34            }
35        ]
36    }
37 }
```

Listing 3.2 – Composition model with erroneous JSON syntax

For such kind of syntax error, the following error message is returned:

*** The syntax of the JSON model is wrong. Possible mistakes: Errors in the brackets | braces | commas | quotes Duplicated keys ***

3.7.4.2 Behavior Properties Verification

The main objective of this section, is to verify the correct behavior of different composition models according to the Reachability, Liveness, and Interoperability properties.

A. **Reachability**

In this test, we consider the composition model example depicted in Figure 3.19, where the data alignment resource (aligning the light radiation data) is connected to the same merging resource input used by the previous alignment resource (aligning the power energy data).

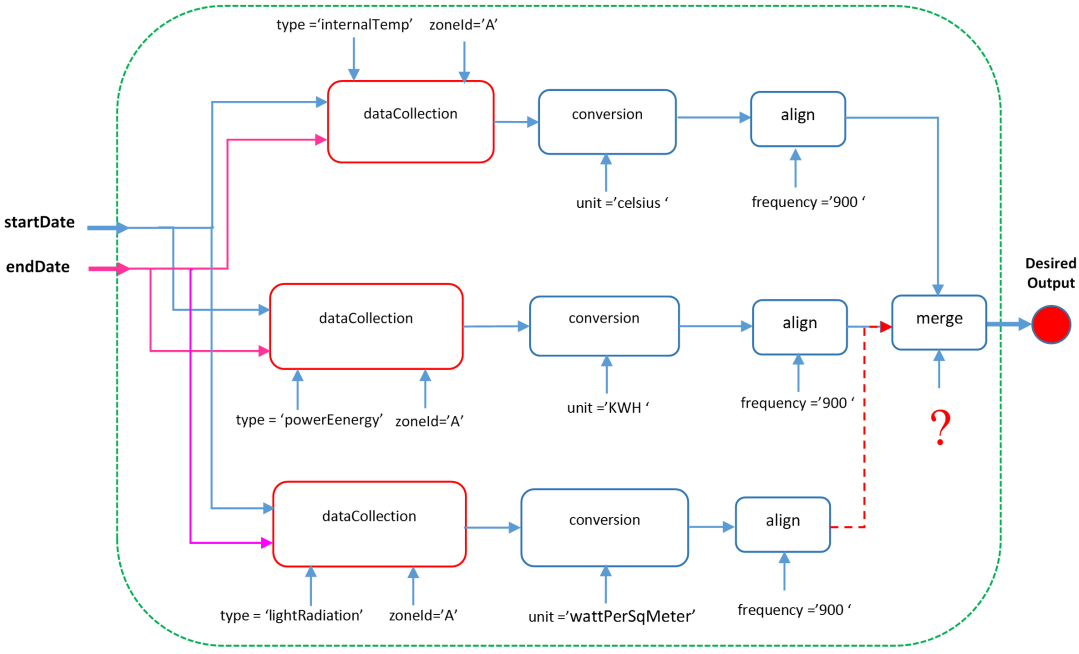


Figure 3.19 – Composition model with a non reachable final state

When verifying such composition, and since the merging resource misses one input, it will not be executed, and the "goal" will not be reached. Thus, the validation engine returns the following error message:

*** Final state is not reachable***
***Not all resources are properly linked ***

B. **Liveness**

In this case, we consider the composition model depicted in Figure 3.20, where an end-loop is occurred to the data conversion due to a linking error. Such loop prevents the data alignment resource from execution (dead resource). When verifying such composition model, the valida-

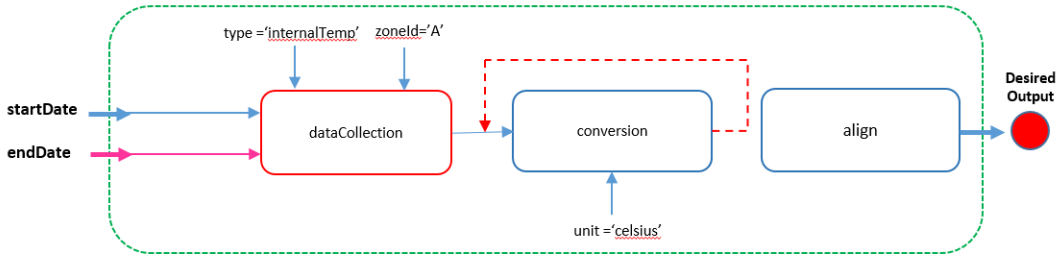


Figure 3.20 – Composition model with an end-loop

tion engine returns the following error message:

*** Final state is not reachable***
***Not all resources are properly linked ***

C. **Interoperability**

For the Interoperability property, we considered the composition model illustrated in Figure 3.21, where two data collection resources are linked.

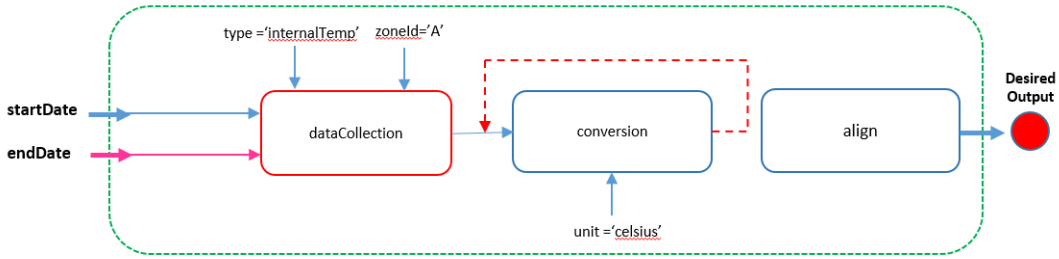


Figure 3.21 – Composition model with non interoprable resources

Due to the mismatch of the I/O datatypes between the data collection resources, the validation engine returns the following message error:

*** Some resources cannot be linked ***

If the resource composition is valid (i.e., there are no identified syntax errors in the composition model, and the behavior properties are verified), the validation engine confirms the following:

*** Resource composition is verified ***

After the composition validation, the conversion engine converts the modeled composition to a Hydra-based description expressed in JSON-LD format, which can be stored into the platform service ontology, and assigns to it a new URL. The user can then execute the composition by calling the corresponding URL, and giving values to the necessary parameters, which are non-fixed parameters as it was defined in the "variables" key of the modeled composition.

Listing 3.3 shows the results of the converted and aligned collected data defined in the composition modeled in Figure 3.16. The results are obtained based on the given parameters: startDate and endDate, i.e, "2018-11-15T09:00-:00" and "2018-11-15T11:00:00" respectively, which are non-fixed parameters as it was defined in the composition model (see Appendix F).

```
{
  "data": [
    ["2018-11-15T09:00:00", 26, 2283, 631],
    ["2018-11-15T09:15:00", 26, 2299, 597],
    ["2018-11-15T09:30:00", 25, 2321, 690],
    ["2018-11-15T09:45:00", 27, 2280, 481],
    ["2018-11-15T10:00:00", 25, 2024, 675],
    ["2018-11-15T10:15:00", 28, 2407, 554],
    ["2018-11-15T10:30:00", 26, 2182, 462],
    ["2018-11-15T10:45:00", 27, 2315, 435],
    ["2018-11-15T11:00:00", 30, 2299, 458]
  ]
}
```

Listing 3.3 – Example of composition results after execution

3.8 Summary

In this chapter, we proposed a solution for verifying the behavior of statically composed RESTful services (resources) before execution. This is done to enable composition designers (end-users) to detect erroneous behavior before actual composition run, and avoid unnecessary composition execution. The solution, which is related to the verification process of the static resource composition in the StARC framework (see Figure 1.10), is based on Colored Petri Nets (CPN), (i.e., a language for the modeling and validation of concurrent and distributed system). In the chapter, we first exposed our CPN-based formalism to model the behavior of resources with their composition. And then, we showed how the verification of composition behavior properties that we identified important to consider (i.e., Reachability, Liveness, and Interoperability) can be applied based on our formal defined model. The work has been illustrated in CPN tools to verify a composition scenario that was defined in the context of the BEMServer Web platform (presented in Chapter 1), proving the applicability of our proposed model. A prototype has been also implemented (within the context of SIBEX project) to verify resource compositions built using data collection and pre-processing resources. This is done through the development of different engines used for: modeling, validating, converting, storing and executing new composed resources.

In the next chapter, we tackle the challenge of resource discovery in hybrid environments (connecting static and dynamic resources), while considering resource location (whenever they are exposed by objects).

Chapter 4

Automatic Location-aware Resource Discovery for Hybrid Web Environments

"Discovery is the journey; insight is the destination"

Gary Hamel

RESTful services (resources) have seen their popularity rising and have shown their potential in composing reliable Web-scale environments (Web applications, Web platforms, Web of Things (WoT), etc.). However, discovering the necessary resources for a composition is becoming more challenging. This is due to: (1) the growing number of published Web resources, (2) the highly dynamic nature of the WoT environment, in which connected objects (devices) exposed as resources, can be connected/disconnected at different instances, and (3) the different locations of the connected objects affecting the accuracy of the collected data. In this chapter, we present an automatic resource discovery, that can be applicable in hybrid Web environments (providing static linked resources established to be always available, and dynamic resources, that can be connected to and removed from the environment at different time periods). The solution is part of the automatic resource composition presented in the proposed StARC framework (see Chapter 1). In the proposed solution, we define a formal representation that models the resources (static and dynamic) in a single graph, and allows the discovery of several resources realizing the same required function using adapted graph-based algorithms. The algorithms use semantic annotations integrated within resource descriptions expressed with Hydra. We also propose a 3-dimensional indexing schema that considers object location for resource discovery, and enhances resource search in large resource environments. Experiments were conducted to evaluate the performance of our solution and showed good results on 4 aspects: dynamicity, multiplicity, efficiency, and scalability.

4.1 Introduction

Nowadays, a plethora of Web environments (Web applications, Web platforms, etc.), publish their functions as RESTful services (resources) that follow the REST architectural style principles (see Section 2.1.2). As the Web has become a major medium of communication, integrating objects (e.g., smart devices) into the Web and taking advantage of its open standards (e.g., REST and HTTP), has created an emerging trend: the Web of Things (WoT) [17]. In the WoT, objects can be (i) stationary (having invariant location), or (ii) mobile (their position changes over time), and are abstracted as resources. A resource, identifiable by a URI, is either (i) **static**, established to be always connected to the environment, or (ii) **dynamic**, connected to and removed from the Web environment at different time periods. Figure 4.1 shows the relations between the type of resources provided by the connected objects and published Web applications. In many cases, a single resource is not

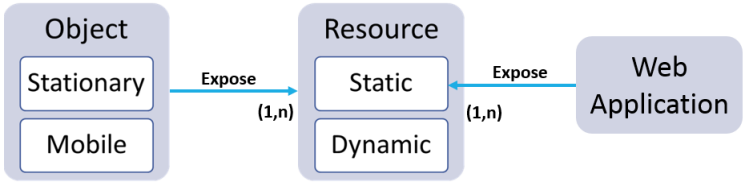


Figure 4.1 – Type of resources exposed by connected objects and Web applications

sufficient to satisfy specific user's requests, and often, resources need to be combined forming a composition that achieves the desired results. However, discovering the necessary resources for a composition remains a challenge, and is becoming even more complex due to: (1) the large number of resources connected to the Web [115], (2) the highly dynamic nature of the WoT resources [11], and (3) the mobility aspect of the resources exposed by objects. Lately, automatic resource discovery has emerged as an active research area [80], where several challenges still need to be addressed:

- **Discover dynamic resources connected at runtime**: With respect to the HATEOAS [45] principle, hypermedia resources' links are included in resources' responses, during their design, to define the next possible resources to call, thus, forming a graph of linked resources. This allows automated tools to navigate through resources' links to discover the upcoming appropriate resources. However, dynamic resources connected at runtime are not linked to the existing graph resources, making their discovery a challenging task.

- **Identify k-resources for a same required function**: Due to WoT dynamic aspect, dynamic resources may be unavailable for execution, even if they were initially identified during the discovery process. Also, sometimes, there are demands that require the coverage of several connected resources in order to be processed correctly. Thus, finding other resources providing the same required function is important to fulfill more efficiently user's requests. With the existing of huge number of

resources in large Web-based environments, and to ensure a better performance in terms of response time, it is essential to allow the identification of k-resources ($k \in \mathbb{N}$), instead of discovering all the resources realizing the same required function (where k=0).

- **Identify the suitable WoT resources based on their object location**: In mobile environments, where objects may continuously move in space, considering objects location is important to select the most interesting ones according to a user's request, to collect relevant data and provide accurate results. Nevertheless, with the existing of many installed connected objects in the Web environment, identifying the most suitable ones for user request, is a challenging task. As such, it requires processing spatial queries [38, 55] (e.g., Range type [18] to identify objects in a specific region; KNN type [67] to locate the K nearest neighbors (objects); and Range-KNN [100] to identity the K nearest objects falling into a specific region).
- **Make resource discovery process fast**: Numerous resources can connect to the Web, forming a large-scale Web resource environment. This makes resource discovery a complex process, especially when dealing with demands that require fast responses. Therefore, discovering resources in an improved response time is necessary to enhance discovery process performance and satisfy user's requests in an effective manner.

Resource discovery has received much attention in the literature and in different domains (Web services [23, 109, 6], sensor networks [57, 73], cloud of things [1], etc.). The works related to the Web service domain as in [23, 63, 6] focused on the discovery of resources exposed by Web applications, without considering the dynamicity and location of resources that can be exposed by objects. Also, these approaches do not consider k-resources discovery for the same required function. The other works [57, 86], which belong to other domains as sensor networks [52], cloud of things [1], and fog computing [70], handled the discovery of dynamic resources exposed by objects and considered their location. This is done without covering their possible linking on the functional level, allowing them to be combined with other resources provided by other objects and by Web applications exposing static resources. Moreover, several approaches have proposed machine-readable REST service descriptions [4, 21] to allow automatic resource discovery. However, these descriptions do not consider resources' dynamicity.

To cope with the aforementioned limitations, we propose in this chapter a graph-based approach for automating k-Resources discovery. Our solution is generic that can be applicable in hybrid Web environments providing static linked resources described through a hypermedia-based language (i.e., Hydra [64] in this thesis), and connecting dynamic resources. In our work, we define a formal representation that models the resources (both dynamic and static) into one single Web resource graph, and extends Hydra to describe dynamic resources and locate the resources exposed by objects. We also define a 3-dimensional indexing schema that maps the resources, supporting HATEOAS [45], to their provided functions and to their relative objects location (whenever they are exposed by objects). The schema is used to identify the necessary data collection resources in the required location, and make

resource discovery fast, especially in large-scale Web environments. Our proposed solution allows the integration of several graph-based algorithms adapted to crawl the Web resource graph, and is able to find k-resources realizing the same required function.

The remainder of this chapter is structured as follows. Section 4.2 presents a scenario to motivate our work, and discusses the main challenges. Section 4.3 presents related works, and shows the originality of our approach. Section 4.4 details our automatic resource discovery solution for hybrid Web environments. Section 4.5 evaluates the performance of the solution. Finally, Section 4.6 concludes the chapter.

4.2 Motivating Scenario and Challenges

Our motivating scenario is illustrated within the BEMServer Web platform, presented in Chapter 1, which exposes several static resources for data collection, data preparation, and data processing. In the scenario, we assume that BEMServer is extended to allow the connection of: (1) stationary objects, O_s, and mobile objects, O_m, providing static/dynamic resources. In this case, BEMServer acts as a hybrid Web environment connecting both static and dynamic resources. Following HATEOAS [45], static resources are linked together based on their provided functions defined in a function graph (see Definition 6), forming a resource oriented directed graph. The links between the resources are included in each resource description, which is expressed in Hydra [64] and registered in a triplestore-based repository. As for dynamic resources (i.e., mainly exposed by building occupants devices), they are accessible through their URIs, without being registered into the repository given their dynamic aspect. Thus, the BEMServer platform is a hybrid environment that copes with registered and non-registered resources, as shown in Figure 4.2. Several requests occur in the extended BEMServer

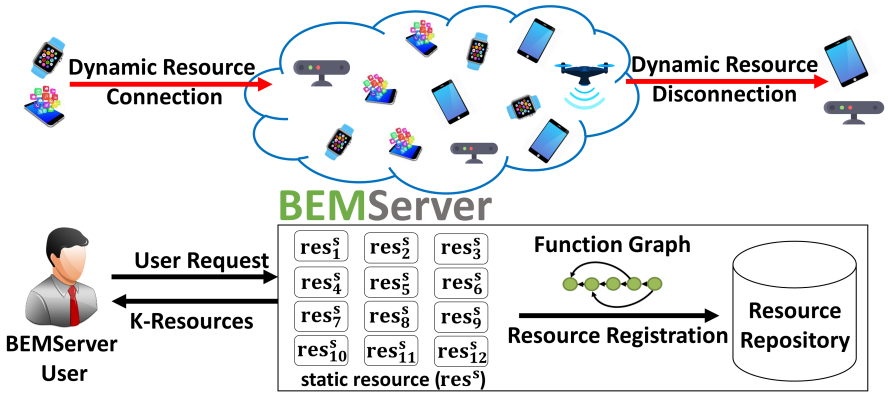


Figure 4.2 – BEMServer, an example of hybrid Web environment

platform. We consider that a building manager wants to predict the heating energy consumption of a specific building zone for the upcoming two hours. To express his demand, he can send 2 different requests in which he specifies the desired function, "EDP" (Energy Demand Prediction):

1. A context aware request, r^{ca}, in which he desires to get the prediction results using temperature data collected from the 3 nearest devices to where he is standing in his office, with a 2 meters (m) range.

2. A non-context aware request, r^{nca}, in which he desires to get the prediction results based on temperature data collected from devices in the conference room A, independently from his current location, as he might be working from abroad.

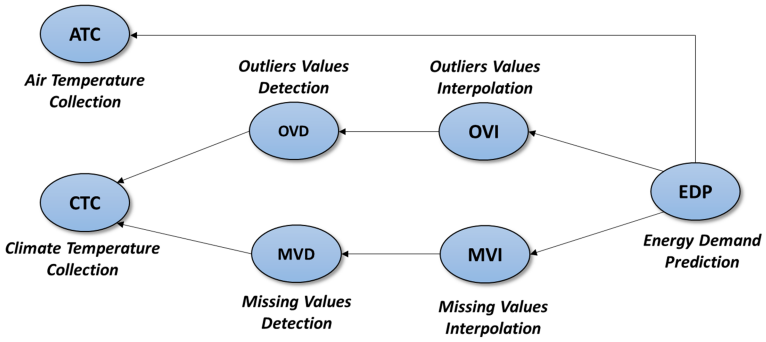


Figure 4.3 – The dependencies between the required functions necessary to realize "EDP" function

To satisfy building manager demand, it is important to identify the necessary resources that provide the functions required to realize "EDP" (see Figure 4.3), and corresponding to the specified zone. However, several challenges arise:

- **Discover external resources**: The dynamic nature of the BEMServer platform, in which external resources can be added and removed dynamically, as o_{m4} and o_{m7} in Figure 4.4, makes it difficult for the building manager to identify the suitable resources answering his request. This is also a complex task even for the automated tools that navigate through resources links to identify the next resources to call, since dynamic resources are not linked to the existing static resources. Thus, to identify both dynamic and static resources (e.g., a resource provided by the smart-phone of the building manager to capture the ambient temperature, and an energy consumption prediction resource embedded within the BEMServer platform), it is important to link them to a resource model.

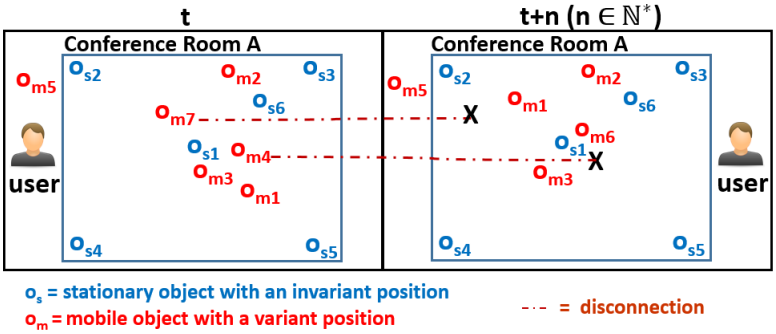


Figure 4.4 – Dynamicity nature of connected resources

- **Identify k-resources providing the same required function**: When a dynamic resource exposed by, for instance, o_{m4} as depicted in Figure 4.4 to collect the ambient temperature, is identified during resource discovery (at instant t), it may be disconnected due to a reboot operation of

the object (at instant t+n). Thus, the provided required function of the resource (i.e., "ATC" function as shown in Figure 4.3) will not be covered. Also, in some cases, the coverage of several resources at once, as collecting the temperature of a big conference room (as conference room A) from different locations using o_{s1}, o_{s2}, and o_{s5} at instant t, might be necessary. For these reasons, finding k-resources providing the same required function is essential.

- **Discover the data collection resources based on their object location**. To have accurate data necessary for user's demand, identifying the resources that are (i) the nearest to the building manager standing point in his office (if his request is r^{ca}), or (ii) located in the conference room A (if his request is r^{nca}), is important. Nevertheless, with the existence of several connected objects having different locations, identifying the most suitable ones to use, is a challenging task. As such, and as illustrated in Figure 4.5, o_{m5} exposes a resource that provides one of the required data collection function, "ATC", but it is not positioned in the conference room A. Thus, it will be inefficient for the processing of $\boldsymbol{r^{nca}}$ at both instants t and t+n. However, at instant t, there are 6 objects ($\boldsymbol{o_{s1}}$, $\boldsymbol{o_{s2}}$, $\boldsymbol{o_{s5}}$, $\boldsymbol{o_{m3}}$, $\boldsymbol{o_{m4}}$, and $\boldsymbol{o_{m7}}$) satisfying "ATC" in the conference room A, and thus, they can answer $\boldsymbol{r^{nca}}$ more efficiently. As for $\boldsymbol{r^{ca}}$, which we assume in this scenario is of type Range-KNN, where it is required to identity the K nearest objects (K=3 in this case) to the building manager falling into a specific region (R=2m), at instant t, $\boldsymbol{o_{s1}}$, $\boldsymbol{o_{m3}}$, and $\boldsymbol{o_{m4}}$ are the most appropriate to use. However, at t+n, $\boldsymbol{o_{s1}}$, $\boldsymbol{o_{m3}}$, and $\boldsymbol{o_{m6}}$ are more convenient, since $\boldsymbol{o_{m4}}$ has been disconnected and a new object, $\boldsymbol{o_{m6}}$, is connected.

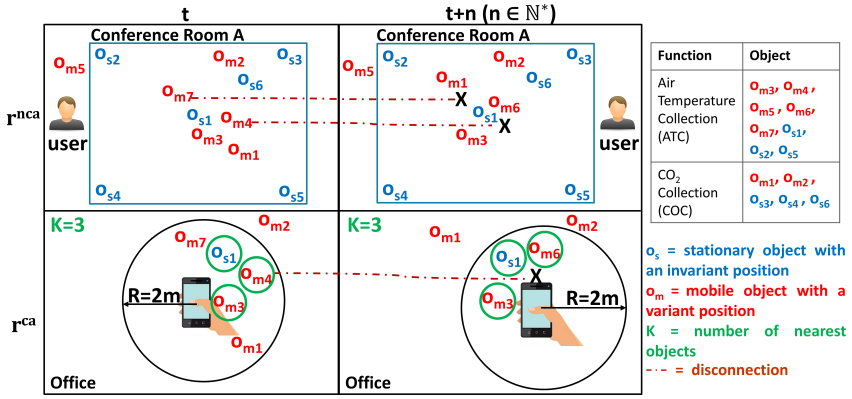


Figure 4.5 – Examples of $\boldsymbol{r^{nca}}$ and $\boldsymbol{r^{ca}}$ in BEMServer

- **Speed-up resource discovery**: Finding the suitable resources in a huge Web environment connecting many resources with an acceptable response time, is important to answer user's request efficiently. For instance, the faster the energy consumption prediction of a building is, the quicker the analysis of the predicted results are, and thus, the performance of systems and devices installed within the building are well managed. Therefore, speeding-up resource discovery within large Web environments, is necessary, but also is a complex task. As such, when the building contains many resources exposed by objects, including the

conference room A, where the building manager desires to predict the energy consumption (see Figure 4.6), locating the suitable resources is challenging, as many objects are installed in different locations.

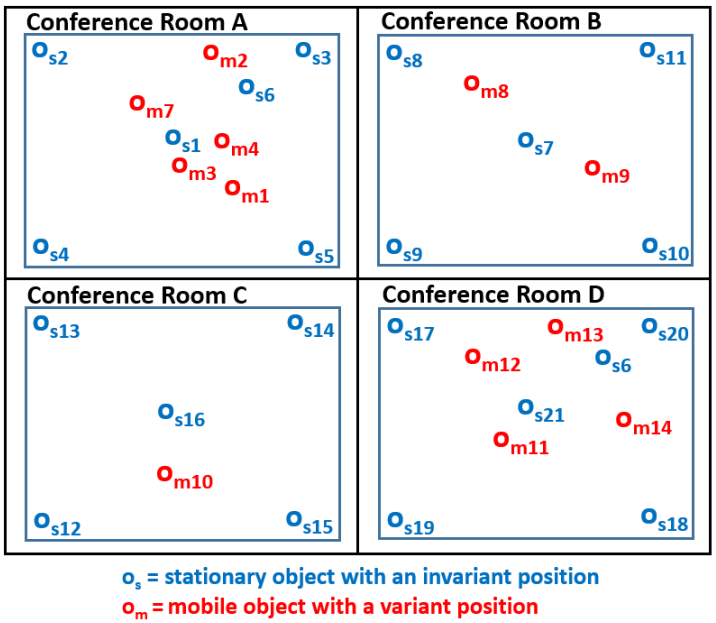


Figure 4.6 – A Web environment connecting many resources exposed by connected objects

To cope with these challenges, we propose a solution for the discovery of static resources supporting HATEOAS, and connected dynamic resources. The solution considers the location of stationary and mobile objects exposing data collection resources (static or dynamic), and identifies the other required resources by following hypermedia resources links with support to HATEOAS. This is done by modeling, first, the resources in a single resource graph, and then, defining an indexing schema that maps resources to their provided functions and corresponding locations (if they are exposed by objects). The indexing schema is used to identify the most appropriate resources from which the discovery process will start its search. Our solution allows to use several graph-based algorithms adapted to explore Hydra-based resource descriptions enriched by semantic annotations, and identifies k-resources for the same required function necessary for user's requests.

4.3 Related Work

Our work relates mainly to Web resource discovery research area, but also to Web resource description. As such, in order to allow a more autonomous usage of Web resources, including resource discovery, it is necessary that the client knows what are the available resources, what are their capabilities (e.g., provided functions), and how they can be invoked (e.g., HTTP method). Therefore, with the aim to reduce the coupling between the client and the resource, a machine understandable resource description is required. In this section, we provide a literature overview on the different existing REST-based approaches relevant to both resource description and discovery

fields. Though there are few discovery and description approaches related to the Web service domain supporting RESTful services comparing to SOAP-based approaches, since REST is an emerging technology, our analysis can show the originality of our solution. During the review, we compared the existing works according to the following criteria:

- **Dynamicity:** denotes the ability to describe or identify suitable resources for a user's request in a dynamic environment connecting both dynamic and static resources.
- **Location-aware:** refers to the ability to specify resources position (whenever they are exposed by mobile/stationary objects) or identify the mot appropriate ones according to their location.
- **Multiplicity:** is the ability to discover several resources providing the same required function necessary to realize user's request.
- **Efficiency and Scalability:** refer to the ability to identify resources with acceptable response time in: (1) large Web environments providing a huge number of resources, and (2) diverse Web environments connecting resources with many different and diverse functions.

We note that resource description approaches were only evaluated according to the Dynamicty and Location-aware criteria.

4.3.1 Resource Description

The authors in [92] present a method called called SemREST (Semantic RESource Tagging) that enriches REST service descriptions with semantic annotations. These annotations provide meaning to what the resource represents, making thus, Client-Service interaction more generic and autonomous. The descriptions are expressed using OpenAPI specification (widely known as Swagger) that is a definition standard proposed by Open API Initiative[1] to describe RESTful Web APIs[2]. As the resources are semantically linked, the method allows service discovery and composition. As such, the semantic resource graph could be generated and host on service side or third party service registry for the clients to query required resources. Nevertheless, the work does not handle dynamic resources or considers resources exposed by objects. Also, it covers the resources callable via "GET" method. In the future, the authors will consider the rest of HTTP verbs, and explore the integration of their method with approaches as Hydra.

Authors in [4], present a model called Resource Linking Language (ReLL) that allows providers to represent RESTful services, with emphasis on the hypermedia characteristic and linked data. The approach provides a natural mapping from the graph-oriented world of RESTful services (resources interlinked by links found in resource representations) to the graph-based model of RDF. Despite from being a rich data format that provides a formal definition of resources and links, it does not support the dynamic aspect of

[1] https://www.openapis.org/
[2] https://swagger.io/

resources and link them to the existing related resources. In addition, it is not dedicated to describe resources exposed by objects.

In [71], an approach that adds semantic annotations to API descriptions at semantic level is proposed. To achieve this goal, authors have extended the Open API Initiative (OAI) specification to create comprehensive APIs description with semantic meaning by linking their properties to concepts in shared vocabularies. The approach focuses on the emerging concept of API Profiling to add descriptive information of data semantics by addressing Dublin Core Application Profile (DCAP) guidelines [3]. However, the work does not consider dynamic resources nor even resource exposed by objects.

In [8] an approach called EXPRESS is presented offering semantic RESTful Web services by exploiting semantic Web resources through a RESTful interface with minimum of design and development overhead. The resources that EXPRESS exploits are entities described semantically in an OWL ontology. As such, a service provider using EXPRESS provides an OWL file describing the provided resources. This is done through an EXPRESS deployment engine to generate URIs for classes, instances and properties. However, the work does not handle the description of dynamic resources, which can be exposed by mobile or stationary objects.

A framework for semantic description of RESTful Web APIs is presented in [99]. The framework is based on annotations added to the application code that associate resources, properties and operations with terms semantically described by vocabularies and ontologies. It allows the generation of representations containing hypermedia controls, which is an important factor for developing clients that are more implementation-independent and resilient to server-side changes. The work is based on the JAX-RS (Java API for RESTful Services) specification that defines a group of APIs for developing Web Services following the REST architectural style principles. These APIs provide a set of annotations that allows regular Java classes to be exposed as resources. Similar to our work, the resources are represented using JSON-LD media type, which provides support for linked data and hypermedia controls, and Hydra Vocabulary enabling the specification of operations that modify resource state. Nevertheless, the framework does not support dynamic resources description, nor even the description of resources exposed by mobile/stationary objects.

In [21], authors define for each resource a Hydra-based descriptor that contains meta-data, mainly about: (i) the HTTP operation used to invoke the resource, (ii) the necessary inputs and the provided outputs, and (iii) information about other related resources w.r.t. the HATEOAS principle. The descriptors are designed as resources following the REST principles. Thus, each descriptor has a specific URI, which can be accessed after calling the correspondent resource URI (though HEAD or GET operation), from the HTTP resource response Header. Figure 4.7 shows how a resource descriptor URI can be accessed.

[3] `https://www.dublincore.org/specifications/dublin-core/profile-guidelines/`

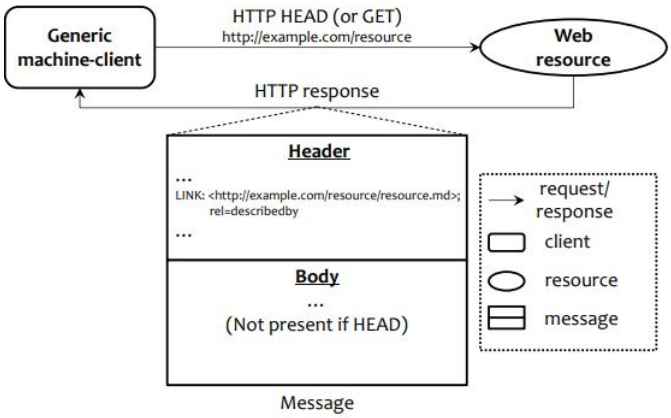


Figure 4.7 – Access to a resource descriptor URI

In order to get the content description of a resource descriptor, a GET operation on the descriptor URI is made. Semantic annotations are integrated into these descriptors in [23], on the resources HTTP operations, and their links to other resources, to automate the resource discovery process. In the work, semantic annotations on resources links can be:

- **isSimilar:** denoting that the resources provide a similar function, though they may vary in terms of non-functional properties.
- **isComplementary:** denoting that the 2 resources are complementary, and thus, they can be combined in the same process to answer user's needs.
- **isIncompatible:** meaning that the 2 resources are incompatible, and therefore, they cannot be involved together in the same process.

However, the descriptions represent only static resources and do not consider resources exposed by objects. In our work, we extended the annotated Hydra-based descriptors presented in [23], in order to allow the descriptions of both static and dynamic resources, while also considering the location of an object exposing a resource.

4.3.2 Resource Discovery

In [23], a resource discovery approach is proposed. It uses a BFS-based algorithm (Breadth First Search) [19] to discover resources, and explores the semantically annotated resource descriptions defined in [21] to determine if a resource suits the required functions. Apart from neglecting the dynamicity aspect of resources and resources exposed by mobile/stationary objects, the discovery process finds one resource for each required function, and thus, prevents the identification of other resources providing similar functions. This is an important criteria to consider in dynamic Web environments, as it is possible to identify more qualified resources to answer user's request, and substitute resources in case of non-availability. Moreover, the proposed discovery algorithm requires a given initial resource URI from the end user in order to be able to crawl the resource graph. Such task is not obvious for non-expert users. In our work, end-users express their demands simply through a single function selected from a given list. Also, our solution allows

the use of different graph-based algorithms (i.e., Breadth First Search (BFS) and Depth First Search (DFS) [97]) to traverse the resource graph, and optimizes resource discovery using an indexing schema that maps resources to their provided functions.

In [109], a Web service description and interaction approach for automatic Web service discovery called RESTdesc, is proposed. The approach is based on Notation3 RDF (Resource Description Framework) syntax to describe REST services, and uses operational semantics of Notation3 in order to allow a flexible discovery. Although it respects the HATEOAS principle, the description does not allow the discovery of dynamic resources, and resources exposed by objects are not covered. Moreover, the solution is used to crawl the related resources without identifying more than one resource for a required task, and depends from a complex logic language, Notation3, which is a superset of RDF. Notation3 is difficult to use, even for expert users, compared to Hydra (the language adopted in our solution) that is expressed through JSON, a comprehensible and easy to learn format for humans.

In [6], services are described through RAD (REST API Description), which is a format that considers the REST uniform interface constraint, including the hypermedia as the engine of application state. RAD is implemented in JSON to allow the generation of human readable documentation, and in Microdata that can be embedded in the HTML service description. In Microdata or JSON implementation, services descriptions are parsed to generate a graph that captures state transitions in an activity layer, as well as resources, transitions, and response semantics in a semantic layer. Using graph queries, the graph is traversed for service discovery and composition. However, the approach does not support dynamic resources that can be exposed by objects. Moreover, the solution requires that the user knows the Schema.org data model, which has been extended with a set of concepts to semantically annotate resource descriptions. Also, user needs are expressed through a graph query that requires knowledge from end-users.

Authors in [63] present an ontology-based approach for personalized RESTful Web service discovery. The approach is based on the use of a profile ontology that highlights users experience with the services, by capturing their feedback and those of similar users. A collaborative filtering is also proposed to recommend services based on their utility to the users. The filtering method used filters returned services with respect to a predicted utility (or satisfaction) value of the returned service, based on the opinions of the users. Though it is an original work that takes into account users experience and feedback for resources discovery, the work is not dedicated to discover dynamic resources that can be exposed by objects. Moreover, efficiency/scalability tests performance are to be examined.

Authors in [58] propose RESTDoc, a combination of Microformats-style markup and RDFs, to provide a comprehensive framework for describing, discovering and composing RESTful services. In the approach, semantic annotations are added into the already existing documentation of RESTful

services expressed in HTML code, and extensions are integrated to link the services w.r.t. the HATEOAS principle, enabling thus, automatic resource discovery and composition. The work distinguishes between two different aspects of the RESTful service discovery: (1) discovery as you browse, which concerns client-side browsers and relies on HTML Link elements on a Web site to point to other resource descriptions, and (2) automated discovery, which refers to the ability of a service to access and link to other related resources in the same application domain. Contrary to RESTDoc, in our solution, the resource description is separated from the resource representation, and is located in what we called a descriptor. This can facilitate the design task for developers. Also, the work is not dedicated to handle dynamic resources or resources exposed by mobile/stationary objects.

Although dynamic resource discovery is not handled in the Web service domain, it is an active research area in other domains, as sensor networks [57, 73] and fog computing [95, 70]. Also, considering the location of resources exposed by objects has been the focus of many works as the ones related to cloud of things [86] and mobile cloud computing [127], allowing the discovery of resources based on their location and context properties (e.g., Accuracy, Precision, and Response time). However, these solutions mainly focus on the discovery of resources exposed by mobile devices that communicate with their existing neighbors, independently from their provided functions, neglecting thus, their possible relations with other resources exposed by objects and Web applications. Contrary to such approaches, our work combines existing linked resources (established to be always available on the Web) and connected dynamic resources into one semantic-based single graph model based on their provided functions. A part from allowing the discovery of both resource types (static and dynamic), our model enables the combination of resources according to their provided functions to create new composed resources realizing complex user's request.

4.3.3 Evaluation Summary

Table 4.1 shows the evaluation summary of existing resource description and discovery approaches related to the Web service domain, based on the required criteria. The works related to resource description are only evaluated according to the Dynamicity and Location-aware aspects. This explains the grey cases of these approaches relative to the rest of the criteria. As it is shown in the table, where we used "-" symbol to express a lack of a criterion coverage, none of the approaches within Web service domain are dedicated to describe dynamic resources exposed by mobile/stationary objects. For the multiplicity criterion, the resources discovery approaches mainly focus on the discovery of one resource for each required function, since resources are always available in the Web environment, contrary to a dynamic one. As for the efficiency/scalability, the current approaches lack in evaluating their work in large Web environments consisting of many resources with diverse functions.

Table 4.1 – Evaluation of existing works related to the Web service domain and used for resource description and discovery w.r.t. the identified criteria

		Dynamicity	Location-aware	Multiplicty	Efficiency/Scalability
Resource Description Approaches	[92]	-	-		
	[4]	-	-		
	[71]	-	-		
	[8]	-	-		
	[99]	-	-		
	[21]	-	-		
Resource Discovery Approaches	[23]	-	-	-	-
	[109]	-	-	-	-
	[6]	-	-	-	-
	[63]	-	-	-	-
	[58]	-	-	-	-

4.4 Automatic Location-aware Approach for k-resources Discovery

In this section, we present our solution to discover automatically k-resources (i.e., static and/or dynamic) responding to user's request, r (see Definition 7), while considering data collection objects location. In the request, the user specifies his desired function, f (such as "EDP" function). f is selected from a generated list of functions that can be provided by the available resources connected to the Web environment at the current instant.

4.4.1 General Overview

Before elaborating on our approach, we define a function graph, FG, containing the functions provided by the available connected resources, RES, and their dependencies (if they exist). FG is a directed acyclic graph, such that:

Definition 6. ***FG = (F, O)****, where:*

- ***F*** *is the set of all the functions provided by RES. Formally,* $F=\{f_i\} / f_i \in res$ *and* $res \in RES$*, with res denoting a resource.*
- $\boldsymbol{O} = \{<\} \cup \phi$*, is a binary order relation on F. Such binary relation is a subset of* $F \times F$ *linking two functions when one precedes another in the ordering. As such, if* f_1 *precedes* f_2*, it is denoted as* $f_1 < f_2$*. FG can also include functions that are not dependent of any other, we refer to these functions as "terminal".*

As for the user's request, r, it is a spatial query that can be:

1. Context aware, r^{ca}, in which the processing considers the requesting device location at the time of request.
2. Non-context aware, r^{nca}, whose processing is independent from the requesting device position.

In this work, $r^{ca} = r^{ca}_{range} | r^{ca}_{knr} | r^{ca}_{rknr}$, such that:

- r^{ca}_{range} is a range query

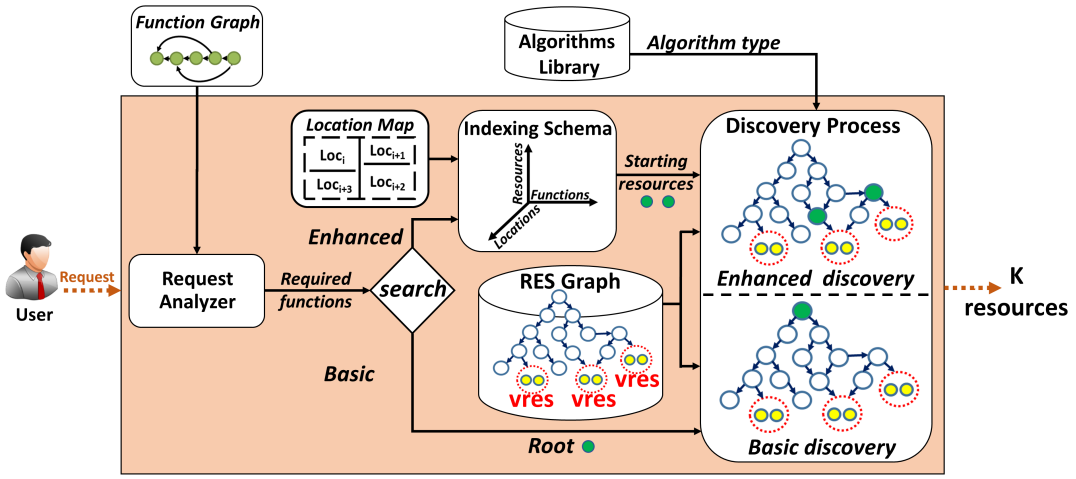


Figure 4.8 – Overview of the resource discovery approach

- r^{ca}_{knr} is a K Nearest Resource (KNR) query, where the nearest resources are provided by the nearest objects to the requesting device location
- r^{ca}_{rknr} is a Range-K Nearest Resource (R-KNR) query

More formally, we define r as:

Definition 7. $\boldsymbol{r = (f, P, k)}$*, where:*

- $\boldsymbol{f}$*, a mandatory element, is the user's requested function.* $f \in F$*, with* $F \in FG$*.*
- $\boldsymbol{P}$*, is the set of parameters necessary for the execution of f, such that* $P = \bigcup_{i=1}^{\mathbb{N}^*} \{\boldsymbol{p_i}\}$*, with* $\boldsymbol{p_i}$ *= (key:value), and where: key denotes the parameter name, and value is the parameter value given by the user. In P, we define: (i) a location, Loc, such that Loc = (Location:value) with value refers to the desired location (e.g., conference room A and office), and (ii) a scope, S, representing a range, with* $S \in \boldsymbol{R}^+$*. Using Loc and S, the user can specify whether his request depends from data collection objects that are: (1) located in Loc, or (2) situated in a given Loc and covering a certain S, or (3) positioned in S bounded via a circle centered by the user requesting device location, with a specific radius.*
- $\boldsymbol{k} \in \mathbb{N}$*, is the number of the nearest objects used for data collection at the time of request. k denotes also the number of the discovered resources providing functions other than data collection. If k=0, all resources fulfilling the dependent functions necessary to realize f, as defined in FG, will be discovered.*

Figure 4.8 shows the resource discovery solution adaptive to discover k static and/or dynamic resources necessary to answer the user's requested f defined in r. The solution covers the discovery process of the automatic resource composition in the StARC framework (see Figure 1.10). Using FG, the Request Analyzer (RA) component finds the set of functions F' (F' $\subset$ F) required to realize f. Resource search consists in finding the available connected Web resources matching F'. For this aim, we first represent all resources (static and dynamic) into one single graph, denoted as RG (see Definition 8), which originally contains linked static resources. To add the dynamic resources in RG, we define a virtual resource, *vres*, for each function

f in F, and link it to the existing static resources realizing the same function. Each *vres* contains the connected dynamic resources answering its correspondent function (this facilitates the integration of new functions exposed by dynamic resources in the environment). Using RG, the Discovery Process (DP) component runs a graph algorithm to identify the resources matching F′. The algorithm type is specified by the solution administrator from a library of graph-based algorithms. In our work, BFS and DFS have been implemented. Resource discovery can be (1) basic, where the algorithm starts crawling RG from the root, or (2) enhanced, where the algorithm starts RG traversal from resources pointed by a defined indexing schema. Such schema maps the set of available linked resources described with Hydra, to their provided functions defined in FG and to their locations (whenever they are exposed by objects). It allows identifying: (i) the resources providing data collection functions (if required for user's request) in the necessary location (defined in a location map), or (ii) the resource(s) from which DP will begin crawling the Web resource graph, i.e., RES, when no data collection functions are required. Thus, the indexing schema returns the appropriate resources from which the discovery algorithm will begin its search, instead of traversing the resources of RG starting from the graph root.

4.4.2 Static and Dynamic Resource-based Graph

With respect to HATEOAS, Web resources are linked together, forming a resource graph, RG. RG contains at least static resources established to be always available on the Web. However, the Web is a hybrid environment that allows connecting dynamic resources. Given their dynamic aspect, these resources are not linked to RG, thus, making their discovery a challenging task. To address this challenge, we define a virtual resource, *vres*, for each function f in F (F $\in$ FG). A *vres* can be permanent, $vres^p$, or temporary, $vres^t$. The $vres^p$ contains the connected dynamic resources realizing its related function that exists in F, and is linked to the static resources providing that same function via the "isSimilar" relation. As such, if a static resource provides both f_1 and f_2, it will be linked to the virtual resources, $vres_1^p$ and $vres_2^p$, defined for each of these functions respectively. And, when a dynamic resource, res^d, providing f is connected, it will be included in the $vres^p$ defined for f. Thus, we obtain a resource oriented directed graph, RG, combining dynamic and static resources. Mainly, the connected dynamic resource of RG provides a function that exists in F. However, if res^d realizes a function that is not included in F, it will be added in a temporary virtual resource, $vres^t$, defined specifically to the new function. $vres^t$ disappears when all of its related dynamic resources are disconnected, contrary to the $vres^p$, which is always present in RG. $vres^t$ is linked to $vres^p$ through "isRelated" relation. Such linking is based on the dependencies between $vres^p$ and $vres^t$ related functions[4]. The process of linking dynamic resources to static ones in a single graph is illustrated in Figure 4.9.

An example of the relations between the different types of resources based on the dependencies of the functions required to realize a function (i.e., "EDP"

[4]Currently the new functions are added randomly in FG. Their real dependencies with other functions will be explored in subsequent work

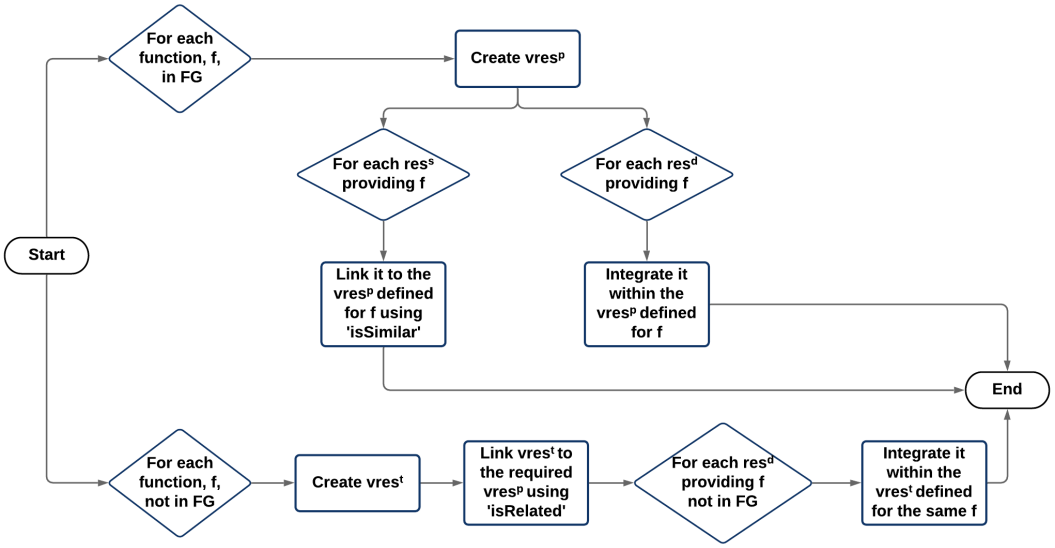


Figure 4.9 – Flowchart of the process linking dynamic resources to static ones

standing for Energy Demand Prediction) is shown in Figure 4.10. In the example, we assume that a resource provides one single function.

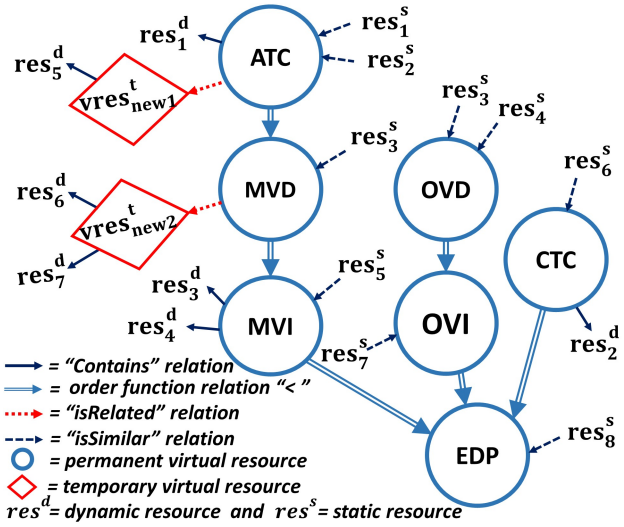


Figure 4.10 – An example model of the relations between the resources used to predict energy demand

Another example of linked static and dynamic resources, using defined permanent virtual resources, is presented in Figure 4.11. The example consists on the resources used to predict the air temperature values based on 3 main functions defined in the function graph, FG: ATC (Air Temperature Collection), TCC (Temperature Conversion to Celsius), and ATP (Air Temperature Prediction). As represented in Figure 4.11, the defined permanent virtual resources for each required function (ATC, TCC and ATP), are linked to static resources via the "isSimilar" relation, and to dynamic resources via the "contains" relation. As for the static resources providing these different functions, they are related through the "isComplementary" relation based on their order dependencies defined in FG.

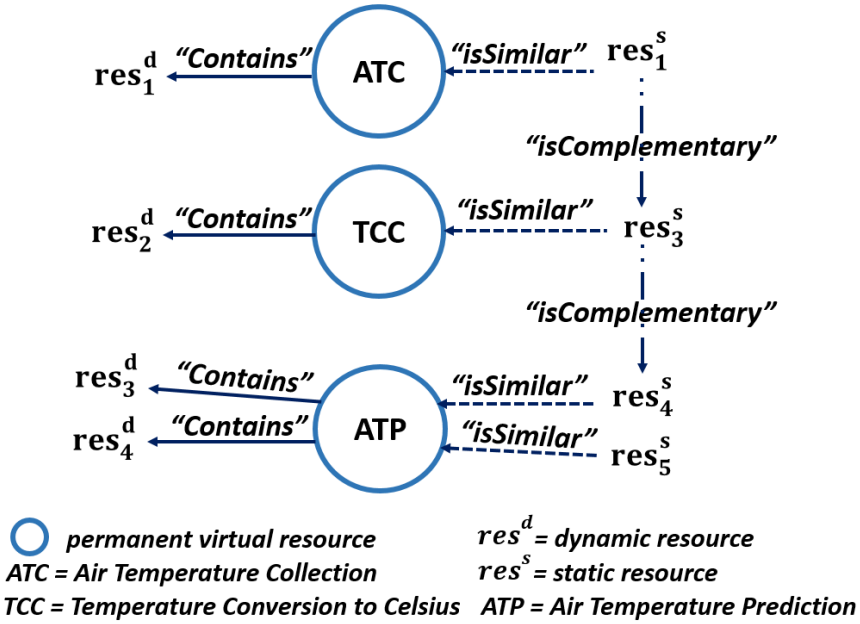


Figure 4.11 – An example model linking static and dynamic resources used to predict air temperature values

Formally, we define RG as:

Definition 8. RG = (*RES, Root, REL, f, g, t*), *where:*

- ***RES*** *is the set of all the static, dynamic, and virtual resources connected to the Web environment.* $RES = RES^S \cup RES^D \cup RES^V$, *with:*
  - $RES^S = \{res^s_{i \in \mathbb{N}}\}$, *is the set of static resources*
  - $RES^D = \{res^d_{i \in \mathbb{N}}\}$, *is the set of dynamic resources*
  - $RES^V = VRES^P \cup VRES^T$, *is the set of permanent and temporary virtual resources, with* $VRES^P = \{vres^p_{i \in \mathbb{N}}\}$ *and* $VRES^T = \{vres^t_{i \in \mathbb{N}}\}$
- ***Root*** $\in RES$, *is the set of any resource, res, that is not being pointed by any other resource in the graph. In this work, such set is formed by static resources.*
- ***REL*** $= R \cup C \cup T$, *is the set of relations linking the resources to each other, where:*
  - ***R*** *refers to the relations used to link static resources to other resources, i.e., static or permanent virtual.*
    $R = \{\simeq, <\}$, *where '*$\simeq$*' denotes "isSimilar" relation, and '*$<$*' denotes "isComplementary" relation*
  - ***C*** *refers to the "contains" relation, such that* $C = \{\in\}$. *It is used to link virtual resources to dynamic resources*
  - ***T*** *refers to the "isRelated" relation used to link permanent virtual resources to temporary virtual ones providing new functions not included in FG, with* $T = \{\rightarrow\}$
- ***f*** *is the function linking static resources together, and static resources to permanent virtual resources, using R, such that f:* $RES^S \overset{R}{\mapsto} RES^S | RES^S \overset{R}{\mapsto} VRES^P$

- ***g*** *is the function relating virtual resources to dynamic resources, using C, such that g:*$RES^V \overset{C}{\mapsto} RES^D$
- ***t*** *is the function relating permanent virtual resources to temporary ones, using T, with t:*$VRES^P \overset{T}{\mapsto} VRES^T$

A resource, *res*, can be static, dynamic or virtual (permanent or temporary), and is defined as:

Definition 9. $\boldsymbol{res = res^s \mid res^d \mid vres^p \mid vres^t}$

Each static/dynamic resource is formally represented as:

Definition 10. $\boldsymbol{res^{s|d}}$ **= (c, id, loc, F, L)**, *where:*

- ***c***, *is the context Web address referring to terms linked to existing data models (e.g., ontologies [85]). Such terms are used to link res properties to concepts defined in their correspondent data models.*
- ***id***, *refers to the URI Web address used to invoke* $res^{s|d}$
- ***loc***, *is the location of the object exposing* $res^{s|d}$*. It is equal to null if it is not the case, i.e., exposed by a Web application.*
- $\boldsymbol{F = \bigcup_{i=1}^{\mathbb{N}^*} \{f_i\}}$*, designates the set of functions provided by res, such that:* $\boldsymbol{f_i = (n, I, O, m)}$*, and where:*
  - ***n***, *refers to* $\boldsymbol{f_i}$ *name*
  - ***I***, *denotes the input(s) of* $\boldsymbol{f_i}$
  - ***O***, *denotes the output(s) of* $\boldsymbol{f_i}$
  - ***m***, *is the HTTP verb used to call* $\boldsymbol{f_i}$
- ***L*** *refers to the set of linked resources (if they exist) to* $\boldsymbol{res^{s|d}}$*. It is always Null for* $\boldsymbol{res^d}$*, however, it can be defined for each* $\boldsymbol{res^s}$ *as* $L = \bigcup_{i=1}^{\mathbb{N}^*} \{l_i\}$*, where:*
  - $\boldsymbol{l_i}$ *=* $(\boldsymbol{res.f}, \boldsymbol{r})$*, with* $\boldsymbol{res.f}$ *is the function provided by the linked* ***res****, such that* $\boldsymbol{res = res^s|vres^p}$*, and r* $\in$ *R (with R* $\in$ *REL)*

A permanent/temporary virtual resource is defined as:

Definition 11. $\boldsymbol{vres^{p|t}}$ **= (id, f, D, L)**, *where:*

- ***id***, *refers to the URI Web address used to invoke vres and access its correspondent dynamic resources*
- ***f*** *=* $(\boldsymbol{n, GET})$ *refers to the function related to vres, with:*
  - ***n*** *is the name of the function defined for vres*
  - ***GET*** *refers to the HTTP verb used to call vres*
- ***D*** *designates the set of dynamic resources supporting the same function defined for vres, with D=*$\{res^d_{i\in\mathbb{N}}\}$*,* $res^d_{i\in\mathbb{N}} \in RES^D$*, and vres contains D through "C" relation (C* $\in$ *REL).*

- ***L** refers to the set of linked resources (if they exist) to $vres^{p|t}$. It is always Null for $vres^t$, however, it can be defined for each $vres^p$ as $L = \bigcup_{i=1}^{\mathbb{N}^*} \{l_i\}$, where:*
  - *l_i = ($vres^t.f$, r), such that $vres^t.f$ is the function provided by the linked $vres^t$, and $r \in T$*

In our work, RES^S and RES^D provide the initial functions defined in FG, while $VRES^T$ answers newly added functions.

Based on the resource definitions, we extended Hydra to allow the descriptions of dynamic and virtual resources, and to add the location of both dynamic and static resources (whenever they are exposed by objects). In the descriptions, shown in Appendix K, the term "Operation" denotes the provided function by the resource, and the term "function" refers to a function defined in FG. The inputs and outputs of an operation are included in the "expects" and "returns" fields respectively. They are represented as key:value pair, where key is the term mapped to a specific data model, and value is the concept to which each input and output is referring to.

4.4.3 Indexing Schema for an Enhanced Resource Search

Exploring graphs like the Web environment requires graph traversal algorithms that traverse RG nodes (i.e., resources) to find the appropriate ones realizing user's request. The most popular algorithms are Depth First Search (DFS) and Breadth First Search (BFS) [97]. We conducted several tests to compare both BFS and DFS. The results, available in Appendix J, show that the performance of each algorithm depends on the functions distribution and the localization of the requested function in FG. Originally, the resource discovery algorithm starts from the resource(s), included within RG Root (see Definition 8), leading to a huge response time when dealing with large graphs. To optimize the time of the resource discovery, we define an indexing schema that maps (i) the resources (i.e., RES^S, RES^D, and $VRES^T$ if they exist) to their provided functions, and (ii) static/dynamic resources used for collecting data to their locations (whenever they are exposed by objects). In order to identify data collection resources suitable for the requested location in r, we assume that there is a location map describing the geographic area of the targeted environment. Such description is based on different levels of location granularity defined in a geographic hierarchy. For example, in our motivating scenario related to smart buildings domain, we consider that Zone $\xrightarrow{isPartOf}$ Floor $\xrightarrow{isPartOf}$ Building, with Zone, Floor, and Building are types of location (i.e., subclasses of Location), as illustrated in Figure 4.12-(a). The location value presented in the description of each data collection resource exposed by an object refers to the smallest location granularity, i.e., a specific zone, Z_i, with i $\in \mathbb{N}$ in our case. We suppose that object's location is updated periodically based on a predefined time interval.

To identify the resources realizing user's request, and locate the suitable data collection objects, we use the indexing schema, IdS, that consists of 3 dimensions: Functions, Resources, and Locations, as presented in Figure 4.12-(b). IdS is formally defined as:

Definition 12. ***IdS = (o, F, R, L**), where:*

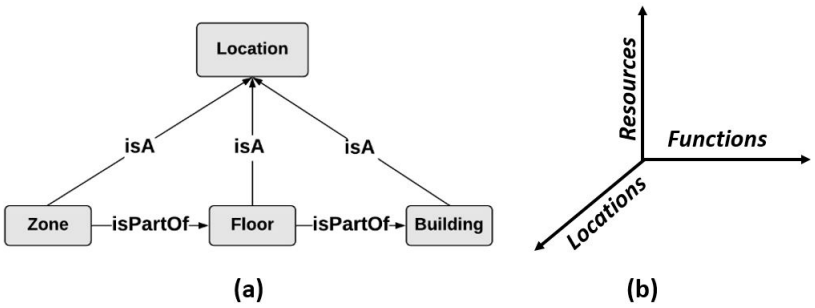


Figure 4.12 – An example of a geographic hierarchy VS The indexing schema

- ***o***, *is the origin positioned at (0, 0, 0). It denotes a special resource containing the set of the static resources, and the dynamic resources used for data collection. o denotes also an empty function provided by the resources, and the set of the smallest location granularity.*
- ***F***=$\{x\}$, *is the abscissa axis values referring to the indices of the functions realized by the static resources, and the data collection functions provided by the dynamic ones. Each x has a fsignature consisting of the functions indices required to realize f, as defined in FG, such that:*
  - *fsignature* = $\{x'\}$, *where* $x' \in F$, $x' \neq x$, *and* $\exists\ x' \in$ *fsignature with* x'*.fsignature*=$\varnothing$ *denoted as "terminal".*
- ***R***=$\{y\}$, *is the ordinate axis values referring to the set of the static resources, and the dynamic resources providing data collection functions. Only y representing a static resource has a rsignature, which consists of the resources indices related to it via semantic relations, i.e., isSimilar and isComplementary, and such that:*
  - *rsignature*=$\{y'\}$, *with* $y' \in R$ *and* $y' \neq y$
- ***L***=$\{z\}$, *is the applicate axis values referring to the smallest location granularity set (e.g., zone) of the objects exposing static/dynamic resources for collecting data.*

IdS is used by the discovery process to identify the resources providing the terminal functions that are not dependent of any other (in this work, they are data collection functions), and relative to the necessary location for r.

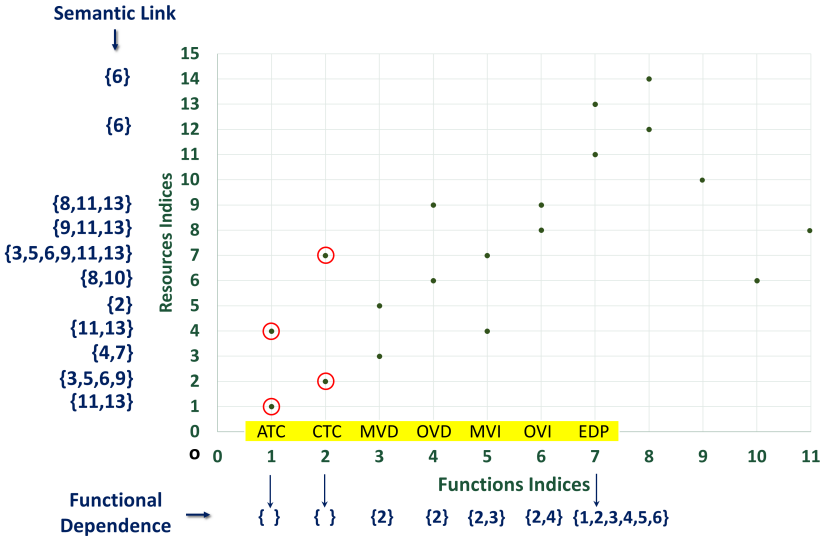


Figure 4.13 – IdS linking resources to their functions and used to identify the initial resources necessary to realize "EDP" function

As shown in Figure 4.13, where we assume that all resources are within the same required location, f_7 (referring to the "EDP" function) is preceded by 6 functions, $\{1,2,3,4,5,6\}$[5]. Based on the analysis of these functions' signatures, f_1 ("ATC" function) and f_2 ("CTC" function) are terminals. The resources realizing such two functions are res_1, res_2, res_4, and res_7 (circled in red), and will act as initial resources from which the discovery algorithm will begin its process, instead of starting from the graph Root.

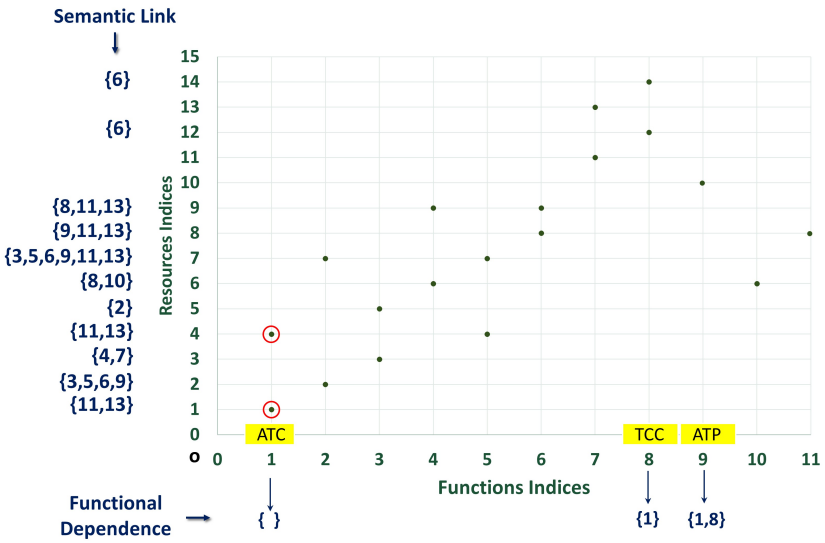


Figure 4.14 – IdS linking resources to their functions and used to identify the initial resources necessary to realize "ATP" function

Another example of using the indexing schema is illustrated in Figure 4.14. The example consists on identifying the resources from which the resource discovery process will begin its search, in order to find the necessary resources used to predict air temperature values. As shown in Figure 4.14, where we consider that all of the necessary resources are within the required

[5]Refers respectively to the ATC, CTC, MVD, OVD, MVI, and OVI functions

location, the "ATP" (Air Temperature Prediction) function depends of two other functions: "TCC" and "ATC". From the analysis made on the fsignatures of these two functions, only "ATC" is terminal. The resources realizing "ATC" are res_1 and res_4 (circled in red), and will act as initial resources from which the discovery algorithm will start its process.

In appendix L, we show the performance evaluation of IdS construction while considering the resources in the same location. The results show that the response time and memory usage increase with the evolution of both functions and resources numbers. Such evolution is huge when the number of functions and resources is high. Thus, updating the IdS dynamically without regenerating it again is an improvement that we seek to do in the future.

4.4.4 Resources Discovery Process

Our discovery solution identifies static and/or dynamic resources, and adapts several graph-based algorithms for RG traversal, while considering resources location (whenever they are exposed by objects). The algorithm to be used is given as input to the resource discovery process based on a library of algorithms (i.e., BFS or DFS in this work). Algorithm 2 presents the pseudo code of the defined discovery process having the following entry data:

- **algoType** (string): denotes the algorithm type to be used
- **f** (string): is the user's requested function
- **k** (integer): is the maximum number (defined in user's request) of the discovered resources providing identical functions

The output is the **discovered** array, containing the pairs [**f**, **id**] that correspond to the discovered resources necessary to realize f. During the discovery process, several main functions are used:

- **IdS(string,integer)**, identifies the data collection functions required for f and gets the relative resources corresponding to the specified location in r, using the geographic hierarchy and the indexing schema.
- **funResMap(array of [string, array of string])**, is called to generate an array of [string, string] that maps each resource to its provided function.
- **getResDesc(string)**, is called to access the description of a resource, identified by its id (i.e., URI), and retrieve the related resources that can be traversed next.
- **discover(array of string)**, traverses the resource Web graph starting from a set of identified resource to discover the rest of the resources necessary to realize f. Its pseudo code is presented in Algorithm 3.
- **functionMatch(string, string)**, checks if two functions are identical.

DP starts by identifying the resources realizing the data collection functions necessary to f, and located in the required location, using the **IdS(f,k)** function (line 8). The data collection functions with their relative identified

Algorithm 2 : Pseudo code of the Discovery Process (DP)

```
1  algoType string
2  input: f string
3  input: k integer
4  output: discovered array of [string, string]
5  dataCollectionRes: array of [string, array of string] // contains the data collection functions relative to f with their resources corresponding to
     the required location
6  resToExploreNext: array of string
7  discRes: array of [string, string]
8  dataCollectionRes = IdS(f,k) // gets k resources for each data collection function required for f in the necessary location
9  discovered = funResMap(dataCollectionRes)
10 foreach func in dataCollectionRes do
11     foreach id in dataCollectionRes[func][1] do
12         Descriptor desc = getResDesc(id) // get the description of a resource using its id
13         if not id.L.empty() then
14             resToExploreNext.insert(id.L) // insert the next related resources to id, as defined in its description in resToExploreNext
15     if dataCollectionRes[func][1].empty() then
16         outputMessage ("no resources are identified in the required location")
17 discRes = discover(algoType, resToExploreNext) // start resource graph traversal from the resources in resToExploreNext to discover other
     required resources providing the necessary functions
18 discovered.insert(discRes)
19 return discovered
```

resources are added into the **dataCollectionRes** array. **funResMap(dataCollectionRes)** function is used to map each function to its identified resource, and store the results in the **discovered** array (line 9). For each data collection function, func, in **dataCollectionRes**, DP retrieves the corresponding resource description (expressed in Hydra in this work) using the **getResDesc(id)** function (line 12) to get the linked resources ids that will be traversed next. These ids, stored in the **resToExploreNext** array (line 13), are later used by the **discover(algoType, resToExploreNext)** function, presented in Algorithm 3, to identify the resources providing the other required functions. When there are no resources providing a data collection function in the required location, a message will be sent (lines 15-16). Once the necessary resources are inserted into **resToExploreNext** array, they will be used by the **discover(algoType, resToExploreNext)** function to traverse the resource graph, and identify the rest of the resources necessary to answer f (line 17). All of the discovered resources realizing the required functions are saved in the returned **discovered** array (line 19).

Based on the given **algoType**, the discover function, presented in Algorithm 3, runs the corresponding algorithm (**runAlgoType(algoType)**) that will explore RG starting from the resources included in the **resToExploreNext** array. **currentId** refers to the resource that is being processed, which is initially the first resource of the **resToExploreNext** array. For each unvisited resource, the algorithm gets the relative Hydra description through **getResDesc()** (line 12). If the operation function matches one of the functions in **F′** using the **functionMatch()** (line 14), the algorithm checks whether the resource is virtual or static (lines 15 to 20). When it is virtual, the ids of the dynamic resources included in the description (line 16) are inserted with their relative function in the **discRes** array. When it is static, the corresponding resource id is stored in the **discRes** array with the relative function (line 21). If the number of the discovered resources realizing the current function is equal to **k** (lines 18 and 22), the function is removed from **F′**. Such number is calculated using the **resFound()** that we implemented apart. To explore other resources, the algorithm follows the resource semantic annotated links (lines 24 to 26).

Algorithm 3 : Pseudo code of discover(agloType, resToExploreNext) function

```
1  algoType string
2  input: resToExploreNext array of string // set of resources from which RG traversal will start
3  output: discRes array of [string, string]
4  visited: array of string
5  F': array of string // contains the non data collection functions required to realize f as defined in FG
6  currentId = resToExploreNext[0]
7  F'= FunG(f) // Gets the required non data collection necessary to answer f
8  runAlgoType(algoType): // the execution of the algorithm corresponding to the given algoType
9  while not F'.empty() do
10     if not currentId in visited then
11         visited.insert(currentId)
12         Descriptor desc = getResDesc(currentId) foreach operation in desc.Operation do
13             foreach f in F' do
14                 if functionMatch(operation.function, f) then
15                     if not desc.RESD.empty() then
                           // the description contains a collection of dynamic resources
16                         foreach adhoc in desc.RESD do
17                             discRes.insert([f, id])
18                             if resFound(discRes, f) = K then
19                                 F'.remove(f)
20                     else
21                         discRes.insert([f, currentId])
22                         if resFound(discRes, f) = K then
23                             F'.remove(f)
24         foreach link in desc.Link do
25             if (link.relationType = isSimilar or link.relationType = isComplementary or link.relationType = isRelated) then
26                 resToTraverse.insert(link.entrypoint) // stores the resources linked to the current traversed resource
27         currentId = resToTraverse.selectNext() // selects the next resource to traverse
28     else
29         currentId = resToExploreNext.next()
30 return discRes;
```

4.5 Evaluation and Discussion

In this section, we evaluate the performance of our resource discovery solution in different function and resource graphs topologies, by varying different parameters, such as: the number of functions, the number of resources providing, each, one function included in FG, and the number of the needed data collection resources in the required location. The tests are divided into two main scenarios. In the first scenario, we considered that all resources are located within the same necessary location for user's request, and studied our approach performance in 2 different forms: (1) basic, where the search starts from the resource(s) included in the graph Root of RG, and (2) enhanced, where the search starts from the resource(s) pointed by IdS. This is done without considering the best algorithm type to use at each function graph topology (as it is not the purpose of this work). Therefore, we only show the experiments using one algorithm type, i.e, the BFS. In the second scenario, we focused on analyzing the approach according to the "Location" dimension, while varying (i) the number of the needed data collection resources in the required location, (ii) the number of locations relative to r (r^{nca}), and (iii) the number of the required data collection functions provided by resources located in the necessary location. To consider worst case scenarios, all of the tests consist of dynamic resources providing functions existing in FG, regardless of whether they were originally defined or newly added.

4.5.1 Environment Setups

The function and resource graphs are dynamically generated based on simulations, where we varied several criteria, i.e., number/order of functions,

number/type of resources, number of resources (k) providing the same function. This is done to study our approach in different functions/resources graphs topologies. In the tests[6], conducted on a Linux Debian (64 bits) virtual machine, with 1 dedicated Intel® Core™ i7-46000 CPU @ 2.10GHz 2.70GHz processor and 1 GB RAM, we show the algorithm response time (in milliseconds) based on an average of 5 sequential executions.

4.5.2 Scenario 1: Basic Search vs Enhanced Search Evaluation

Due to the lack of standard benchmark to evaluate existing related works [23, 109, 6], we propose the following criteria for our resource discovery approach evaluation:

- **Dynamicity**: the ability to identify appropriate resources for a user's request in a dynamic environment connecting both dynamic and static resources
- **Multiplicity**: the ability to discover k-resources responding to the same required function
- **Efficiency**: the ability to identify suitable resources in a large Web environment with acceptable response time
- **Scalability**: the ability to identify resources in large graphs containing resources that provide numerous different functions

For each aspect, we generated results by running the basic and the enhanced search forms of the BFS algorithm. For the **dynamic aspect**, the tests were executed on 5 resource graphs containing, each, 200 resources, based on a FG with 20 ordered functions. The number of dynamic and static resources varies from a graph to another with the variation of k. As such, the graph consisting of 200 static resources ([200,0]), contains 10 resources (k=10) for each function in FG. However, graph [100,100] includes 5 static and dynamic resources (k=5 for each type) realizing the same function. The aim of the tests was to find 1-resource for each function required to answer a function dependent of 6 other functions. Figure 4.15-(a) shows that the presence of dynamic resources reduces the response time of the resource discovery in both algorithm forms. This is explained by the existence of virtual resources from which the algorithm can access several dynamic resources realizing the same function at once.
In the dynamicity aspect tests, in particularly where RG contains only 200 static resources [200,0], the response time of the executed resource discovery process in its enhanced form, is the same as the response time of the resource discovery proposed in the work [23]. This is due to the fact that in [23], where dynamic resources are not considered, it is assumed that the user knows the URL of the first resource from which the discovery process sill start, thus the similarity with the enhanced search in our case. However, in our solution, the end-user expresses his request through a single function, which is selected form a list of functions provided by the available resources in the Web environment at the current instance.

[6]The prototype code is available online: http://tinyurl.com/y7e78n24

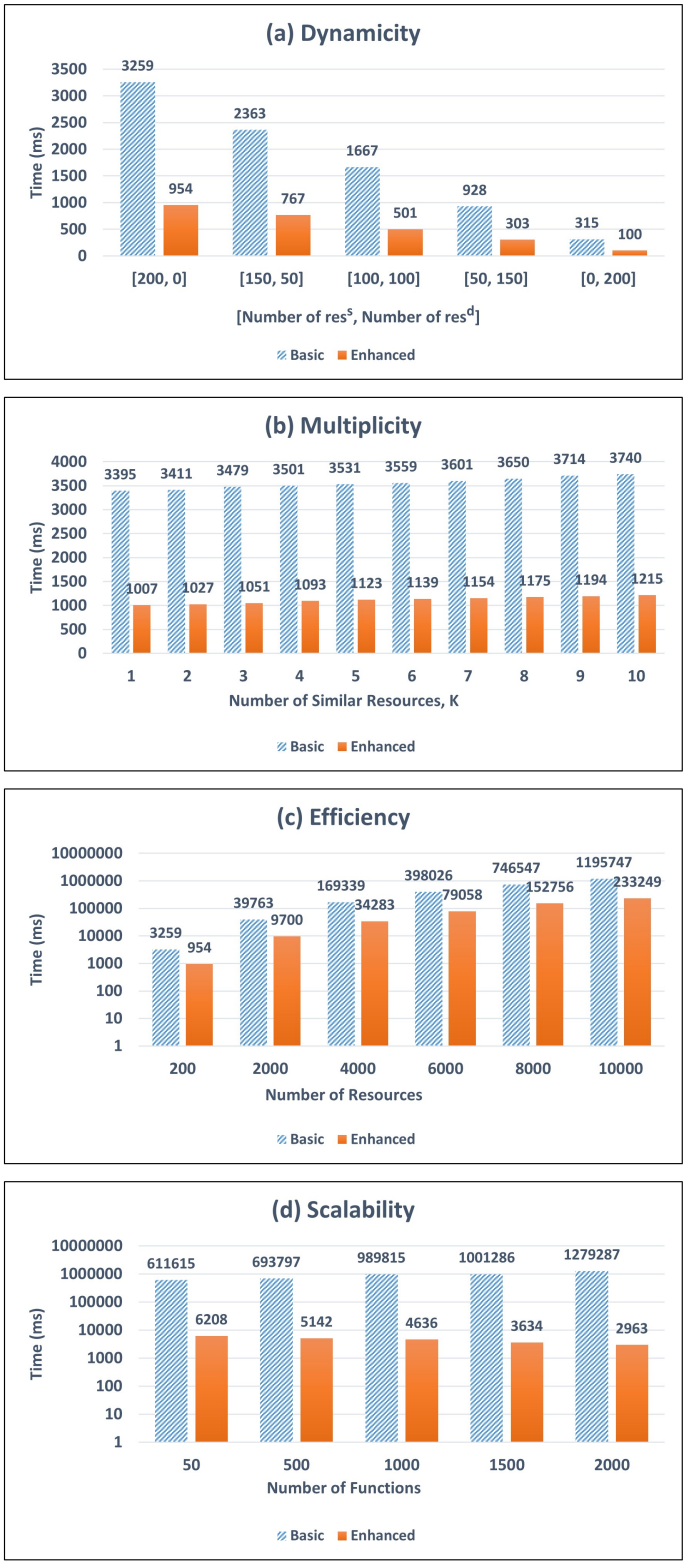


Figure 4.15 – Performance results

For the **multiplicity aspect**, we built a graph of 400 resources (200 static and 200 dynamic), and run our tests while varying k. Figure 4.15-(b) shows that the response time increases with the evolution of k in both algorithm forms. This is due to the additional resources that the discovery process will have to find realizing a single required function. However, the results are more satisfactory using the enhanced search.

For the **efficiency aspect**, Figure 4.15-(c) shows that the response time of the enhanced search is better than the basic one, when increasing the number of resources from 200 to 10000[7]. This highlights the utility of the indexing schema in large Web graphs.

As for the **scalability aspect**, we fixed the number of resources to 10000, and varied the number of functions (from 50 to 2000). In the first 3 tests, RG graphs consist of k dynamic and static resource for each function, with k decreasing from 100, 10, and 5 respectively. In the rest 2 tests, k is defined unequally between static and dynamic resources of the RG graphs. As such, when the number of functions is 2000, RG graph includes 2 dynamic resources (k=2) and 3 static resources (k=3) for each function. Figure 4.15-(d) shows an increase in the response time with the evolution of the functions number with the basic search. This is due to the variety of resources providing numerous functions that are different from the required ones. However, the response time decreases in the enhanced search. This is explained by the reduction of the number of resources providing the necessary functions related to user's request, since the functions number increases while the total resources number is fixed.
Figure 4.15-(a) results highlight the benefit of defining virtual resources containing dynamic resources realizing the same function. It facilitates and speeds-up the access to k-resources at once during resource discovery. The tests in Figures 4.15-(b), 4.15-(c) and 4.15-(d) show that the time curve in both algorithm forms generally increases with the evolution of the functions number, the resources number, and the similar resources number, k. Except for the scalability aspect, and with the enhanced search, the time curve decreases. This is due to the fixed number of resources while increasing the number of the provided functions, which leads to a decrease in the number of resources providing the required functions necessary to realize user's request. Our experiments prove that using our indexing schema optimizes resource discovery in all graphs setups. This can be seen through the difference between the results of the basic and the enhanced searches, especially when the resources number is high (10000 resources) as shown in Figures 4.15-(c) and 4.15-(d).

4.5.3 Scenario 2: Discovery Evaluation based on Resource Location

In the tests of the second scenario, we identify all the resources responding to the required functions necessary for r, where k=0. As such, all the resources providing the required data collection functions within the needed location are discovered, along with all the resources realizing the other required functions. In the experiments, the generated FG consists of 50 ordered functions. The response time covers the search in IdS for the necessary data collection

[7]Each graph contains the same exact number of static/dynamic resources

resources, and the traversal of the resources links, using BFS, to identify the other required resources. Figure 4.16 shows the generated results of the resource discovery performance in 3 different cases. The tests in the first case were executed on a resource graph containing 2000 resources (1000 static and 1000 dynamic). The data collection resources were distributed on 12 different locations. With the aim to identify the resources needed to answer a given function, we varied the number of the resources providing the required data collection function within the needed location. The results in Figure 4.16-(a) show that the response time increases with the evolution of the number of relative data collection resources existing in the required location. This is due to the growing number of the necessary resources in the needed location, and to the increasing number of the HATEOAS links to be traversed, which are included in the description of each identified data collection resources. In

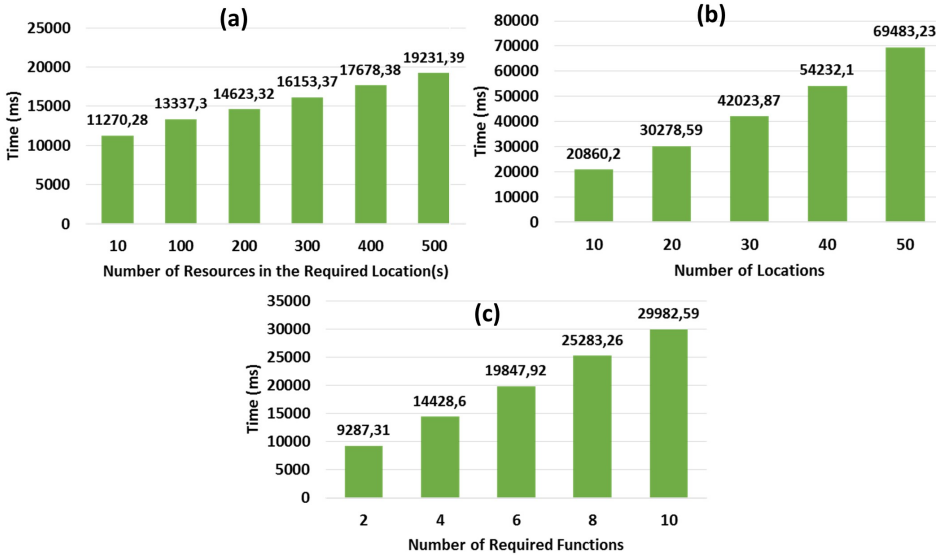


Figure 4.16 – Performance results of the resource discovery

the second case, we built 5 graphs of 4000 resources containing, each, 2000 static and 2000 dynamic, and run the tests while varying the number of the required locations. Each location contained several resources, including 20 resources providing the required data collection function for r. The results in Figure 4.16-(b) show that the increasing number of locations relative to r leads to an evolution in response time during resource discovery. This is due to the growing number of resources to discover in the different locations, and the time taken to crawl their HATEOAS links. In the third case, a graph of 2000 resources (1000 static and 1000 dynamic), is generated, and data collection resources were distributed on 12 different locations. In the tests, we varied the number of data collection functions required in the needed location. Each function is provided by 40 resources. The results in Figure 4.16-(c) show an evolution in the response time with the growing number of required functions. This is explained by the increase number of resources that are to be identified providing the necessary functions. The generated graphs in all cases show a positive linear curve, denoting that the time of resource discovery process increases linearly with the number of resources in the required location, the number of locations relative to r, and the number of required data collection functions. The results also highlight the important increase of

response time with the growing number of locations relative to r, comparing to the other graph curves.

4.6 Summary

In this chapter, we proposed an automatic location-aware solution for k-resources discovery in hybrid Web environments connecting: (1) static resources that are established to be always available and respect the HATEOAS principle, and (2) dynamic resources, which can be connected to and removed form the environment at different instances. The solution, which covers the resource discovery process of the automatic resource composition in the StARC framework, consists of a formal representation that models the resources (both dynamic and static) in one single resource graph. The resource graph can be traversed by several graph-based algorithms adapted to follow the semantic annotations integrated in the descriptions (expressed with Hydra vocabulary) of the traversed resources, to discover the necessary k-resources realizing the same required function. In the approach, we defined an original 3-dimensional indexing schema that maps the resources to their provided functions and corresponding location (whenever they are exposed by objects). As shown in our conducted experiments, the indexing schema allows identifying data collection resources based on their location. It also enhances resource search in large Web environments to avoid crawling the resource graph from the Root and causing big search time. This has been proved in the evaluation of our solution performance on 4 aspects: dynamicity, multiplicity, efficiency, and scalability.

In the next chapter, we tackle the challenge of selecting the appropriate resources from the ones identified during resource discovery, to form the suitable compositions realizing user's request in hybrid Web environments.

Chapter 5

QoR-based Resource Selection for Hybrid Web Environments

"Imagine the possibilities"

Ralph Marston

As many resources are being published on the Web, selecting the appropriate ones to form a composition of resources satisfying user different needs is becoming a challenging task. This is due to: (1) the growing number of resources providing identical functions, which calls for the use of Quality of Resource (QoR) to distinguish between them, and (2) the transient nature of resource availability in hybrid environments connecting not only static resources (established to be always available) but also dynamic resources, that can be connected/disconnected at different instances. In this chapter, we present a QoR-driven approach for resource selection that can be applicable in hybrid environments. The proposed selection solution is the next process to be considered after the discovery process, which is relative to the automatic resource composition in the StARC framework (presented in Chapter 1). It allows selecting the appropriate resources to form i-compositions (with $i \in \mathbb{N}^*$), offering different implementation alternatives, taking into account resource availability, QoR constraints, matching of resources Inputs/Outputs and user different needs (e.g., compositions with the highest QoR, and compositions having acceptable QoR but formed in satisfactory delays). Experiments are conducted in different environments setups (i.e., different resource and function graphs topologies) to study the performance of our work, and analysis are made to evaluate our resources/compositions' quality model against existing ones.

5.1 Introduction

In this chapter, we present our proposed resource selection approach, which is the next step to consider after the k-resources discovery solution presented in Chapter 4. Similar to the discovery approach, resource selection can be applicable in hybrid Web environments providing:

- **Static resources**, i.e., established to be always available on the Web. These resources can be exposed by Web applications, connected WoT objects [17] (stationary or mobile) as smart devices, etc.
- **Dynamic resources**, i.e., exposed by stationary/mobile WoT objects, that can be connected to and removed from the environment at different instances.

As many resources can be identified during resource discovery, selecting the suitable ones to form a composition that satisfies user needs is becoming a complex task. As such, several challenges arise:

1. **Selection of the appropriate resource for a function**: With the growing number of resources responding to the same function, selecting the appropriate one, while considering user constraints (if they are given), is a non-trivial task for end-users. For this matter, taking into account the Quality of Resource (QoR) attributes used to differentiate resources having identical functions [116], is important to select the suitable resource for a function. The increasing number of candidate resources and their various QoR attributes [59] (e.g., Availability and Cost) require an automatic approach that facilitates the task for end-users, and accelerates the selection process. Also, during selection, considering the matching of the input and output (I/O) parameters of the related resources, is essential to generate composition solutions that fit efficiently users needs.
2. **Forming different composition alternatives**: In hybrid Web environments connecting dynamic resources, the selected resource(s) for a composition may be unavailable for execution (disconnected from the environment). In order to avoid repeating the discovery and selection processes to form new suitable composition, providing i-compositions (i $\in \mathbb{N}^*$), i.e., a set of compositions having different implementation alternatives, during resource selection becomes important. These compositions achieve the workflow (representing the dependencies between different functions to be satisfied by multiple resources) that is necessary for user's request by using, each, a different set of resources. This gives the possibility to substitute a composition that misses a resource (due to a disconnection from the environment for instance) by another one consisting of available resources. Thus, a selection approach that considers resource dynamicity is necessary. Furthermore, in some cases, users require optimal composition having the highest possible scores, others may need optimistic composition having acceptable scores obtained in more satisfactory delays, and in other cases, users ask for solutions having acceptable scores while considering resource dynamicity (whenever a dynamic resource is unavailable during a composition execution, there is always another composition consisting of available

resources that can take over). Therefore, forming compositions that are adaptive to different user needs becomes essential.

In the literature, many approaches (i.e., REST-based and SOAP-oriented) addressed service selection [22, 25, 114, 96]. Some works [22, 114, 120] were based on Quality of Services (QoS) to select the most suitable ones according to user constraints or preferences, without taking into account I/O service matching and service dynamicity. Others [96, 66] dealt with the service selection problem as an AI planning problem aiming at finding a sequence of services starting from given inputs and leading to the desired outputs, without considering service matching on the functional level, their QoS, and dynamicity. Also, and to the best of our knowledge, none of the existing service composition approaches [41, 24], is adaptive to form several types of compositions realizing different user needs (e.g., optimal compositions having the highest scores, optimistic compositions having acceptable scores but obtained in more satisfactory delays, etc.).

To address the aforementioned challenges and existing limitations, we present, in this chapter, a QoR-driven resource selection approach that forms i-compositions in hybrid Web environments (connecting static and dynamic resources). To do so, we first propose a formal QoR-based graph that models the identified resources (static and dynamic) during resource discovery, with their relations. In the model, where the nodes represent the discovered resources and arcs represent resource relations based on their provided functions, we define a QoR score for each resource, and a quality score for each possible composition. Then, we define a selection strategy adaptor that forms i-compositions based on user QoR constraints and I/O matching of related resources, while considering resource dynamicity and user's requested composition type (e.g., optimal compositions having the highest scores, and optimistic compositions having acceptable scores but obtained in more satisfactory delays). Resource selection is automatic, i.e., based on semantic annotations integrated within resource descriptions expressed using Hydra [64] in our work.

The rest of the chapter is organized as follows. Section 5.2 motivates our work, and describes the main challenges and needs. Section 5.3 discusses related work and shows the originality of our solution. Section 5.4 details our resource selection approach. Section 5.5 evaluates the performance of the proposed solution and compare our defined QoR model against existing works. Finally, Section 5.6 concludes the chapter.

5.2 Motivation, Challenges and Needs

We motivate our work with a scenario illustrated in the BEMServer Web-based management platform of HIT2GAP and SIBEX projects. In the scenario, we assume that BEMServer, and apart from connecting static resources, is extended to allow connecting dynamic resources exposed by objects. Several requests can occur in such Web platform. We consider a building manager that wants to predict the temperature of a specific building zone. To express his demand, specified by "ATP" (Air Temperature Prediction) function, he can send 2 different requests, as explained before in Section 4.4 of Chapter 4: (1) a context aware request, r^{ca}, in which the prediction results

depends from data collected from the 3 nearest devices in his office (with a 2m range), and (2) a non-context aware request, r^{nca}, in which the required data is collected from devices located in the conference room A, independently from the building manager position. In the example, we consider that the required resources providing the needed functions to realize "ATP", have been already identified during resource discovery.

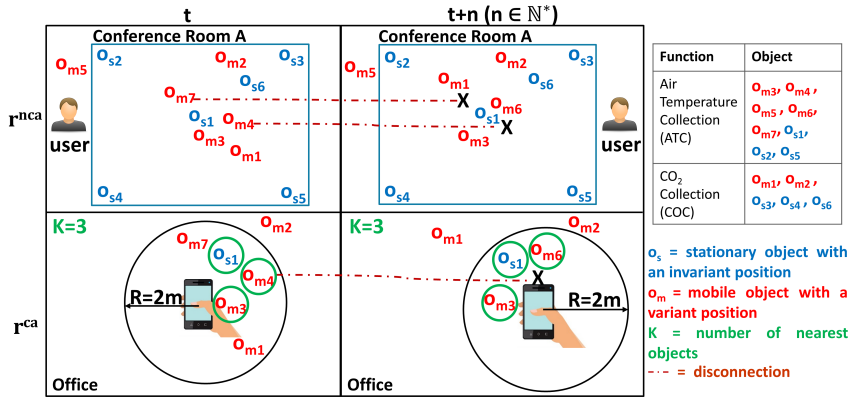


Figure 5.1 – r^{nca} and r^{ca} examples in BEMServer

To satisfy building manager demand, several challenges should be addressed, as illustrated in Figure 5.1:

1. **Select the appropriate resources to form a suitable composition**. When several resources are identified responding to the same required function as "ATC" (Air Temperature Collection), selecting the appropriate ones among all others (o_{s1}, o_{m3}, and o_{m4} for r^{ca} at instant t) is not an obvious task for end-users, as the building manager. For this matter, QoR plays an essential role to select the suitable resources. For instance, as shown in Table 5.1, object o_{s1} is better than o_{m3} and o_{m4} as it has: (i) full battery capacity denoting a full availability, (ii) continuous connectivity to the environment (since it is static), (iii) cost free when using it, and (iv) high usage rate (i.e., it has been invoked many times). Considering these non functional aspects (QoR), allows selecting the appropriate resources among other candidates. As many candidate resources may

Table 5.1 – Quality aspects of the ATC objects

	Battery Capacity (Availability)	Continuous Connectivity (i.e., 0 if static and 1 if dynamic)	Cost	Rate
o_{s1}	100	0	0	65
o_{s2}	90	0	10	60
o_{s5}	80	0	25	60
o_{m3}	75	1	0	60
o_{m4}	65	1	75	20
o_{m5}	70	1	0	75
o_{m6}	41	1	0	50
o_{m7}	45	1	10	10

be connected to the BEMServer with various QoR, resource selection requires an automatic approach that facilitates and accelerates the task for end-users. Such approach should also consider the suitable I/O matching between the linked resources. This is done to: (1) guarantee efficient composition results, and (2) allow forming other potential compositions

by replacing dynamic resources unavailable for composition execution with other resources that can be suitably linked to the rest of resources in the composition. Moreover, in some cases, the building manager may require:

(a) **Prediction results using the most qualified resources among others.** In this case, the resources to be selected are the ones having the highest values of quality aspects, and independently of the selection response time, since the building manager may desire to adjust the conference room A temperature necessary for a meeting that will start in the late afternoon. As such, and as per Figure 5.1, among the ATC objects located in the conference room A at t+n for r^{nca}, o_{s1} is the most qualified object to use, as it has the highest quality aspects values (see Table 5.1).

(b) **Fast but good prediction results.** In this case, and as the building manager might be feeling very hot in his office, he needs fast prediction results to regulate his office temperature at instant t using r^{nca}. This is done by selecting the first resource realizing his demand without the need to check others. Though requiring fast results, it is important that the resources to be selected have minimal quality aspects values (e.g., Availability $\geqslant$ 75, Continuous connectivity, Cost $\leqslant$ 25, and Rate $\geqslant$ 60), guarantying good predictions. In this context, and based on Table 5.1, o_{s2} has good quality values, therefore if it is identified first before the other objects, it will be selected.

(c) **Good and always available results.** In this case, the building manager requires to have results using resources with good quality aspect values, but at anytime of his request, i.e., even if dynamic resources are disconnected from the Web platform there are always other resources that can take over. Thus, the ATC object that will be selected should be always connected to the environment (i.e., static) at both instants t and t+n, and having good quality aspects. As such, using r^{nca}, all three static objects at t and t+n, i.e., o_{s1}, o_{s2}, and, o_{s5}, have good quality aspects, thus, the first one identified will be selected.

In other particular cases, and for each of these previous requirements, he may need to have:

- **Trusted results**, generated by only static resources already provided by the Web platform (e.g., o_{s1}, o_{s2}, and o_{s5}).
- **Cost free results**, using resources without any charge (i.e., Cost = 0 as o_{s1}, o_{m3}, o_{m5}, and o_{m6}).
- **Effective results**, using resources with high availability rate (e.g., o_{s1}, o_{s2}, o_{s5}, and o_{m3}).
- **Efficient results**, using resources that have been used several times in other scenarios (e.g., o_{s1} and o_{m5}).
- **Qualified results**, using resources having the highest QoR values (e.g., o_{s1}).

- **Reliable results**, using resources that can be linked in the most proper way (i.e., best I/O matching between the related resources).

Thus, it is necessary to consider user needs and constraints, and adapt resource selection accordingly.

2. **Form several composition alternatives**. Selected dynamic resource(s) for a composition may be unavailable during execution. As such, for r^{nca} and at instant t, 5 mobile objects providing "ATC" are positioned in the conference room A. If o_{m4} provides the appropriate resource among these objects, it will be selected to take part in the composition. However, at t+n, o_{m4} is disconnected, and thus, the composition will miss a resource if the composition execution time is $\geqslant$ t+n. The same applies to r^{ca} for which o_{m4} is no longer available. To avoid repeating both resource discovery and selection processes, to form a new suitable solution with available resources, it is important to identify i-compositions during resource selection, with i $\in \mathbb{N}^*$.

To address these challenges and respond to the different user needs, we propose a QoR-driven resource selection solution adapted to: (i) different requested composition types (e.g., Optimal, Optimistic and Optimistic Cost-free), and (ii) user constraints expressed in spatial queries. Our selection approach considers I/O matching between the related resources and resource dynamicity (when it is necessary), to form the required i-compositions.

5.3 Related Work

As we are handling resource selection to form i-compositions, we present in this section different approaches related to such research area. Due to the few works covering REST services, our review concerns several approaches that we considered interesting to our work, independently from the protocol/technology used for Web services implementation. In the review, we categorized the service selection approaches into three groups: (1) QoS-based approaches that consider QoS, (2) I/O similarities-based approaches that rely on services I/O matching, and (3) k-services compositions approaches that form k-compositions (i.e., more than one solution). During the survey, we compare the approaches according to the following criteria:

- **QoS-based:** denotes the ability to consider quality of service attribute(s) during service selection. It is an essential factor that helps in distinguishing services providing identical functions.

- **Consider I/O matching:** is the ability to consider the inputs/outputs matching of related services. This is important to ensure an efficient matching between services, and thus reliable composition results.

- **Dynamicity-aware:** denotes the ability in considering resource dynamic aspect, and allowing thus the possibility to substitute a composition including unavailable resources with another one.

- **Generate different compositions types:** is the ability to form several compositions types that fit different user needs, e.g., optimal compositions having the highest scores, optimistic solutions having acceptable scores but obtained in more satisfactory delays, etc.

5.3.1 QoS-based Approaches

In [22], a quality-driven solution for selecting semantically described RESTful services is presented. The approach uses a set of quality attributes (i.e., Performance, Availability, and Reputation) incorporated into each resource description expressed with Hydra, and implements a skyline-based algorithm that reduces the set of candidates for a given task. The selection phase aims at selecting the best candidate for each task to obtain an overall QoS that matches with the user's QoS profile. In the work, resource selection is done at the same time of resource discovery, but in different configurations: (1) On the fly selection, in which the selection process is executed at the same time as the resource discovery goes on, and (2) N-periodic selection, during which the launching of the selection process is done every time there are N new candidates identified for a given task.

In [25], a heuristic-based approach is proposed to solve the QoS-aware Web Service composition problem. The solution aims at maximizing the QoS of the overall Web Service composition, while considering preferences and constraints defined by the user. In the work, a heuristic called H1_RELAX_IP is proposed that uses a backtracking algorithm on the results computed by a relaxed integer program. The evaluation of H1_RELAX_IP have revealed good results compared to a linear integer programming based solution with regard to the computation time, especially while increasing the number of candidate Web Services and process tasks. Moreover, two meta-heuristics have been defined to improve H1_IP_RELAX results: (1) H2_SWAP, which tries to find a composition having a higher QoS by randomly replacing Web Services of the execution plan calculated by H1_IP_RELAX, and (2) H3_SIM_ANNEAL, which temporarily accepts worse solutions during the optimization process to be able to leave local optima and possibly find the global optimum. The solution supports only sequential Web service compositions.

Authors in [126] propose a resource selection approach that considers design preference in cloud manufacturing system. The solution presents a QoS ontology from which customers express their preference, and providers declare the policy they are using. In the work, the resource are described with five QoS properties: Cost, Time, Reliability, Availability, and Reputation. The Reputation value is calculated based on data provided as a feedback from resource users, e.g., very high, high, normal, low, and verylow. In order to quantify such given data, the proposed solution defines a QoS computation model based on a fuzzy theory using a triangular fuzzy number [10]. Based on the defined model, the particle swarm optimization (PSO) [14] algorithm is applied to select the service composition.

In [114], an approach for a service selection based on both qualitative and quantitative user QoS preference with services trust properties is presented.

The solution is applicable in Big Data Web environments consisting of massive migrated services to the cloud, i.e., business applications. In the work, user preferences are of three aspects: (1) quantitative QoS property, (2) conditional preference on qualitative QoS property (i.e., they are not expressed in numerical value), and (3) the relative importance about one QoS property to another. The proposed service selection approach consists first on handling inconsistent quantitative QoS properties by normalizing them in a range of [0,1]. Second, QoS match degree calculation for each quantitative/qualitative QoS property of the candidate services by integrating also trust with user constraints. And finally, defining a linear weighting function to rank how each service matches the user's requirements through a Multi-objective Constrained Model.

In [120], several algorithms with different techniques have been developed for QoS-aware service selection. The algorithms are based on the artificial bee colony (ABC) [61], an implementation of swarm intelligence that is used in solving several real-world problems, especially the numerical optimization problems. Using an approximate approach for the neighborhood search of ABC, the developed algorithms enable an effective local search in the discrete space of service selection, in a way that is analogical to the search in a continuous space.

Discussion: Although, these approaches take into account QoS attributes and user constraints/preferences, they do not consider I/O matching between the linked services, and service dynamicity. Moreover, they are not adapted to generate different composition types to realize different user needs.

5.3.2 I/O similarities-based Approaches

The work in [96], presents a graph-based framework for automatic service composition, by focusing on the I/O semantic parameter matching of services. Starting from a given user requirements in terms of inputs and expected outputs, the framework produces a service composition graph that is generated on the basis of the relevant services matching user requirements. The generated graph contains all possible service compositions that satisfy user's request. In the work, different techniques (e.g., the interface dominance optimisation allowing to substitute the original services of the graph by abstract interfaces that capture the functionality of the dominant or equivalent services) are used to group and reduce the number of services, and an optimal search is performed over the reduced graph to identify the optimal composition.

In [66], a formal model, i.e., Causal Link Matrix (CLM), is provided for an AI planning-oriented service composition. It pre-computes the I/O semantic similarity between a finite set of services, according to causal links, which are logical dependencies among input and output parameters of different services. Inputs and outputs parameters are concepts in an ontology. The CLM, which aims at storing all valid causal links between the existing services, consists on columns, rows and entries. The columns of the CLM are

labelled by the inputs parameters of the services and the concepts describing the desired goal, and the rows are labelled by the inputs of the services. Each entry is defined as a set of pairs (S, score), where score is the semantic matching degree between an output parameter of S (whose input matches the corresponding row concept) and the corresponding column concept. The work also proposes a regression-based approach that uses the CLM to identify the services matching the required goal concepts.

The work in [15] proposes a solution for a semantic Web service composition. It formalizes the composition problem using a directed acyclic graph (DAG) representation of services that can be composed to obtain the desired service. As such, the composition problem is defined as finding automatically a DAG of semantically matched services in terms of inputs/outputs and pre-conditions/post-conditions, to realize user's request expressed with a set of inputs, pre-conditions, outputs, and post-conditions. Services are described semantically in an ontology, i.e., OWL-S[1]. The work proposes a methodology to compute the trust rating of the composition solutions based on individual ratings of service providers. An automatic generation of OWL-S descriptions of the new composite service is also presented to allow the execution and the registration of the composition.

Discussion: Although the aforementioned approaches consider I/O services matching (on the semantic level), they do not consider the functional aspect of the related services, nor even QoS attributes. Also, these works are not intended to be applied in hybrid Web environments that provide dynamic services.

5.3.3 k-service Compositions Approaches

A top-k automatic service composition solution is presented in [41]. The solution consists of three phases. First, the prepossessing phase, in which the services are transformed into rules formed, each, by the inputs, outputs and QoS of a service. From these rules, a rule repository is constructed. The second phase is service filtering that reduces the set of the services candidates. The filtering is based on the I/O semantic matching between services, starting from a set of inputs given by the user. The final step is related to the top-compositions identification approach, which adopts the idea of MapReduce, by mapping the top-k service compositions into multiple tasks that can be executed in parallel. The output of each parallel task are the generated solution subgraphs. The generation of these subgraphs is done using a backtracking algorithm based on Depth First Search [97]. A central agent then merges the generated solution subgraphs for each concept of the user's requested outputs, to produce the final top-compositions. The solution considers one quality of service, i.e., response time.

In [24], an approach for composing the top-k DaaS (Data as a Service) services is proposed to answer user fuzzy preference queries. The latter are

[1] `https://www.w3.org/Submission/OWL-S/`

based on fuzzy terms (e.g., "cheap" for service price), and expressed in a modified version of SPARQL. In the solution, different constraints inclusion methods are used to compute the matching degrees between the services' fuzzy constraints (describing each service) and the fuzzy preferences involved in the user query. A fuzzy score is associated for each service using the Fuzzy-Pareto-Dominance method [13], to rank/order the services. The scores computed are then leveraged to compute each composition and find the top-k ones. The work focuses on fuzzy constraints rather than some given quality of service.

In [56], an algorithm called Key-Path-Based Loose (KPL), is used to address top k query of QoS-aware automatic service composition. KPL is extended to support multiple QoS measurements, and uses generated directed acyclic graphs representing, each, a composition. Each composition has an overall QoS score computed based on its atomic constituent services. In the work, the compositions are generated by worsening the optimal QoS, without identifying all possible composition alternatives that guarantee the optimal QoS.

Discussion: Although these works produce several service compositions, and consider QoS attributes, they are not designed to handle service dynamicity, nor the generation of different composition types answering user needs. Also, only in [41] I/O matching and QoS attributes are used in the same approach.

5.3.4 Evaluation Summary

Table 5.2 shows the evaluation summary of existing service selection approaches based on the identified criteria. We used "+" symbol to express a positive coverage for a criterion, and "-" symbol to express a lack of a criterion coverage. As seen in the table, each category of approaches covers mainly one criterion, with the exception of the work [41] that considers both QoS attributes and I/O matching of services.

Table 5.2 – Evaluation of existing service selection approaches w.r.t. the identified criteria

		QoS-based	Consider I/O Matching	Dynamic-aware	Generate Different Compositions Types
QoS-based Approaches	[22]	+	-	-	-
	[25]	+	-	-	-
	[126]	+	-	-	-
	[114]	+	-	-	-
	[120]	+	-	-	-
I/O Similarities-based Approaches	[96]	-	+	-	-
	[66]	-	+	-	-
	[15]	-	+	-	-
k-service Compositions Approaches	[41]	+	+	-	-
	[24]	+	-	-	-
	[56]	+	-	-	-

5.4 A QoR-driven Resource Selection for i-compositions

5.4.1 General Overview

Figure 5.2 shows the process overview of our resource selection approach, relative to the automatic resource composition in the StARC framework. The solution is used to form i-compositions responding to user's request, and adapted to user's request type. The latter includes one of the following desired compositions types: (i) optimal, denoting the compositions having the highest scores, (ii) optimistic, referring to the compositions having acceptable scores, i.e., $\geqslant$ a specific computed threshold (see Section 5.4.4), or (iii) hybrid, denoting compositions having acceptable scores but whose dynamic aspect is considered, guaranteeing the existence of a composition at any instance. The composition types can be followed optionally by other subtypes (e.g., trusted, denoting that only static resources can be part of the compositions, and cost-free, denoting that cost-free resources can be involved in the compositions). The user's request, r, defined in Chapter 4 (see Definition 7), is extended in this chapter to include user constraints, C. Thus, we obtain the new formal definition of r as:

Definition 13. $\boldsymbol{r = (f, P, k, C)}$*, where:*

- $\boldsymbol{f}$*,* $\boldsymbol{P}$*, and* $\boldsymbol{k}$ *are the same as in Definition 7*
- $\boldsymbol{C}$*, is the given user constraints according to which, i-compositions are obtained.* $\boldsymbol{C = Q_c \cup i \cup W \cup d}$*, with:*
  - $\boldsymbol{Q_c = Q_c^{res} \cup Q_c^f}$*, refers to the set of constraints given to the resources (*$\boldsymbol{Q_c^{res}}$*) and to their provided functions (*$\boldsymbol{Q_c^f}$*), with* $\boldsymbol{Q_c^{res} = \bigcup_{i=1}^{n} \{q_i^{res}\}}$*, and* $\boldsymbol{Q_c^f = \bigcup_{j=1}^{m} \left\{q_j^f\right\}}$*, and where:*
    - *n is the number of attributes used to describe a resource and m the number of attributes describing its provided functions. In this work, we use 4 basic attributes: "Dynamicity" and "Availability" to describe a resource, and "Cost" and "Usage" to describe resource functions.*
    - $\boldsymbol{q_i^{res}|q_j^f = [min_{i|j}\text{-}max_{i|j}]}$*, where* $\boldsymbol{min_{i|j}}$*,* $\boldsymbol{max_{i|j}}$ *are, respectively, the minimum and maximum values desired by the user for* $\boldsymbol{q_i^{res}}$ *or* $\boldsymbol{q_j^f}$*. User constraints values can be given to the basic attributes, and for other attributes that can be added later on in resource descriptions.*
  - $\boldsymbol{i \in \mathbb{N}^*}$*, is the desired number of the formed compositions. By default i=1, and can be only specified for optimal and optimistic compositions main types. For the hybrid composition type, the number of solutions depends from resource dynamicity aspect of the formed compositions (see Section 5.4.4).*
  - $\boldsymbol{W = \{w_{qor}, w_{io}\}}$*, are the weights given respectively to the score of the resources and their I/O matching, while computing compositions score (see Section 5.4.3).* $\boldsymbol{w_{qor}, w_{io} \in R^+}$ *and are bounded by [0, 1]. By default,* $\boldsymbol{W = \{1, 1\}}$*.*

- ***d***, *is the degree value rate (in %) of a computed threshold, T (see Section 5.4.4), that refers to the minimal acceptable score of the i-compositions, i.e., optimal and hybrid.*

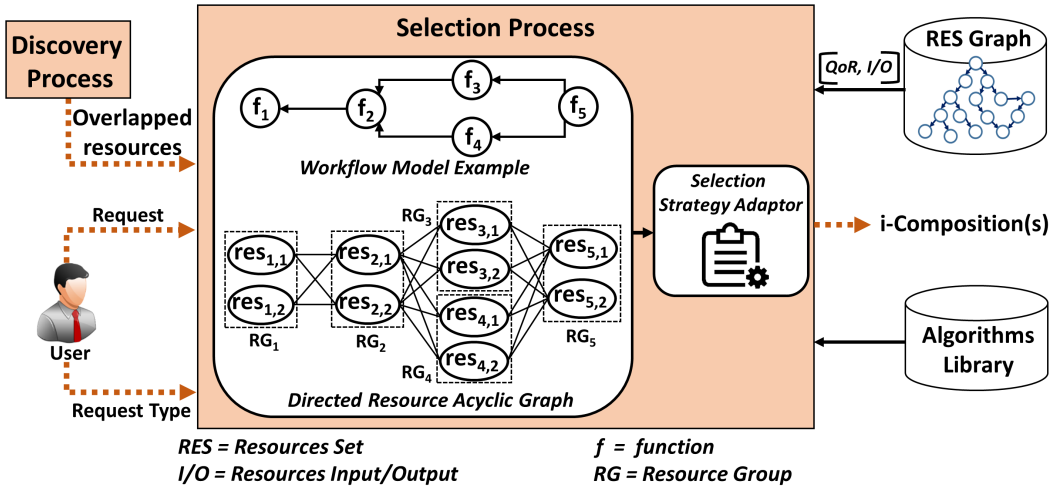


Figure 5.2 – Overview of the resource selection approach

To briefly describe our resource selection solution, we present in Figure 5.2, the framework overview of the Selection Process (SP). SP is executed when there are several candidate resources identified during resource discovery, for at least one required function necessary to realize user's request. The required functions form a Workflow Model, WM, as defined in the function graph, FG (see Definition 6). During the process, the resources realizing an identical function are grouped into the same resource group, RG. Each resource of a RG can be linked to the resources included in the RG related to the next function (as defined in WM), forming a Directed Resource Acyclic Graph, DRAG. Using a graph algorithm, SP traverses DRAG to compute the I/O matching of the linked eligible resources (whose QoR values respect user constraints defined in Q_c), and form the necessary compositions. However, in order to satisfy user different requested composition types (e.g., optimal compositions having the highest scores, and optimistic compositions obtained in more satisfactory delays), we define a Selection Strategy Adaptor (SSA) that adapts to user needs to form the required i-set of resource compositions. SSA allows not only to produce optimal compositions having the highest scores, but also compositions having acceptable score, i.e., $\geqslant$ a computed threshold, T (see Section 5.4.4, obtained in more satisfactory delays. Such threshold is computed for each of the desired composition types: optimistic and hybrid.

5.4.2 Preliminaries

Before elaborating on our resource selection approach, we extend in this section, the static/dynamic resource definition, $res^{s|d}$ (see Definition 10 in Chapter 4). This is done to include the set of quality attributes, Q_{res}, describing the non-functional properties of a resource, and the set of attributes describing its providing functions. As such, $res^{s|d}$ formal definition is extended as follows:

Definition 14. $res^{s|d} = (c, id, loc, F, L, Q_{res})$, *with:*

- c, id, loc, *and* L *are the same as in Definition 10*
- $F = \bigcup_{i=1}^{\mathbb{N}^*} \{f_i\}$, *is the set of functions provided by res.* $f_i = (n, I, O, m, Q_f)$, *where:*
  - n, *refers to* f_i *name*
  - I, *denotes the input(s) of* f_i
  - O, *denotes the output(s) of* f_i
  - m, *is the HTTP verb used to call* f_i
  - $Q_f = \bigcup_{i=1}^{\mathbb{N}^*} \{(qf_i : vf_i)\}$, *refers to the set of quality attributes related to* f_i, *with* qf_i *the name of the attribute (e.g., Cost and Usage), and* $vf_i \in \mathbb{R}^+$
- $Q_{res} = \bigcup_{i=1}^{\mathbb{N}^*} \{(qres_i : vres_i)\}$, *is the set of attributes related to res, with* $qres_i$ *the name of the attribute (e.g., Dynamicity and Availability), and* $vres_i \in \mathbb{R}^+$.

In the literature, we find various QoR attributes, e.g., Availability, Usability, Cost, that are used to differentiate resources with identical functions [116]. In this thesis, we consider 4 basic attributes, where some are directly related to the resource itself (Q_{res}), and others related to each of its provided functions (Q_f):

- **Dynamicity**, is the quality aspect of whether the resource is always available (static) or not (dynamic). It is either equal to 0, i.e., the resource is static, or equal to 1, i.e., the resource is dynamic. In r, users can specify if they want to have dynamic and/or static resource in the i-compositions. As such, for the maximum and minimum values given to the "Dynamicity" attribute, q_1^{res}, in user constraints, Q_c^{res}:
  - If q_1^{res} = [1-0], only static resources can be part of the compositions
  - If q_1^{res} = [1-1], static and/or dynamic resources can be part of the compositions
  - If q_1^{res} = [0-1], only dynamic resources can be part of the compositions
- **Availability**, is the degree (%) to which a resource is operational or ready for immediate use. For resources provided by stationary/mobile objects, it denotes the battery capacity of these objects.
- **Cost**, is the amount of money to pay, in a specific currency, to use a function of a resource. It can be defined by the resource provider (i.e., the organization or person that developed the resource) and/or by the object provider (i.e., the person connecting the object to the environment).
- **Usage**, is a value that is incremented every time a resource function is used. By default it is equal to 0. For each dynamic resource, and to avoid the re-initialization of the usage attribute when being disconnected from the Web, we define a TTL (Time To Live) value denoting the maximum amount of time during which a dynamic resource can be disconnected before the usage attribute value decrements by 1.

QoR are classified into 2 groups [9]: (i) maximization attributes, whose values should be maximized, i.e., the higher the value the higher the quality (e.g., Availability), and (ii) minimization attributes, whose values should be minimized, i.e., the higher the value the lower the quality (e.g., Cost). QoR are used to compute, for each provided resource function, a global score that is defined as: score(res_f) = $\sum_{i=1}^{\mathbb{N}^*} \{vres_i\} + \sum_{i=1}^{\mathbb{N}^*} \{vf_i\}$, such that $vres_i$ (excepting the "Dynamicity" attribute value) and vf_i are normalized using equation (5.1) or (5.2) presented below. In fact, and due to the different QoR dimensions and units, normalizing their values is necessary for the calculation of score(res_f). As in [81], equations (5.1) and (5.2) are used to normalize a QoR value, q_i, with $q_i = qres_i | qf_i$, and $q_i^{'}$=1 if $max(q_i) - min(q_i)$=0. $max(q_i)$ and $min(q_i)$ refers respectively to the maximum and minimum values of q_i among the resources in DRAG.

$$q_i^{'} = \frac{q_i - min(q_i)}{max(q_i) - min(q_i)} \qquad (5.1) \qquad q_i^{'} = \frac{max(q_i) - q_i}{max(q_i) - min(q_i)} \qquad (5.2)$$

Based on the presented res definition, we extend Hydra-based resource description in this chapter, as shown Listing 5.1, to include resource QoR values.

```
{
    "@context": "http://www.h2g.eu/h2g/resdesc/context.jsonld",
    "@id": "http://www.h2g.eu/resdesc/getairtemp.md",
    "location": "Z1",
    "Operation": [{
        "method": "GET",
        "expects": ["h2g:startdate", "h2g:enddate"],
        "returns": ["schema:DateTime", "schema:Float"],
        "function": "ATC",
        "Qf": [{
            "Cost": "15"
        }, {
            "Usage": "4"
        }]
    }],
    "Link": [{
        "entrypoint": "http://www.h2g.eu/predairtemp",
        "method": "GET",
        "relationType": "isComplementary",
        "function": "ATP"
    }],
    "Qres": [{
        "Dynamicity": "0"
    }, {
        "Availability": "75"
    }]
}
```

Listing 5.1 – Extended Hydra with QoR attributes

5.4.3 Formal modeling of a QoR-based Resource Graph

The functions required for user's request define with their dependencies, a Workflow Model, WM, with WM $\subset$ FG. Based on the functions order in WM, the resources identified during the discovery process are linked together,

forming a Directed Resource Acyclic Graph, DRAG. Formally, DRAG is defined as:

Definition 15. ***DRAG = (DRES, Rel, f_{DRES}, f_{Rel})****, where:*

- ***DRES****, is the set of the discovered static and dynamic resources obtained from the resource discovery process (see Chapter 4).*
- ***Rel****, is the set of relations linking the resources to each other.*
- f_{DRES}*, is the function computing the score of each resource function based on QoR values (e.g., Availability and Cost), included in our study in Hydra-based resources description.*
- f_{Rel}*, is the function linking the resources together based on WM, and computing their link score based on their I/O similarities.*

The resources discovered for the same function, form a resource group, RG_f, relative to that function, where: $RG_f = \bigcup_{i=1}^{m} \left\{ res_{(f,i)} \right\}$, with m is the number of candidate resources realizing function f, and $res_{(f,i)}$ refers to the resource res_i providing f. A resource composition, RC, consists of a set of resources included, each, in a different RG_f, where: $\mathbf{RC} = \bigcup_{f=1}^{n} \left\{ res_{(f,i)} \right\}$, such that n is the number of functions in WM, and $i \in m$, with m denotes the number of resources in the correspondent RG_f. During selection, I/O matching between linked eligible resources is computed, forming the score link of these resources. Such score is calculated as: $\text{sim}(res_{(f,i)}, res_{(f',j)}) = \sum_{u=1}^{U} \sum_{v=1}^{V} sim(out_u^{res(f,i)}, in_v^{res(f',j)})$, with:

- $res_{(f,i)}$, $res_{(f',j)}$, denote resources that belong, respectively, to RG_f and $RG_{f'}$, where f precedes f' in WM
- $out_u^{res(f,i)}$, is an output of $res_{(f,i)}$, and U is the number of $res_{(f,i)}$ outputs
- $in_v^{res(f',j)}$, is an input of $res_{(f',j)}$, and V is the number of $res_{(f',j)}$ inputs

In our work, we consider that the matching score between an output of a res and an input of another, is computed using a similarity measure function between keywords (as Jaccard measure [84]), and such that $\text{sim}(res_{(f,i)}, res_{(f',j)}) \in [0, 1]$.
Each RC in DRAG has a score, score(RC), computed using (i) the score of each involved resource providing the needed function, score(res_f), and (ii) the score of the link relating each 2 eligible resources, such that: **score(RC) = Score(RES) + Score(Rel)**, where:

- **Score(RES)** = $\sum_{f=1}^{n} score(res_{(f,i)})$, is the sum of the scores of the involved resources realizing the required functions, such that: n is the total number of functions in WM and i ∈ m, with m denotes the number of candidate resources in the correspondent RG_f.
- **Score(Rel)** = $\sum sim(res_{(f,i)}, res_{(f',j)})$, is the sum of I/O similarity scores of each 2 eligible linked resources in RC, where: f precedes f' in WM, i ∈ [1,m] and j∈ [1,m'], with m and m' denoting the numbers of resources in RG_f and $RG_{f'}$ respectively.

Score(RES) and Score(Rel) can be multiplied, each, by a weight value defined in W in user request's, r (see Definition 13), allowing users to assign them a priority during compositions score calculation.

An example of a DRAG formed by discovered resources is given in Figure 5.3. As per the illustrated Figure, and based on the Workflow model showing the dependencies between the required functions to realize f_5, the identified resources during resource discovery process are grouped into the same group relative to their provided same function. Each resource has a score computed according to the set of the quality attributes values related to its provided necessary function, i.e., vf_i, and the attributes related to the resource itself, i.e., $vres_i$ (see Definition 14). Also, each link relating two resources (e.g., $res_{3,2}$ and $res_{5,2}$) has a similarity measure score (e.g., $sim(res_{(3,2)}, res_{(5,2)})$ between $res_{3,2}$ and $res_{5,2}$). A score(RC) is assigned to each possible resource composition in DRAG, which is represented by a path (see the red circled resources in Figure 5.3) linking one resource belonging, each, to different resource group.

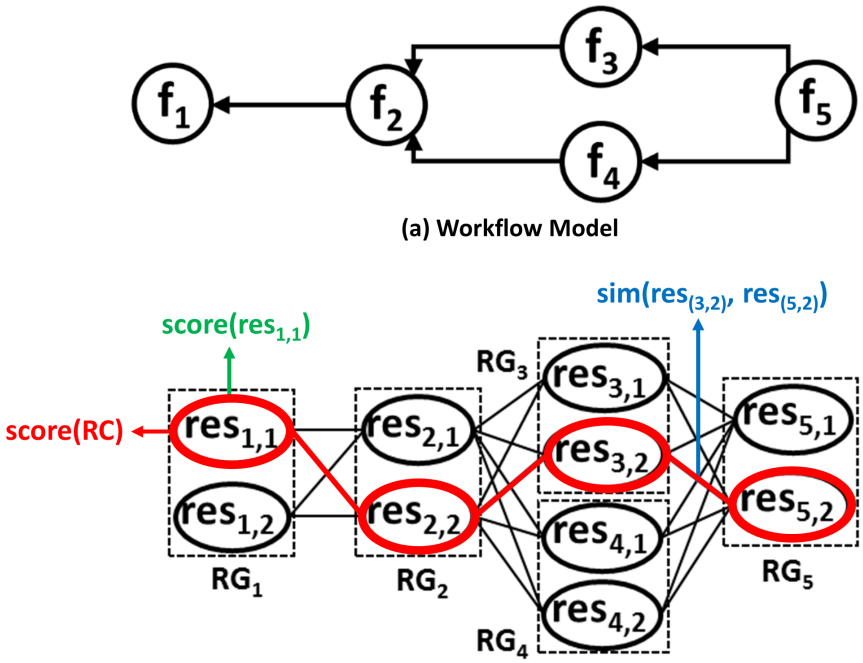


Figure 5.3 – An example of a DRAG showing the scores defined for each of the involved resources, their I/O matching, and each possible composition

During selection, DRAG can be traversed by a graph-based algorithm (e.g., BFS and DFS [19]) to form the compositions with the highest scores, whenever they are requested by the user in the given request type (i.e., optimal). However, in order to allow: (i) retrieving potential solutions having acceptable scores with more satisfactory delays without computing all compositions scores (i.e., composition type = optimistic), and (ii) guaranteeing the existence of a composition despite resource dynamicity (i.e., composition type = hybrid), SP uses the Selection Strategy Adaptor (SSA) described next, to form i-compositions satisfying different user needs.

5.4.4 Selection Strategy Adaptor for i-compositions

In order to satisfy different user needs, we define a Selection Strategy Adaptor (SSA) that allows the generation of 3 main compositions types:

1. **Optimal**, refers to the compositions having the highest score(RC). Examples of such compositions, highlighted in grey, are shown in Table 5.3.

Table 5.3 – Examples of 3 optimal compositions having the highest scores

f_1	f_2	f_3	f_4	f_5	Score(RC)
$res^d_{1,1}$	$res^s_{2,1}$	$res^d_{3,1}$	$res^s_{4,1}$	$res^d_{5,2}$	65
$res^s_{1,1}$	$res^s_{2,2}$	$res^d_{3,2}$	$res^d_{4,2}$	$res^s_{5,2}$	64
$res^s_{1,2}$	$res^s_{2,2}$	$res^d_{3,1}$	$res^d_{4,2}$	$res^s_{5,2}$	64
$res^s_{1,1}$	$res^s_{2,1}$	$res^d_{3,1}$	$res^d_{4,2}$	$res^s_{5,2}$	50
-	-	-	-	-	-
$res^s_{1,2}$	$res^s_{2,2}$	$res^s_{3,1}$	$res^s_{4,1}$	$res^s_{5,1}$	10

2. **Optimistic**, designates compositions having acceptable score(RC) based on a computed minimum threshold. Examples of optimistic compositions having a score $\geqslant 50$, are highlighted in grey in Table 5.4.

Table 5.4 – Examples of 4 optimistic compositions having acceptable score $\geqslant 50$

f_1	f_2	f_3	f_4	f_5	Score(RC)
$res^d_{1,1}$	$res^s_{2,1}$	$res^d_{3,1}$	$res^s_{4,1}$	$res^d_{5,2}$	55
$res^s_{1,1}$	$res^s_{2,2}$	$res^d_{3,2}$	$res^d_{4,2}$	$res^s_{5,1}$	32
$res^s_{1,2}$	$res^s_{2,2}$	$res^d_{3,1}$	$res^d_{4,2}$	$res^s_{5,2}$	51
$res^d_{1,2}$	$res^s_{2,2}$	$res^s_{3,1}$	$res^s_{4,1}$	$res^s_{5,1}$	10
-	-	-	-	-	-
$res^s_{1,2}$	$res^s_{2,1}$	$res^s_{3,1}$	$res^s_{4,1}$	$res^d_{5,1}$	50

3. **Hybrid**, denotes compositions having acceptable score(RC) based on a minimum threshold, and whose resources dynamicity is taken into account to guarantee, at any instance, the existence of an available composition, i.e., it includes available resources that provide all the necessary functions for r. Table 5.5 shows examples of compositions having acceptable score $\geqslant 50$, including one consisting of only static resources. This is done to ensure that even if some dynamic resources are not available for execution (e.g., $res^d_{1,1}$ in the first composition), there is a resource composition whose resources are always available (i.e., static).

For each of these compositions types, we define the following subtypes that can be given optionally:

(A) **Trusted**, refers to compositions having only static resources that are already provided by the Web environment, i.e., their Dynamicity=0.

(B) **Cost-free**, designates compositions that consist of resources used without any charge, i.e., their Cost=0.

Table 5.5 – Examples of 4 hybrid compositions having acceptable score $\geqslant 50$, including one (the latest) consisting of static resources

f_1	f_2	f_3	f_4	f_5	Score(RC)
$res^d_{1,1}$	$res^s_{2,1}$	$res^d_{3,1}$	$res^s_{4,1}$	$res^d_{5,2}$	55
$res^s_{1,1}$	$res^s_{2,2}$	$res^d_{3,2}$	$res^d_{4,2}$	$res^s_{5,1}$	32
$res^s_{1,2}$	$res^s_{2,2}$	$res^d_{3,1}$	$res^d_{4,2}$	$res^s_{5,2}$	51
$res^d_{1,2}$	$res^s_{2,2}$	$res^s_{3,1}$	$res^s_{4,1}$	$res^s_{5,1}$	10
-	-	-	-	-	-
$res^s_{1,2}$	$res^s_{2,1}$	$res^s_{3,1}$	$res^s_{4,1}$	$res^s_{5,1}$	54

(C) **Efficient**, denotes compositions that include resources with high normalized usage value, i.e., their Usage $\geqslant 0.75$.

(D) **Effective**, refers to compositions containing resources with high normalized availability value, i.e., their Availability $\geqslant 0.75$.

(E) **Qualified**, denotes compositions in which the score of each involved resource providing the needed function, score(res_f) $\geqslant$ [(n $\times$ 0.75) + (m $\times$ 0.25)], where n is the number of QoR maximization attributes relative to each resource and its provided needed functions (with the exception of the "Dynamicity" attribute), and m is the number of QoR minimization attributes (e.g., Cost).

(F) **Reliable**, designates compositions whose Score(Rel) $\geqslant$ (l $\times$ 0.75), with l refers to the number of dependencies links between the required functions defined in WM.

Table 5.6 – QoR values of optimistic and hybrid compositions subtypes

	Dynamicity	**Availability**	**Cost**	**Usage**
Trusted	0	$\geqslant 0.5$	$\leqslant 0.25$	$\geqslant 0.5$
Cost-free	0 | 1	$\geqslant 0.5$	0	$\geqslant 0.5$
Efficient	0 | 1	$\geqslant 0.5$	$\leqslant 0.25$	$\geqslant 0.75$
Effective	0 | 1	$\geqslant 0.75$	$\leqslant 0.25$	$\geqslant 0.5$
Qualified	0 | 1	$\geqslant 0.75$	$\leqslant 0.25$	$\geqslant 0.75$
Reliable	0 | 1	$\geqslant 0.5$	$\leqslant 0.25$	$\geqslant 0.5$

The composition subtypes are defined according to either a specific QoR attribute value, or a set of QoR attributes values, or I/O similarity scores. However, and in addition to these constraints, both optimistic and hybrid composition types should respect other QoR attributes constraints, as defined in Table 5.6. This is done to ensure having compositions with an acceptable score(RC), and thus, good compositions results. For (A)-(E) compositions subtypes preceded by optimistic or hybrid types, Score(Rel) $\geqslant$ (l $\times$ 0.5), with l referring to the number of dependencies links between the required functions defined in WM. As for the optimal compositions, they have the maximum values of score(RC).

Based on user's request type, i-compositions are formed to answer his request. The i value ($\in \mathbb{N}^*$) can be determined by the user in r, for the 2 main types: optimal and optimistic. As for hybrid compositions, i depends from the resources dynamic aspect. As such, when optimal compositions are required, SSA computes all possible compositions scores and retrieves

the i-compositions having the best scores. When optimistic compositions are needed, SSA stops forming compositions until having i-compositions with acceptable scores. If hybrid compositions are required, SSA generates the solutions having acceptable scores, and stops until having a composition containing only static resources, guaranteeing thus the existence of a composition at any instance. Moreover, and in case optimal compositions subtypes are needed (e.g., optimal trusted, optimal cost-free, and optimal qualified), a filtering process is necessary before score(RC) calculations. Such filtering is based on specific constraints defined for the different composition subtypes:

- **Optimal Trusted**: Only static resources are used.
- **Optimal Cost-free**: Resources (static or dynamic) having Cost = 0 are used.
- **Optimal Efficient**: Resources (static or dynamic) having Max(Usage) are used, with Max(Usage) refers to the maximum value of the Usage attribute among the resources in DRAG.
- **Optimal Effective**: Resources (static or dynamic) having Max(Availability) are used, with Max(Availability) refers to the maximum value of the Availability attribute among the resources in DRAG.
- **Optimal Qualified**: Compositions having max(Score(RES)) are returned.
- **Optimal Reliable**: Compositions having max(Score(Rel)) are retrieved.

If the user defines in his request, r, constraints that do not align with the designated composition subtype constraints, the latter are considered.
When optimistic or hybrid compositions are required, the SSA applies several steps:

1. **Compute the minimum score of an acceptable composition**. A composition is considered acceptable, if its score(RC) is $\geqslant$ a specific Threshold, T. Based on user's request type (i.e., optimistic, optimistic effective, hybrid, hybrid cost-free, etc.), SSA computes the necessary T. When optimistic or hybrid composition types are requested (without subtypes), T is defined as:
   T = $[(n \times Avg(Q_c)) + (l \times \mathbf{0.5})] \times (d/\mathbf{100})$, where:
   - **n** is the total number of functions in WM.
   - **Avg(Q_c)** are the average of the normalized user QoR constraints defined in r for each resource (excepting the "Dynamicity" attribute). If Q_c are not given, the average of each QoR is calculated according to their maximum values among DRAG resources.
   - **l**, is the number of the dependencies links between WM functions. We consider that there is, at least, an I/O similarity match (=0.5) between any two linked resources.
   - **d**, is the composition acceptance degree value (in %) given by the user in r.

If optimistic or hybrid composition types are succeeded by a subtype (e.g., trusted, cost-free, efficient, etc.) in user's request type, T is represented as: $T_{subtype} = [(n \times Q) + (l \times s)] \times (d/100)$, with Q denoting the minimum values of the attributes as defined in Table 5.6 (excepting the "Dynamicity" attribute), and s $\in$ [0,1] refers to the minimum I/O similarity matching score between any two linked resources. s = 0.75 whenever subtype = reliable, and s = 0.5 for the rest of subtype values.

2. **Compute the score of each composition formed by eligible resources**. To do so, a generator is used to get the possible compositions without score calculation. Table 5.7 shows some examples of generated RC where each corresponds to a set of resources achieving the required workflow. During RC generation, the following conditions are performed:

Table 5.7 – Examples of generated compositions achieving, each, the required workflow without score calculation,

f_1	f_2	f_3	f_4	f_5
$res^d_{1,1}$	$res^s_{2,1}$	$res^d_{3,1}$	$res^s_{4,1}$	$res^d_{5,2}$
$res^s_{1,1}$	$res^s_{2,1}$	$res^d_{3,1}$	$res^d_{4,2}$	$res^s_{5,2}$
-	-	-	-	-
$res^s_{1,2}$	$res^s_{2,2}$	$res^s_{3,1}$	$res^s_{4,1}$	$res^s_{5,2}$

(i) If a resource in RC is not eligible, it is registered in array, **arr_notEl**, and another possible RC is generated

(ii) If all the resources of RC are eligible, score(RC) is computed. If score(RC) $\geqslant$ **T**, RC is saved into the suitable compositions array, **arr_suitRC**, if not, another possible RC is generated

Table 5.8 – Examples of returned i-compositions with score calculation

f_1	f_2	f_3	f_4	f_5	Score(RC)
$res^d_{1,1}$	$res^s_{2,1}$	$res^d_{3,1}$	$res^s_{4,1}$	$res^d_{5,2}$	$\geqslant$ T
$res^s_{1,1}$	$res^s_{2,1}$	$res^d_{3,1}$	$res^d_{4,2}$	$res^s_{5,2}$	$\geqslant$ T
-	-	-	-	-	$\geqslant$
$res^s_{1,2}$	$res^s_{2,2}$	$res^s_{3,1}$	$res^s_{4,1}$	$res^s_{5,2}$	$\geqslant$ T

While analyzing each RC, if a resource is in **arr_notEl**, another possible RC is generated. If not, conditions (i) and (ii) are tested. When optimistic compositions are required, the generator stops when having i-compositions respecting T. However, when hybrid solutions are needed, the generator stops when having a composition that respects T, and contains only static resources, i.e., always available in the environment.

SSA results are the set of RC included in **arr_suitRC**. An example of returned suitable compositions with score calculation (where type = hybrid) is shown in Table 5.8.

Figure 5.4 shows the flowchart of the selection process and its related SSA, to form i-compositions based on the given user's request and request type.

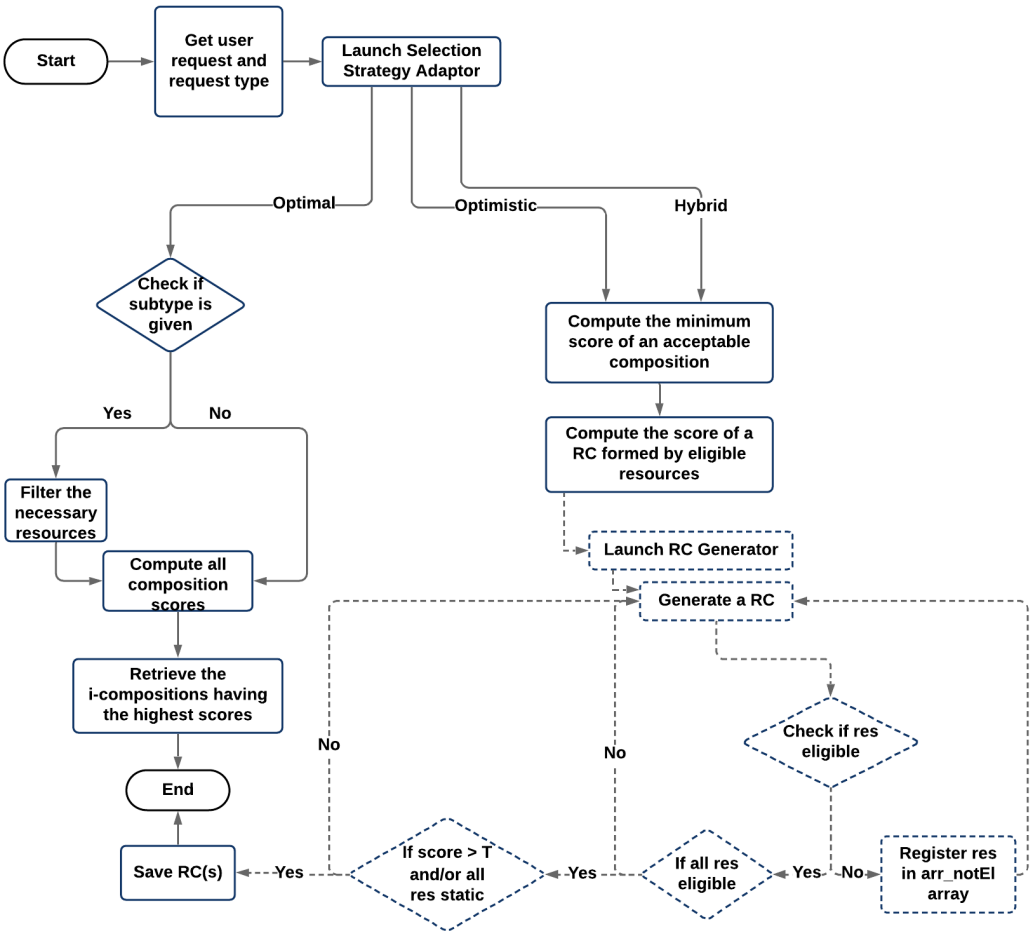


Figure 5.4 – Flowchart of the selection process and its related Selection Strategy Adaptor

5.5 Evaluation and Discussion

In this section, we first evaluate the performance of our resource selection approach in different environment setups (e.g., varying the number of candidate resources per function, and varying the number of required functions in the workflow). Then, we compare our QoR model to existing works.

5.5.1 Resource Selection Performance Evaluation

In the experiments conducted for the resource selection performance evaluation, the function and resource graphs[2] are dynamically generated based on simulations. The tests were applied on a Linux Debian (64 bits) virtual machine, with 1 dedicated Intel® Core™ i7-46000 CPU @ 2.10GHz 2.70 GHz processor and 1 GB RAM. In the results, we show the response time (in milliseconds) of the resource selection process (without including resource discovery time), based on an average of 5 sequential executions.

During the tests, we evaluated our selection approach performance by considering that the requested composition type = hybrid, to focus on resources dynamcity aspect while forming the compositions. The evaluation consists on mainly two cases: (1) varying the number of resources (dynamic and static) per function in DRAG, and (2) varying the number of functions required in WM. For each of the two cases, we applied several scenarios:

(i) All static resources in DRAG are eligible

(ii) 50% of the static resources in DRAG are eligible

(iii) All DRAG resources are dynamic and eligible

For (i) and (ii), the first generated possible compositions (without score calculation) include only dynamic resources, thus, the selection process will continue generating compositions until having one with score(RC) $\geqslant$ T (with d = 100%), and consisting of only static resources. In the best case scenarios, static resources are traversed first, and the selection process responds more rapidly. This is shown in Table 5.9 results. For (iii), and since DRAG consists of eligible dynamic resources, the score of each possible composition is calculated, and all compositions are returned. This case is almost similar to the case where type = optimal, in which all possible compositions are computed, however, i-compositions with the i-top scores are retrieved. In the tests, FG consists of 50 functions, each resource has 2 inputs and 2 outputs, and user QoR constraints are given to the Dynamicity, Availability, Cost, and Usage attributes. Moreover, in the experiments, static and dynamic resources can be part of the compositions (i.e., q_1^{res} related to the Dynamicity attribute constraint = [1-1] in Q_c^{res}), and the related resources can be linked (we assumed that the I/O similarity score between a resource and another related one is $\geqslant$ 1).

[2]Resource graph is used only for resource discovery

Table 5.9 – Response time (in ms) of SP while varying resource number per function (m), and number of required functions (n)

	n = 2	**n = 3**	**n = 4**	**n = 5**
m = 5	61.7	106	152	214
m = 20	105	172	261	332

In Figure 5.5, where the workflow consists of 3 functions, response time evolves with the growing number of resources. Comparing Figure 5.5-(a) to Figure 5.5-(b), response time increments less significantly with the existence of static eligible resources, as the selection process stops when having a composition of static resources. Figure 5.5-(b) represents the worst case scenario (when type = hybrid), where DRAG contains only dynamic resources that are eligible. Thus, all compositions scores are computed, leading to an important response time with the evolution of resources number.

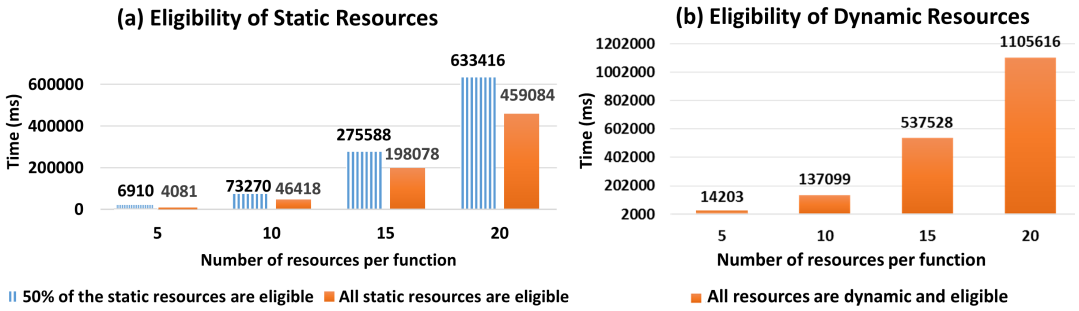


Figure 5.5 – Selection results while varying the number of resources

In the tests results of Figure 5.6-(a), in which the number of resources per function is fixed to 5, the response time increases with the number of required functions in WM. Similar to the first case (see Figure 5.5), the selection process is faster when there are static eligible resources, as the selection process stops before the generation of the rest of the possible compositions. As for Figure 5.6-(b), response time evolves significantly, since DRAG contains dynamic eligible resources, thus, all compositions scores are calculated.

The results highlight the importance of the existence of static eligible resources in DRAG (when type = hybrid), as the selection process stops when having a composition consisting of static resources with an acceptable score. When dynamic resources exist, the response time of the selection process increases, since their dynamic aspect is taken into consideration, as they might be unavailable for execution. The results also show that the growing number of resources affects more the response time, comparing to the evolution of the number of functions.

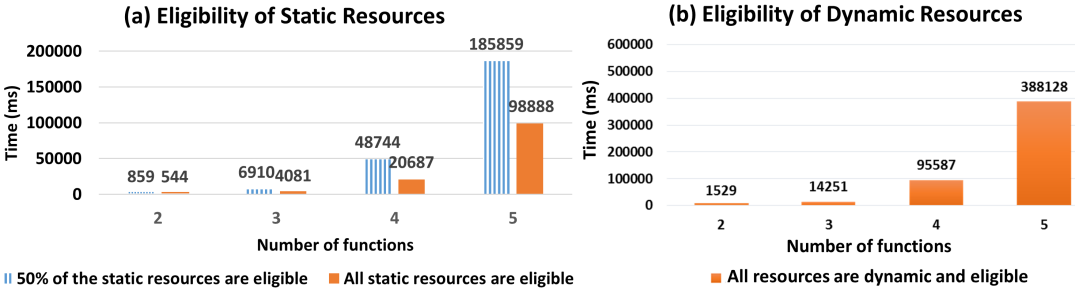


Figure 5.6 – Selection results while varying the number of functions

5.5.2 Comparison with Existing QoS Models

In this section, we analyze QoS model of existing works [22, 25, 114, 120], and compare them to our QoR model. This is done independently from the number and type of the quality attributes used, since our solution can support various attributes as long as they are presented in resource descriptions.

The work in [22] uses 3 attributes: Performance: [0-10], Availability: [0-100], and Reputation: [0-5]. Based on user constraints, a service s1 is chosen over a candidate service s2 if all of s1's QoS are equal or better than s2's QoS, preventing thus having compositions with acceptable overall QoS in satisfactory delays. Also, and apart from neglecting I/O matching of the linked services, by applying our threshold formula(s), a service with a high attribute value (Availability = 90) and a very low value for another one (Performance = 2) will be selected over a service with acceptable values for both attributes (Availability = 70 and Performance = 6), since QoS are not normalized.

In [25], several QoS are used to describe services as Response Time and Availability. Contrary to our work, user constraints are given to the global composition (e.g., the overall response time should be < 50s) and not to each service, thus, aggregation functions are used for every QoS parameter. Moreover, weights are given to each QoS while computing the composition score. However, in our approach, user constraints are given to each of the involved services, and weights are assigned to (i) the global services score, Score(RES), and (ii) the overall I/O services matching score, Score(Rel), which is not considered in [25].

In [114], a service has a score based on the sum of weighted utility functions relative to each QoS attribute. Similar to our work, QoS are normalized and user constraints are given to every service in a composition. However, the work does not define a global composition score, as the service of a specific task (i.e. function) with the highest score is selected. Also, the work does not consider I/O matching of the related services.

The work in [120] defines an overall composition formula that is based on a function, f_i, related to the composite value of an attribute i. The quality attributes are normalized and each f_i of an attribute can be multiplied by a weight. The goal of the approach is to obtain a composition solution that maximizes the overall formula. However, the work does not consider I/O matching score.

In Table 5.10, we show the summary evaluation of existing works according to the following service/composition quality related criteria:

- **QoS Normalization,** denotes if the considered QoS attributes during selection are normalized.
- **Overall Composition Score,** denotes whether an overall score is computed and assigned to each possible composition.
- **Service Score,** denotes whether a score is computed and assigned to each service.
- **I/O Matching,** denotes if the I/O matching of the related services in a composition is taken into account.

- **Weights,** denotes if weights are assigned to each QoS attribute during composition/service score calculation.

In the comparison, we used "+" symbol to express a positive coverage for a criterion, and "-" symbol to express a lack of a criterion coverage.

Table 5.10 – Summary evaluation of existing works w.r.t. the service/ composition quality related criteria

Approaches	QoS Normalization	Overall Composition Score	Service Score	I/O Matching	Weights
[22]	-	-	-	-	-
[25]	+	+	-	-	+
[114]	+	-	+	-	+
[120]	+	+	-	-	+

5.6 Summary

In this chapter, we presented a QoR-driven resource selection approach related to the selection process of the automatic resource composition in the StARC framework. The proposed solution can be applicable in hybrid Web environments providing static resources (established to be always available), and dynamic resources (connected/disconnected at different instances). It allows to form several compositions alternatives based on different requested compositions types (i.e., Optimal, Optimistic, and Hybrid), and subtypes (e.g., Trusted, Cost-free, and Qualified) given by the users to answer their different needs. For this aim, we provided a formal graph representation of the resources identified in the resource discovery process (previously presented in Chapter 4), and defined a score for each resource (i.e., a graph node), and each possible composition (i.e., a graph path). Using the formal graph, we proposed a Selection Strategy Adaptor that allows forming several composition alternatives (i-compositions with $\in \mathbb{N}^*$) answering user's requests. This is done while considering user QoR constraints (i.e., Dynamicity, Cost, Availability, and Usage), resource Inputs/Outputs matching and dynamicity (whenever it is required). Several tests have been conducted to study the performance of our proposed solution in different environment setups (varying the number of candidate resources providing a similar required function, and the number of functions in the workflow needed to realize use demand). Analysis have been also made to compare our QoR model with existing works.

Chapter 6

Conclusion

"The best way to predict the future is to create it"

Abraham Lincoln

6.1 Recap

In this thesis, we presented a framework for static and automatic RESTful service (resource) composition. In the framework, entitled StARC[1], we focused on the: (1) behavior verification of static resource compositions (built manually by the user), (2) automatic resource discovery, and (3) automatic resource selection. In our work, the verification approach is adopted in Web environments providing static resources (established to be always available), whereas both automatic discovery and selection approaches can be applicable in hybrid Web environments connecting also dynamic resources (i.e., connected to/removed from the Web at different instances). The proposed solutions are generic and can be applicable in different Web environments domains. However, in this thesis, we motivated our work using scenarios illustrated in the smart buildings domain to help building actors in managing their buildings energy behavior. This is done by allowing them to create new composed resources that can answer complex needs requiring the combination of several resources together.

In **Chapter 1**, we started by defining the context of the thesis, by giving first an insight on the World Wide Web, i.e., its main developments and technologies: (i) the Web services concept (including resources following the REST principles), (ii) the Web of Things (where objects are exposed as resources), and (iii) the Service-Oriented Architecture (SOA) with a real-world SOA-based architecture example related to two projects: HIT2GAP and SIBEX. Then, we presented the scope of the thesis, including the collaborators and the main objectives. A motivating scenario illustrated in the HIT2GAP/SIBEX service-oriented platform (the BEMServer), was presented, through which the thesis addressed 3 main research challenges: (1) the verification of a static resource composition behavior before execution, (2) the automatic resource

[1] It stands for **St**atic and **A**utomatic **R**esource **C**omposition

discovery in hybrid Web environments connecting static and dynamic resources, (3) the automatic resource selection to form the suitable compositions in hybrid Web environments. The contributions of the thesis were presented in a generic framework, entitled StARC, and detailed in the remaining chapters.

In **Chapter 2**, we presented some background information for the full understanding of the context of our work. We first gave preliminaries about Web services, the main protocol/principles supported during their implementation, and how they can be semantically be described so their properties (provided functions, Inputs/Outputs, etc.) can be understandable to machines. Then, we presented a review on the existing languages used to describe RESTful services (the type of services supported in this thesis) also known as resources, with a focus on hypermedia-driven languages as Hydra vocabulary (the one adopted in our work to describe the resources).

In **Chapter 3**, we proposed a solution for the verification of static resource composition (built manually by the user) before being executed. For this aim, we defined a formal model used to map the behavior of resources with their composition to Colored Petri Nets (CPN) (i.e., a graphical oriented language for design, specification, simulation and verification of systems). Based on the defined CPN-based model, we were able to use CPN behavioral properties (i.e., Interoperability to check if the linked resources are compatible according the their related Input/Output data types, Reachability to ensure that the final desired state is reachable, and Liveness to ensure that all resources can be executed during composition execution), to verify resource compositions behavior before their execution. The approach has been experimented in CPN tools to verify a composed resource illustrated in the building energy management domain, proving the applicability of our proposed CPN-based model to verify resource compositions. In the chapter, we also presented a prototype that has been developed in the context of the SIBEX project to verify building-oriented resource compositions before execution. The prototype allows modeling, validating, converting, and executing verified composed resources, through different implemented engines. Several tests have been conducted to validate the correct execution of the different developed engines including the validation engine that, based on our defined CPN-based model, allows verifying resource compositions.

However, currently the developed prototype is a standalone module that runs independently of the BEMServer platform, as it allows composing resources using resources hosted outside the platform. This is due mainly to the lack of support of the necessary REST principles while implementing the resources provided by the BEMServer during the development phase. In the future, we seek to integrate the prototype within the BEMServer, so it enables the creation of new composed resources using the resources embedded within the platform, including the advanced services.

In **Chapter 4**, we presented an automatic location-aware resource discovery for hybrid Web environments connecting: (1) static linked resources (established to be always available) following the HATEAOS principle, and (2)

dynamic resources that can be connected to and removed form the environment at different instances. For this aim, we proposed a formal model representation that links resources (i.e., dynamic and static) in one single resource graph. This is done by defining virtual resources that can be connected to static resources, and hold dynamic ones. The resource graph can be traversed by several adapted graph-based algorithms (i.e., BFS and DFS in our work) to discover k-resources ($k \in \mathbb{N}$) for each required function for user's request. This is done using semantic annotations integrated in the resource descriptions (expressed with Hydra vocabulary in this thesis). In the chapter, we also defined an original 3-dimensional indexing schema that maps the resources to their provided functions and location (whenever they are exposed by objects). The indexing schema is used to identify data collection resources based on their location, and enhance resource search in large Web environments. Several tests were conducted to evaluate the solution performance in different Web environment setups (e.g., varying the number of resources, varying the number of required functions for user's request, etc.), and on 4 aspects: dynamicity, multiplicity, efficiency, and scalability. The results highlight the importance of using the indexing schema, especially in large Web environments, to enhance resource discovery response time.

Nevertheless, currently the proposed resource discovery approach is not integrated within the BEMServer. In fact, it has been tested in simulated Web environments based on dynamically generated function and resource graphs, having different configurations in terms of: number of resources, number of required functions, and the number of resources providing similar functions. Moreover, in the experiments, we have considered that each resource provides one single function. Therefore, in the future, we plan to increase the number of functions provided by the resources, and study the resource discovery performance while considering several parameters simultaneously, rather than focusing on one parameter (e.g., number of resources) for each aspect (i.e., dynamicity, multiplicity, efficiency, and scalability). Also, in the current approach, the algorithm type (i.e., BFS or DFS in this work) to be used by the resource discovery process, is specified manually. The results available in Appendix J show that the performance of each of these algorithms depends on the functions' distribution and the localization of the requested function in the function graph. Thus, we intend to propose dynamically the most suitable algorithm according to the current function graph topology. As for the indexing schema, the experiments presented in appendix L show that the response time and memory usage increase with the evolution of both functions and resources numbers. Such evolution is important when the number of functions and resources is high. Therefore, updating the indexing schema dynamically without regenerating it again is an improvement that we seek to do in the future.

In **Chapter 5**, we have proposed an automatic resource selection approach that can be applicable in hybrid Web environments connecting static and dynamic resources. In the approach, we first defined a formal model that links the identified resources during the resource discovery process in a directed acyclic graph, based on their providing functions. Then, we presented a selection strategy adaptor that allows selecting the appropriate resources to

form several alternative compositions with different types answering user different needs (e.g., optimal compositions having the highest scores, optimistic compositions having acceptable scores but obtained in more satisfactory delays, etc.). The selection process takes into consideration user QoR constraints, resource I/O matching of related resources, and resource dynamicity. Several tests have been conducted to study and evaluate the performance of our proposed automatic resource solution in different environment setups (e.g., varying the number of resource candidates and the number of required functions answering user's request). Also, analysis were made to compare our QoR model with existing works.

Nevertheless, and similar to the automatic resource discovery approach, resource selection is not integrated within the BEMServer platform. As such, it has been tested in simulated Web environments based on dynamically generated function and resource graphs. Furthermore, currently, we have evaluated our selection approach performance according to one requested composition type (i.e., Hybrid), without considering any composition subtypes (e.g., trusted, cost-free, qualified, etc.). This will be handled in the future. Also, our proposed resource selection consists, in its current version, on verifying resource eligibility (if they respect user QoR constraints) before they can be allowed to be used to form the needed compositions. Such verification is impacting negatively the response time of the selection process. Thus, we seek to integrate the resource eligibility verification during resource discovery. As such, no resource that is not eligible can pass to the selection process. In addition, in the current selection approach, we consider the similarity measure between the I/O of the linked resources, without taking into account their matching on the semantic level. The latter is an important criteria to consider in the future, to ensure an efficient resource compositions involving semantically linked resources.

6.2 Future Works

There are several steps that we can apply in the future to further (i) validate, (ii) improve, and (iii) extend the proposed contributions of this thesis. These steps are discussed next on the basis of the several identified limitations.

6.2.1 Integrate the Static/Automatic Resource Composition in Real-world Environments

In Chapter 3, we presented our approach for the verification of static resource compositions behavior. Although we have developed a prototype implementing the Colored Petri Net-based model that we defined, the prototype is a standalone module that uses data collection and pre-processing resources hosted outside of the BEMServer Web platform (see Section 1.1.1.3 in Chapter 1). In the future, we seek to integrate the prototype as a service in the Management level of the BEMServer, so it enables the creation of new composed resources using the resources provided by the platform, including the advanced services. In order to allow such integration, the resources of the BEMServer should: (i) follow the REST principles (i.e., they are identified by a Web address, a URI, and callable through HTTP methods (e.g., GET, POST,

PUT, and DELETE), and (ii) be described using the Hydra Vocabulary (e.g., provided functions, inputs/outputs, etc.).

As for the automatic resource composition aspect, we plan to test the proposed automatic resource discovery and selection solutions in real Web environments providing: static and dynamic resources. To allow such integration, it is essential that the provided Web resources support HATEOAS, which consists on linking resources together based on their providing functions forming one resource graph.

6.2.2 Extend the Automatic Resource Discovery

In Chapter 4, we presented our automatic static and dynamic resource discovery solution, while considering resource location (whenever it is exposed by an object). We aim to extend the resource discovery approach, by integrating/adapting other graph algorithms (besides BFS and DFS that were considered in this work), as A star (A^*) and Best First Search [97], to explore the resource graph and identify the required resources in a better response time. Moreover, we plan to consider the semantic matching between the required functions necessary for user request and the functions of the traversed resources in the resource graph, rather than testing their exact match, as it is the case in the current approach. Also, in our proposed resource discovery solution, the new functions provided by dynamic resources are randomly linked to the existing function graph. In the future, we seek to study the measures that define the dependencies between the new and the existing functions.

6.2.3 Improve the Automatic Resource Selection Performance

In Chapter 5, we provided a solution for selecting automatically the appropriate static/dynamic resources to form suitable and different compositions alternatives answering user's request. This is done by taking into account user QoR constraints, resources I/O matching and resource dynamicity. The proposed resource selection is executed after the resource discovery process. In order to improve the performance of the selection process, we seek to launch the selection process in parallel with resource discovery when two compositions types are requested: Optimistic (i.e., compositions having acceptable scores) and Hybrid (i.e., compositions having acceptable scores and including one consisting only on static resources). This can improve the selection performance in terms of response time.

6.2.4 Propose an Automatic Resource Orchestration Approach

For the automatic resource composition aspect, we have proposed solutions for both automatic resource discovery and selection, without considering resource composition execution held by the Execution Process of the StARC framework. Therefore, in the future, we intend to ensure automatically the correct interaction between the selected resources forming the suitable composition(s) for their execution. In this context, resource orchestration process takes part [88], where the involved resources are controlled by a single endpoint central process (another resource). To ensure an automatic resource orchestration, a semantic approach that links correctly the resource outputs

to the next related resource inputs is important. Such approach should also guarantee the correct linking between a resource and parallel ones, and vice versa.

Appendix A

WSDL 2.0 Example

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns            = "http://www.w3.org/ns/wsdl"
    targetNamespace  = "http://jenkov.com/MyService"
    xmlns:tns        = "http://jenkov.com/MyService"
    xmlns:stns       = "http://jenkov.com/MyService/schema"
    xmlns:wsoap      = "http://www.w3.org/ns/wsdl/soap"
    xmlns:soap       = "http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsdlx      = "http://www.w3.org/ns/wsdl-extensions" >
    <documentation>
      This is the web service documentation.
    </documentation>
    <types>
      <xs:schema
        xmlns:xs           = "http://www.w3.org/2001/XMLSchema"
        targetNamespace    = "http://jenkov.com/MyService/schema"
        xmlns              = "http://jenkov.com/MyService/schema">
      <xs:element      name = "latestTutorialRequest"
                       type = "typeLatestTutorialRequest"/>
      <xs:complexType name = "typeLatestTutorialRequest">
        <xs:sequence>
          <xs:element name = "date" type="xs:date"/>
        </xs:sequence>
      </xs:complexType>
        <xs:element name = "latestTutorialResponse" type = "xs:string"/
            >
        <xs:element name = "invalidDateError" type = "xs:string"/>
      </xs:schema>
    </types>
    <interface    name = "latestTutorialInterface">
      <fault      name = "invalidDateFault"  element = "
          stns:invalidDateError"/>
      <operation name = "latestTutorialOperation"
            pattern = "http://www.w3.org/ns/wsdl/in-out"
            style = "http://www.w3.org/ns/wsdl/style/iri"
            wsdlx:safe = "true">
      <input messageLabel = "In" element = "stns:latestTutorialRequest"
           />
      <output messageLabel = "Out" element = "
          stns:latestTutorialResponse" />
      <outfault messageLabel = "Out" ref ="tns:invalidDateFault" />
      </operation>
    </interface>
    <binding name = "latestTutorialSOAPBinding"
       interface = "tns:latestTutorialInterface"
       type = "http://www.w3.org/ns/wsdl/soap"
       wsoap:protocol = "http://www.w3.org/2003/05/soap/bindings/HTTP/"
           >
       <fault ref="tns:invalidDateFault" wsoap:code="soap:Sender"/>
```

```
45          <operation ref = "tns:latestTutorialOperation"
46          wsoap:mep = "http://www.w3.org/2003/05/soap/mep/soap-response"/>
47      </binding>
48      <service
49          name      = "latestTutorialService"
50          interface = "tns:latestTutorialInterface">
51       <endpoint name = "latestTutorialEndpoint"
52                 binding = "tns:latestTutorialSOAPBinding"
53                 address = "http://jenkov.com/latestTutorial"/>
54      </service>
55 </description>
```

Appendix B

WADL Example

```
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
  xmlns:tns="urn:yahoo:yn" xmlns:yn="urn:yahoo:yn" xmlns:ya="
      urn:yahoo:api"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://wadl.dev.java.net/2009/02">
   <grammars>
     <include href="NewsSearchResponse.xsd"/>
     <include href="Error.xsd"/>
   </grammars>

   <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
     <resource path="newsSearch">
       <method name="GET" id="search">
         <request>
           <param name="appid" type="xsd:string" style="query" required
               ="true"/>
           <param name="query" type="xsd:string" style="query" required
               ="true"/>
           <param name="type" style="query" default="all">
             <option value="all"/>
             <option value="any"/>
             <option value="phrase"/>
           </param>
           <param name="results" style="query" type="xsd:int" default="
               10"/>
           <param name="start" style="query" type="xsd:int" default="1"
               />
           <param name="sort" style="query" default="rank">
             <option value="rank"/>
             <option value="date"/>
           </param>
           <param name="language" style="query" type="xsd:string"/>
         </request>
         <response status="200">
           <representation mediaType="application/xml" element="
               yn:ResultSet"/>
         </response>
         <response status="400">
           <representation mediaType="application/xml" element="
               ya:Error"/>
         </response>
       </method>
     </resource>
   </resources>
 </application>
```

Appendix C

HAL Example

```
{
    "_links": {
        "self": {
            "href": "http://www.h2g.eu/h2g/resource/getairtemperature"
        }
    },
    "_embedded": {
        "next": {
            "_links": {
                "self": {
                    "href": "http://www.h2g.eu/h2g/resource/predheatengcons"
                }
            }
        }
    },
    "Operation": [{
        "method": "GET",
        "expects": ["startdate", "enddate"],
        "returns": ["DateTime", "Float"],
        "function": "ATC"
    }]
}
```

Appendix D

SIREN Example

```
{
    "class": ["datacollection"],
    "properties": {
       "data": "airtemperature"
    },
    "entities": [],
    "actions": [
       {
          "name": "getairtemperature",
          "title": "Get Air Temperature",
          "method": "GET",
          "href": "http://www.h2g.eu/h2g/resource/getairtemperature",
          "type": "application/x-www-form-urlencoded",
          "fields": [{
                "name": "startdate",
                "type": "datetime"
             },
             {
                "name": "enddate",
                "type": "datetime"
             }
          ]
       }
    ],
    "links": [{
          "rel": ["self"],
          "href": "http://www.h2g.eu/h2g/resource/getairtemperature"
       },
       {
          "rel": ["next"],
          "href": "http://www.h2g.eu/h2g/resource/predheatengcons"
       }
    ]
}
```

Appendix E

MASON Example

```
{
   "ID": 1,
   "Title": "Get Air Temperature",
   "Description": "A resource that collects the air temperature",
   "@meta": {},
   "@controls": {
      "self": {
         "href": "http://www.h2g.eu/h2g/resource/getairtemperature"
      },
      "is:next": {
         "title": "Energy Heat Prediction",
         "description": "Predicts building energy heat",
         "href": "http://www.h2g.eu/h2g/resource/predheatengcons"
      }
   }
}}
```

Appendix F

Resource Composition Modeling

```
{
   "composition": {
      "name": "Conversion and alignment of collected data",
      "description": "This composition collects data from bldg A,
          convert them to the desired unit, and align them on a same
          frequency (expressed in secs)",
      "variables": ["startDate", "endDate"],
      "resources": [{
            "url": "http://sibex/measures",
            "method": "GET",
            "name": "measures",
            "param": [{
                  "type": "internalTemp"
               },
               {
                  "zoneid": "BldgA"
               },
               {
                  "startdate": "startDate"
               },
               {
                  "enddate": "endDate"
               }
            ],
            "output": "A"
         },
         {
            "url": "http://sibex/measures",
            "method": "GET",
            "name": "measures",
            "param": [{
                  "type": "powerEnergy"
               },
               {
                  "zoneid": "BldgA"
               },
               {
                  "startdate": "startDate"
               },
               {
                  "enddate": "endDate"
               }
            ],
            "output": "B"
         },
         {
            "url": " http://sibex/measures ",
            "method": "GET",
            "name": "measures",
```

```
48              "param": [{
49                    "type": "lightRadiation"
50                 },
51                 {
52                    "zoneid": "BldgA"
53                 },
54                 {
55                    "startdate": "startDate"
56                 },
57                 {
58                    "enddate": "endDate"
59                 }
60              ],
61              "output": "C"
62           },
63           {
64              "url": "http://sibex/dataconversion",
65              "method": "GET",
66              "name": "Data conversion",
67              "param": [{
68                 "data": "A",
69                 "unit": "celsius"
70              }],
71              "output": "D"
72           },
73           {
74              "url": "http://sibex/dataconversion",
75              "method": "GET",
76              "name": "Data conversion",
77              "param": [{
78                 "data": "B",
79                 "unit": "KWH"
80              }],
81              "output": "E"
82           },
83           {
84              "url": "http://sibex/dataconversion",
85              "method": "GET",
86              "name": "Data conversion",
87              "param": [{
88                 "data": "C",
89                 "unit": "wattPerSqMeter"
90              }],
91              "output": "F"
92           },
93           {
94              "url": "http://sibex/datalignment",
95              "method": "GET",
96              "name": "dataAlignment ",
97              "param": [{
98                    "data": "D"
99                 },
100                {
101                   "frequency": "900"
102                }
103             ],
104             "output": "G"
105          },
106          {
107             "url": "http://sibex/datalignment",
108             "method": "GET",
109             "name": "dataAlignment ",
110             "param": [{
```

```
                    "data": "E"
                },
                {
                    "frequency": "900"
                }
            ],
            "output": "H"
        },
        {
            "url": "http://sibex/datalignment",
            "method": "GET",
            "name": "dataAlignment ",
            "param": [{
                    "data": "F"
                },
                {
                    "frequency": "900"
                }
            ],
            "output": "I"
        },
        {
            "url": " http://sibex/merge3data",
            "method": "GET",
            "name": "Merge 3 data",
            "param": [{
                    "data": "G"
                },
                {
                    "data": "H"
                },
                {
                    "data": "I"
                }
            ],
            "output": "J"

        }
    ],
    "goal": "J"
  }
}
```

Appendix G

Hydra-based Composed Resource Description

```
{
        "@context": {
        "@vocab": "http://www.w3.org/ns/hydra/core#",
        "schema": "http://schema.org/",
        "ifcTC1": "http://www.buildingsmart-tech.org/ifcOWL/IFC2X3_TC1/",
        "ifcFinal": "http://www.buildingsmart-tech.org/ifcOWL/
            IFC2X3_Final/",
        "zoneid": "ifcTC1:globalId_IfcRoot",
        "sbxBI": "http://sibex/sbxBI/",
        "sbxOcc": "http://sibex/sbxOccupant/",
        "sbxProp": "http://sibex/sbxProperty/",
        "ssn": "https://www.w3.org/TR/vocab-ssn/",
        "qudt": "http://quadt.org/schema/qudt/",
        "type": "ssn:SOSAResult",
        "timestamp": "schema:DateTime",
        "startdate": "schema:DateTime",
        "enddate": "schema:DateTime",
        "updatedDate": "schema:DateTime",
        "frequency": "sbxProp:Frequency",
        "unit": "qudt:Unit",
        "quality": "schema:Float",
        "Workflow": "Collection",
        "resources": "Collection",
        "data": "schema:Text",
        "goal": "schema:Text",
        "id": "schema:identifier",
        "url": "schema:url",
        "acronym": "schema:Text",
        "value":"schema:Float",
        "tabCollVal": {
                "@id": "tabCollVal",
                        "@container" : "@set",
                        "@values" : [
                {"@type" : "quality"},
                                {"@type" : "timestamp"},
                {"@type" :  "updatedDate"},
                {"@type" :  "value"}]
                },
        "tabValues":    {
          "@id": "tabValues",
                    "@container" : "@set",
                    "@values" : [
              {"@type" :  "timestamp"},
              {"@type" :  "value"} ]
                },
        "tabValues2":    {
```

```
46          "@id": "tabValues2",
47                      "@container" : "@set",
48                      "@values" : [
49              {"@type" :  "timestamp"},
50              {"@type" :  "value"} ]
51               },
52          "tabValues3":   {
53            "@id": "tabValues3",
54                      "@container" : "@set",
55                      "@values" : [
56              {"@type" :  "timestamp"},
57              {"@type" :  "value"} ]
58               },
59          "tabTimestamp":   {
60            "@id": "tabTimestamp",
61                      "@container" : "@set",
62                      "@values" : [
63              {"@type" :  "initDate"},
64              {"@type": "endDate"} ]
65               }
66      },
67      "@id": "http://sibex/usecase1",
68      "@type": "composed",
69      "description": "This composed resource coverts several time data to
            the necessary unit and align them to the same frequency",
70      "title": "Data preparation for the energy consumption prediction
            module",
71      "operation": [{
72          "method": "GET",
73          "expects": {
74              "startDate": "",
75              "endDate": ""
76          },
77              "returns": {"tabValues": [""]}
78      }],
79      "Workflow": {
80          "members": [{
81                  "@id": "http://sibex/measures",
82                  "method": "GET",
83                  "expects": [{
84                          "type": "Temperature"
85                      },
86                      {
87                          "zoneid": "BldgA"
88                      },
89                      {
90                          "startdate": "startDate"
91                      },
92                      {
93                          "enddate": "endDate"
94                      }
95                  ],
96                  "returns": [{
97                      "data": "A"
98                  }]
99              },
100             {
101                 "@id": " http://sibex/measures",
102                 "method": "GET",
103                 "expects": [{
104                         "type": "PowerEnergy"
105                     },
106                     {
```

```
                "zoneid": "BldgA"
            },
            {
                "startdate": "startDate"
            },
            {
                "enddate": "endDate"
            }
        ],
        "returns": [{
            "data": "B"
        }]
    },
    {
        "@id": " http://sibex/measures",
        "method": "GET",
        "expects": [{
                "type": "LightRadiation"
            },
            {
                "zoneid": "BldgA"
            },
            {
                "startdate": "startDate"
            },
            {
                "endate": "endDate"
            }
        ],
        "returns": [{
            "data": "C"
        }]
    },
    {
        "@id": "http://sibex/dataconversion",
        "method": "GET",
        "expects": [{
            "data": "A"
        }],
        "returns": [{
            "data": "D"
        }]
    },
    {
        "@id": "http://sibex/dataconversion",
        "method": "GET",
        "expects": [{
            "data": "B"
        }],
        "returns": [{
            "data": "E"
        }]
    },
    {
        "@id": "http://sibex/dataconversionr",
        "method": "GET",
        "expects": [{
            "data": "C"
        }],
        "returns": [{
            "data": "F"
        }]
    },
```

```
170             {
171                 "@id": "http://sibex/datalignment",
172                 "method": "GET",
173                 "expects": [{
174                         "data": "D"
175                     },
176                     {
177                         "frequency": "900"
178                     }
179                 ],
180                 "returns": [{
181                     "data": "G"
182                 }]
183             },
184             {
185                 "@id": "http://sibex/datalignment",
186                 "method": "GET",
187                 "expects": [{
188                         "data": "E"
189                     },
190                     {
191                         "frequency": "900"
192                     }
193                 ],
194                 "returns": [{
195                     "data": "H"
196                 }]
197             },
198             {
199                 "@id": "http://sibex/datalignment",
200                 "method": "GET",
201                 "expects": [{
202                         "data": "F"
203                     },
204                     {
205                         "frequency": "900"
206                     }
207                 ],
208                 "returns": [{
209                     "data": "I"
210                 }]
211             },
212             {
213                 "@id": "http://sibex/Merge3data",
214                 "method": "GET",
215                 "expects": [{
216                         "data": "G"
217                     },
218                     {
219                         "data": "H"
220                     },
221                     {
222                         "data": "I"
223                     }
224                 ],
225                 "returns": [{
226                     "data": "J"
227                 }]
228             }
229         ],
230         "goal": "J"
231     }
232 }
```

Appendix H

SIBEX Resource Description using Hydra

```
{
   "@context": {
      "@vocab": "http://www.w3.org/ns/hydra/core#",
      "schema": "http://schema.org/",
      "ifcTC1": "http://www.buildingsmart-tech.org/ifcOWL/IFC2X3_TC1/",
      "ifcFinal": "http://www.buildingsmart-tech.org/ifcOWL/
          IFC2X3_Final/",
      "regionId": "ifcTC1:globalId_IfcRoot",
      "sbxBI": "http://sibex/sbxBI/",
      "sbxOcc": "http://sibex/sbxOccupant/",
      "sbxProp": "http://sibex/sbxProperty/",
      "ssn": "https://www.w3.org/TR/vocab-ssn/",
      "qudt": "http://quadt.org/schema/qudt/",
      "type": "ssn:SOSAResult",
      "timestamp": "schema:DateTime",
      "startdate": "schema:DateTime",
      "enddate": "schema:DateTime",
      "updatedDate": "schema:DateTime",
      "frequency": "sbxProp:Frequency",
      "unit": "qudt:Unit",
      "quality": "schema:Float",
      "Workflow": "Collection",
      "resources": "Collection",
      "data": "schema:Text",
      "goal": "schema:Text",
      "id": "schema:identifier",
      "zoneid": "schema:identifier",
      "url": "schema:url",
      "acronym": "schema:Text",
      "value":"schema:Float",
      "tabCollVal": {
            "@id": "tabCollVal",
                  "@container" : "@set",
                  "@values" : [
            {"@type" : "quality"},
                        {"@type" : "timestamp"},
            {"@type" :  "updatedDate"},
            {"@type" :  "value"}]
            },
      "tabValues":   {
        "@id": "tabValues",
                  "@container" : "@set",
                  "@values" : [
           {"@type" :  "timestamp"},
           {"@type" :  "value"} ]
            },
```

```
46        "tabValues2":    {
47          "@id": "tabValues2",
48                    "@container" : "@set",
49                    "@values" : [
50              {"@type" :  "timestamp"},
51              {"@type" :  "value"} ]
52               },
53        "tabValues3":    {
54          "@id": "tabValues3",
55                    "@container" : "@set",
56                    "@values" : [
57              {"@type" :  "timestamp"},
58              {"@type" :  "value"} ]
59               },
60        "tabTimestamp":    {
61          "@id": "tabTimestamp",
62                    "@container" : "@set",
63                    "@values" : [
64              {"@type" :  "initDate"},
65              {"@type": "endDate"} ]
66               }
67    },
68     "resources": [{
69          "@id": "http://sibex/measure",
70          "@type": "dataCollection",
71          "description": "This resource allows data collection",
72          "title": "Get Measures",
73          "operation": [{
74             "method": "GET",
75             "expects": {
76                    "type": "",
77                    "zoneid": "",
78                    "startdate": "",
79                    "enddate": ""
80                 },
81             "returns": {"tabCollVal": [""]},
82             "acronym": "DC"
83          }],
84          "Workflow": ""
85       },
86         {
87
88          "@id": "http:// sibex/blankdetection",
89          "@type": "preprocessing",
90          "description": "This resource detects the missing values",
91          "title": "Blanks Detection",
92          "operation": [{
93             "method": "GET",
94             "expects": {"tabValues": [""]},
95             "returns":{"tabTimestamp": [""]},
96             "acronym": "BD"
97          }],
98          "Workflow": ""
99       },
100       {
101
102          "@id": "http://sibex/outlierdetection",
103          "@type": "preprocessing",
104          "description": "This resource detects outlier values",
105          "title": "Outliers Detection",
106          "operation": [{
107             "method": "GET",
108             "expects": {"tabValues": [""]},
```

```
109             "returns": {"tabTimestamp": [""]},
110             "acronym": "OD"
111          }],
112          "Workflow": ""
113       },
114       {
115         "@id": "http:// sibex/datainterpolation",
116          "@type": "preprocessing",
117          "description": "This resource interpolates on blanks/outliers
                ",
118          "title": "Data Interpolation",
119          "operation": [{
120             "method": "GET",
121             "expects":  { "tabTimestamp":[""],
122                           "tabValues": [""]},
123             "returns": { "tabValues": [""]},
124             "acronym": "DI"
125          }],
126          "Workflow": ""
127       },
128       {
129          "@id": "http://sibex/sumdata",
130          "@type": "preprocessing",
131          "description": "Sum data over a period ",
132          "title": "Sum Data",
133          "operation": [{
134             "method": "GET",
135             "expects": {"tabValues": [""]},
136             "returns":
137                {
138                   "value": ""
139                },
140             "acronym": "SD"
141          }],
142          "Workflow": ""
143       },
144       {
145          "@id": "http://sibex/substractdata",
146          "@type": "preprocessing",
147          "description": "Subtract data over a period",
148          "title": "Subtract Data",
149          "operation": [{
150             "method": "GET",
151             "expects": {"tabValues": [""]},
152             "returns":
153                {
154                   "value": ""
155                },
156             "acronym": "SsD"
157          }],
158          "Workflow": ""
159       },
160       {
161          "@id": "http://sibex/compmeandata",
162          "@type": "preprocessing",
163          "description": "Compute mean value over a period",
164          "title": "Compute Mean Value ",
165          "operation": [{
166             "method": "GET",
167             "expects": {"tabValues": [""]},
168             "returns":
169                {
170                   "value": ""
```

```
                },
            "acronym": "CMD"
        }],
        "Workflow": ""
    },
    {
        "@id": "http://sibex/compmediandata",
        "@type": "preprocessing",
        "description": "Compute median value over a period ",
        "title": "Compute Median Value ",
        "operation": [{
            "method": "GET",
            "expects": {"tabValues": [""]},
            "returns":
                {
                    "value": ""
                },
            "acronym": "CMdD"
        }],
        "Workflow": ""
    },
    {
        "@id": "http://sibex/datapercentage",
        "@type": "preprocessing",
        "description": "Compute data percentage",
        "title": "Compute data percentage",
        "operation": [{
            "method": "GET",
            "expects": {"tabValues": [""],
            "tabValues2": [""],
            "initDate": "",
            "endDate": ""},
            "returns":
                {
                    "value": ""
                },
            "acronym": "DP"
        }],
        "Workflow": ""
    },
    {
        "@id": "http://sibex/datamultiplication",
        "@type": "preprocessing",
        "description": "Data Multiplication",
        "title": "Data Multiplication",
        "operation": [{
            "method": "GET",
            "expects":  { "tabValues": [""],
                "tabValues2": [""]
                },
            "returns": {"tabValues": [""]},
            "acronym": "DM"
        }],
        "Workflow": ""
    },
    {
        "@id": "http://sibex/disaggregate",
        "@type": "preprocessing",
        "description": "Data Disaggregation",
        "title": "Data Disaggregation",
        "operation": [{
            "method": "GET",
            "expects": {
```

```
234                 "tabValues": [""]},
235              "returns":
236                 {"tabValues": [""]},
237              "acronym": "DisD"
238           }],
239           "Workflow": ""
240        },
241        {
242           "@id": "http://sibex/aggregate",
243           "@type": "preprocessing",
244           "description": "Data Aggregation",
245           "title": "Data Aggregation",
246           "operation": [{
247              "method": "GET",
248              "expects": {
249                 "tabValues": [""]},
250              "returns":
251              {"tabValues": [""]},
252              "acronym": "AgD"
253           }],
254           "Workflow": ""
255        },
256        {
257           "@id": "http://sibex/datalignment",
258           "@type": "preprocessing",
259           "description": "Align single time-series data on the same
                frequency",
260           "title": "Data Alignment",
261           "operation": [{
262              "method": "GET",
263              "expects": {"tabValues": [""],
264                 "frequency": ""
265              },
266              "returns":
267                 {"tabValues": [""]},
268              "acronym": "DA"
269           }],
270           "Workflow": ""
271        },
272        {
273           "@id": "http://sibex/conversion",
274           "@type": "preprocessing",
275           "description": "Convert data measures",
276           "title": "Data Conversion",
277           "operation": [{
278              "method": "GET",
279              "expects": {"tabValues": [""],
280                 "unit": ""
281              },
282              "returns":
283                 {"tabValues": [""]},
284              "acronym": "DCv"
285           }],
286           "Workflow": ""
287        },
288        {
289           "@id": "http://sibex/merge2Data",
290           "@type": "preprocessing",
291           "description": "Merge two time series data",
292           "title": "Merge 2 data",
293           "operation": [{
294              "method": "GET",
295              "expects": {"tabValues": [""],
```

```
296                 "tabValues2": [""]
297                 },
298                 "returns":
299                     {"tabValues": [""]},
300                 "acronym": "MG2"
301             }],
302             "Workflow": ""
303         },
304         {
305             "@id": "http://sibex/merge3Data",
306             "@type": "preprocessing",
307             "description": "Merge three time series data",
308             "title": "Merge 3 data",
309             "operation": [{
310                 "method": "GET",
311                 "expects": {"tabValues": [""],
312                 "tabValues2": [""],
313                     "tabValues3": [""]
314                 },
315                 "returns":
316                     {"tabValues": [""]},
317                 "acronym": "MG3"
318             }],
319             "Workflow": ""
320         }
321 
322     ]
323 }
```

Appendix I

Prototype APIs

1. **GET the description of all the resources**

Table I.1 – Request to get resources description

Method	URL	Filters
GET	/resources/description	resourceType
Type	**Parameters**	**Values**
HEAD	UserID	string
HEAD	SourceID	string
HEAD	ProxyToken	string
HEAD	Options	List[string]

Table I.2 – Response for getting resources description

Status	Response	Values
200	[resource{ URL, type, description, title, method, inParameters[inName, Type], inFixParameters[inFixName, Type], outParameters [outName, Type], workflow // optional }]	 string string string string string string string string string string string list[string]
400	{" bad request ":"Invalid options."}	
401	{" unauthorized ":"Invalid proxy token."}	
401	{" unauthorized ":"Unknown user."}	
401	{" unauthorized ":"Unknown source."}	
404	{" unknown service type ":serviceType}	
500	{" error ":"Internal error."}	

2. Create a resource composition

Table I.3 – Request to create a resource composition

Method	**URL**	
POST	/resources/composition	
Type	**Parameters**	**Values**
HEAD	UserID	string
HEAD	SourceID	string
HEAD	ProxyToken	string
HEAD	Options	List[string]

POST	{
	"composition":{
	"name",
	"description",
	"variables":[],
	"resources":[{
	"url": "",
	"method": "",
	"name": "",
	"param": [{"key": "value"},
	{etc.}],
	"output": ""
	}],
	"goal": ""
	}
	}
		
		string
		string
		List[string]
		
		string
		string
		string
		List[string]
		
		string
		
		string

Table I.4 – Response for creating a resource composition

Status	**Response**	
201	{"compID": compID}	
400	{" bad request ":"Invalid options."}	
400	{" bad request ":"Invalid type."}	
401	{" unauthorized ":"Invalid proxy token."}	
401	{" unauthorized ":"Unknown user."}	

403	{
	ValidationResults:{
	Reachability,
	Liveness,
	Interoperability
	}
	}
		
		boolean
		boolean
		boolean

500	{" error ":"Internal error."}	500

3. Store the resource composition

Table I.5 – Request to store the resource composition

Method	URL
POST	/resources/description

Type	Parameters	Values
HEAD	UserID	string
HEAD	SourceID	string
HEAD	ProxyToken	string
HEAD	Options	List[string]
200	[resource{ URL, description, title, method, inParameters[inName, Type], inFixParameters[inFixName, Type], outParameters [outName, Type], workflow // optional }]	 string string string string string string string string string string list[string]
400	{" bad request ":"Invalid options."}	
401	{" unauthorized ":"Invalid proxy token."}	
401	{" unauthorized ":"Unknown user."}	
401	{" unauthorized ":"Unknown source."}	
404	{" unknown service type ":serviceType}	
500	{" error ":"Internal error."}	

Table I.6 – Response for storing the resource composition

Status	Response
201	{"compositionID": compositionID}
400	{" bad request ":"Invalid options."}
400	{" bad request ":"Invalid type."}
401	{" unauthorized ":"Invalid proxy token."}
401	{" unauthorized ":"Unknown user."}
500	{" error ":"Internal error."}

4. Execute the resource composition

Table I.7 – Request to execute the resource composition

Method	URL	
GET	/resources/composition/compositionID	
Type	**Parameters**	**Values**
HEAD	UserID	string
HEAD	SourceID	string
HEAD	ProxyToken	string
HEAD	Options	List[string]

Table I.8 – Response for executing the resource composition

Status	Response	Values

200	{
	CompositionResults:{
	compositionURL,
	values[data]
	}
	}
		string
		List[string]

400	{" bad request ":"Invalid options."}	
401	{" unauthorized ":"Invalid proxy token."}	
401	{" unauthorized ":"Unknown user."}	
401	{" unauthorized ":"Unknown source."}	
404	{" unknown composition ":"compositionID."}	
500	{" error ":"Internal error."}	

Appendix J

Comparative Results between DFS and BFS

In this appendix, we present and analyze several tests to compare the performance of both DFS (Depth First Search) and BFS (Breadth First Search) in terms of response time (ms), while searching for a resource providing a specific function. The experiments were conducted on a simulated graph of 2000 resources (i.e., 1000 static and 1000 dynamic), providing functions defined in two different function graphs:

- Horizontally distributed (Figure J.1), in which 1000 functions are defined in a directed acyclic function graph distributed horizontally. The function graph consists of 50 branches, each, with 20 ordered functions.
- Vertically distributed (Figure J.2), in which 1000 functions are defined in a directed acyclic graph distributed vertically. The function graph consists of 50 dependent functions vertically connected to 2 branches with 450 vertically ordered functions, each.

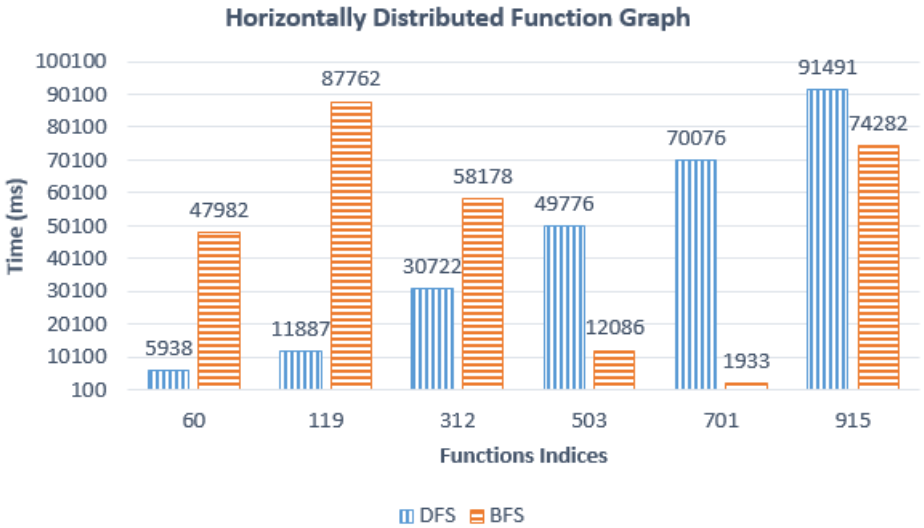


Figure J.1 – Horizontally distributed function graph

As illustrated in Figure J.1, when the functions exist in the lower left side of the function graph, such as f_{60}, f_{119}, and f_{312}, DFS is faster than the BFS. This is due to the exploration of the graph resources in a small number of branches of 20 functions, each, with DFS, while in BFS the exploration is done in an important number of levels consisting of 50 functions, each. However,

when the functions are in the upper right side of the function graph, as f_{503} and f_{701}, BFS is better than DFS, since it reaches the required function by crawling less number of levels comparing to the number of branches. As for the f_{915}, which is located in the lower right side of the function graph, BFS is still better than DFS, but the difference is not important comparing to the two tests applied for the previous functions (i.e., f_{503} and f_{701}) because there is a big number of branches and levels to cross for both DFS and BFS respectively.

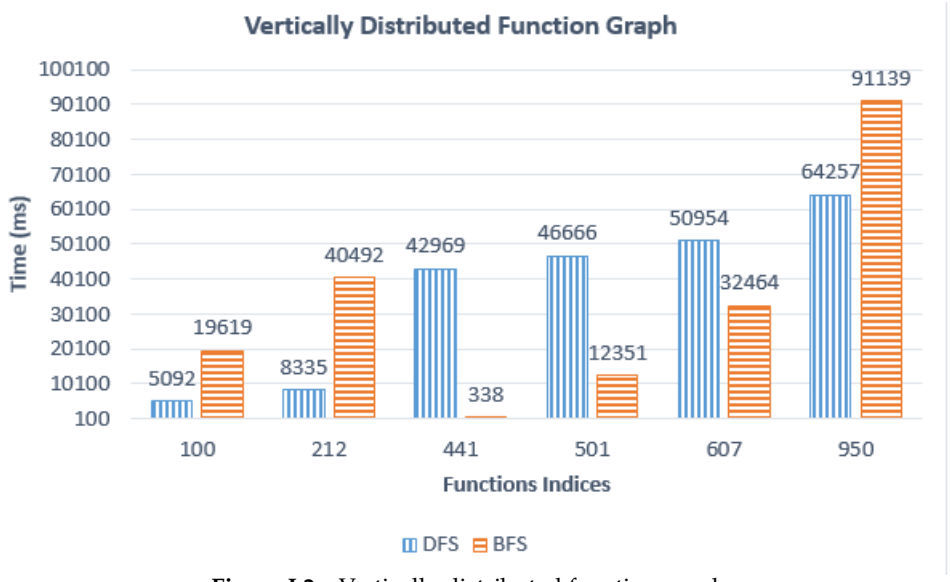


Figure J.2 – Vertically distributed function graph

In Figure J.2, DFS is faster than BFS when searching for a resource answering a function that is located in the upper left of the function graph, as f_{100} and f_{212}. The reason is that DFS traverses a single branch and reaches a function with a lower depth comparing to its total big depth, while BFS crawls the graph by levels and thus it moves from a long branch to another before reaching the required function. Nevertheless, BFS is better than DFS when searching for resources providing functions f_{441}, f_{501} and f_{607}. These functions exist in the upper right of the second branch, thus, they can be reached faster when crawling the graph by levels instead of moving deeper in the first branch with DFS. As for f_{950}, DFS is faster than BFS, but still both algorithms time responses are high comparing to the other tests. This is due to the function existence in the lower right side of the function graph, leading to a big number of levels for the BFS, and to an important depth of both branches for the DFS.

The results show that the performance of each algorithm depends on the functions distribution and the localization of the requested function in the function graph, FG. Currently, the algorithm type used to traverse the resource graph is specified by the solution administrator. In a later phase, it will be calculated dynamically based on the function graph topology.

Appendix K

Hydra Vocabulary Extended

```
{"@context": "http://www.h2g.eu/context.jsonld",
   "@id": "http://www.h2g.eu/resdesc/res-getairtemp.md",
   "location": "Zone1",
   "Operation": [{
      "method": "GET",
      "expects": ["h2g:startdate", "h2g:enddate"],
      "returns": ["schema:DateTime", "schema:Float"],
      "function": "ATC"}],
   "Link": [{
      "entrypoint": "http://www.h2g.eu/res-gethumidity",
      "method": "GET",
      "relationType": "isComplementary",
      "function": "EDP"
      }]
    }
```

Listing K.1 – Static resource description

```
{
     "@context": "http://www.h2g.eu/context.jsonld",
     "@id": "http://www.h2g.eu/resdesc/resd-getairtemp.md",
     "location": "Zone1",
     "Operation": [{
      "method": "GET",
      "expects":["h2g:startdate", "h2g:enddate"],
      "returns": ["schema:DateTime", "schema: Float"],
      "function": "ATC"}]
}
```

Listing K.2 – Dynamic resource description

```
     {
   "@context": "http://www.h2g.eu/context.jsonld",
   "@id": "http://www.h2g.eu/resdesc/vresp-getairtemp.md",
   "Operation": {
      "method": "GET",
      "function": "ATC"
   },
   "Collection": {
      "member": [{
         "@id": "http://www.h2g.eu/resd-getairtemp"
      }]
   },
   "Link": [{
      "entrypoint": "http://www.h2g.eu/vrest-gethumidity",
      "method": "GET",
      "relationType": "isRelated",
      "function": "HC"
   }]
```

```
}
```

Listing K.3 – Permanent virtual resource description

```
{
    "@context": "http://www.h2g.eu/context.jsonld",
    "@id": "http://www.h2g.eu/resdesc/vresp-getairtemp",
    "Operation": {"method": "GET", "function": "ATC"},
    "Collection": {
     "member": [{"@id": "http://www.h2g.eu/resd-getairtemp1"},
               {"@id": "http://www.h2g.eu/resd-getairtemp2"}]
     }
 }
```

Listing K.4 – Temporary virtual resource description

Appendix L

Performance Evaluation of the Indexing Schema Construction

In this appendix, we evaluate the performance of constructing the indexing schema, IdS, by assuming that all resources exposed by objects are withing the same location. Thus, only 2 dimensions i.e., Functions and Resources, are used. To this end, we conducted several experiments using different functions and resources graphs setups, and evaluated the IdS construction in terms of response time (ms) and memory usage (kb). The tests were done on a Linux Debian (64 bits) virtual machine, with 1 dedicated Intel® Core™ i7-46000 CPU @ 2.10GHz 2.70GHz processor and 1 GB RAM.

The construction of IdS consisted of:

- Retrieving the set of functions defined in FG with their corresponding signature (i.e., fsignature) based on the functions dependencies.
- Traversing the existing resource graph containing the static/dynamic resources to get their provided functions with their related resources (i.e., rsignature).
- Linking each function to the set of the resources realizing it.

Figure L.1 and Figure L.2 illustrate respectively the response time and the memory usage of the tests conducted while varying the number of functions defined in FG. The number of the static resource in these tests is 1000. As shown in both figures, the response time and the memory usage increase with the evolution of the number of functions. This is due to the calculations required to get the necessary signature of each function, and to link each function to the set of static resource matching it.

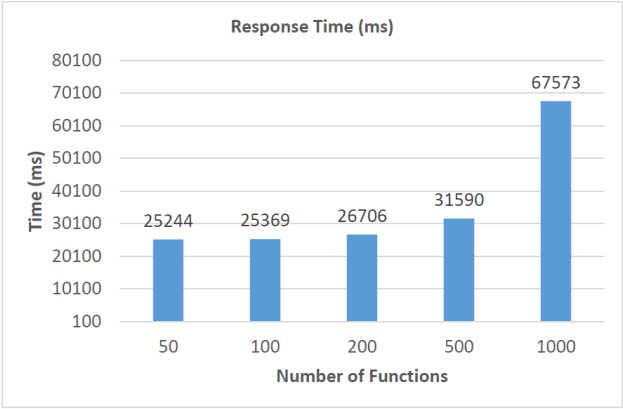


Figure L.1 – The indexing time of the tests conducted while varying the number of functions

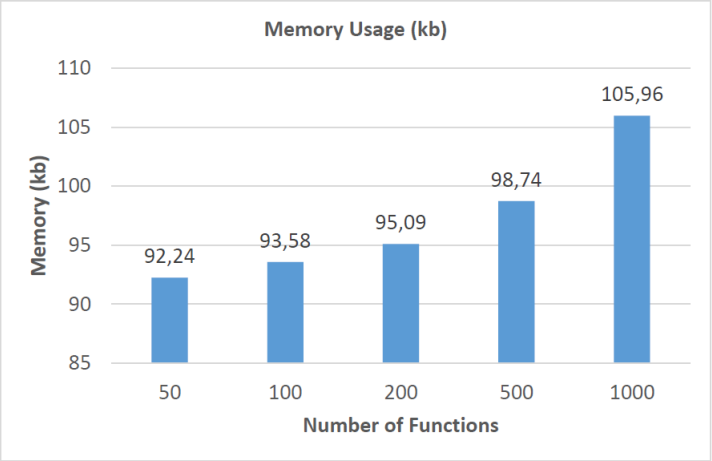


Figure L.2 – The indexing memory usage of the tests conducted while varying the number of functions

Figure L.3 and Figure L.4 show respectively the response time and the memory usage of the tests conducted while varying the number of static resources. In these tests the number of functions is 500. As it is seen in both figures, the response time and the memory usage increase when augmenting the number of resources. This is explained by the graph resource traversal to get the provided function of each resource with its related resources, and to the linking of each function to the set of resources answering it.

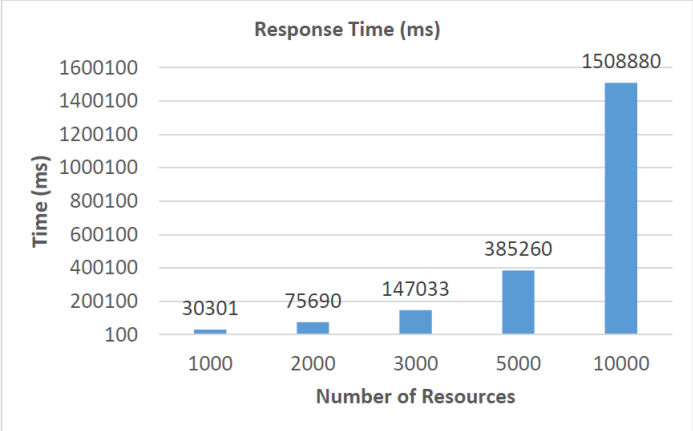


Figure L.3 – The indexing time of the tests conducted while varying the number of resources

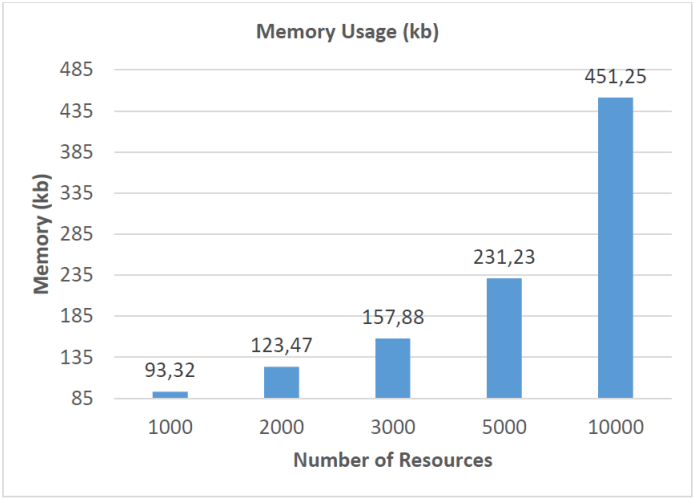


Figure L.4 – The indexing memory usage of the tests conducted while varying the number of resources

The experiments show that increasing the number of functions and resources affects the construction of the indexing schema in terms of response time and memory usage. However, these two aspects increase more with the growth of the resources number comparing to the functions number. In our work, the indexing schema is generated every time there is a change within the function graph (i.e., add/remove functions and change in the functions dependencies) or the resource graph (i.e., connect/disconnect static resources). Nevertheless, the indexing time shown in Figure L.1 has an exponential curve with the increase number of functions. The same graph pace appears in Figure L.3 and Figure L.4 related to the indexing time and indexing memory usage respectively, while increasing the resources number. These curves show that updating the indexing schema by regenerating it from the start requires lot of time and memory space when both functions number and resources number are relatively high. Although in real Web-based environments the number of resources does not normally exceed 10000, nor even the provided function are 1000, we seek in future works to

improve the indexing schema performance construction by updating it dynamically without regenerating it from scratch every time, and thus, avoiding huge response time and lot of memory space.

Bibliography

[1] Mohammad Aazam and Eui-Nam Huh. "Fog computing and smart gateway based communication for cloud of things". In: *2014 International Conference on Future Internet of Things and Cloud*. IEEE. 2014, pp. 464–470.

[2] Ben Adida. "RDFa in XHTML: Syntax and Processing W3C Recommendation". In: *http://www. w3. org/TR/rdfa-syntax/* (2008).

[3] Ahmed Al Amoodi and Elie Azar. "Impact of Human Actions on Building Energy Performance: A Case Study in the United Arab Emirates (UAE)". In: *Sustainability* 10.5 (2018), p. 1404.

[4] Rosa Alarcon and Erik Wilde. "From RESTful services to RDF: connecting the web and the semantic web". In: *arXiv preprint arXiv:1006.2718* (2010).

[5] Rosa Alarcon, Erik Wilde, and Jesus Bellido. "Hypermedia-driven RESTful service composition". In: *International Conference on Service-Oriented Computing*. Springer. 2010, pp. 111–120.

[6] Rosa Alarcon et al. "REST web service description for graph-based service discovery". In: *International Conference on Web Engineering*. Springer. 2015, pp. 461–478.

[7] Fernando Luis Almeida. "Concept and Dimensions of Web 4.0". In: *International Journal of Computers & Technology* 16.7 (2017), pp. 7040–7046.

[8] Areeb Alowisheq, David E Millard, and Thanassis Tiropanis. "EXPRESS: EXPressing REstful semantic services using domain ontologies". In: *International Semantic Web Conference*. Springer. 2009, pp. 941–948.

[9] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. "Selecting skyline services for QoS-based web service composition". In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 11–20.

[10] M Clement Joe Anand and Janani Bharatraj. "Theory of Triangular Fuzzy Number". In: *Proceedings of NCATM* 2017 (2017), p. 80.

[11] Meriem Aziez, Saber Benharzallah, and Hammadi Bennoui. "Service discovery for the Internet of Things: Comparison study of the approaches". In: *2017 4th International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE. 2017, pp. 0599–0604.

[12] Franz Baader, Ian Horrocks, and Ulrike Sattler. "Description logics as ontology languages for the semantic web". In: *Mechanizing mathematical reasoning*. Springer, 2005, pp. 228–248.

[13] Oumayma Bahri, Nahla Ben Amor, and Talbi El-Ghazali. "New Pareto approach for ranking triangular fuzzy numbers". In: *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*. Springer. 2014, pp. 264–273.

[14] Qinghai Bai. "Analysis of particle swarm optimization algorithm". In: *Computer and information science* 3.1 (2010), p. 180.

[15] Srividya Bansal et al. “Generalized semantic Web service composition”. In: *Service Oriented Computing and Applications* 10.2 (2016), pp. 111–133.

[16] Mike Barlow. *Real-time big data analytics: Emerging architecture*. " O'Reilly Media, Inc.", 2013.

[17] Payam Barnaghi, Amit Sheth, and Cory Henson. “From data to actionable knowledge: Big data challenges in the web of things [Guest Editors' Introduction]”. In: *IEEE Intelligent Systems* 28.6 (2013), pp. 6–11.

[18] Ricardo J Barrientos et al. “Range query processing in a multi-GPU environment”. In: *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. IEEE. 2012, pp. 419–426.

[19] Scott Beamer, Krste Asanović, and David Patterson. “Direction-optimizing breadth-first search”. In: *Scientific Programming* 21.3-4 (2013), pp. 137–148.

[20] Sean Bechhofer. “OWL web ontology language reference, W3C Recommendation”. In: *http://www. w3. org/TR/owl-ref/* (2004).

[21] Mahdi Bennara, Michael Mrissa, and Youssef Amghar. “An approach for composing RESTful linked services on the web”. In: *Proceedings of the 23rd International Conference on World Wide Web*. ACM. 2014, pp. 977–982.

[22] Mahdi Bennara, Michael Mrissa, and Youssef Amghar. “Linked Service Selection Using the Skyline Algorithm”. In: *International Conference on Model and Data Engineering*. Springer. 2016, pp. 88–97.

[23] Mahdi Bennara, Michael Mrissa, and Youssef Amghar. “Semantic-Enabled and Hypermedia-Driven Linked Service Discovery”. In: *International Conference on Model and Data Engineering*. Springer. 2016, pp. 108–117.

[24] Karim Benouaret, Djamal Benslimane, and Allel Hadjali. “Top-k web services compositions: A fuzzy-set-based approach”. In: *ACM-Symp. on Applied Computing (SAC)*. 2011, pp. 1038–1043.

[25] Rainer Berbner et al. “Heuristics for qos-aware web service composition”. In: *2006 IEEE International Conference on Web Services (ICWS'06)*. IEEE. 2006, pp. 72–82.

[26] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. DIANE Publishing Company, 2001.

[27] Tim Berners-Lee, James Hendler, Ora Lassila, et al. “The semantic web”. In: *Scientific american* 284.5 (2001), pp. 28–37.

[28] Tim Berners-Lee et al. “World-wide web: the information universe”. In: *Internet Research* (2010).

[29] L Blanes et al. “Integration of Fault Detection and Diagnosis with Energy Management Standard ISO 50001 and Operations and Maitenance of HVAC Systems”. In: *Clima 2013* (2013).

[30] Dan Brickley. “RDF vocabulary description language 1.0: RDF schema”. In: *http://www. w3. org/TR/rdf-schema/* (2004).

[31] Ethan Cerami. *Web services essentials: distributed applications with XML-RPC, SOAP, UDDI & WSDL*. " O'Reilly Media, Inc.", 2002.

[32] Richard Chbeir et al. “OntoH2G: A Semantic Model to Represent Building Infrastructure and Occupant Interactions”. In: *International Conference on Sustainability in Energy and Buildings*. Springer. 2018, pp. 148–158.

[33] Sofiane Chemaa, Raida Elmansouri, and Allaoua Chaoui. "Web services modeling and composition approach using object-oriented petri nets". In: *arXiv preprint arXiv:1304.2080* (2013).

[34] Roberto Chinnici et al. "Web services description language (wsdl) version 2.0 part 1: Core language". In: *W3C recommendation* 26.1 (2007), p. 19.

[35] Nupur Choudhury. "World wide web and its journey from web 1.0 to web 4.0". In: *International Journal of Computer Science and Information Technologies* 5.6 (2014), pp. 8096–8100.

[36] Erik Christensen et al. *Web services description language (WSDL) 1.1*. 2001.

[37] World Wide Web Consortium et al. "RDF 1.1 concepts and abstract syntax". In: (2014).

[38] Rone Ilídio Da Silva, Daniel Fernandes Macedo, and José Marcos S Nogueira. "Spatial query processing in wireless sensor networks–A survey". In: *Information Fusion* 15 (2014), pp. 32–43.

[39] C Dechsupa, W Vatanawood, and A Thongtak. "Formal verification of web service orchestration using colored petri net". In: *Proceedings of the international MultiConference of Engineers and Computer Scientists*. 2016.

[40] Gero Decker et al. "RESTful petri net execution". In: *International Workshop on Web Services and Formal Methods*. Springer. 2008, pp. 73–87.

[41] Shuiguang Deng et al. "Top-k Automatic Service Composition: A Parallel Method for Large-Scale Service Sets". In: *IEEE Transactions on Automation Science and Engineering* 11.3 (2014), pp. 891–905.

[42] Ivan Di Pietro, Francesco Pagliarecci, and Luca Spalazzi. "Semantic Annotation for Web Service Processes in Pervasive Computing". In: *Pervasive Computing*. Springer, 2009, pp. 289–311.

[43] Ivano Alessandro Elia, Nuno Laranjeiro, and Marco Vieira. "Test-Based Interoperability Certification for Web Services". In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2015, pp. 196–206.

[44] Andrea Ferrara. "Web services: a process algebra approach". In: *Proceedings of the 2nd international conference on Service oriented computing*. ACM. 2004, pp. 242–251.

[45] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, USA, 2000.

[46] Roy T Fielding and Richard N Taylor. "Principled design of the modern Web architecture". In: *ACM Transactions on Internet Technology (TOIT)* 2.2 (2002), pp. 115–150.

[47] Wan Fokkink. *Introduction to process algebra*. springer science & Business Media, 2013.

[48] Martin Garriga et al. "RESTful service composition at a glance: A survey". In: *Journal of Network and Computer Applications* 60 (2016), pp. 32–53.

[49] Vijay Gehlot and Carmen Nigro. "An introduction to systems modeling and simulation with colored petri nets". In: *Proceedings of the Winter Simulation Conference*. Winter Simulation Conference. 2010, pp. 104–118.

[50] Marc J Hadley. "Web application description language (WADL)". In: (2006).

[51] Rachid Hamadi and Boualem Benatallah. "A Petri net-based model for web service composition". In: *Proceedings of the 14th Australasian database conference-Volume 17*. Australian Computer Society, Inc. 2003, pp. 191–200.

[52] Guangjie Han et al. "Localization algorithms of wireless sensor networks: a survey". In: *Telecommunication Systems* 52.4 (2013), pp. 2419–2436.

[53] Sok-Min Han et al. "Using Pi-calculus to Model Dynamic Web Services Composition Based on the Authority Model". In: *New Review of Information Networking* 22.2 (2017), pp. 111–123.

[54] Antonio Garrote Hernández and María N Moreno García. "A formal definition of RESTful semantic web services". In: *Proceedings of the First International Workshop on RESTful Design*. ACM. 2010, pp. 39–45.

[55] Haibo Hu, Jianliang Xu, and Dik Lun Lee. "A generic framework for monitoring continuous spatial queries over moving objects". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, pp. 479–490.

[56] Wei Jiang, Songlin Hu, and Zhiyong Liu. "Top K query for QoS-aware automatic service composition". In: *IEEE Transactions on Services Computing* 7.4 (2013), pp. 681–695.

[57] Simon Jirka, Arne Bröring, and Christoph Stasch. "Discovery mechanisms for the sensor web". In: *Sensors* 9.4 (2009), pp. 2661–2681.

[58] Davis John and MS Rajasree. "Restdoc: Describe, discover and compose restful semantic web services using annotated documentations". In: *International journal of Web & Semantic Technology* 4.1 (2013), p. 37.

[59] Katawut Kaewbanjong and Sarun Intakosum. "Qos attributes of web services: A systematic review and classification". In: *Journal of Advanced Management Science Vol* 3.3 (2015).

[60] Lara Kallab et al. "HIT2GAP: Towards a better building energy management". In: *Energy Procedia* 122 (2017), pp. 895–900.

[61] Dervis Karaboga and Bahriye Akay. "A comparative study of artificial bee colony algorithm". In: *Applied mathematics and computation* 214.1 (2009), pp. 108–132.

[62] Jacek Kopeckỳ, Karthik Gomadam, and Tomas Vitvar. "hrests: An html microformat for describing restful web services". In: *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. Vol. 1. IEEE. 2008, pp. 619–625.

[63] Sana Ben Abdallah Ben Lamine et al. "An ontology-based approach for personalized RESTful Web service discovery". In: *Procedia computer science* 112 (2017), pp. 2127–2136.

[64] Markus Lanthaler and Christian Gütl. "Hydra: A Vocabulary for Hypermedia-Driven Web APIs." In: *LDOW* 996 (2013).

[65] Jonathan Douglas Lathem et al. "SA-REST: BRING THE POWER OF SEMANTICS TO REST-BASED WEB SERVICES Electronic Version Approved". In: (2007).

[66] Freddy Lécué and Alain Léger. "A formal model for semantic web service composition". In: *International semantic web conference*. Springer. 2006, pp. 385–398.

[67] Jae Moon Lee. "Fast k-nearest neighbor searching in static objects". In: *Wireless Personal Communications* 93.1 (2017), pp. 147–160.

[68] Li Li and Wu Chou. "Design and describe REST API without violating REST: A Petri net based approach". In: *2011 IEEE International Conference on Web Services*. IEEE. 2011, pp. 508–515.

[69] Dongsheng Liu et al. "Modeling workflow processes with colored Petri nets". In: *computers in industry* 49.3 (2002), pp. 267–281.

[70] Wei Liu et al. "Adaptive resource discovery in mobile cloud computing". In: *Computer Communications* 50 (2014), pp. 119–129.

[71] Meherun Nesa Lucky et al. "Enriching API descriptions by adding API profiles through semantic annotation". In: *International Conference on Service-Oriented Computing*. Springer. 2016, pp. 780–794.

[72] Alexander Maedche and Steffen Staab. "Ontology learning for the semantic web". In: *IEEE Intelligent systems* 16.2 (2001), pp. 72–79.

[73] Raluca Marin-Perianu, Hans Scholten, and Paul Havinga. "Prototyping service discovery and usage in wireless sensor networks". In: *32nd IEEE Conference on Local Computer Networks (LCN 2007)*. IEEE. 2007, pp. 841–850.

[74] Brian McBride. "The resource description framework (RDF) and its vocabulary description language RDFS". In: *Handbook on ontologies*. Springer, 2004, pp. 51–65.

[75] Deborah L McGuinness, Frank Van Harmelen, et al. "OWL web ontology language overview". In: *W3C recommendation* 10.10 (2004), p. 2004.

[76] Sheila A McIlraith, Tran Cao Son, and Honglei Zeng. "Semantic web services". In: *IEEE intelligent systems* 16.2 (2001), pp. 46–53.

[77] Daniel Mihályi and Valerie Novitzká. "What about linear logic in computer science". In: *Acta Polytechnica Hungarica* 10.4 (2013), pp. 147–160.

[78] María V Moreno, Miguel A Zamora, and Antonio F Skarmeta. "User-centric smart buildings for energy sustainable smart cities". In: *Transactions on emerging telecommunications technologies* 25.1 (2014), pp. 41–55.

[79] A Moreno-Munoz et al. "Distributed DC-UPS for energy smart buildings". In: *Energy and Buildings* 43.1 (2011), pp. 93–100.

[80] Debajyoti Mukhopadhyay and Archana Chougule. "A survey on web service discovery approaches". In: *Advances in Computer Science, Engineering & Applications*. Springer, 2012, pp. 1001–1012.

[81] Yohei Murakami, Donghui Lin, and Toru Ishida. *Services Computing for Language Resources*. Springer, 2018.

[82] Tadao Murata. "Petri nets: Properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.

[83] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. "An analysis of public rest web service apis". In: *IEEE Transactions on Services Computing* (2018).

[84] Suphakit Niwattanakul et al. "Using of Jaccard coefficient for keywords similarity". In: *Proceedings of the international multiconference of engineers and computer scientists*. Vol. 1. 6. 2013, pp. 380–384.

[85] Natalya F Noy. "Semantic integration: a survey of ontology-based approaches". In: *ACM Sigmod Record* 33.4 (2004), pp. 65–70.

[86] Luiz Henrique Nunes et al. "Multi-criteria IoT resource discovery: a comparative analysis". In: *Software: Practice and Experience* 47.10 (2017), pp. 1325–1341.

[87] In Lih Ong, Pei Hwa Siew, and Siew Fan Wong. "A five-layered business intelligence architecture". In: *Communications of the IBIMA* (2011).

[88] Hye-young Paik et al. "Web Service Composition: Overview". In: *Web Service Implementation and Composition Techniques*. Springer, 2017, pp. 149–158.

[89] Mike P Papazoglou and Willem-Jan Van Den Heuvel. "Service oriented architectures: approaches, technologies and research issues". In: *The VLDB journal* 16.3 (2007), pp. 389–415.

[90] Peter F Patel-Schneider. "OWL web ontology language semantics and abstract syntax, W3C Recommendation". In: *http://www. w3. org/TR/2004/REC-owl-semantics-20040210/* (2004).

[91] Cesare Pautasso. "RESTful web services: principles, patterns, emerging technologies". In: *Web Services Foundations*. Springer, 2014, pp. 31–51.

[92] Cong Peng and Guohua Bai. "Using Tag based Semantic Annotation to empower Client and REST Service Interaction". In: *COMPLEXIS 2018*. 2018, pp. 64–71.

[93] Franck Pommereau. "SNAKES: a flexible high-level petri nets library (tool paper)". In: *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer. 2015, pp. 254–265.

[94] Anass Rachdi, Abdeslam En-Nouaary, and Mohamed Dahchour. "Liveness and reachability analysis of BPMN process models". In: *Journal of computing and information technology* 24.2 (2016), pp. 195–207.

[95] Zeineb Rejiba et al. "F2C-aware: Enabling discovery in Wi-Fi-powered fog-to-cloud (F2C) systems". In: *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE. 2018, pp. 113–116.

[96] Pablo Rodriguez-Mier et al. "An integrated semantic web service discovery and composition framework". In: *IEEE transactions on services computing* 9.4 (2015), pp. 537–550.

[97] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[98] Khouloud Salameh et al. "OntoH2G: A Semantic Model to Represent Building Infrastructure and Occupant Interactions". In: *Sustainability in Energy and Buildings 2018: Proceedings of the 10th International Conference in Sustainability on Energy and Buildings (SEB'18)*. Vol. 131. Springer. 2018, p. 148.

[99] Ivan Luiz Salvadori and Frank Siqueira. "A framework for semantic description of restful web apis". In: *2014 IEEE International Conference on Web Services*. IEEE. 2014, pp. 630–637.

[100] Zhou Shao, David Taniar, and Kiki Maulana Adhinugraha. "Range-kNN queries with privacy protection in a mobile environment". In: *Pervasive and Mobile Computing* 24 (2015), pp. 30–49.

[101] Amit P Sheth, Karthik Gomadam, and Jon Lathem. "SA-REST: Semantically interoperable and easier-to-use services and mashups". In: *IEEE Internet Computing* 11.6 (2007), pp. 91–94.

[102] Manu Sporny et al. "JSON-LD 1.0". In: *W3C Recommendation* 16 (2014), p. 41.

[103] Jun Sun et al. "PAT: Towards flexible verification under fairness". In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 709–714.

[104] Murata Tadao. "Petri nets: properties, analysis and applications". In: *Proceedings of the IEEE* 77.4 (1990).

[105] Maurice H Ter Beek, Antonio Bucchiarone, and Stefania Gnesi. "Formal methods for service composition". In: *Annals of Mathematics, Computing & Teleinformatics* 1.5 (2007), pp. 1–10.

[106] Baojun Tian and Yanlin Gu. "Formal modeling and verification for web service composition". In: *Journal of software* 8.11 (2013), pp. 2733–2738.

[107] Thanh Tung Tran. "Verification of timed automata: reachability, liveness and modelling". PhD thesis. Université de Bordeaux, 2016.

[108] Balaji Varanasi and Sudha Belida. "HATEOAS". In: *Spring REST*. Springer, 2015, pp. 165–174.

[109] Ruben Verborgh et al. "Description and interaction of restful services for automatic discovery and execution". In: *2011 FTRA International workshop on Advanced Future Multimedia Services (AFMS 2011)*. Future Technology Research Association International (FTRA). 2011.

[110] Ruben Verborgh et al. "Survey of semantic description of REST APIs". In: *REST: Advanced Research Topics and Practical Applications*. Springer, 2014, pp. 69–89.

[111] Tiziano Villa et al. *Synthesis of finite state machines: logic optimization*. Springer Science & Business Media, 2012.

[112] Nikolaos S Voros, Wolfgang Mueller, and Colin Snook. "An Introduction to Formal Methods". In: *UML-B Specification for Proven Embedded Systems Design*. Springer, 2004, pp. 1–20.

[113] Henry Vu, Tobias Fertig, and Peter Braun. "Verification of Hypermedia Characteristic of RESTful Finite-State Machines". In: *Companion Proceedings of the The Web Conference 2018*. International World Wide Web Conferences Steering Committee. 2018, pp. 1881–1886.

[114] Hongbing Wang et al. "Effective bigdata-space service selection over trust and heterogeneous QoS preferences". In: *IEEE Transactions on Services Computing* 11.4 (2015), pp. 644–657.

[115] Jian Wang et al. "A web service discovery approach based on common topic groups extraction". In: *IEEE Access* 5 (2017), pp. 10193–10208.

[116] Lijuan Wang, Jun Shen, and Jianming Yong. "A survey on bio-inspired algorithms for web service composition". In: *Proceedings of the 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE. 2012, pp. 569–574.

[117] Sanjiva Weerawarana et al. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005.

[118] Ian H Witten et al. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[119] Xi Wu and Huibiao Zhu. "Formalization and analysis of the REST architecture from the process algebra perspective". In: *Future Generation Computer Systems* 56 (2016), pp. 153–168.

[120] Xiaofei Xu et al. "Novel artificial bee colony algorithms for QoS-aware service selection". In: *IEEE Transactions on Services Computing* 12.2 (2016), pp. 247–261.

[121] YanPing Yang, QingPing Tan, and Yong Xiao. "Verifying web services composition based on hierarchical colored petri nets". In: *Proceedings of the first international workshop on Interoperability of heterogeneous information systems*. ACM. 2005, pp. 47–54.

[122] Ting Yuan et al. "Formalization and Verification of REST on HTTP Using CSP". In: *Electronic Notes in Theoretical Computer Science* 309 (2014), pp. 75–93.

[123] Liangzhao Zeng et al. "QoS-aware middleware for web services composition". In: *IEEE Transactions on software engineering* 30.5 (2004), pp. 311–327.

[124] Xia Zhao. "A linear logic approach to RESTful web service modelling and composition". In: (2013).

[125] Xia Zhao, Enjie Liu, and Gordon J Clapworthy. "A two-stage restful Web service composition method based on linear logic". In: *2011 IEEE Ninth European Conference on Web Services*. IEEE. 2011, pp. 39–46.

[126] Hao Zheng, Yixiong Feng, and Jianrong Tan. "A fuzzy QoS-aware resource service selection considering design preference in cloud manufacturing system". In: *The International Journal of Advanced Manufacturing Technology* 84.1-4 (2016), pp. 371–379.

[127] Bowen Zhou et al. "A context sensitive offloading scheme for mobile cloud computing service". In: *2015 IEEE 8th International Conference on Cloud Computing*. IEEE. 2015, pp. 869–876.

[128] Ivan Zuzak, Ivan Budiselic, and Goran Delac. "A finite-state machine approach for modeling and analyzing restful systems". In: *Journal of Web Engineering* 10.4 (2011), p. 353.