HAL
open science

Dynamic Composition of Functionalities of Networked Devices in the Semantic Web

Sattisvar Tandabany

To cite this version:

Sattisvar Tandabany. Dynamic Composition of Functionalities of Networked Devices in the Semantic Web. Computer Science [cs]. Université Joseph-Fourier - Grenoble I, 2009. English. NNT : . tel-00436707v2

HAL Id: tel-00436707

https://theses.hal.science/tel-00436707v2

Submitted on 27 Nov 2009

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

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

UNIVERSITÉ JOSEPH FOURIER – GRENOBLE 1

École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

Dynamic Composition of Functionalities of Networked Devices in the Semantic Web

THÈSE

présentée et soutenue publiquement le 23 novembre 2009

pour l'obtention du

Doctorat de l'Université Joseph Fourier – Grenoble 1

(spécialité informatique)

par

Sattisvar TANDABANY

Directeur de thèse : Marie-Christine ROUSSET

Composition du jury

Président :	Christine COLLET	Professeur ENSIMAG Grenoble INP – Grenoble
Rapporteurs :	Djamal BENSLIMANE	Professeur à l'UCBL – Lyon
	Farouk TOUMANI	Professeur à l'Université du Sud – Toulon
Directeur de thèse :	Marie-Christine ROUSSET	Professeur à l'UJF – Grenoble

Laboratoire d'Informatique de Grenoble

The gift given without any hope for reward, with a sense of duty, is given in proper place and time to a deserving person, that gift is said to be pure.

Bhagavad-Gita, XVII

Acknowledgment

The path taken by this thesis was not predictable in its beginning. The tortuous meanders it has made me follow have uncovered significant people to interact with. I wish to express in these few lines my gratitude to all those people with whom I was able to complete this thesis.

In particular, I extend my sincere thanks to my supervisor Professor Marie-Christine Rousset, who guided me throughout these years of my PhD, who committed to this adventure, who has undeniably managed to show me that the light at the end of this tunnel was not the headlight of another locomotive. What the research means to me and how much I am attached to it is thanks to her.

I feel particular deference to Professor Yuzuru Tanaka who welcomed me for one year in his laboratory in Sapporo, Japan. I owe him for sharpening my scientific cultivation through lengthy discussions. With him I learned the proverb that framed my concept of time:

> "Nothing is more precious than time, so there is no greater generosity than to waste it without counting"

I also thank Professors Djamal Benslimane and Farouk Toumani, for their careful reading of this manuscript and their report, and Professor Christine Collet for chairing the jury of my PhD.

My research was enhanced by teaching, and this activity let me explore a different facet of research: pedagogy. I thank therefore Michel Burlet and Jean-Pierre Peyrin for their invaluable advice regarding the transmission of knowledge.

My work took place primarily within the team HADAS at Grenoble. I thank its members for their help and support through the last days. I think especially of my fellow office and other doctoral students who shared with me moments of distraction: Remi, Benjamin, Noha, Charlotte, Christine, Diana, and Naga.

My stays in Japan made me mingle with people from across the world with different cultures and have opened a critical eye on both my Indian and Western heritages. I thank Yu Asano, Akio Takashima, Hajime Imura, Fubuki Susuki, Tsuyoshi Sugibuchi, Aran Lunzer and all my friends at the Friendship House. A special thanks to my two roommates Vivi and Kama for the long winter evenings gazing at the Sapporo snow.

Outside of daily work and scientific exchanges, in need of body and mind to be entertained, I would like to express my gratitude to all my family who supported me and cheered at every step; to my father for his determination to explain that we can move mountains provided that it is done stone by stone; to my mother for her ability to talk to me about various and entertaining subjects. My evening of tango and lindy hop, as well as the dancers I met then, also gave me a second wind when writing.

I think very tenderly of Helia who has contributed significantly to both the serenity and the tumult of my life during my PhD. Also, I think with emotion of Florence for our sincere discussions on our slices of life and philosophies of science. And Étienne who interfered with my life like noone could infer.

In addition, special mention to my friends who have welcome my unclassifiable mind and gave it the space to flourish. I refer to Christelle, Celine, David, Lucrezia, Mathias, Kathrin, Géraldine, Claudia, Julie, Fleur.

Finally, I have an endless list of friends around the world, met when out traveling, and who have each introduced their speck of sand into the cogwheels of my mind and made me what I am today. I will not take up the challenge of listing them all because of the risk of omitting some. I am grateful to all of them.

Et que mettant mon âme à côté du papier,
Je n'ai tout simplement qu'à la recopier.

Cyrano de Bergerac, II, 3, *Cyrano*,
Edmond Rostand

Remerciements

Le cheminement pris par cette thèse n'avait rien de prévisible en ses débuts. Les méandres tortueux qu'il m'aura fait prendre m'auront permis de découvrir des gens marquants et d'échanger avec eux. Je désire exprimer dans ces quelques lignes ma reconnaissance envers toutes ces personnes grâce à qui j'ai pu mener à terme ce travail de thèse.

En particulier, j'adresse mes remerciements les plus sincères à ma directrice le Professeur Marie-Christine Rousset qui m'a guidé tout au long de ces années de doctorat, s'est investie dans cette aventure, et qui a indéniablement su me montrer que la lumière au bout de ce tunnel n'était pas le phare avant d'une autre locomotive. Ce que la recherche représente à mes yeux et combien j'y suis attaché a pu se révéler grâce à elle.

J'éprouve une déférence particulière à l'égard du Professeur Yuzuru Tanaka qui m'a accueilli durant un an dans son laboratoire à Sapporo au Japon. Je lui dois l'affûtage de ma culture scientifique à travers de très longues discussions. C'est avec lui que j'ai appris le proverbe qui a dessiné ma conception du temps:

> « Rien n'est plus précieux que le temps, il n'y a donc pas de plus grande générosité qu'à le perdre sans compter. »

Je remercie également les Professeurs Djamal Benslimane et Farouk Toumani, pour leur lecture attentive de ce manuscrit et leur rapport, ainsi que le Professeur Christine Collet pour avoir présidé le jury de ma soutenance.

Ma recherche s'est agrémentée d'enseignements et cette activité a permis d'explorer une autre facette du chercheur, la pédagogie. Je remercie à ce sujet Michel Burlet et Jean-Pierre Peyrin pour leurs conseils inestimables en matière de transmission de savoir.

Mon travail s'est déroulé d'abord au sein de l'équipe HADAS à Grenoble dont je remercie les membres pour leur aide et leur soutien jusque dans les derniers jours. Je pense particulièrement à mes compagnons de bureau et les autres doctorants qui ont partagé avec moi mes moments d'égarements: Rémi, Benjamin, Noha, Charlotte, Christine, Diana et Naga.

Puis mon séjour au Japon m'a fait côtoyer des personnes de l'autre bout de la planète avec une culture différente et m'ont ouvert un oeil critique étonnant sur mes deux cultures indienne et occidentale. Je remercie Yu Asano, Akio Takashima, Hajime Imura, Fubuki Susuki, Tsuyoshi Sugibuchi, Aran Lunzer et tous les amis de la House Friendship. Un merci particulier pour mes deux colocatrices Vivi et Kama pour les longues soirées d'hiver à contempler la neige de Sapporo.

En dehors du travail quotidien et des échanges scientifiques, dans la nécessité du corps et de l'esprit de se divertir, je voudrais exprimer ma gratitude à toute ma famille pour m'avoir supporté et réconforté à chaque pas; à mon père pour sa détermination à m'expliquer que l'on peut même déplacer des montagnes pourvu qu'on s'y prenne pierre après pierre; à ma mère pour sa capacité à me parler de sujets variés et distrayants. Mes soirées de tango et de lindy-hop et tous les danseurs que j'y ai rencontrés m'ont aussi donné un second souffle lors de la rédaction.

J'ai une pensée très tendre à l'endroit d'Hélia qui a contribué pour beaucoup à la fois à la sérénité et au tumulte de ma vie de doctorant. Aussi je pense avec émotion à Florence pour nos discussions sincères sur nos tranches de vie et sur les philosophies des sciences. Et Étienne qui a interféré dans ma vie comme personne n'aurait pu l'inférer.

Encore, une mention spéciale à mes amis qui ont su accueillir mon esprit inclassable et lui donner l'espace nécessaire pour son épanouissement. Je pense ici à Christelle, Céline, David, Lucrezia, Mathias, Kathrin, Géraldine, Claudia, Julie, Fleur.

Enfin, j'ai une infinie liste d'amis de par le monde, rencontrés au gré des voyages et qui ont chacun déposé leur grain de sable dans les rouages de ma pensée et qui font de moi ce que je suis aujourd'hui. Je ne prendrai pas le pari de tous les énumérer sans risquer d'en omettre certains, je leur suis reconnaissant à tous.

Je dédie cette thèse
à mes parents,
à ma soeur Satya,
et à mon frère Steve

Contents

Acknowledgment **i**

Remerciements **iii**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

1.1 Problem statement 2

1.2 Illustrative scenarios 3

1.2.1 Mixing console 3

1.2.2 Viewing a webcam knowing its location 3

1.2.3 Transferring files 3

1.2.4 Displaying in a distant location 4

1.2.5 Screens side by side 4

1.2.6 Printing a pdf file on a postscript printer 4

1.2.7 Alert in a house 4

1.2.8 Requirements 4

1.3 Sketch of the approach 5

1.4 Contributions 6

1.4.1 A logical class-based language with functions 6

1.4.2 A feasability study of inferring compositions on demand using a PROLOG engine 6

1.4.3 A solution for a decentralised deployment of the approach 6

2 A logic-based language to describe devices **7**

2.1 Description of classes, instances and properties 8

2.1.1 Taxonomies 8

2.1.2 Instances 8

2.1.3 Predicates 9

2.2 Description of functionalities . . . 11
2.2.1 The use of functions . . . 12
2.2.2 The signature and the assertions of a functionality . . . 12
2.3 Logical description of a device . . . 14
2.4 Composition of functionalities . . . 16
2.4.1 The relay race composition . . . 16
2.4.2 The assembly line composition . . . 17
2.5 Specification of queries . . . 17
2.5.1 Yes-no queries . . . 19
2.5.2 Wh- queries . . . 19
2.5.3 Composed functionalities as answers to queries . . . 20
2.6 Specification of virtual objects . . . 20
2.6.1 Virtual devices . . . 20
2.6.2 Virtual functionalities . . . 21
2.6.3 Combination of virtual devices and virtual functionalities . . . 22
2.7 Extension to heterogeneous descriptions . . . 22

3 Centralised reasoning for dynamic composition 25
3.1 Using PROLOG as a reasoning engine . . . 25
3.2 Experiments . . . 26
3.2.1 The basis of the scenarios . . . 26
3.2.2 About the curves . . . 27
3.3 Different scenarios . . . 27
3.3.1 Scenario almost without type constraint . . . 27
3.3.2 Best case: a single possible composition of depth 2 . . . 27
3.3.3 A single possible composition of depth n . . . 29
3.3.4 n possible compositions of depth 2 . . . 31
3.3.5 n^2 possible compositions of depth 2 . . . 33
3.3.6 $2^{n/2}$ possible compositions of depth $n/2$ . . . 34
3.4 Summary . . . 36

4 Decentralised reasoning for dynamic composition 39
4.1 Reminders and preliminaries . . . 40
4.1.1 Logical description of a device and its functionalities . . . 40
4.1.2 Queries . . . 42
4.1.3 Example . . . 44
4.1.4 Relevant devices for a query . . . 45

4.2 Propositional encoding: principles and properties 48
4.2.1 Encoding of device descriptions and queries 48
4.2.2 Properties 52
4.2.3 Examples 57
4.3 Look-up by decentralised propositional reasoning 59
4.3.1 Look-up: definition and properties 59
4.3.2 Decentralised computation of $\mathrm{Lookup}\ (Q)$ using SOMEWHERE 61
4.4 Heterogeneous descriptions 62

5 Related Work and Conclusion **63**
5.1 Semantic Approach of Dynamic Web Services Composition 64
5.2 Web Service Composition via Planning 67
5.3 Logic based language for Web services composition 68
5.4 Conclusion 71

A Glossary **73**

B Index **74**

C Bibliography **77**

List of Figures

1.1 Architecture of an opportunistic network . . . 2

2.1 Graphical representation of the taxonomy shown in table 2.1 . . . 9
2.2 Two outputs and their ordered trees of composition . . . 18
(a) Expressions of outputs . . . 18
(b) Ordered trees of composition . . . 18
(c) Semantic of each functionality and device . . . 18
2.3 Example of mappings between taxonomies . . . 23

3.1 Schema of connections in the scenario 3.3.2 . . . 28
3.2 Scenario with a single possible composition of depth 2 . . . 28
(a) Time consumption . . . 28
(b) Memory usage . . . 28
3.3 Schema of connections in the scenario 3.3.3 . . . 29
3.4 Scenario with a single possible composition of depth n . . . 30
(a) Time consumption . . . 30
(b) Memory usage . . . 30
3.5 Schema of connections in the scenario 3.3.4 . . . 31
3.6 Scenario with n possible compositions of depth 2 . . . 32
(a) Time consumption . . . 32
(b) Memory usage . . . 32
3.7 Schema of connections in the scenario 3.3.5 . . . 33
3.8 Scenario with n^2 possible compositions of depth 2 . . . 34
(a) Time consumption . . . 34
(b) Memory usage . . . 34
3.9 Schema of connections in the scenario 3.3.6 . . . 35
3.10 Scenario with 2^p composition of depth $p = \lfloor n/2 \rfloor$ . . . 35
(a) Time consumption . . . 35
(b) Memory usage . . . 35

4.1 A derivation tree of a query with respect to a set of rules . . . 43
(a) Derivation tree . . . 43
(b) Set of rules . . . 43
4.2 Example of the encoding of a taxonomy . . . 51
(a) The taxonomy used for the schema . . . 51
(b) Partial encoding of the taxonomy 4.2(a) in the context of the predicate *located* . . . 51
4.3 The encoding of a description shown in a graph . . . 58
4.4 Derivation tree of the query Q with respect to the rules in table 4.1 . . . 60

4.5 Propositional implicants of $\mathrm{Code}\,(Q)$ with respect to the encoding of the rules 60
4.6 Schema of the whole process in a device 61

List of Tables

2.1 Definition of a taxonomy with the syntax of first-order logic 9
2.2 The RDFS notation and the syntax of first-order logic 10
2.3 Description of a functionality in the general case 13
2.4 Example of the description of a functionality in high-level language and first-order logic 14
2.5 Another example of the description of a functionality 14
(a) Description in high-level language 14
(b) Description in first-order logic 14
2.6 Rewriting of the high-level language in first-order logic 15
2.7 Rewriting mappings using only inclusion 23
2.8 Semantic in first-order logic of the mappings 24

3.1 Data of the scenario with a single possible composition of depth 2 29
3.2 Data of the scenario with a single possible composition of depth n 31
3.3 Data of the scenario with n possible compositions of depth 2 32
3.4 Data of the scenario with n^2 possible compositions of depth 2 33
3.5 Data of the scenario with 2^p compositions of depth $p = \lfloor n/2 \rfloor$ 36
3.6 Summary of all the scenarios 36

4.1 Descriptions of two devices and their encoding 58
(a) Rules that enable the inheritance of types 58
(b) Description of a computer 58
(c) Description of a server 58

Chapter 1

Introduction

Contents

1.1 Problem statement **2**
1.2 Illustrative scenarios **3**
1.2.1 Mixing console 3
1.2.2 Viewing a webcam knowing its location 3
1.2.3 Transferring files 3
1.2.4 Displaying in a distant location 4
1.2.5 Screens side by side 4
1.2.6 Printing a pdf file on a postscript printer 4
1.2.7 Alert in a house 4
1.2.8 Requirements 4
1.3 Sketch of the approach **5**
1.4 Contributions **6**
1.4.1 A logical class-based language with functions 6
1.4.2 A feasability study of inferring compositions on demand using a PROLOG engine 6
1.4.3 A solution for a decentralised deployment of the approach 6

Technology's constant progress brings more and more independent devices dedicated to different goals. Moreover, an easier access to the Internet, and also cheaper and smaller devices, offer wider possibilities to achieve complex tasks by combining many simple tasks together. However, the problem is to find the appropriate functionalities; to contact the devices which can perform it; and to explicit how to combine them.

In this thesis, we address composition of distributed resources. Pervasive computing deals not only with Web services, but also with services provided by physical smart objects mutually connected through Peer-to-peer (P2P) networks. By 'resource', we mainly mean devices and components, but also data, Web services and computational resources accessible through the Web. Furthermore, service composition deals with all these resources as component services, and focuses on their composition. Our model considers Web services and physical smart objects respectively as logical and physical devices. Consequently, it can model service composition as functionalities composition.

Any smart object which can be connected by a cable or via a network to other objects can be considered as a device. The interest of the devices is in their functionalities. As we are faced with a growing number of devices and their various functionalities, it would be of great help to be able to look for a device (or a group of devices) which realises a certain functionality or matches certain criteria. It would

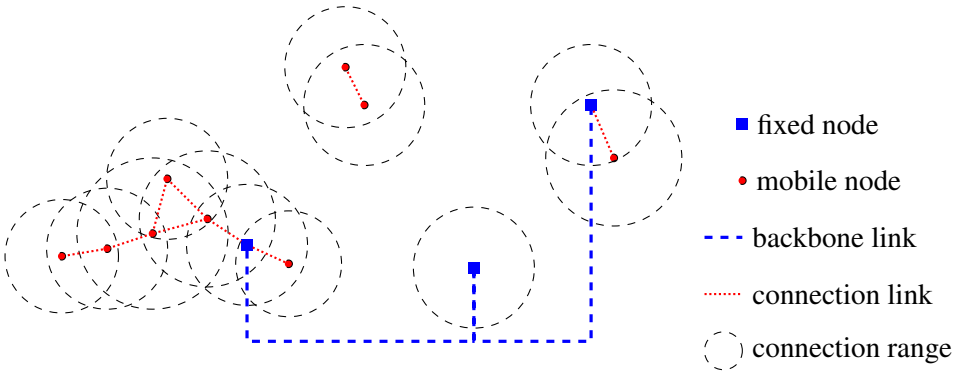


Figure 1.1: Architecture of an opportunistic network

also be useful to compose functionalities from interoperable devices automatically, to build new functionalities on the fly and to name them in order to reuse them easily.

In order to join the network, a new peer chooses any of the peers in the current network and provides a description of the functionalities that it offers. To enable it to provide such a description, we propose a language in which the functionalities are first-class citizens. We want to make it possible to define, infer and query possibly composed functionalities, and to retrieve the (group of) devices realising them.

In the continuity of the *Semantic Web* which was first introduced by [5] and extends the current Web with well-defined meaning and machine-understandable information, we add semantics to the description.

We handle a network of resources which can have either a centralised repository of the descriptions of each peer; or each peer can be autonomous and none of them may have the knowledge of the whole topology of the network. Our work is thus split in two steps. In the first part, we assume that every description is stored in a central repository on which we can use a reasoning system. We will establish the language of description of the resources and the language of queries. Then, we will extend the case to a decentralised description storage, where each peer stores its own description.

1.1 Problem statement

An opportunistic network of devices is a network of connected (possibly by wireless) devices either mobile or fixed. The network topology may change due to device mobility or device activation and deactivation. The devices provide at least the following two main abilities:

- Discovery: A device is able to discover other network nodes in direct communication range;
- One-hop message exchange: A device is able to send and receive arbitrary data in form of a message to or from any other node in direct communication range.

An opportunistic network is a quite a low-sized network. But if some fixed node are linked, a wider backbone network can appear to support many opportunistic networks. For instance some access points connected to the Internet may play this role. But still, this is not mandatory. Figure 1.1 shows such an architecture.

On top of such a dynamic network of devices, resource aggregation is facilitated. A mobile device can effectively 'dock' into the network thereby enabling resources to be shared between devices within the network. This ability can be used in many scenarios as detailed in section 1.2. Therefore, sharing

dynamic resources is of interest if and only if one is able to look for functionalities or devices available through the dynamic network. This is the first problem we want to address.

Moreover, resources shared are even more interesting if the composition of the functionalities is possible. A composition of two (or more) functionalities has to meet some condition:

- The functionalities to be composed have to share data, thus, a connection between the devices that provide the functionalities must exist in the network;
- The data shared must be of the same kind;
- If the functionalities require some conditions to be achieved, we need to check them first.

When the right devices providing the right functionalities meeting the above conditions are found, we can claim that a composition is computed. The composition, due to the dynamic character of the network, has to be computed each time it is needed, according to the current devices and functionalities present. Finding a composition in an opportunistic network is the second problem we are addressing.

Finally, and this is an inherent part of the two first problems, a description language that permits us to look for devices and functionalities and also to find composition needs to be defined.

1.2 Illustrative scenarios

In order to figure out what the description language — and consequently also the query language — that we will need, we introduce seven scenarios to illustrate each requirement of the description language and the query language.

1.2.1 Mixing console

Let us consider an audio mixing console. This device has many functionalities like, for instance, combining two or more audio streams together and possibly applying filters. Filters can amplify the volume, reduce noises, *etc*. Each filter applied can be described as a separate functionality, which takes an audio stream as input and provides a modified audio stream as output. As a result, the type of the input and the type output are not enough to distinguish those functionalities. Thus, we need to label them — *id est* to type them.

1.2.2 Viewing a webcam knowing its location

Suppose you have a device which is able to display a video stream coming from a webcam. Suppose also that you have to decide whether you can go to ski in a specific winter sports resort. To help your choice you may want to search a webcam near the resort so that you can check whether there is powder snow or whether the resort is crowded. So, the query will be about a functionality which captures video of the environment and which is performed by a device found by its location — near a certain winter sports resort. The properties, such as the location, of the devices enable us to narrow the scope of a query.

1.2.3 Transferring files

Once you came back from the resort of the previous scenario, you may have to send pictures or videos to your friends or relatives. As these data can represent many mega octets, emailing them is not suitable. Then, the need raises to exchange data between two distant devices (computers). If both computers are connected to the Internet, the first user can publish its file on the Web and give the URL to the second one, so that he can fetch it. Otherwise, we will have to find and handle a possibly long path of computers

in between. The data will be copied from one computer to the next one throughout the path. At every step, we must have a way to know that the copied data represents the same data.

1.2.4 Displaying in a distant location

Here is a more complete scenario. Suppose you have slides in a file stored somewhere in your computer, and you need to display it through a projector located in a distant room where people are waiting for your presentation. So, you are looking for a device that can display your file in that room. It sounds like the first scenario, except that the projector cannot display files from any computer but from the one connected to it. This means that the expected answer to the query has to be a composed functionality:

1. Your file must be transferred to the computer connected to the projector;
2. This computer has to convert the copy of your file into a video stream;
3. This video stream has to be sent to the projector in order to be displayed.

1.2.5 Screens side by side

Suppose that in the distant room of the previous scenario, we do not have one but two screens which can both display. Moreover, we have a device which can divide a video stream into a right and a left part splitting the video down the middle. It is quite natural to display the right part of the incoming video stream on the right screen and the left part on the left screen. Then, once for all, we can set these two screens up to work together, considering them as one single virtual device with a displaying functionality (with a higher resolution). This scenario leads to the idea of a virtual device made of a set of devices working together.

1.2.6 Printing a pdf file on a postscript printer

Postscript printers are printers which can only print files in the *postscript* format. However, articles and reports are usually stored in the Portable Document File (PDF) format and converters from PDF to *postscript* are not always available. A solution could be to give the printer a newly defined functionality which will, in fact, use an available converter (if there is one) before printing. This scenario shows that one could need to set up a new functionality, depending on the context — *id est* the available devices and functionalities. Following the example of the virtual device, this is a virtual functionality .

1.2.7 Alert in a house

Let us consider a house where simple devices like a phone, a television or lights can be reached and remotely controlled. Each of these devices has different functionalities (ring, display subtitles, blink, *etc*.). If we want the users to pay attention on a peculiar thing, we can let the lamp blink, display a message on the TV screen, or ring the phone. Then any of those functionalities could serve this purpose, even though they are typed differently and have different aims. We need somehow to say that a ringing functionality is also a warning functionality, that ringing is a subtype of warning. This leads to classify the types in a hierarchy.

1.2.8 Requirements

The scenarios introduced in this section suggest the following requirements for the description language. We need of course to describe a device by specifying its functionalities. The important point is that we

need a rich description of the functionalities. In particular, the more we constrain the functionalities, the easier we will find those matching a given specification or query. Therefore, we use taxonomies to type (see scenario 1.2.7) devices, functionalities (see scenario 1.2.1) and their inputs and output. We will allow the specification of preconditions in order to express the conditions under which the functionality can be realised. We will also allow the specification of post-conditions in order to express properties of the output after its realisation.

Furthermore, we need to specify the properties which are specific to each device — like its physical location (see scenario 1.2.2), its availability or its capacity — as well as the properties which are specific to each functionality. The connection between devices and the sameness of two data are peculiar properties (see scenario 1.2.3) which require a particular treatment.

Many devices interact with their environment. Some of them (*e.g.* sensors, cameras, captors, *etc.*) get the inputs for some of their functionalities from the environment. Some others (*e.g.* heater, speaker, projector, *etc.*) act on their environment through the outputs of some of their functionalities.

We also need to express complex queries where the searched (group of) devices, the inputs of the functionalities or their outputs are specified. Doing so makes composition possible (see scenario 1.2.4). Then, the next idea is to provide a way to store a query, or to define devices or composed functionalities as an answer to a query. This will lead us to deal with virtual devices (see scenario 1.2.5) and virtual functionalities (see scenario 1.2.6).

1.3 Sketch of the approach

In this section we will sketch the approach used in this thesis to address the problems defined in section 1.1. We want to design a language to describe the devices and their functionalities so that we can address at the same time the look up problem and the composition problem. In order to search for a devices or a functionality through the network, if we only need to specify the type and/or some of its properties such as the location for a device or such as the kind of input or output for a functionality, a description stored in a simple database can be sufficient to answer. However, we do need more than a simple database when we have to find compositions of functionalities. That's why the requirements, mentioned in section 1.2.8, drive us to use a logical description of the devices and their functionalities. More specifically, as we need to make compositions at query-time, the description of the functionalities requires a rich logical formalism, in particular using functions.

Thus, we build a first-order logic based language to describe the resources using a taxonomy of class to constrain the type of the resources. Each device connecting to a device in the network has to provide its description to be able to exchange with others nodes. Yet the language provides some flexibility as we reckon on different taxonomies on the devices. The language is easily translated into the language used by the reasoner (PROLOG).

We specify also the corresponding query language which again can be translated easily into a PROLOG query. Then, we use PROLOG as a reasoner to answer the queries. If the query needs to find compositions, the reasoner eventually finds them with no cost and provides an effective composed functionalities using functions.

As there is no distributed PROLOG solution that can fit our needs, in order to deploy our approach in an opportunistic network, we need to narrow the space of search to a subset of devices of the network that are sufficient to find all solutions of a query. For that purpose, we encode our first-order logic based language into propositional logic, so that we can use SOMEWHERE [20] as a look up service. Once a subset of devices has been retrieved, we get their descriptions and use the reasoner.

1.4 Contributions

1.4.1 A logical class-based language with functions

One of the contributions of this thesis is to set up a first-order logic language to describe the devices, keeping in mind that we want to make queries upon the description by using a reasoner.

The language is class-based. Every object handled is assigned to a class and moreover the classes are organised in taxonomies. The main difference with other semantic Web languages such as Resource Document Framework (RDF) [11] is that we use functions to express the composition between functionalities. The function that link an output of a functionality to its input permits us to deliver a composition as an answer of a query with no additional cost.

1.4.2 A feasability study of inferring compositions on demand using a PROLOG engine

The main point of building a description language based on logic and a query language is to use a reasoner to answer to the queries. Queries will mostly lead to build composition of functionalities on the fly.

We use a PROLOG engine. The nature of our language introduces infinite loops of two kinds that PROLOG itself cannot avoid. One infinite loop is generated by an infinite possible compositions of functionalities, mostly due to the inherent nature of some functionalities. For instance, considering a functionality f taking an input of type A and providing an output of type B; and a functionality g taking a input of type B and providing an output of type A. These two functionalities can be composed over and over. This is avoided by limiting the computation to an upper bound of composition depth. In one hand, doing so prevent the reasoner to give some of the answers but in the other hand it can explore more answers than if it remained inside that loop.

Another infinite loop is generated by the description language that makes recursive rules. For instance, the equivalence between classes or the symmetrical property of some predicate introduce such recursion. XSB [23] is a PROLOG engine that can deal with recursion loop by maintaining a table of local sub-goals already visited during the resolution process [21, 22].

Also, the language may not be decidable. It is half-way between Datalog and PROLOG. Our language have some restriction such as absence of negation and any variable in the conclusion of a rule is guaranteed to appear in a clause of the premise of this rule. However, we allow terms with functions as argument of predicate.

1.4.3 A solution for a decentralised deployment of the approach

Once we dig into a decentralised deployment of our approach, we meet some problems. There is no fully distributed PROLOG that we can use to simply translate the problem in an opportunistic network. Some works [**?**] address the problem of parallel PROLOG, where a program is known by all the computation unit and the resolution is distributed. However, we have the opposite need, where the program is distributed and not fully known by none of the node of the network. The difficulty of making a distributed first-order logic reasoner remains in the fact that the definition of a predicate is not necessarily where the computation is made.

Nevertheless, we tried in this thesis to address the problem. The process can be divided in two steps. First, according to a query, a look up is launched through the network to find a subset of devices that contain relevant descriptions to answer to the query. Then the device which issued the query download those descriptions in order to treat locally the query using the PROLOG engine.

Chapter 2

A logic-based language to describe devices and their functionalities

Contents

2.1 Description of classes, instances and properties . . . 8
2.1.1 Taxonomies . . . 8
2.1.2 Instances . . . 8
2.1.3 Predicates . . . 9
2.2 Description of functionalities . . . 11
2.2.1 The use of functions . . . 12
2.2.2 The signature and the assertions of a functionality . . . 12
2.3 Logical description of a device . . . 14
2.4 Composition of functionalities . . . 16
2.4.1 The relay race composition . . . 16
2.4.2 The assembly line composition . . . 17
2.5 Specification of queries . . . 17
2.5.1 Yes-no queries . . . 19
2.5.2 Wh- queries . . . 19
2.5.3 Composed functionalities as answers to queries . . . 20
2.6 Specification of virtual objects . . . 20
2.6.1 Virtual devices . . . 20
2.6.2 Virtual functionalities . . . 21
2.6.3 Combination of virtual devices and virtual functionalities . . . 22
2.7 Extension to heterogeneous descriptions . . . 22

This chapter is dedicated to defining the formal model of the description language. As we want to describe devices and their functionalities in order to compose them on the fly, we chose to represent them in logic.

Composed functionalities will be built at query time as answers to queries. Moreover, the reasoning is constrained by taxonomies of class for the objects involved in our description.

We will use either the Resource Document Framework Schema (RDFS) notation or the PROLOG notation whenever it is convenient, to express the taxonomies, the predicates, the rules and the functions. In the PROLOG notation, variables are denoted by words beginning with an upper-case letter. Each time

one of the two notations will be introduced, the equivalence in the other notation will be provided. To simplify the description for the end-user, we provide a high-level language whose syntax will be introduced throughout the document. We introduce the high-level language to simplify the understanding.

2.1 Description of classes, instances and properties

The language handles four different kinds of objects: devices, functionalities, data, and places. In order to constrain the reasoning, we associate every objects with a class. That is what we shall call 'typing'. Class names are constants of the language. Moreover, a taxonomy of the classes is provided.

2.1.1 Taxonomies

The taxonomy is defined using the triple notation of RDFS where

$$a \quad \texttt{rdf:subClassOf} \quad b$$

expresses that the constant a denotes a class which is a subclass of the class denoted by the constant b.

The semantics of such a RDFS triple is first-order logic and corresponds to the following formula used to shorten the RDFS notation of the corresponding predicate:

$$subclassof(b, a)$$

The high-level language defines taxonomies by giving the subclass relation between classes. For instance, in order to express that *computer* and *printer* are subclasses of *device*, we write:

$$\begin{array}{ll} (\textbf{define} & (\textit{device} :: \textit{computer}) \\ & (\textit{device} :: \textit{printer}) \ ) \end{array}$$

As a syntactic sugar we can express successive subclass relations in an easy way in the high-level language. For instance, in order to express that *city* is a subclass of *country*, which is a subclass of *place*, we write:

$$(\textbf{define} \qquad (\textit{place} :: \textit{country} :: \textit{city}) \ )$$

Examples of taxonomies

A sketch of a taxonomy used in our language can be seen in table 2.1, and its graphical representation on figure 2.1. The classes *device*, *data*, *funct*, *place*and *top* are provided by the language. If there is a need to add a class name, one can define it.

2.1.2 Instances

We declare instances of the previously introduced classes using the following notations in RDFS:

$$e \quad \texttt{rdf:type} \quad t$$

expresses that the constant e denotes an instance of the class denoted by the constant t; the semantics in first-order logic is the following formula:

$$type(\texttt{e}, t)$$

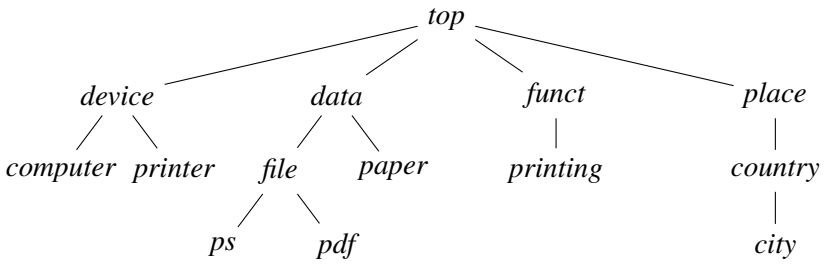


Figure 2.1: Graphical representation of the taxonomy shown in table 2.1

The inheritance is also expressed for any instance by a generic rule

$$\forall \mathrm{X}, \mathrm{C}, \mathrm{D} \qquad \mathit{type}(\mathrm{X}, \mathrm{C}) \wedge \mathit{subclassof}(\mathrm{C}, \mathrm{D}) \rightarrow \mathit{type}(\mathrm{X}, \mathrm{D})$$

A class name followed by a comma-separated list of its instances is the syntax used in the high-level language to define instances. There is a special treatment for the subclasses of the class *funct* because we link a device to a functionality (see section 2.2.1). Below, here is the declaration of some instances in the high-level language.

```
(define (country  france, japan)
        (building  ensimag)
        (room  roomD101, roomD302)
        (ps  myps)
        (pdf  apdf)
        (computer  mypc, yourpc)
        (printer  prn) )
```

2.1.3 Predicates

In our language, the predicates are used to define properties and valued attributes for objects.

In RDFS notation,

$$p \;\; \texttt{rdfs:domain} \;\; a$$

resp.

$$p \;\; \texttt{rdfs:range} \;\; b$$

Table 2.1: Definition of a taxonomy with the syntax of first-order logic

subclassof (*device*, *top*)	*subclassof* (*file*, *data*)
subclassof (*data*, *top*)	*subclassof* (*ps*, *file*)
subclassof (*funct*, *top*)	*subclassof* (*pdf*, *file*)
subclassof (*place*, *top*)	*subclassof* (*paper*, *data*)
subclassof (*computer*, *device*)	*subclassof* (*country*, *place*)
subclassof (*printer*, *device*)	*subclassof* (*city*, *country*)
subclassof (*printing*, *funct*)	

expresses that the domain (resp. range) of the predicate named p is the class denoted by the constant a (resp. by the constant b). The semantics in first-order logic is the following formula:

$$\forall \mathrm{X}, \mathrm{Y} \qquad p(\mathrm{X}, \mathrm{Y}) \rightarrow \mathit{type}(\mathrm{X}, a)$$

resp.

$$\forall \mathrm{X}, \mathrm{Y} \qquad p(\mathrm{X}, \mathrm{Y}) \rightarrow \mathit{type}(\mathrm{Y}, b)$$

Here is a list of predicates provided in our language. This list is not exhaustive and according to his purpose, a user can define more predicates if their domain and their range are provided. All the predicates shown here are binary but this is not mandatory.

- *connected* takes two devices as arguments and holds if and only if the former is connected to the latter. $\mathit{connected}(\mathtt{a}, \mathtt{b})$ holds when the device `a` can start a communication with `b`. Once the communication is established, `a` and `b` can exchange through the channel. But, it is possible for `b` not to have any means to start a communication with `a`. This predicate is not symmetrical.

  `connected` `rdfs:domain` *device*
  `connected` `rdfs:range` *device*

  If the connection is assumed once for all, this predicate can be declared. Otherwise, if the connection is temporary, we have to make a check on this predicate at query time, to make sure that it mirrors the real connection.

- *stored* takes two arguments, δ a data object and `d` a device. $\mathit{stored}(\delta,\ \mathtt{d})$ holds if and only if δ is stored in `d`, for instance to express that a file is stored in a certain computer.

  `stored` `rdfs:domain` *data*
  `stored` `rdfs:range` *device*

  This predicate can be declared or infered as a post-condition of a functionality.

- *hasfunct* holds if and only if the device given as the first argument has the functionality given as the second argument. It is an explicit link between a device and its functionality. If there is a generic way to define the association of a class of functionality with a class of device, this predicate can be infered in a rule — for instance, to declare that every printer has the functionality of printing. But, this predicate can also be declared for specific instances of devices and functionalities.

  `hasfunct` `rdfs:domain` *device*
  `hasfunct` `rdfs:range` *funct*

Table 2.2: The RDFS notation and the syntax of first-order logic

RDFS notation	Syntax and semantics of first-order logic
b `rdf:subClassOf` a	$\mathit{subclassof}(b, a)$
e `rdf:type` t	$\mathit{type}(\mathtt{e}, t)$
p `rdfs:domain` a	$\forall \mathrm{X}, \mathrm{Y} \quad p(\mathrm{X}, \mathrm{Y}) \rightarrow \mathit{type}(\mathrm{X}, a)$
p `rdfs:range` b	$\forall \mathrm{X}, \mathrm{Y} \quad p(\mathrm{X}, \mathrm{Y}) \rightarrow \mathit{type}(\mathrm{Y}, b)$

- $same$ holds if and only if the two data given are identical — for example, a file stored in a computer and a copy of this file stored elsewhere.

  `same` `rdfs:domain` *data*
  `same` `rdfs:range` *data*

  This predicate comes also with some generic rules that show its reflexiveness and its transitivity:

$$\forall \mathrm{I}, \quad same(\mathrm{I}, \mathrm{I})$$
$$\forall \mathrm{I}, \mathrm{J}, \mathrm{K}, \quad same(\mathrm{I}, \mathrm{J}) \wedge same(\mathrm{J}, \mathrm{K}) \rightarrow same(\mathrm{I}, \mathrm{K})$$

- $located$ associates a device with its location. It holds if and only if the device given as the first argument is located in a place denoted by the second argument. The place should be a constant name, an instance of a subclass of *place*.

  `located` `rdfs:domain` *device*
  `located` `rdfs:range` *place*

  This predicate can be declared, or in the case of mobile devices it can be modified.

- *inside* expresses that a location is inside another.

  `inside` `rdfs:domain` *place*
  `inside` `rdfs:range` *place*

  This predicate comes also with a rule. If a device D is located in a place A and if A is assumed to be inside a place B, then the device D is also located in B. This is expressed by the following rule:

$$\forall \mathrm{D}, \mathrm{L}, \mathrm{M} \quad location(\mathrm{D}, \mathrm{L}) \wedge inside(\mathrm{L}, \mathrm{M}) \rightarrow location(\mathrm{D}, \mathrm{M})$$

For any predicate defined above, the properties can be expressed in the high-level language. For instance to declare that `myps`, an instance of the class *ps*, is stored in `mypc`, an instance of the class *computer*, we can write:

(**define** (`myps` <u>stored</u> `mypc`))

2.2 Description of functionalities

First of all, we have to describe the functionalities of each subclass of *funct*. All the functionalities of a certain class share the same description. Then, we have to associate devices with their functionalities. The description of a functionality consists in typing the inputs and the output, expressing preconditions and post-conditions. In order to make this description, we have to use some functions that we will first introduce. Then, we will present the description in detail.

2.2.1 The use of functions

Let us define two functions used in both the declaration and the description of functionalities. Functions are not specified in RDF, so we shall use only the Prolog notation.

This function is called out in reference to 'output'. It expresses the output of a function F fed by a list of inputs $\vec{I}$. As constructor of a new object, it allows to express properties associated with the output, remembering the functionality and the inputs. This function makes the expression of composition possible (see section 2.4).

Those functions (with indices), applied to a device, provide one of the functionalities of that device. This notation associates a device with one of its functionalities.

There are two cases. Let *a* be a subclass of *device* and *b* be a subclass of *funct*. If all instances of *a* have a functionality, which is an instance of *b*, then we write:

$$\forall \mathrm{D}, \qquad \mathit{type}(\mathrm{D}, a) \;\rightarrow\; \mathit{type}(f_1(\mathrm{D}), b) \;\wedge\; \mathit{hasfunct}(\mathrm{D}, f_1(\mathrm{D}))$$

Otherwise, if a device, which is an instance of class *a* and which does not have a functionality instance of class *b*, exists, then we cannot write such a rule. Thus, for each particular device which is an instance of class *a* and which does have a functionality instance of class *b*, we have to declare explicitly this association by writing:

$$\mathit{type}(f_1(d), b) \;\wedge\; \mathit{hasfunct}(d, f_1(d))$$

The choice of the name of those functions can be left to the care of a naming system. Normally, the end-user can completely ignore the existence of these functions and that is why they are not present in the high-level language.

In the high-level language, to express, for example, that every printer has the printing functionality, we write:

$$(\mathbf{define} \quad (\mathbf{every} \quad \mathit{printer} \quad \mathbf{has} \quad \mathit{printing})\,)$$

In the second case, to express for instance that only `mypc` has the *upload* functionality, we write:

$$(\mathbf{define} \quad (\texttt{mypc} \quad \mathbf{has} \quad \mathit{upload})\,)$$

After having typed the data and the devices, we have also typed the functionalities and we have bound them to devices. Then, as we keep in mind that we want to compose functionalities on the fly whenever it is possible, we will describe in the following section the specification of the functionalities that will enable us to compose them.

2.2.2 The signature and the assertions of a functionality

As explained earlier in this chapter, there are two kinds of constraint to compose functionalities. Thus, while describing a functionality we have to describe these constraints.

Definition 1 : The signature of a functionality is the class of each of its inputs and the class of its output. The signature is mandatory to describe a functionality. A functionality may not have any inputs but it always has a single output. □

Let us consider a functionality F, and let us assume that it provides an output of class T. In order to feed this output as an input of a functionality G, T must also be a class of that input: this is the typing

$$
\begin{array}{lll}
r: & & \mathit{type}(F,T) & \text{Type the functionality} \\
 & \wedge & \mathit{hasfunct}(D,F) & \text{Specify the device} \\
 & \wedge & \bigwedge_{k=1}^{n} \mathit{type}(I_k,T_k) & \text{Type the inputs} \\
 & \wedge & \mathit{Prec}(\vec{I},D) & \text{Define preconditions} \\
 & \rightarrow & \mathit{type}(\mathit{out}(F,[\vec{I}]),T_o) & \text{Type the output} \\
 & \wedge & \mathit{Post}(\vec{I},\mathit{out}(F,[\vec{I}]),D) & \text{Define post-conditions}
\end{array}
$$

where

- T is the type of the described functionality;
- D is the device that has the functionality;
- n is the number of inputs for this class T of functionalities;
- I_k is one of the inputs;
- out is a function representing the output of a functionality F on its inputs $[\vec{I}]$;
- $\vec{I}$ represents $I_1, \ldots, I_n$;
- $Prec$ is a conjunction of preconditions where the inputs and the device D are involved;
- $Post$ is a conjunction of post-conditions where the inputs, the output and the device D are involved;

Table 2.3: Description of a functionality in the general case

constraint. That explains why so far we needed to type the objects handled, and why now we constrain the class of the inputs and the class of the output. A functionality that provides only output corresponds to services that provides only output called by [4] 'Data-Providing' services.

Definition 2 : The assertions of a functionality are all the preconditions required and all the post-conditions claimed to be true once the functionality is completed. Neither the preconditions nor the post-conditions are mandatory. □

The device which performs a functionality and the devices from which the functionality possibly get its input can be involved in the preconditions and the post-conditions of the functionality. A functionality that has post-conditions is equivalent to Web services that changes the state of the system called by [4] 'Effect-Providing' services.

Thus, the rule that describe a functionality links its output with its inputs, provides the class of the output, and then expresses the preconditions (if there is any) about the inputs and the device (connection, location, ...) and the post-conditions (if any) about the output. The description in the general case written in logic is showntable 2.3.

Examples: The example shown in table 2.4 defines the uploading functionality. The class of the output is the same class *T* as the first input and it has to be a subclass of *file*. The class of the second input is *computer*, it represents the server where the file should be uploaded. The functionality needs two preconditions: the file input must be stored in the device performing the functionality, and the device performing

(**define** (*upload Fu*	$type(\text{Fu}, upload)$
(*file* :: *T* I_1 **as** $Fu.In$)	$\wedge\ type(I_1, T)\ \wedge\ subclassof(T, file)$
(*computer* Srv **as** $Fu.In$)	$\wedge\ type(Srv, computer)$
(I_1 <u>stored</u> $Fu.Dev$)	$\wedge\ hasfunct(D, Fu)\ \wedge\ stored(I_1, D)$
($Fu.Dev$ <u>connected</u> Srv)	$\wedge\ connected(D, Srv)$
$\Rightarrow$	$\rightarrow$
(T O **as** $Fu.Out$)	$type(\text{out}\ (Fu,\ [I_1, Srv]), T)$
(O <u>stored</u> Srv)	$\wedge\ stored(\text{out}\ (Fu,\ [I_1, Srv]), Srv)$
(O <u>same</u> I_1)	$\wedge\ same(\text{out}\ (Fu,\ [I_1, Srv]), I_1)$
)	
)	

Table 2.4: Example of the description of a functionality in high-level language and first-order logic

the functionality must be connected to the target computer. There are also two post-conditions for this functionality. Once the upload is done, the output is a file stored in the target computer, and the output file is a copy of the input file (expressed by the predicate <u>same</u>). Moreover, in the high-level language, the list of inputs is ordered by the order in which the inputs appear in the description.

The example in table 2.5 is the description of a printing functionality. The class of the output is *paper*; the class of the described functionality is *printing*, the class of the input should be *ps*, and the connection between the device providing the *ps* file as an input and the device actually printing is required as a precondition. There is no post-condition in this particular example. First we show the decription written in the high-level language and then its semantics in logic.
The notation $Fp.Dev$ refers to the device which has the functionality named Fp (Fp is the local name of the functionality described here). The same way, the notation $Fun.Dev$ (resp. $Fun.Out$) refers to the device which has the functionality Fun (resp. the output of the functionality Fun). Then, the line

$$(I_1 \quad \textbf{is} \quad Fun.Out)$$

leads to replace I_1 with the expression $\text{out}\ (Fun,\ X)$. The fact that $Fun.Dev$ appears in a precondition leads to replace it with a new variable $FunD$ and to constrain it by adding $hasfunct(FunD,\ Fun)$. This way, $Fun.Dev$ alias $FunD$ is the device which has the functionality Fun.

2.3 Logical description of a device

A device d stores its description: a set of facts and rules in first-order logic. It contains:

- a set of facts representing:

Table 2.5: Another example of the description of a functionality

(a) Description in high-level language	(b) Description in first-order logic
(**define** (*printing* *Fp*	$type(Fp,\ printing)$
(*ps* I_1 **as** $Fp.In$)	$\wedge\ type(\text{out}\ (Fun,\ X),\ ps)$
(I_1 **is** $Fun.Out$)	$\wedge\ hasfunct(D, Fp)$
($Fun.Dev$ <u>connected</u> $Fp.Dev$)	$\wedge\ hasfunct(FunD, Fun)$
$\Rightarrow$ (*paper* O **as** $Fp.Out$))	$\wedge\ connected(FunD, D)$
	$\rightarrow\ type(\text{out}\ (Fp,\ [\text{out}\ (Fun,\ X)]),\ paper)$

Table 2.6: Rewriting of the high-level language in first-order logic

High-level language	First-order logic
(**define** $(\alpha :: \beta)$)	$subclassof(\beta, \alpha)$
(**define** $(\alpha :: \beta :: \gamma :: \delta)$)	$subclassof(\beta, \alpha)$ $subclassof(\gamma, \beta)$ $subclassof(\delta, \gamma)$
(**define** $(\tau \quad \mathtt{e}_1, \mathtt{e}_2, \ldots)$)	$type(e_1, \tau)$ $type(e_2, \tau)$ $\vdots$
(**define** $(\mathtt{e}_1 \quad \underline{\mathtt{p}} \quad \mathtt{e}_2)$)	$p(\mathtt{e}_1, \mathtt{e}_2)$
(**define** (**every** δ **has** ϕ))	$type(D, \delta) \rightarrow type(f_k(D), \phi)$ $\wedge\ hasfunct(D, f_k(D))$
(**define** $(\mathtt{d}$ **has** $\phi)$)	$type(f_k(d), \phi) \ \wedge\ hasfunct(d, f_k(d))$
(**define** $(\phi \quad F$ $(\tau_1 \quad I_1$ **as** $F.In$) $\vdots$ $(\tau_n \quad I_n$ **as** $F.In$) $(Prec_1)(\ldots)$ $\Rightarrow$ $(\tau_o \quad O$ **as** $F.Out$) $(Post_1)(\ldots)$))	$type(F, \phi) \quad \wedge \quad hasfunct(D, F)$ $\wedge \quad \bigwedge_{k=1}^{n} type(I_k, \tau_k)$ $\wedge \quad Prec(\vec{I}, \quad D)$ $\rightarrow$ $type(\text{out}\left(F,\ [\vec{I}]\right), \quad \tau_o)$ $\wedge \quad Post(\vec{I}, \quad \text{out}\left(F,\ [\vec{I}]\right), \quad D)$

- $\alpha,\ \beta, \ldots$ are names of class;
- **keyword** represents keywords of the high-level language;
- a, b, ... are instances;
- A, B, ... stands for variables;
- pred represents predicates in the high-level language;
- $Prec_1$ and $Post_1$ stands for preconditions and post-conditions respectively.

- $instanceof$ relations in the taxonomies: $type(d,\ c)$;
- $subclass$ relations in the taxonomies: $subclassof(c_1,\ c_2)$;
- the assertions of known properties about the device:
  * $type(d,\ c)$;
  * $connected(d,\ d')$;
  * $located(d,\ l)$, $type(l,\ c_l)$;
  * $inside(l_1,\ l_2)$;
  * $stored(o,\ d)$, $type(o,\ c_o)$;
  * $hasfunct(d,\ f(d))$, $type(f(d),\ c_f)$; where f is a skolem function denoting one of the functionalities of the device d.

- a set of rules $type(d,\ c_d) \rightarrow hasfunt(d,\ f(d)), type(f(d),\ c_f)$, to assert that every device of type c_d has a functionality of type c_f
- a set of rules R describing the functionalities of the device, where a rule r has the form shown in table 2.3;
- a set of rules expressing the inheritance of types. It can also include transitivity, symmetry, reflexivity of some properties.

Finally, for a device d, $\mathrm{Descr}\,(d)$ denotes the union of facts and rules mentioned above that describe the device. By extension, for a set D of devices:

$$\mathrm{Descr}\,(D) = \bigcup_{d \in D} \mathrm{Descr}\,(d)$$

2.4 Composition of functionalities

Composing two functionalities F and G consists either in using the output of F as an input of G, or in performing F so that its effects satisfy the preconditions of G.

2.4.1 The relay race composition

Definition 3 : The relay race composition[1] is the composition of two or more functionalities made by matching up their signatures. Before completing the relay race composition of two functionalities F and G, we have to check whether the type of the output of F and the type of the input of G match. □

Let us consider the relay race composition between two functionalities F and G where the k-th input of G is provided by F. Thus, we can express this composition by writing the output of G this way

$$\mathrm{out}\,(G,\ [I_1, \ldots, I_{k-1}, \mathrm{out}\,(F,\ X)\,, I_{k+1}, \ldots])$$

This expression permits us to read directly the structure of the relay race composition and leads us to define the ordered tree of composition of an output.

Definition 4 : An ordered tree T is recursively defined by its root r and the ordered list $[s_1, \ldots, s_n]$ of the sub-trees that are sons of r. We write

$$T = (r, [s_1, \ldots, s_n])$$

[1] named after the relay race where the runners pass one another the baton

If r is a leaf, then

$$T = (r, [\,])$$

□

Definition 5 : The ordered tree of composition T_Ω of an output Ω is an ordered tree that reflects the structure of Ω:

- If $\Omega = c$, where c is a constant of the language, then

$$T_\Omega = (c, [\,])$$

- If $\Omega = \mathrm{out}\,(f,\ [X_1, \ldots, X_n])$, where f is a functionality and each X_i is either an expression of an output or a constant of the language, then

$$T_\Omega = (f, [T_{X_1}, \ldots, T_{X_n}])$$

  where T_{X_i} is the ordered tree of composition of the output X_i.

□

Then, we can define the depth of a relay race composition.

Definition 6 : The depth of a relay race composition is the height of its ordered tree of composition. □

An example of outputs and their ordered trees of composition can be seen on figure 2.2. The depth of the output on the right hand side is 2, and the depth of the output on the left hand side is 3.

2.4.2 The assembly line composition

Definition 7 : The assembly line composition[2] is the composition of two or more functionalities made by matching up their assertions. Before completing the assembly line composition of two functionalities F and G, we have to check whether the preconditions of G can be fulfilled by the post-conditions of F. □

Note that the expression of an output with the function out does not permit to express the assembly line composition. The way to express an assembly line composition will be seen in section **??**.

2.5 Specification of queries

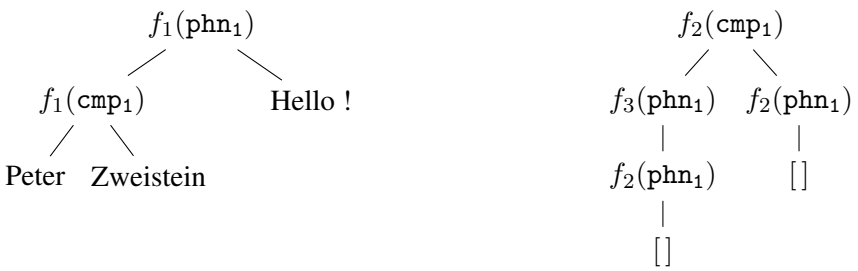
Here are two kinds of queries and their translation in the high-level language and its semantics in logic. For each of them, a detailed answer and an interpretation is given.

Definition 8 : A query Q is defined by a set of rules (denoted by $\mathrm{Def}\,(Q)$) of the form:

[2] named after the moving assembly line model of production developed by Henry Ford

$$\begin{array}{l}
\mathrm{out}(f_1(\mathtt{phn}_1), [\\
\quad \mathrm{out}(f_1(\mathtt{cmp}_1), [\text{"Peter"}, \text{"Zweistein"}]), \\
\quad \text{"Hello!"} \\
])
\end{array}
\qquad
\begin{array}{l}
\mathrm{out}(f_2(\mathtt{cmp}_1), [\\
\quad \mathrm{out}(f_3(\mathtt{phn}_1), [\\
\quad\quad \mathrm{out}(f_2(\mathtt{phn}_1), [])) \\
\quad]), \\
\quad \mathrm{out}(f_2(\mathtt{phn}_1), []) \\
])
\end{array}$$

(a) Expressions of outputs



(b) Ordered trees of composition

$\mathtt{phn}_1$: a phone ; $\mathtt{cmp}_1$: a computer

$f_1(\mathtt{phn}_1)$	send a SMS	1: a phone number 2: the message to send
$f_1(\mathtt{cmp}_1)$	find a phone number	1: a first name 2: a last name
$f_3(\mathtt{phn}_1)$	list of price of carriers	
$f_2(\mathtt{phn}_1)$	list of carriers	
$f_2(\mathtt{cmp}_1)$	return the key of the max	1: a list of value to sort
		2: a list of keys

(c) Semantic of each functionality and device

Figure 2.2: Two outputs and their ordered trees of composition

Def (Q): $$\bigwedge_{i=1}^{n} R_i(t_i^1, t_i^2) \rightarrow Q(X_0, \ldots, X_p)$$

where t_i^j are terms of the language or variables, R_i are predicates and for all k in $[\![1 ; p]\!]$ it exists i in $[\![1 ; n]\!]$ and j in $\{1, 2\}$ so that X_k appears in t_i^j. Afterwards $X_0, \ldots, X_p$ will be denoted by $\bar{v}$ and called *variables of interest* . In the high-level language, a variable of interest begins with a question mark. □

Definition 9 : Let D be a set of devices. The answers to the query Q against the union of description $\mathrm{Descr}\,(D)$ — denoted by $\mathrm{Answer}\,(Q,\ D)$ — is the set of tuples $\bar{t}$ on Herbrand universe of $\mathrm{Descr}\,(D)$ instantiated terms for which $Q(\bar{t})$ can be logically entailed from the devices descriptions and the definition of the query.

$$\mathrm{Answer}\,(Q,\ D) = \{\bar{t} \in H(\mathrm{Descr}\,(D))^p \quad / \quad \mathrm{Descr}\,(D)\,, \mathrm{Def}\,(Q) \models Q(\bar{t})\}$$

□

2.5.1 Yes-no queries

A yes-no query is a query without variable of interest. An answer to this kind of queries is either 'yes' or 'no'. It corresponds to the yes-no questions in English. For example:

'Is the text file `mytxt` stored in the computer `mycmp` ?'

In the high-level language, we write:

$$(\textbf{query} \quad \texttt{q1} \quad (\texttt{mytxt} \quad \underline{\text{stored}} \quad \texttt{mycmp})\,)$$

which is in logic:

$$\mathit{stored}(\texttt{mytxt}, \texttt{mycmp}) \rightarrow q_1$$

Another example:

'Is there any printer in the room `r` ?'

In the high-level language, we write:

$$\begin{array}{l}(\textbf{query} \quad \texttt{q2} \quad (\mathit{printer} \quad P) \\ \qquad\qquad\qquad\quad (P \quad \underline{\text{location}} \quad \texttt{r})\,)\end{array}$$

which is in logic:

$$\mathit{type}(\mathrm{P}, \mathit{printer}) \ \wedge \ \mathit{location}(\mathrm{P}, \texttt{r}) \rightarrow q_2$$

The second example contains a variable but it is not a variable of interest. So, the answer of q_2 will be 'yes' or 'no' rather than an instantiation of the variable P.

2.5.2 Wh- queries

A wh- query is a query with variables of interest. An answer to this kind of queries is an instantiation of the variables of interest so that the formula of the query holds. It corresponds to the wh- questions in English. For example:

'Which device with a printing functionality is `mypc` connected to ?'

In the high-level language, we write:

$$\begin{array}{l}(\textbf{query} \quad \texttt{q3} \quad (\texttt{mypc} \quad \underline{\text{connected}} \quad ?F.Dev) \\ \qquad\qquad\qquad\quad (\mathit{printing} \quad F\,))\end{array}$$

which corresponds, in logic, to:

$$\mathit{connected}(\texttt{mypc}, D) \ \wedge \ \mathit{hasfunct}(D, F) \ \wedge \ \mathit{type}(F, \mathit{printing}) \rightarrow q_3(D)$$

2.5.3 Composed functionalities as answers to queries

Wh- queries can also retrieve composed functionalities. In our approach, the composition of functionalities is encoded in the expression of the output with the function out. So, to fetch a possible composition, it is sufficient to make a query where a variable of interest is the output of a functionality. For instance, if we want to print a file, we will look for a (possibly composed) functionality which gets a file and provides a printed paper, under the constraint that the file should be a copy of the file we want to print. Moreover, we need the printer to be located in a particular room `r`. The word 'printer' should actually be read 'the device which performs the searched functionality'. In the high-level language we will write:

$$
\begin{array}{lll}
(\textbf{query} & \texttt{q4} & (Fclass \quad F) \\
 & & (\textit{file} \quad I_1 \quad \textbf{as} \quad F.In\ ) \\
 & & (I_1 \quad \underline{\text{same}} \quad \texttt{myps}) \\
 & & (F.Dev \quad \underline{\text{location}} \quad \texttt{r}) \\
 & & (\textit{paper} \quad ?Output \quad \textbf{as} \quad F.Out\ )\ )
\end{array}
$$

which is in logic:

$$
\begin{array}{l}
\textit{type}(F,\ Fclass\ ) \\
\wedge\ \textit{type}(I_1,\ \textit{file}\ ) \\
\wedge\ \textit{same}(I_1,\ \texttt{myps}\ ) \\
\wedge\ \textit{hasfunct}(Dev,\ F)\ \wedge\ \textit{location}(Dev,\ \texttt{r}\ ) \\
\wedge\ \textit{type}(\text{out}\ (F,\ [I_1])\ ,\ \textit{paper}\ ) \\
\rightarrow \quad q_4(\text{out}\ (F,\ [I_1])\ )
\end{array}
$$

An answer to such queries is an instance of the variables of interest. Considering the form of the variable of interest, the form of the answer will be out $(F,\ [\ldots])$. Thus, the structure of the composition can be read in the answer.

As we have described the devices and their functionalities, we have just been able to make queries on this knowledge. We will now introduce views to define virtual objects.

2.6 Specification of virtual objects

The idea of virtual objects has already been introduced in both scenario 1.2.5 and scenario 1.2.6. The former brings the concept of virtual devices while the latter brings the concept of virtual functionalities. Virtual objects are views. They are written in such a way that it is simple to reuse them as normal objects in other queries.

The advantages of virtual objects are the following ones. Rather than referring to an absolute device or to a composition of functionalities defined beforehand, the query-based definition of the virtual object is evaluated: so it reflects the dynamic environment. Though the evaluation may be different each time, the definition remains the same.

2.6.1 Virtual devices

A virtual device is defined by a query which retrieves a list of devices. Then, all the properties and functionalities have to be redefined. For example, the location of the virtual device has to be set. Sometimes, it is necessary to keep the instantiations of some of the variables which are present in the definition of a virtual device. For that purpose, the name of the virtual device can be a function of those variables.

Let us go back to the scenario 1.2.5. In this scenario, two screens in the same room could possibly work together as a new virtual device to display a wider resolution of a video stream.

In the high-level language, the keyword **view** is used to introduce the virtual device, and to give it a name, a type, and some of its properties. Then, the keyword **as** is used to introduce the specification of the query which will be the definition of the view. Thus, we should write:

$$
\begin{array}{rl}
(\textbf{view} & (\textit{screen} \quad \texttt{bigscreen}\,) \\
 & (\texttt{bigscreen} \quad \underline{\text{location}} \quad X) \\
\textbf{as} & (\textit{screen} \quad ?L,\ ?R\,) \\
 & (\textit{room} \quad X) \\
 & (?L \quad \underline{\text{location}} \quad X) \\
 & (?R \quad \underline{\text{location}} \quad X)
\end{array}
$$

which first-order logic semantics is:

$$
\begin{array}{rl}
 & \textit{type}(L,\ \textit{screen}\,) \ \wedge\ \textit{type}(R,\ \textit{screen}\,) \\
 & \wedge\ \textit{type}(X,\ \textit{room}\,) \\
 & \wedge\ \textit{location}(L, X) \ \wedge\ \textit{location}(R, X) \\
\rightarrow & \textit{type}(\texttt{bigscreen}([L, R]), \textit{screen}\,) \\
 & \wedge\ \textit{location}(\texttt{bigscreen}([L, R]), X)
\end{array}
$$

Suppose that `leftscr` and `rightscr` are two screens in the same room `r`, then according to the previous rules, the class of `bigscreen`([`leftscr`, `rightscr`]) is *screen* and its location is `r`.

2.6.2 Virtual functionalities

Let us go back to the scenario on section 1.2.6. To define the virtual functionality which prints files in PDF format, we write in the high-level language:

$$
\begin{array}{lll}
1 & (\textbf{view} & (\textit{pdfprinting} \quad \texttt{vf} \\
2 & & \quad (\textit{pdf} \quad I \quad \textbf{as} \quad \texttt{vf}.\textit{In}) \\
3 & & \Rightarrow \quad (\textit{paper} \quad O \quad \textbf{as} \quad \texttt{vf}.\textit{Out}) \\
4 & &) \\
5 & \textbf{as} & (I \quad \textbf{is} \quad G.\textit{In}) \\
6 & & (\textit{ps} \quad P_2 \quad \textbf{as} \quad G.\textit{Out}) \\
7 & & (X \quad \underline{\text{same}} \quad P_2) \\
8 & & (\textit{ps} \quad X \quad \textbf{as} \quad F.\textit{In}) \\
9 & & (O \quad \underline{\text{same}} \quad F.\textit{Out}) \\
10 & & (\texttt{vf}.\textit{Dev} \quad \textbf{is} \quad F.\textit{Dev}) \\
11 &) &
\end{array}
$$

Let us use the line numbering to explain step by step what it is said in this definition. In line 1, the class of the virtual functionality is declared. This class name has to be inserted in the taxonomy beforehand. Lines 2 and 3, the signature of the currently described functionality. The signature is mandatory here. Line 5, the keyword **as** introduce the query which defines the functionality.

The input of the virtual functionality should be the input of a functionality G (line 5) whose output is a *ps* file (line 6). Then, the input of the functionality F (line 8) should be the same as the output of G (line 7). This let us the possibility to find a way to transfer the data between G and F in case the devices that perform them are not connected. The output of the virtual functionality will be the same as the

output of F (line 9). Finally the device which has this virtual functionality is the device that performs the last functionality F (line 10).

Thus, here is the semantics of the previous definition, in first-order logic:

$$
\begin{array}{l}
\quad \mathit{type}(\mathrm{I},\ \mathit{pdf}) \\
\quad \wedge\ \mathit{type}(\mathrm{out}\ (G,\ [\mathrm{I}])\ ,\ \mathit{ps}) \\
\quad \wedge\ \mathit{same}(J,\ \mathrm{out}\ (G,\ [\mathrm{I}])) \\
\quad \wedge\ \mathit{type}(J,\ \mathit{ps}) \\
\quad \wedge\ \mathit{type}(\mathrm{out}\ (F,\ [J])\ ,\ \mathit{paper}) \\
\quad \wedge\ \mathit{hasfunct}(D,\ F) \\
\rightarrow \\
\quad \mathit{type}(\mathrm{out}\ (vf_1(D),\ [\mathrm{I}])\ , \mathit{paper}) \\
\quad \wedge\ \mathit{same}(\mathrm{out}\ (vf_1(D),\ [\mathrm{I}])\ , \mathrm{out}\ (F,\ [J])) \\
\quad \wedge\ \mathit{hasfunct}(D, vf_1(D)) \\
\quad \wedge\ \mathit{type}(vf_1(D), \mathit{pdfprinting})
\end{array}
$$

Therefore, it is possible to access the new virtual functionality as a normal one. For any device D which has a functionality providing papers, the virtual functionality described here is associated with D. Then the query $\mathit{same}(\mathrm{out}\ (vf_1(D),\ [I])\ , X)$ instantiate the variable X with the corresponding output which is hidden behind the output of the virtual functionality.

2.6.3 Combination of virtual devices and virtual functionalities

In section 2.6.1 we have introduced virtual devices, using the scenario 1.2.5 where two screens were stitched together to make a composed screen with higher resolution. But in that scenario, there was nothing to split the video stream. So now, we will look for a functionality which performs such an operation and we will package it together with the two screens as the displaying functionality of the virtual device. This way, the virtual device will be associated with a virtual functionality .

2.7 Extension to heterogeneous descriptions

So far we defined the language of the description but each class used in the taxonomy. In the language, this have never been really defined and we implicitly understood that there were a common taxonomy among the devices. Yet, this is a constraint that is usually not met in real world, and even though a device which connects to the network must describe itself in the defined language, it can afford a different taxonomy. The effort of bringing a widely common ontology is usually justified by the will to free oneself from the complexity of the natural language. Nevertheless, we can spare us one official way of thinking by providing flexibility to our language, dealing with heterogeneous descriptions.

We regard heterogeneous descriptions as the fact that two different devices may have used different taxonomies in their description (with other things the same). Now if taxonomies are different, we need a way to link one to the other, that is to say a mapping between the different taxonomies.

To prevent class name to overlap, we suppose that classes name of a taxonomies of a certain device are always prefixed with the identifier of that device. Figure 2.3 shows an example of such a mapping where a class name of the taxonomy of the first device is prefixed with 1 while for the second device, it is prefixed with 2.

We define three kinds of mappings, all illustrated in figure 2.3:

- the inclusion;

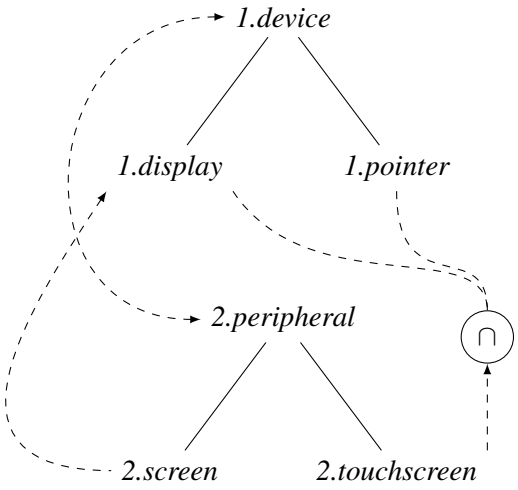


Figure 2.3: Example of mappings between taxonomies

Table 2.7: Rewriting mappings using only inclusion

$(a_1 \cup a_2) \subset b$	$\wedge \begin{array}{c} a_1 \subset b \\ a_2 \subset b \end{array}$
$a \subset (b_1 \cap b_2)$	$\wedge \begin{array}{c} a \subset b_1 \\ a \subset b_2 \end{array}$
$a = b$	$\wedge \begin{array}{c} a \subset b \\ b \subset a \end{array}$

- the inclusion to the intersection of more than one class.
- the equivalence.

Note that, as we don't allow negation in our language, there is no way to express disjointness of classes.

In figure 2.3, the dashed arrows have the following meaning. *1.device* and *2.peripheral* are equivalent and the equivalence is represented by a dashed two-headed arrow. *2.screen* is a *1.display*. It means that any device which type is *2.screen* are considered as *1.display*. This inclusion is represented by a dashed arrow pointing towards *1.device*. A *2.touchscreen* is both a *1.display* and a *1.pointer*. It is represented by a dashed arrow pointing to a cap joining *1.display* and *1.pointer*.

Once a device connects to a network (actually it connects to one or more devices that are already in the network), we assume that it brings is own description and that somehow a mapping is provided. It can be asked to the user or automatically discovered, but this is out of our scope.

Finally, it is possible to express every mapping into inclusion only as shown in table 2.7. Therefore, table 2.8 provides the semantics in first-order logic of the mappings.

Table 2.8: Semantic in first-order logic of the mappings

Mappings between classes	Semantic in first-order logic
$a \subset b$	$type(X, a) \rightarrow type(X, b)$
$(a_1 \cup a_2) \subset b$	$\wedge \begin{array}{l} type(X, a_1) \rightarrow type(X, b) \\ type(X, a_1) \rightarrow type(X, b) \end{array}$
$a \subset (b_1 \cap b_2)$	$\wedge \begin{array}{l} type(X, a) \rightarrow type(X, b_1) \\ type(X, a) \rightarrow type(X, b_2) \end{array}$
$a = b$	$\wedge \begin{array}{l} type(X, a) \rightarrow type(X, b) \\ type(X, b) \rightarrow type(X, a) \end{array}$

Chapter 3

Centralised reasoning for dynamic composition of resources

Contents

3.1 Using PROLOG as a reasoning engine **25**
3.2 Experiments **26**
3.2.1 The basis of the scenarios 26
3.2.2 About the curves 27
3.3 Different scenarios **27**
3.3.1 Scenario almost without type constraint 27
3.3.2 Best case: a single possible composition of depth 2 27
3.3.3 A single possible composition of depth n 29
3.3.4 n possible compositions of depth 2 31
3.3.5 n^2 possible compositions of depth 2 33
3.3.6 $2^{n/2}$ possible compositions of depth $n/2$ 34
3.4 Summary **36**

In this chapter, we assume that it exists a central repository of the descriptions of each devices of the network. The description language and the query language as they have been defined in chapter 2, may be undecidable. The language has some restriction compared to the Horn clause language. We know that satisfiability of definite Horn clause in first-order logic is undecidable. However, in one hand, our language contains no existential quantifiers, in the other hand it uses functions. The question of decidability have not been explored further.

There are reasoner like PROLOG that can answer queries using SLD-resolution methods. Those fit better our needs then some other tableaux-based reasoners or rule-base semantic reasoner for instance.

3.1 Using PROLOG as a reasoning engine

We need a reasoner to infer the possible composition of functionalities according to the description of each device. As seen in chapter 2, the description and the queries languages are translated in logic, so that we can use a reasoner to answer a query. We use PROLOG as a reasoner in order to answer the queries.

PROLOG instantiate the variables of a query so that the query with the substitutions applied is a logical consequence of the description. This is achieved by traversing the search space and looking for solution to the problem. While PROLOG find a possible solution it will dig into the success branch until either the set of sub-goals becomes empty or a failure appears. In the latter case, a backtracking process is launched finding another solution in one of the previous alternative not tried yet. Doing so, it is impossible to avoid infinite loop.

However, XSB [23] uses the concept of tabling [21, 22] to store goal matches and previously partial answers met during the resolution. This is like carrying a transitive closure of the solution to avoid simple loop recursion of recursive loop. But not all loop can be caught this way.

3.2 Experiments

We aim to check whether the process of retrieving answers to queries is scalable for the case where the whole description is centralised and accessible. For that purpose, we will make six variants of a scenario. Those variants are easily generated by the high-level language introduced in chapter 2. All variants will declare the same devices, and use the same functionalities. Moreover, each scenario will be repeated with a variable number of devices. In one of the variants we will almost remove the typing while in the others the typing described previously will be present. Among those five latter scenarios, the graph of connections will be different. As connections between devices are necessary for composition in this case, the graph of connections will determine both the number of the compositions and their depth (see definition 6).

3.2.1 The basis of the scenarios

The scenarios are basically about printing a file stored in a computer. We will repeat each scenario for a variable number of devices. In this chapter, the variable n will stand for that number of devices.

All of the following scenarios are made of:

- n printers $(\mathtt{prn}_i)_{i \in [\![1\,;\,n]\!]}$
- $n+1$ computers
  - `mypc`: the computer where data are stored;
  - $(\mathtt{srv}_i)_{i \in [\![1\,;\,n]\!]}$: computers.

The file `myps` is stored in the computer `mypc`. Every computer can upload any of its files onto a computer to which it is connected. Every printer has a printing functionality.

The query for each scenario is kept identical. In regular english, the query is ‘We want a printed version of the file `myps` (which is stored in the computer `mypc`)’ while in the high-level language it is:

$$
\begin{array}{lll}
(\textbf{query} & \mathtt{q1} & (\textit{funcF}) \\
 & & (\textit{file} \quad I_1 \quad \textbf{as} \quad F.In) \\
 & & (I_1 \quad \underline{\text{same}} \quad \mathtt{myps}) \\
 & & (\textit{paper} \quad ?O \quad \textbf{as} \quad F.Out) \\
) & &
\end{array}
$$

Definition 10 : A oriented graph $G = (V, E)$ is called graph of connections if and only if the following conditions are all met:

- $V \subset \{\ d \quad | \quad type(d, device)\ \}$
- $(d_1, d_2) \in E \quad \Leftrightarrow \quad connected(d_1, d_2)$

In other words a graph of connections is a graph with devices as vertices and where there is an arc from a device d_1 to a device d_2 if and only if the atom $connected(d_1, d_2)$ holds. □

The graph of connections between the devices makes the singularity of each scenario as it changes the number and the depth of the compositions retrieved for the query. That graph will be detailed for each scenario.

3.2.2 About the curves

For each scenario, two curves are drawn. The first one represents the time consumption depending on the number of devices, and the second one represents the memory consumption depending on the number of devices. To find the best polynomial function passing as near as possible to all the measured points, the easiest is to find a straight line using the least squares method.

Thus, drawing a curve with a logarithmic scale for the x-axis and the y-axis (called afterwards logarithmic coordinate system) permits us to compute directly the degree of the nearest polynomial function. A curve with a logarithmic scale for the y-axis only (called afterwards semi logarithmic coordinate system) permits us to show that the curve is exponential.

3.3 Different scenarios

3.3.1 Scenario almost without type constraint

In our model, typing is so fundamental that it is impossible to simply remove the types. However, we can prevent types from being meaningful by providing each object with the same type. Actually, in our approach we still need to be able to distinguish a functionality from a data, from a device, *etc*. Thus, we use the following meaningless types:

- *devnull* for every device;
- *funcnull* for every functionality;
- *locnull* for every location;
- *datanull* for every data.

We use the same graph of connections as for the scenario described in section 3.3.2. Nevertheless, there are many more answers to the query than in the equivalent scenario with types, because some of the compositions retrieved cannot be performed, due to a lack of type constraint. Even worse, some functionality are claimed to be feasible while they are not when there is typing constraint. Except for the scenario 3.3.2, all the others when removing types tend to be exponential in time.

3.3.2 Best case: a single possible composition of depth 2

In this scenario, we have the following connections represented by the graph of connections on figure 3.1:

$$connected(\texttt{mypc}, \texttt{srv}_1)$$
$$connected(\texttt{srv}_1, \texttt{prn}_1)$$

For any number n of devices, there are only two connections. Thus, there is only one possible composition: `mypc` uploads the file `myps` onto the server $\mathtt{srv}_1$ which is in charge to send it to the printer $\mathtt{prn}_1$. But as n grows, the other devices become noises for the search of the solution.

Statistics and curves

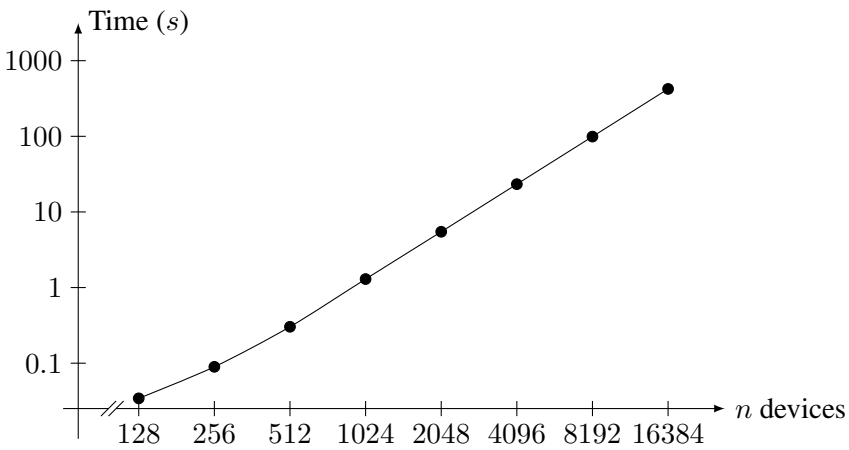
Statistical curves depending on the number of devices are drawn on figure 3.2 in a logarithmic coordinate system. The curve on figure 3.2(a) represents the time consumption. With the least squares method, we find out that time consumption is quadratic with n. The memory consumption due to the XSB table is represented by curve on figure 3.2(b). The memory consumption is linear with n.

Interpretation

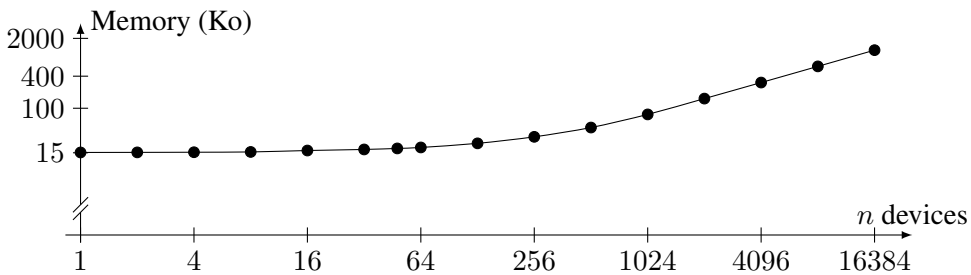
We have to consider the PROLOG engine used to make the experiments in order to analyse the result. We use XSB. Before the evaluation of a query, XSB creates an indexed table of the predicates. Then, during

$$\begin{array}{lll} \mathtt{mypc} \longrightarrow & \mathtt{srv}_1 \longrightarrow & \mathtt{prn}_1 \\ & \mathtt{srv}_i & \mathtt{prn}_i \\ & \mathtt{srv}_n & \mathtt{prn}_n \end{array}$$

Figure 3.1: Schema of connections in the scenario 3.3.2



(a) Time consumption



(b) Memory usage

Figure 3.2: Scenario with a single possible composition of depth 2

Number of devices	Time (s)	Time for table	Memory (Ko)
128	0.04	0.03	22
256	0.10	0.12	30
512	0.32	0.26	44
1024	1.28	1	77
2048	5.06	2.3	150
4096	20.2	16.1	298
8192	80.4	70	593
16384	321.7	300	1183

Table 3.1: Data of the scenario with a single possible composition of depth 2

the evaluation, XSB can access the table to prove a sub-goal. In this experiment, we have to remove the time taken to build up the XSB table as it is done only once. That is why in table 3.1 there is a column showing the time taken to build the XSB table.

The results of the query $type(X, Y)$ are stored in the XSB table. They represent a portion of size n of the table, since there are as many lines as devices. The results of the queries $same(X, Y)$ and $stored(X, Y)$ are also in the XSB table and their size is constant because they appear only for the possible connection.

Now, to find all the answers to the query, we have to check every printer (n) and for everyone, we have to look up in the table of size n. This takes a linear time. We can conclude that the time consumption is quadratic (n^2).

3.3.3 A single possible composition of depth n

In this scenario, we have the following connections represented by the graph of connections on figure 3.3:

$$\forall i \in [\![1; n-1]\!] \qquad \begin{array}{c} connected(\texttt{mypc}, \texttt{srv}_1) \\ connected(\texttt{srv}_i, \texttt{srv}_{i+1}) \\ connected(\texttt{srv}_n, \texttt{prn}_n) \end{array}$$

There are $n + 1$ connections, so that there is only one possibility to print, through the whole chain of servers, on the printer $\texttt{prn}_n$.

Statistics and curves

Statistical curves depending on the number of devices are drawn on figure 3.4 in a logarithmic coordinate system. The curve on figure 3.4(a) represents the time consumption. It appears to be quartic with n. The

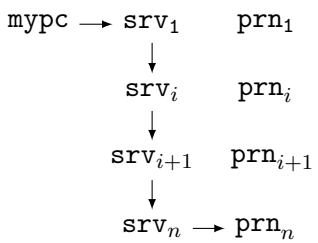


Figure 3.3: Schema of connections in the scenario 3.3.3

curve on figure 3.4(b) represents the memory consumption due to the XSB table. The memory space used is cubic with n.

Interpretation

The results of the query $same(X, Y)$ are stored in the XSB table. There are all the following answers:

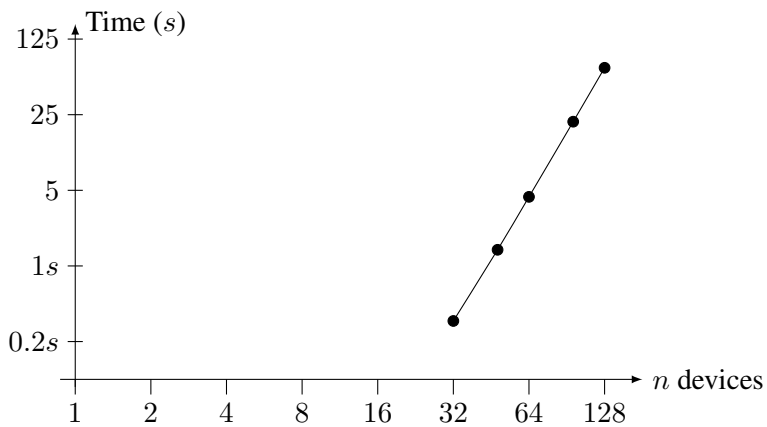
$$\forall i, j \in [\![1 \,;\, n]\!] \qquad \begin{array}{l} same(\underbrace{\text{out}\,(f(srv_{i-1}),\ [\text{out}\,(f(srv_{i-2}),\ [..., srv_{i-1}])\,, srv_i])}_{\text{depth } i}, \\ \underbrace{\text{out}\,(f(srv_{j-1}),\ [\text{out}\,(f(srv_{j-2}),\ [..., srv_{j-1}])\,, srv_j])}_{\text{depth } j}) \end{array}$$

For any i and j between 1 and n, there is a line in the XSB table; and the size of this line is $i + j$. Thus, this portion of the table represents a portion of size

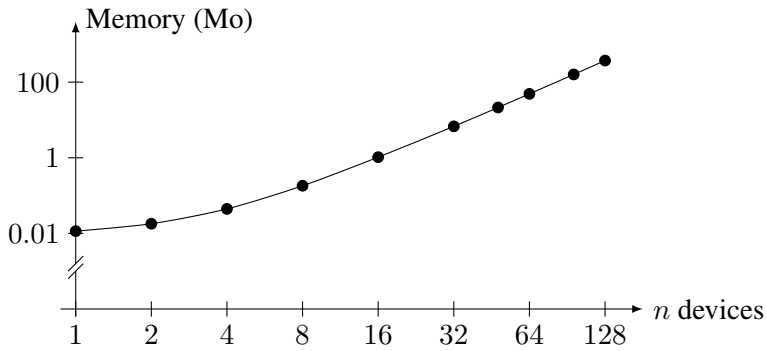
$$\sum_{i=1}^{n} \sum_{j=1}^{n} i + j \sim n^3$$

of the XSB table.

There is also the results of the query $type(X, Y)$ which still represents a portion of size n; and the results of the query $stored(X, Y)$, which are:



(a) Time consumption



(b) Memory usage

Figure 3.4: Scenario with a single possible composition of depth n

Number of devices	Time (s)	Memory (Ko)
1	0.01	11
2	0.01	18
4	0.01	44
8	0.01	181
16	0.02	1 Mo
32	0.31	6 Mo
64	4.3	48 Mo
128	68	371 Mo

Table 3.2: Data of the scenario with a single possible composition of depth n

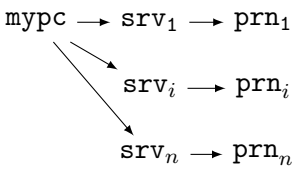


Figure 3.5: Schema of connections in the scenario 3.3.4

$$\forall i \in [\![1\,;\, n]\!] \qquad stored(\underbrace{\text{out}\,(f(srv_{i-1}),\ [\text{out}\,(f(srv_{i-2}),\ [...,srv_{i-1}])\,,srv_i])}_{\text{depth } i}, \texttt{srv}_i)$$

They represents a portion of size n^2 of the XSB table.

Thus, the global size of the table is n^3.

Now, to find all the answers to the query, we have to check every server (n) of the chain and for everyone, we have to look up in the table of size n^3. Thus, the time consumption is quartic.

3.3.4 n possible compositions of depth 2

In this scenario, we have the following connections represented by the graph of connections on figure 3.5:

$$\forall i \in [\![1; n]\!] \qquad \begin{cases} connected(\texttt{mypc}, \texttt{srv}_i) \\ connected(\texttt{srv}_i, \texttt{prn}_i) \end{cases}$$

There are $2n$ connections, so that there are n possibilities to print the file `myps`. For any i, uploading to the server i and then printing on the printer i is a possible answer to the query.

Statistics and curves

Statistical curves depending on the number of devices are drawn on figure 3.6 in a logarithmic coordinate system. The time consumption is represented on the curve on figure 3.6(a). It appears to be cubic with n. The curve on figure 3.6(b) represents the memory consumption due to the XSB table. The memory space used is quadratic with n.

Interpretation

The results of the query $same(X, Y)$ are stored in the XSB table and there are all the following answers:

Number of devices	Time (s)	Memory (Ko)
8	0.01	58
16	0.01	162
32	0.04	492 Ko
64	0.29	1.7 Mo
128	2.20	6.3 Mo
256	17.4	24 Mo
512	133	96 Mo

Table 3.3: Data of the scenario with n possible compositions of depth 2

$$\forall i, j \in [\![1 \,;\, n]\!] \qquad same(\text{out}\,(f(\texttt{mypc}),\ [\texttt{myps}, \texttt{srv}_i])\,, \text{out}\,(f(\texttt{mypc}),\ [\texttt{myps}, \texttt{srv}_j]))$$

They represent a portion of size n^2 of the XSB table.

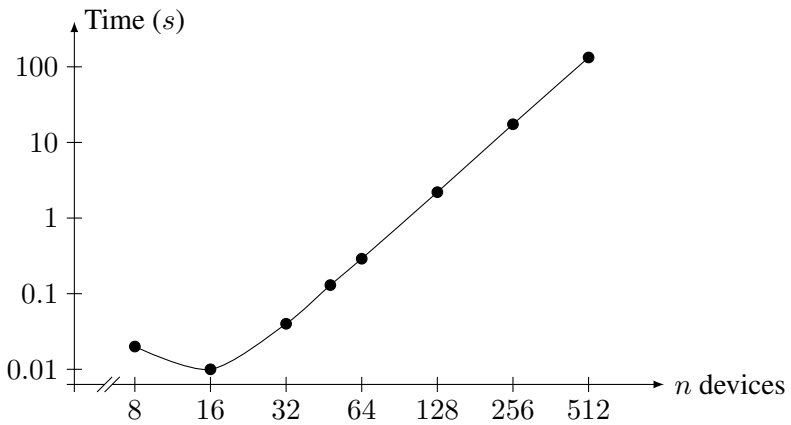
The results of the query $stored(X, Y)$ are:

$$\forall i \in [\![1 \,;\, n]\!] \qquad stored(\text{out}\,(f(\texttt{mypc},\ [\texttt{myps}, \texttt{srv}_i]), \texttt{srv}_i))$$

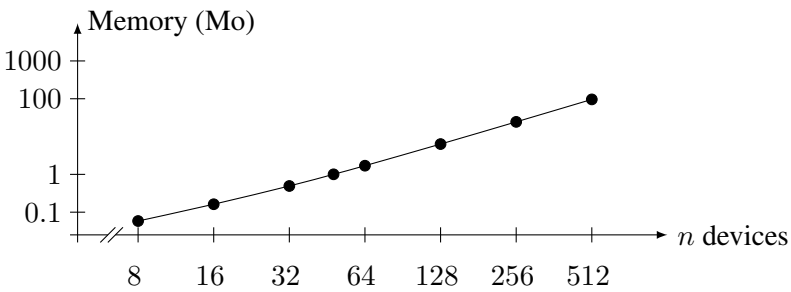
They represent a portion of size n of the XSB table.

Thus, the global size of the table is n^2.

Finally, to find each of the n final answers to the main query, we have to access the n^2-sized table. Thus, the time consumption is cubic.



(a) Time consumption



(b) Memory usage

Figure 3.6: Scenario with n possible compositions of depth 2

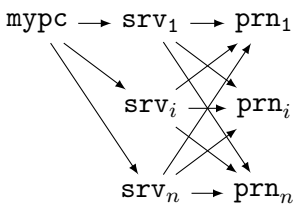


Figure 3.7: Schema of connections in the scenario 3.3.5

3.3.5 n^2 possible compositions of depth 2

In this scenario, we have the following connections represented by the graph of connections on figure 3.7:

$$\forall i, j \in [\![1; n]\!] \qquad \begin{cases} \mathit{connected}(\texttt{mypc}, \texttt{srv}_i) \\ \mathit{connected}(\texttt{srv}_i, \texttt{prn}_j) \end{cases}$$

There are n^2 connections and there are n^2 possibilities to print the file `myps`. For any i and j, it is made possible by the connections to upload the file onto the server i and then to print it on the printer j.

Statistics and curves

Statistical curves depending on the number of devices are drawn on figure 3.8 in a logarithmic coordinate system. The curve on figure 3.8(a) represents the time consumption. It appears to be quartic with n. The curve on figure 3.8(b) represents the memory consumption due to the XSB table. The memory space used is quadratic with n.

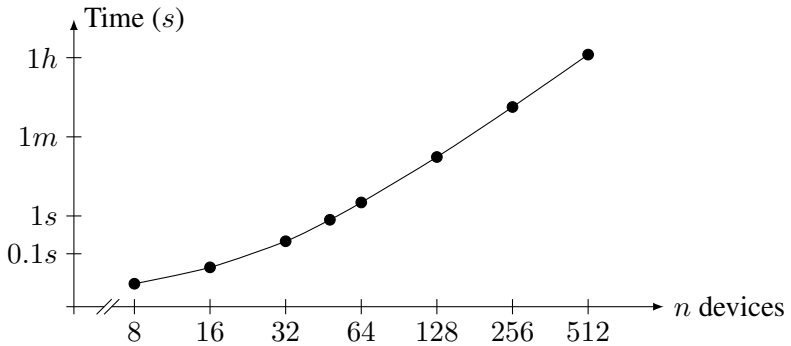
Number of devices	Time (s)	Memory (Ko)
1	0.02	11 Ko
2	0.02	16 Ko
4	0.03	28 Ko
8	0.03	58 Ko
16	0.07	162 Ko
32	0.27	493 Ko
64	2	1.7 Mo
128	21	6.3 Mo
256	280	24.5 Mo
512	4217	96 Mo

Table 3.4: Data of the scenario with n^2 possible compositions of depth 2

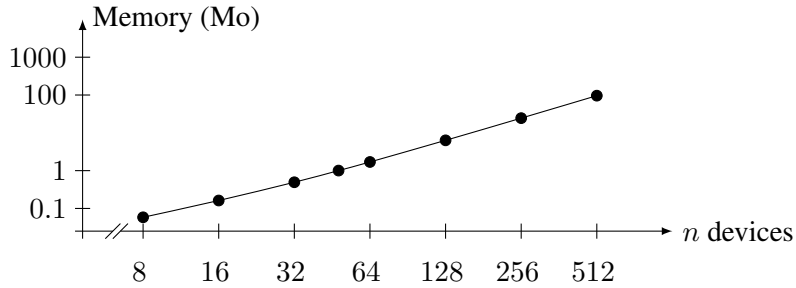
Interpretation

The results of the query $\mathit{type}(X, \mathit{paper})$ are stored in the XSB table and there are all the following answers:

$$\forall i, j \in [\![\, 1 \,;\, n \,]\!] \qquad \mathit{type}(\text{out}\ \big(g(\texttt{prn}_j),\ [\text{out}\ (f(\texttt{mypc}),\ [\texttt{myps}, \texttt{srv}_i])]\big)\ , \mathit{paper})$$



(a) Time consumption



(b) Memory usage

Figure 3.8: Scenario with n^2 possible compositions of depth 2

They represent a portion of size n^2 of the XSB table.

The results of the query $stored(X, Y)$ are stored in the XSB table and there are all the following answers:

$$\forall i \in [\![1 \,;\, n]\!] \qquad stored(\text{out}\,(f(\texttt{mypc}),\ [\texttt{myps}, \texttt{srv}_i])\,, \texttt{srv}_i)$$

These answers represent a portion of size n of the XSB table.

The results of the query $same(X, Y)$ are stored in the XSB table and there are all the following answers:

$$\forall i, j \in [\![1 \,;\, n]\!] \qquad same(\text{out}\,(f(\texttt{mypc}),\ [\texttt{myps}, \texttt{srv}_i])\,, \text{out}\,(f(\texttt{mypc}),\ [\texttt{myps}, \texttt{srv}_j]))$$

They represent a portion of size n^2 of the XSB table.

Thus, the size of the whole XSB table is n^2.

Finally, to find each of the n^2 final answers to the main query, we need to access the n^2-sized table. Thus the time consumption is quartic.

3.3.6 $2^{n/2}$ possible compositions of depth $n/2$

In this scenario, we suppose that $n = 2p$. We have the following connections represented by the graph of connections on figure 3.9:

$$\forall i, j \in \{0, 1\} \times [\![1; p]\!] \qquad \texttt{srv}_{2i-j+2} \equiv \texttt{srv}^i_j$$

$$\forall i \in \{0,1\} \quad \forall j \in [\![1; p-1]\!] \quad \begin{cases} connected(\mathtt{srv}_j^i, \mathtt{srv}_{j+1}^i) \\ connected(\mathtt{srv}_j^i, \mathtt{srv}_{j+1}^{1-i}) \\ connected(\mathtt{mypc}, \mathtt{srv}_1^i) \\ connected(\mathtt{srv}_p^i, \mathtt{prn}_i) \end{cases}$$

There are only $2 \times n$ connections but there are 2^p possibilities to print the file. For any x, if $\overline{i_0, \ldots, i_p}$ is the binary writing of x, then it is made possible by the connections to upload the file onto the server i_0, then onto the server i_1, ..., and finally onto the server i_p before printing it with the printer $i_p + 1$.

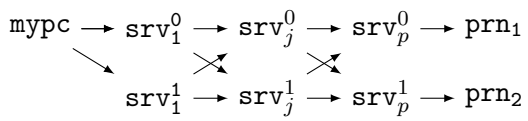
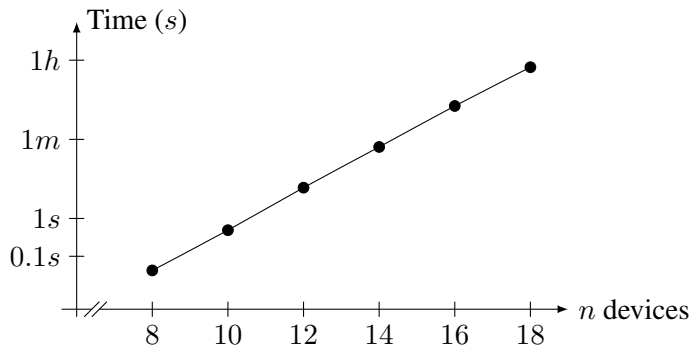


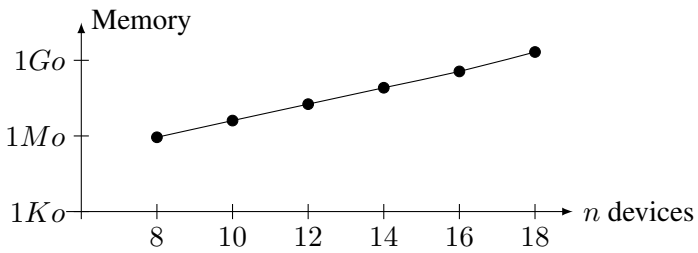
Figure 3.9: Schema of connections in the scenario 3.3.6

Statistics and curves

Statistical curves depending on the number of devices are drawn on figure 3.10. Attention should be paid to the fact that here the curves are drawn in a semi-logarithmic coordinate system. The curve on figure 3.10(a) represents the time consumption. It appears to be exponential with n. The curve on figure 3.10(b) represents the memory consumption due to the XSB table. The memory space used is exponential with n.



(a) Time consumption



(b) Memory usage

Figure 3.10: Scenario with 2^p composition of depth $p = \lfloor n/2 \rfloor$

Number of devices	Time (s)	Memory (Ko)
2	0.01	16 Ko
4	0.02	48 Ko
8	0.06	885 Ko
10	0.53	4 Mo
12	5.3	18.3 Mo
14	48	81.6 Mo
16	441	362 Mo
18	4503	3897 Mo

Table 3.5: Data of the scenario with 2^p compositions of depth $p = \lfloor n/2 \rfloor$

Interpretation

The results of the query $type(X, paper)$ are stored in the XSB table and there are 2^p answers of depth p. Thus, they represent a portion of size $p2^p$ of the table.

The results of the query $stored(X, Y)$ are stored in the XSB table and represent a portion of size $p2^p$ of the table.

The results of the query $same(X, Y)$ are stored in the XSB table and represent a portion of size $p^2 2^p$ of the table.

Thus, the size of the whole XSB table is $p^2 2^p$.

Then, to find each of the 2^p final answers to the main query, we need to access the $p^2 2^p$-sized table. Thus the time consumption is $n^2 2^n$.

3.4 Summary

This bunch of scenarios shows the behaviour of a quite common scenario with respects to the number of devices involved. We have seen with the scenario 3.3.2 how the typing is important. Typing narrows the field to explore in the description in order to find answers to a query.

There is a summary of all the scenarios seen in this chapter, on table 3.6.

Scenarios	3.3.2	3.3.3	3.3.4	3.3.5	3.3.6
possible compositions	1	1	n	n^2	2^n
depth of the compositions	2	n	2	2	n
time consumption	n^2	n^4	n^3	n^4	$n^2 2^n$
memory usage	n	n^3	n^2	n^2	$n^2 2^n$

Table 3.6: Summary of all the scenarios

Let us compare the scenario 3.3.2 — where there is a single possible composition of depth 2 — with the scenario 3.3.3 — where the depth of the single possible composition is n. The time consumption and the memory usage are multiplied by n^2, when the depth of the compositions are only multiplied by a n. The comparison between the scenario 3.3.2 and the scenario 3.3.4, considering a number of possible compositions multiplied by n, we can see that time and memory are multiplied by n too. Cross verifying with the comparison between scenario 3.3.4 and the scenario 3.3.5 which have the same difference as in the previous comparison, does not show any differences with the memory usage. This can be explain if we go back to the scenario and if we see precisely what is stored in memory. There is nothing stored in

the XSB table that associates a printer and a server in this particular scenario which is the main difference in the graph of connections.

Chapter 4

Decentralised reasoning for dynamic composition of resources

Contents

4.1 Reminders and preliminaries 40

4.1.1 Logical description of a device and its functionalities 40

4.1.2 Queries 42

4.1.3 Example 44

4.1.4 Relevant devices for a query 45

4.2 Propositional encoding: principles and properties 48

4.2.1 Encoding of device descriptions and queries 48

4.2.2 Properties 52

4.2.3 Examples 57

4.3 Look-up by decentralised propositional reasoning 59

4.3.1 Look-up: definition and properties 59

4.3.2 Decentralised computation of $\mathrm{Lookup}\,(Q)$ using SOMEWHERE 61

4.4 Heterogeneous descriptions 62

We assume in this chapter that there is not a central repository of the descriptions of each devices. Instead every device will store their own descriptions. Also, each device has only a few neighbours to which it is connected. We assume that it cannot exchange data, messages and descriptions but with its very neighbours. In this context, the first problem to solve for a device that is queried (by a user or by another device) is to find in the network the relevant devices whose (possibly composed) functionalities can answer the query. This is a difficult problem since each device has a partial knowledge of the networked devices and their functionalities.

A flooding strategy could be applied: it would consist of spreading the query from a device to all its neighbours and so on. Each device that could answer part of the query would transmit back the corresponding answers. All the answers finally received by the initially requested device would have to be combined. Such a flooding strategy can be applied to P2P systems sharing files (*e.g.* gnutella) because the queries are simple keywords and the expected answers are single file fitting the keywords of the query instead of a set of tuples. In our setting, queries contain variables that can be unified possibly within functions in many ways. A flooding strategy would require to spread complex queries and to send back complex answers that would have to be combined by the queried device, anyway.

Our approach to answer queries in this decentralised setting is in two steps. First, a look up step that looks in the network for the relevant devices (*i.e.* the ones whose descriptions are sufficient to answer to the query). Then, the second step, which is achieved by the requested device, consists of getting the descriptions stored in the relevant devices previously found and to locally reason upon them using its local PROLOG engine in order to answer the query.

In this chapter, we focus on the look-up step which is the main issue in a decentralised setting — it corresponds to the (B), (C) and (D) steps in figure 4.6. Our approach can be seen as a flooding strategy applied to an abstraction in propositional logic of the first-orderquery and the device descriptions. This way, queries are abstracted into 'keywords' queries where the 'keywords' are propositional variables; and device descriptions are abstracted into propositional rules. The look-up is then carried out by a decentralised reasoning in propositional logic which is implemented by the existing SOMEWHERE platform [20].

We will begin with a set of reminders and preliminaries to define the set of devices that are relevant to a query. Then we describe the propositional encoding of first-order description that we propose with its properties. Next, we explain how a decentralised reasoning on this encoding with the encoding of a query can be exploited to find a subset of relevant devices that are still sufficient to get all the answer to the query. Finally,

4.1 Reminders and preliminaries

4.1.1 Logical description of a device and its functionalities

A device d stores its description: a set of facts and rules in first-order logic. It contains:

- a set of facts representing:
  - $instanceof$ relations in the taxonomies: $type(d,\ c)$;
  - $subclass$ relations in the taxonomies: $subclassof(c_1,\ c_2)$;
  - the assertions of known properties about the device:
    - $type(d,\ c)$;
    - $connected(d,\ d')$;
    - $located(d,\ l)$, $type(l,\ c_l)$;
    - $inside(l_1,\ l_2)$;
    - $stored(o,\ d)$, $type(o,\ c_o)$;
    - $hasfunct(d,\ f(d))$, $type(f(d),\ c_f)$; where f is a skolem function denoting one of the functionalities of the device d.
- a set of rules $type(d,\ c_d) \rightarrow hasfunt(d,\ f(d)), type(f(d),\ c_f)$, to assert that every device of type c_d has a functionality of type c_f
- a set of rules R describing the functionalities of the device, where a rule r has the form:

$$
\begin{array}{lll}
r: & & type(F,T) & \text{Type the functionality} \\
 & \wedge & hasfunct(D,F) & \text{Specify the device} \\
 & \wedge & \bigwedge_{k=1}^{n} type(I_k,T_k) & \text{Type the inputs} \\
 & \wedge & Prec(\vec{I},D) & \text{Define preconditions} \\
 & \rightarrow & type(out(F,[\vec{I}]),T_o) & \text{Type the output} \\
 & \wedge & Post(\vec{I},out(F,[\vec{I}]),D) & \text{Define post-conditions}
\end{array}
$$

where

- T is the type of the described functionality;
- D is the device that has the functionality;
- n is the number of inputs for this class T of functionalities;
- I_k is one of the inputs;
- out is a function representing the output of a functionality F on its inputs $[\vec{I}]$;
- $\vec{I}$ represents $I_1, \ldots, I_n$;
- $Prec$ is a conjunction of preconditions where the inputs and the device D are involved;
- $Post$ is a conjunction of post-conditions where the inputs, the output and the device D are involved;

- a set of rules expressing the inheritance of types. It can also include transitivity, symmetry, reflexivity of some properties.

Finally, for a device d, $\mathrm{Descr}\,(d)$ denotes the union of facts and rules mentioned above that describe the device. By extension, for a set D of devices:

$$\mathrm{Descr}\,(D) = \bigcup_{d \in D} \mathrm{Descr}\,(d)$$

For convenience, for a rule r, we define $\mathrm{Descr}^{-1}\,(r)$

$$d = \mathrm{Descr}^{-1}\,(r) \Leftrightarrow r \in \mathrm{Descr}\,(d)$$

For a rule r, $\mathrm{Condition}\,(r)$ denotes the set of the atoms that are present in the premise of the rule r and $\mathrm{Conclusion}\,(r)$ denotes the set of the atoms that are present in the conclusion of the rule r. We consider an atomic formula as a rule with an empty condition. Thus, we can extend the definition to atomic formulæ:

$$\mathrm{Condition}\,(\pi(t_1,t_2)) = \emptyset$$

$$\mathrm{Conclusion}\,(\pi(t_1,t_2)) = \{\pi(t_1,t_2)\}$$

The definition of a predicate π, denoted by $\mathrm{Def}\,(\pi,\ D)$, is the union of facts of the form $\pi(t_1,t_2)$ and of the rules having an atom $\pi(t_1,t_2)$ in their conclusion. As a fact is a rule without conditions, we can simply write:

$$\mathrm{Def}\,(\pi,\ D) = \{r \in \mathrm{Descr}\,(D) \quad / \quad \exists t_1,t_2 \qquad \pi(t_1,t_2) \in \mathrm{Conclusion}\,(r)\}$$

it is important to note that in a decentralised setting, $\mathrm{Def}\,(\pi,\ D)$ may be distributed among several devices.

Moreover, we define a relation between predicates. A predicate π logically depends on a predicate π' among the set of devices D if and only if π' appears in the conditions of a rule in the definition of π.

$$\pi \preceq \pi' \Leftrightarrow \exists r \in \mathrm{Def}\,(\pi,\ D)\,, \quad \exists t_1, t_2 \ / \ \pi'(t_1, t_2) \in \mathrm{Condition}\,(r)$$

4.1.2 Queries

A query Q is defined by a set of rules (denoted by $\mathrm{Def}\,(Q)$) of the form:

$$\mathrm{Def}\,(Q): \qquad \bigwedge_{i=1}^{n} R_i(t_i^1, t_i^2) \rightarrow Q(X_0, \ldots, X_p)$$

where t_i^j are terms of the language or variables, R_i are predicates and for all k in $[\![\,1\,;\,p\,]\!]$ it exists i in $[\![\,1\,;\,n\,]\!]$ and j in $\{1, 2\}$ so that X_k appears in t_i^j. Afterwards $X_0, \ldots, X_p$ will be denoted by $\bar{v}$ and called variables of interest.

Let D be a set of devices. The answers to the query Q against the union of description $\mathrm{Descr}\,(D)$ — denoted by $\mathrm{Answer}\,(Q,\ D)$ — is the set of tuples $\bar{t}$ on Herbrand universe of $\mathrm{Descr}\,(D)$ instantiated terms for which $Q(\bar{t})$ can be logically entailed from the devices descriptions and the definition of the query.

$$\mathrm{Answer}\,(Q,\ D) = \{\bar{t} \in H(\mathrm{Descr}\,(D))^p \quad / \quad \mathrm{Descr}\,(D)\,, \mathrm{Def}\,(Q) \models Q(\bar{t})\}$$

Definition 11 : *Derivation tree*

In first-order logic, a derivation tree of the query Q with respect to the description of a set of devices D is a tree that reflects a proof (or a refutation) of Q according to the set of descriptions. It is denoted by $\mathrm{DT}\,(Q,\ D)$. Each node of the tree is a (possibly empty) list of atoms. The root node is $Q(\bar{v})$. Let $N = (A_1, \ldots, A_k)$ (with $1 \leq k$) be a node in the tree. Then, for each rules

$$r = \textstyle\bigwedge_{i=1}^{p} B_i \rightarrow \pi_1(x, y) \in \mathrm{Def}\,(\pi_1,\ D)$$

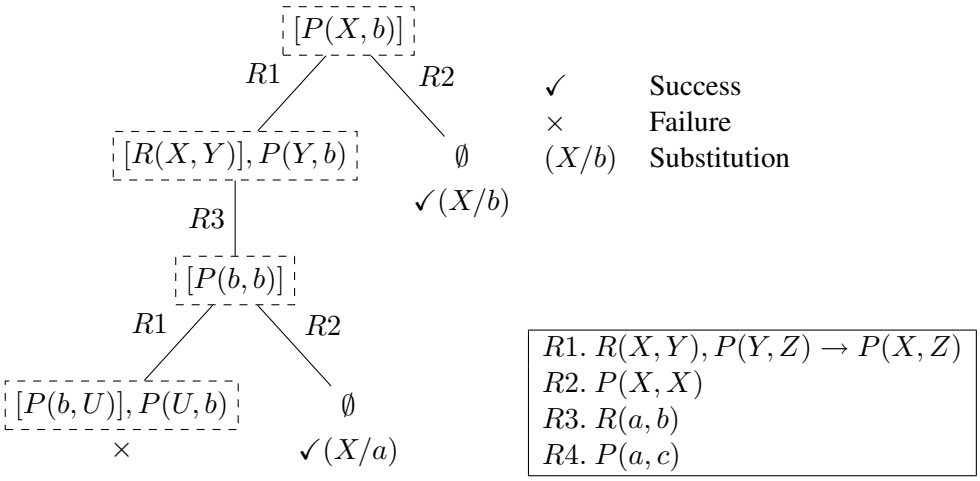
(or facts if $p = 0$) such that $\pi_1(x, y)$ and A_1 are unifiable with σ their Most General Unifier (MGU), the node has a child which is the node N where A_1 has been replaced by the B_i and the substitution has been applied:

$$B_i\sigma, \ldots, B_p\sigma, A_2\sigma, \ldots, A_k\sigma$$

The edge from the node N to this child his labelled by r. If $p = 0$ (*i.e.* the atom A_1 unifies with a fact) then we simply erase A_1 from the list and apply the substitution σ. Also if $k = 1$ and $p = 0$ (*i.e.* the node is reduced to one atom unifying with a fact) the child will be an empty clause. Nodes that are empty clause have no children and are called 'success' leaves. Nodes that are not empty clause and have no children are called 'failure' leaves. By extension, the set of paths in the tree that end with a success leaf (resp. failure leaf) is called the set of success paths (resp. failure paths) of the tree and is denoted by $\mathrm{DT}_+\,(Q,\ D)$ (resp. $\mathrm{DT}_-\,(Q,\ D)$).

Formally, a derivation tree T is a couple made of a node — which is a tuple of atoms — and a list of sub-trees with the label of the corresponding edge.

$$T = (R, [(T_1, r_1), \ldots, (T_n, r_n)])$$



(a) Derivation tree (b) Set of rules

Figure 4.1: A derivation tree of a query with respect to a set of rules

where $R = (A_1, \ldots, A_k)$ is the root of the tree, A_i are atoms, $T_i = (X_i, Y_i)$ are derivation trees and r_i are rules so that:

$$\text{Descr}\,(D)\,, R, r_i \models X_i \qquad \square$$

Definition 12 : The height of a derivation tree is:

- zero for a success or a failure

$$\text{height}\,((X, ())) = 0$$

- one more than the height of the highest sub-tree

$$\text{height}\,((X, ((T_1, r_1), \ldots, (T_n, r_n)))) = 1 + \max_{1 \leq i \leq n} \{\text{height}\,(T_i)\} \qquad \square$$

Definition 13 : We define $\text{Rules}\,(T)$, the set of all rules that appear in the labels of the edges of the tree T.

$$\text{Rules}\,((X, ())) = \emptyset$$
$$\text{Rules}\,((X, ((T_1, r_1), \ldots, (T_n, r_n)))) = \{r_1, \ldots, r_n\} \cup \bigcup_{i=1}^{n} \text{Rules}\,(T_i)$$

The figure 4.1 shows an example of a derivation tree T of a query with respect to a set of rules and facts. According to the figure, we have:

$$\begin{aligned} \text{height}\,(T) &= 3 \\ \text{Rules}\,(T) &= \{R1, R2, R3\} \end{aligned} \qquad \square$$

4.1.3 Example

The following represents the logical description of three devices. There are parts of the descriptions that are common for the three devices and concern the taxonomy , the properties of some predicates, the inheritance of types, *etc*. We introduce them before the description of the devices. Identifiers beginning with an upper-case letter stand for variables.

r1: $\mathit{subclassof}(\mathrm{C},\ \mathrm{D}), \mathit{type}(\mathrm{X},\ \mathrm{C}) \rightarrow \mathit{type}(\mathrm{X},\ \mathrm{D}).$

r2: $\mathit{same}(\mathrm{X},\ \mathrm{Y}) \rightarrow \mathit{same}(\mathrm{Y},\ \mathrm{X}).$

r3:
$$\left\{\begin{array}{l}
\mathit{subclassof}(\mathit{printer},\ \mathit{device}).\\
\mathit{subclassof}(\mathit{room},\ \mathit{place}).\\
\mathit{subclassof}(\mathit{printing},\ \mathit{func}).\\
\mathit{subclassof}(\mathit{download},\ \mathit{func}).\\
\mathit{subclassof}(\mathit{paper},\ \mathit{data}).\\
\mathit{subclassof}(\mathit{computer},\ \mathit{device}).\\
\mathit{subclassof}(\mathit{pc},\ \mathit{computer}).\\
\mathit{subclassof}(\mathit{pocketpc},\ \mathit{computer}).\\
\mathit{subclassof}(\mathit{file},\ \mathit{data}).\\
\mathit{subclassof}(\mathit{postscript},\ \mathit{file}).
\end{array}\right.$$

The rule r1 expresses the inheritence of types; the rule r2, the symmetry of the predicate same. And the set of facts r3 are an extract of the facts expressing the taxonomies in a logical way.

Consider first, a printer `prn`, which is located in `roomD`. Here is its description

$$\begin{array}{ll}
 & \mathit{type}(\mathtt{prn},\ \mathit{printer}).\\
 & \mathit{located}(\mathtt{prn},\ \mathtt{roomD}).\\
 & \mathit{type}(\mathtt{roomD},\ \mathit{room}).\\
\text{r4:} & \mathit{type}(\mathrm{D},\ \mathit{printer}) \rightarrow \mathit{type}(g_0(\mathrm{D}),\ \mathit{printing})\ \wedge\ \mathit{hasfunct}(\mathrm{D},\ g_0(\mathrm{D})).
\end{array}$$

The rule r4 expresses that every `printer` has a *printing* functionality. The following rule describes the printing functionality.

r5:
$$\begin{array}{l}
\mathit{type}(\mathrm{out}\ (\mathrm{G},\ \mathrm{J})\ ,\ \mathit{postscript})\\
\wedge\ \mathit{type}(\mathrm{F},\ \mathit{printing})\\
\wedge\ \mathit{hasfunct}(\mathrm{E},\ \mathrm{G})\\
\wedge\ \mathit{hasfunct}(\mathrm{D},\ \mathrm{F})\\
\wedge\ \mathit{connected}(\mathrm{E},\ \mathrm{D})\\
\rightarrow \mathit{type}(\mathrm{out}\ (\mathrm{F},\ [\mathrm{out}\ (\mathrm{G},\ \mathrm{J})])\ ,\ \mathit{paper}).
\end{array}$$

The rule r5 describes the printing functionality of a device D. It takes an input of type *postscript* and provides an output of type *paper* provided that the printing device D is connected to a device E that has a functionality able to output postscript file. Note that $\mathrm{out}\ (\mathrm{F},\ [\mathrm{out}\ (\mathrm{G},\ \mathrm{J})])$ expresses the output of the composed functionality.

Consider a PC `srv`, which is connected to the printer `prn`.

$$\begin{array}{ll}
 & \mathit{type}(\mathtt{srv},\ \mathit{pc})\\
 & \mathit{connected}(\mathtt{srv},\ \mathtt{prn})\\
\text{r6:} & \mathit{type}(\mathrm{D},\ \mathtt{pc}) \rightarrow \mathit{type}(g_1(\mathrm{D}),\ \mathit{download})\ \wedge\ \mathit{hasfunct}(\mathrm{D},\ g_1(\mathrm{D})).
\end{array}$$

The rule r6 expresses that every *pc* has a *download* functionality. The rule r7 describes the downloading functionality of a device D. The functionality requires two inputs: a computer S, which is connected to D,

and a file I, which is stored in the computer S and is of a given format C. The functionality provides, as output, a file of format C, stored in D and with the same content as I.

$$
\text{r7:}\quad
\begin{array}{l}
\quad \mathit{type}(\mathrm{I},\ \mathrm{C}) \\
\quad \wedge \mathit{type}(\mathrm{S},\ \mathit{computer})\ \wedge\ \mathit{hasfunct}(\mathrm{D},\ \mathrm{F})\ \wedge\ \mathit{connected}(\mathrm{S},\ \mathrm{D}) \\
\quad \wedge\ \mathit{subclassof}(\mathrm{C},\ \mathit{file})\ \wedge\ \mathit{stored}(\mathrm{I},\ \mathrm{S})\ \wedge\ \mathit{type}(\mathrm{F},\ \mathit{download}) \\
\rightarrow \mathit{type}(\mathrm{out}\ (\mathrm{F},\ [\mathrm{I},\mathrm{S}])\ ,\ \mathrm{C}) \\
\wedge\ \mathit{stored}(\mathrm{out}\ (\mathrm{F},\ [\mathrm{I},\mathrm{S}])\ ,\ \mathrm{D}) \\
\wedge\ \mathit{same}(\mathrm{out}\ (\mathrm{F},\ [\mathrm{I},\mathrm{S}])\ ,\ \mathrm{I})
\end{array}
$$

Consider a pocket PC denoted by the constant `mypc`, storing a postscript file denoted by the constant `myps` and connected to the PC `srv`. It is described by the following facts.

$$
\begin{array}{l}
\mathit{type}(\texttt{myps},\ \mathit{postscript}). \\
\mathit{type}(\texttt{mypc},\ \mathit{pocketpc}). \\
\mathit{connected}(\texttt{mypc},\ \texttt{srv}). \\
\mathit{stored}(\texttt{myps},\ \texttt{mypc}).
\end{array}
$$

Finally, consider the query q_1 defined by:

$$
\begin{array}{l}
\quad \mathit{hasfunct}(\mathrm{D},\ \mathrm{F})\ \wedge\ \mathit{located}(\mathrm{D},\ \texttt{roomD}) \\
\quad \wedge\ \mathit{type}(\mathrm{out}\ (\mathrm{F},\ \mathrm{Is})\ ,\ \mathit{paper}) \\
\quad \wedge\ \mathit{member}(\mathrm{I},\ \mathrm{Is})\ \wedge\ \mathit{type}(\mathrm{I},\ \mathit{file})\ \wedge\ \mathit{same}(\mathrm{I},\ \texttt{myps}) \\
\rightarrow q_1(\mathrm{F},\ \mathrm{Is})
\end{array}
$$

It asks for a functionality F and a list of inputs Is so that

- the device having the functionality F is located in `roomD`;
- the type of the output of F is *paper*;
- one of the input from Is is a file which has the same content as the file `myps`.

An answer to q_1, against the description introduced above, is the following tuple:

$$(\mathrm{F}, \mathrm{Is}) = (g_0(\texttt{prn}), [\mathrm{out}\ (g_1(\texttt{srv}),\ [\texttt{myps}, \texttt{mypc}])])$$

where $g_0(\texttt{prn})$ is the printing functionality of `prn` and its input Is is the output of the functionality $g_1(\texttt{srv})$, which is the download functionality of the server `srv`, applied to the inputs `myps` and `mypc`. It expresses that to get `myps` (stored in `mypc`) printed in `roomD` we have to compose the downloading functionality of the PC `srv` and the printing functionality of the printer `prn`. Such a composition is possible because `mypc` is connected to `srv` and itself connected to `prn`.

4.1.4 Relevant devices for a query

Given a query Q asked to a given device d among a set of devices D, the problem is to find a subset $\mathrm{Relevant}\ (Q,\ D)$ of D so that the union of their descriptions are sufficient to compute all the answers to Q.

$$\mathrm{Answer}\ (Q,\ \mathrm{Relevant}\ (Q,\ D)) = \mathrm{Answer}\ (Q,\ D)$$

The query Q is nothing but a predicate, so we define $\mathrm{Relevant}\ (\pi,\ D)$ for any predicate π. We first define $\mathrm{Support}\ (\pi,\ D)$, which are the rules or facts that are in the definition of π or in the definition of any predicate on which π logically depends. We formally define it by induction:

$$\text{Support}\,(\pi,\ D)\ = \bigcup_{i=0}^{\infty} \text{Support}_i\,(\pi,\ D)$$

where $$\text{Support}_0\,(\pi,\ D)\ = \text{Def}\,(\pi,\ D)$$

and $$\text{Support}_{i+1}\,(\pi,\ D)\ = \bigcup_{\pi \preceq \pi'} \text{Support}_i\,(\pi',\ D)$$

We get the following immediate property:

$$\text{Support}\,(\pi,\ D) = \text{Def}\,(\pi,\ D) \cup \bigcup_{\pi \preceq \pi'} \text{Support}\,(\pi',\ D)$$

in particular we have

$$\pi \preceq \pi' \Rightarrow \text{Support}\,(\pi',\ D) \subset \text{Support}\,(\pi,\ D)$$

The relevant devices for a predicate π are the devices whose descriptions contain rules or facts belonging to support of π

$$\text{Relevant}\,(\pi,\ D) = \{d \in D\ /\ \text{Support}\,(\pi,\ D) \cap \text{Descr}\,(d) \neq \emptyset\}$$

Since the set of devices — and therefore the set of their description — is finite, so is $\text{Relevant}\,(\pi,\ D)$ and $\text{Support}\,(\pi,\ D)$. More precisely, for any predicate π it exists an integer N such that

$$\text{Support}\,(\pi,\ D) = \bigcup_{i=0}^{N} \text{Support}_i\,(\pi,\ D)$$

We prove in lemma 1 that the rules that appear in the labels of the edges of the derivation tree of an instantiated query belong to the support of the query.

Lemma 1 : Let Q be a query, D a set of devices and $\bar{t}$ an answer to Q. Then

$$\text{Rules}\,(\text{DT}\,(Q(\bar{t}),\ D)) \subset \text{Support}\,(Q,\ D)$$

Proof:

Since the SLD strategy of resolution — which is implemented in PROLOG engines — is complete for definite clauses, for any answers $\bar{t}$, it exists a deduction of $Q(\bar{t})$ from the union of $\text{Descr}\,(D)$ and $\text{Def}\,(Q)$, which can be summarised by a derivation tree.

Let us prove the inclusion by induction on the height of the derivation tree of $Q(\bar{t})$.

Basis:

Let us assume that the height of the derivation tree of $Q(\bar{t})$ is 1. It means that $Q(\bar{t})$ unifies with a fact F, which belongs to the support of Q by definition. Thus,

$$\text{Rules}\,(\text{DT}\,(Q(\bar{t}),\ D)) = \{F\} \subset \text{Support}\,(Q,\ D)$$

Inductive step:

Let us assume that for a certain integer n,

if $$\text{height}\,(\text{DT}\,(Q(\bar{t}),\ D)) \leq n$$

then $$\text{Rules}\left(\text{DT}\left(Q(\bar{t}),\ D\right)\right) \subset \text{Support}\left(Q,\ D\right)$$

Let us prove the property for a height of $n+1$. Let us assume that the height of the derivation tree of $Q(\bar{t})$ is $n+1$.

$$\text{DT}\left(Q(\bar{t}),\ D\right) = \left(Q(\bar{t}), [(T_1, r_1), \ldots, (T_k, r_k)]\right)$$

with $$\forall i \quad \text{height}\left(T_i\right) \leq n$$

if we denote $T_i = (\pi_i(\overline{t_i}), Y_i) = \text{DT}\left(\pi_i(\overline{t_i}),\ D\right)$, the inductive hypothesis can be applied and we deduce that

$$\text{Rules}\left(T_i\right) \subset \text{Support}\left(\pi_i,\ D\right)$$

Now, $$\text{Rules}\left(\text{DT}\left(Q(\bar{t}),\ D\right)\right) = \bigcup_{i=0}^{k} \{r_i\} \cup \bigcup_{i=0}^{k} \text{Rules}\left(T_i\right)$$

and $$r_i \in \text{Support}\left(Q,\ D\right)$$

Also $$Q \preceq \pi_i$$

thus $$\text{Support}\left(\pi_i,\ D\right) \subset \text{Support}\left(Q,\ D\right)$$

and finally $$\text{Rules}\left(\text{DT}\left(Q(\bar{t}),\ D\right)\right) \subset \text{Support}\left(Q,\ D\right)$$

$\diamondsuit$

Theorem 1 : Let Q be a query, D a set of devices and d a device in D. Then

$$\text{Answer}\left(Q,\ \text{Relevant}\left(Q,\ D\right)\right) = \text{Answer}\left(Q,\ D\right)$$

Proof:

As $\text{Relevant}\left(Q,\ D\right)$ is a subset of D we have one inclusion.

Let us go back to the definitions to prove the other inclusion. Let $\bar{t}$ be an answer to Q against the descrptions of D.

$$\text{Descr}\left(D\right), \text{Def}\left(Q\right) \models Q(\bar{t})$$

According to the definition of the derivation tree that correspond to the SLD strategy of resolution we have

$$\text{Rules}\left(\text{DT}\left(Q(\bar{t}),\ D\right)\right) \models Q(\bar{t})$$

Now, we have proved in lemma 1 that

$$\text{Rules}\left(\text{DT}\left(Q(\bar{t}),\ D\right)\right) \subset \text{Support}\left(Q,\ D\right)$$

And by definition of $\text{Relevant}\left(Q,\ D\right)$, we have

$$\text{Support}\left(Q,\ D\right) \subset \text{Def}\left(Q\right) \cup \text{Descr}\left(\text{Relevant}\left(Q,\ D\right)\right)$$

We can then deduce

$$\text{Descr}\left(\text{Relevant}\left(Q,\ D\right)\right), \text{Def}\left(Q\right) \models Q(\bar{t})$$

■

4.2 Propositional encoding: principles and properties

The principle of our encoding is to abstract the first-order device descriptions and the queries into a set of facts and rules in propositional logic, while preserving some important properties. The main property of our encoding is that a propositional backward-reasoning from the *encoded* query, using the *encoded* device descriptions, leads to the identification of the devices whose *first-order* descriptions are sufficient to compute the set of all the answers of the *first-order* query. To relate explicitly a given device d to the encoding of its description, we introduce a particular propositional variable, denoted $d^\star$, and we add it as a condition into the propositional rules encoding the first-order rules in the description of d. We call those propositional variables *device propositional variables* . We have shown (theorem 3) that the devices whose descriptions are sufficient for answering the query are obtained from the propositional *implicants* of the encoded query which are conjunctions of device propositional variables. In section 4.2.1, we provide the encoding function by defining first the encoding of a device, then the encoding of first-order (conjunction of) atoms, and finally the encoding of first-order rules. In particular, the encoding of the class taxonomies is made specific to each device description by encapsulating the inheritance between classes into as many propositional rules as there are subclass relations for each first-order atoms involving arguments typed by classes. Section 4.2.2 is dedicated to the transfer property (theorem 3) of our encoding, which shows that the sufficient devices for answering a first-order query can be obtained by an appropriate decoding of the proper prime implicants of the encoding of the query. The way this transfer property can be exploited and implemented in a decentralised setting using SOMEWHERE will be explained in section 4.3.

4.2.1 Encoding of device descriptions and queries

We define the encoding function denoted by Code.

Encoding of a device

For a device d, assuming that d is an identifier of the device in the network, we define

$$\text{Code}\left(d\right) = d^\star$$

The definition is extended to a set of devices $\{d_1, \ldots, d_n\}$:

$$\text{Code}\left(\{d_1, \ldots, d_n\}\right) = \bigwedge_{i=1}^{n} d_i^\star$$

We define also the inverse function:

$$\text{Decode}\left(\bigwedge_{i=1}^{n} d_i^\star\right) = \{d_1, \ldots, d_n\}$$

In the following, the letter F (or F', F_0, *etc*) denotes conjunction of atoms that are made of neither the predicate $type$ nor the predicate $subclassof$; the letter C (or C', C_0, *etc*) denotes conjunction of atoms which are exclusively made of the predicate $type$; the letter S (or S', S_0, *etc*) denotes conjunction

of atoms which are exclusively made of the predicate $subclassof$. Atoms of the form $type(t,\ \tau)$and $subclassof(\tau_1,\ \tau_2)$ are called typing atoms afterwards.

We distinguish in the description of a device d the following sets of rules:

$$\begin{aligned} C_0(d) &= \{\rightarrow type(t,\ \tau) \in \mathrm{Descr}\,(d)\} \\ S_0(d) &= \{\rightarrow subclassof(\tau,\ \theta) \in \mathrm{Descr}\,(d)\} \\ T_0(d) &= C_0(d) \cup S_0(d) \\ R_0(d) &= \{X' \rightarrow F \wedge C \in \mathrm{Descr}\,(d)\} \end{aligned}$$

$T_0(d)$ plays a particular role in the encoding of the descriptions of d and is called the set of typing facts of the device d.

Encoding of an atom

The encoding function comes with two flavours. With the symbol i as a subscript, it provides propositional variables that encode atoms present in the condition of a rule. Those propositional variables are called 'in-propositional variables' afterwards. With the symbol o as a subscript, it provides propositional variables that encode atom in the conclusion of a rule. Those propositional variables are called 'out-propositional variables' afterwards.

We first define the encoding of an atom $\pi(t_1, t_2)$ — where π is neither the predicate $type$ nor the predicate $subclassof$ — denoted by $\mathrm{Code_i}\,(\pi(t_1, t_2)\,|T\,)$ (resp. $\mathrm{Code_o}\,(\pi(t_1, t_2)\,|T\,)$) because it depends on the typing atoms of t_1 and t_2. T is the context of the encoding of $\pi(t_1, t_2)$. It is a set of atoms, including the typing of t_1 (resp. t_2). This typing can be explicit with the predicate $type$ or inferred from the domain (resp. range) of the predicate π.

if	$type(t_i,\ \tau_i) \in T$
or	$type(t_i,\ \mathrm{X}), subclassof(\mathrm{X},\ \tau_i) \in T$
then	$\mathrm{Code_i}\,(\pi(t_1, t_2)\,|T\,) = \pi.\tau_1.\tau_2{}^{\mathrm{i}}$
	$\mathrm{Code_o}\,(\pi(t_1, t_2)\,|T\,) = \pi.\tau_1.\tau_2{}^{\mathrm{o}}$

We also encode the atoms of the form $type(t,\ \tau)$ the following way:

$$\begin{aligned} \mathrm{Code_i}\,(type(t,\ \tau)\,|T\,) &= type.\tau.\mathbb{T}^{\mathrm{i}} \\ \mathrm{Code_o}\,(type(t,\ \tau)\,|T\,) &= type.\tau.\mathbb{T}^{\mathrm{o}} \end{aligned}$$

where $\mathbb{T}$ denotes the type of the class names. Note that in this case, the encoding is independent of T and we can write $\mathrm{Code_i}\,(type(t,\ \tau))$ as well.

Encoding of a conjunction of atoms

For convenience, a conjunction of atoms can be seen as a set of atoms. For a conjunction of atoms A, the encoding of A associated with a set of typing atoms is the conjunction of the encoding of each atom of A associated with $A \cup T$ as set of typing atoms.

$$\begin{aligned} \mathrm{Code_i}\,(A\,|T\,) &= \bigwedge_{a \in A} \mathrm{Code_i}\,(a\,|T \cup A\,) \\ \mathrm{Code_o}\,(A\,|T\,) &= \bigwedge_{a \in A} \mathrm{Code_o}\,(a\,|T \cup A\,) \end{aligned}$$

For instance, suppose we have the following conjunction of atoms.

Set of atoms A	Set of typing facts T
$\mathit{stored}(\texttt{myps},\ \texttt{mypc})$.	$\mathit{type}(\texttt{mypc},\ \mathit{pocketpc})$.
$\mathit{type}(\texttt{myps},\ \mathit{postscript})$.	
$\mathit{connected}(\texttt{mypc},\ \texttt{srv})$	

then,

$$\mathrm{Code_i}\,(A\,|T\,) = \mathit{stored.postscript.pocketpc}^{\mathrm{i}} \wedge \mathit{type.postscript}.\mathbb{T}^{\mathrm{i}} \wedge \mathit{connected.pocketpc.device}^{\mathrm{i}}$$

The propositional variable $\mathit{connected.pocketpc.device}^{\mathrm{i}}$ is made from the inferred type of `srv`, which is inferred from the range of the predicate $\mathit{connected}$.

Encoding of a rule

We now define the encoding of a rule R, which is stored in the device d

$$R: \quad F' \wedge T' \rightarrow F \wedge C$$

Let $T_0(d)$ be the set of typing atoms stored in d. $T_0(d)$ represents the context of the encoding of all the rules in the description of d.

$$\mathrm{Code}\,(R\,|T_0(d)\,) = d^{\star} \wedge\ \mathrm{Code_i}\,(F' \wedge T'\,|T_0(d)\,) \rightarrow \mathrm{Code_o}\,(F \wedge C\,|T_0(d) \cup T'\,)$$

It is important to note that the encoding of a first-order logic rule provides a propositional logic rule with a conjunction of in-propositional variables in the condition side and with a conjunction of out-propositional variables in the conclusion side.

The typing facts that are in $C_0(d)$ are particular rules of the form:

$$\rightarrow \mathit{type}(t,\ \tau)$$

then their encoding is

$$\mathrm{Code}\,(\mathit{type}(t,\ \tau)) = d^{\star} \rightarrow \mathit{type}.\tau.\mathbb{T}^{\mathrm{o}}$$

Encoding of the taxonomy

The encoding of a fact

$$\rightarrow \mathit{subclassof}\,(\tau,\ \tau')$$

is defined in the context $\mathrm{Descr}\,(d)$.

In the following, if it exists the fact $\rightarrow \mathit{subclassof}\,(\tau,\ \tau')$ in the description of the device d, we say that τ is a strict subclass of τ'. The global subclass relation is the transitive and reflexive closure of the strict subclass relation. τ is a global subclass of τ', denoted $\tau \sqsubseteq \tau'$ if and only if either $\tau = \tau'$ or it exists τ'' such that τ is a strict subclass of τ'' and τ'' is a global subclass of τ'. Therefore, if $\tau \sqsubseteq \tau'$ appears in the context of the encoding, it means that it exists $\tau_1, \ldots, \tau_n$ such that for all i between 1 and $n-1$, the rule $\rightarrow \mathit{subclassof}\,(\tau_i,\ \tau_{i+1})$ belongs to the context of the encoding.

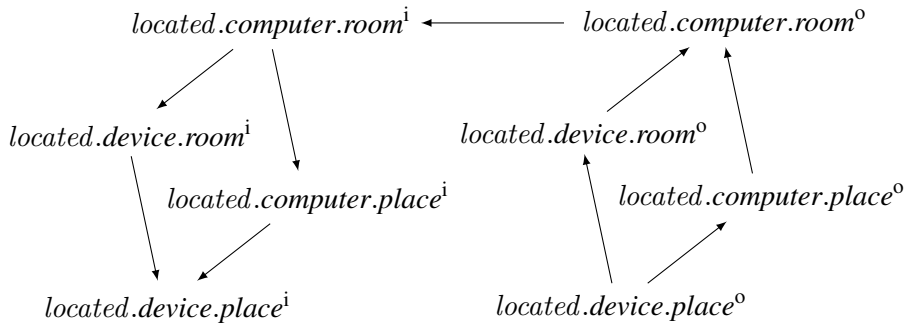
For each rule $\rightarrow \mathit{type}(t,\ \tau_0)$ such that τ_0 is a global subclass of τ we encode all the instantiation on $\mathrm{Descr}\,(d)$ of the following rule:

$$\forall \mathrm{X}, \mathrm{C}, \mathrm{D} \qquad \mathit{type}(\mathrm{X}, \mathrm{C}) \wedge \mathit{subclassof}\,(\mathrm{C}, \mathrm{D}) \rightarrow \mathit{type}(\mathrm{X}, \mathrm{D})$$

this way:

$$\begin{aligned}\mathrm{Code}\,(\mathit{subclassof}\,(\tau_1,\ \tau_2)\,|\!\rightarrow \mathit{type}(t,\ \tau_0), \tau_0 \sqsubseteq \tau_1\,) = \{\ & \mathit{type}.\tau_2.\mathbb{T}^{\mathrm{o}} \rightarrow \mathit{type}.\tau_1.\mathbb{T}^{\mathrm{o}}, \\ & \mathit{type}.\tau_1.\mathbb{T}^{\mathrm{i}} \rightarrow \mathit{type}.\tau_2.\mathbb{T}^{\mathrm{i}}, \\ & \mathit{type}.\tau_0.\mathbb{T}^{\mathrm{o}} \rightarrow \mathit{type}.\tau_0.\mathbb{T}^{\mathrm{i}}\ \}\end{aligned}$$

(a) The taxonomy used for the schema



(b) Partial encoding of the taxonomy 4.2(a) in the context of the predicate *located*

Figure 4.2: Example of the encoding of a taxonomy

It is important to pay attention to the fact that rules for the in-propositional variables and for the out-propositional variables are opposite. For instance when a first-order rule conclude on a object typed say *postscript*, it will satisfy any query on objects of type *file* which is a super-class of *postscript*. In the propositional encoding the following rule accounts for it:

$$type.file.\mathbb{T}^{o} \rightarrow type.postscript.\mathbb{T}^{o}$$

The same way, when a first-order rule has an object typed *file* in its conditions, then any object typed *postscript* which is a subclass of *file* would satisfy the request. This corresponds to the following rule in the propositional encoding:

$$type.postscript.\mathbb{T}^{i} \rightarrow type.file.\mathbb{T}^{i}$$

When other predicates appear in $\mathrm{Descr}\,(d)$ we encapsulate the subclass relation in the propositional encoding in a similar way. More precisely, for each predicate π we define:

$$\mathrm{Code}\left(\rightarrow subclassof\,(\tau_1,\ \tau_2)\ \middle|\ \begin{array}{l}\{\ \pi(t_d, t_r),\\ \quad type(t_d,\ \tau_d),\ \tau_d \sqsubseteq \tau_1,\\ \quad type(t_r,\ \tau_r),\ \tau_r \sqsubseteq \alpha\ \ \}\end{array}\right) = \begin{array}{l}\{\ \pi.\tau_1.\alpha^{i} \rightarrow \pi.\tau_2.\alpha^{i},\\ \quad \pi.\tau_2.\alpha^{o} \rightarrow \pi.\tau_1.\alpha^{o},\\ \quad \pi.\tau_d.\tau_r{}^{o} \rightarrow \pi.\tau_d.\tau_r{}^{i}\ \}\end{array}$$

$$\mathrm{Code}\left(\rightarrow subclassof\,(\tau_1,\ \tau_2)\ \middle|\ \begin{array}{l}\{\ \pi(t_d, t_r),\\ \quad type(t_d,\ \tau_d),\ \tau_d \sqsubseteq \alpha,\\ \quad type(t_r,\ \tau_r),\ \tau_r \sqsubseteq \tau_1\ \}\end{array}\right) = \begin{array}{l}\{\ \pi.\alpha.\tau_1{}^{i} \rightarrow \pi.\alpha.\tau_2{}^{i},\\ \quad \pi.\alpha.\tau_2{}^{o} \rightarrow \pi.\alpha.\tau_1{}^{o},\\ \quad \pi.\tau_d.\tau_r{}^{o} \rightarrow \pi.\tau_d.\tau_r{}^{i}\ \}\end{array}$$

Finally, the encoding of a rule $\rightarrow subclassof\,(\tau_1,\ \tau_2)$ in the context of $\mathrm{Descr}\,(d)$ is obtained bu taking the union of all the above encoding:

$$\mathrm{Code}\,(\rightarrow subclassof\,(\tau_1,\ \tau_2)\,|\mathrm{Descr}\,(d)\,) = \bigcup_{A \subset \mathrm{Descr}(d)} \mathrm{Code}\,(\rightarrow subclassof\,(\tau_1,\ \tau_2)\,|A)$$

Figure 4.2(b) shows the encoding of the subclass relations in the taxonomy of figure 4.2(a) in the context of a description containing

$$located(\texttt{pc}_1,\ \texttt{roomD}),\ type(\texttt{pc}_1,\ computer),\ type(\texttt{roomD},\ room)$$

The propositional rules are represented as edges in a graph where vertices are propositional variables and arrows between two propositional variables a and b denotes a rule $a \rightarrow b$.

Encoding of a whole description

The encoding of the description of a device d is the union of the encoding of its rules and the encoding of its taxonomy description.

$$\text{Code}\,(\text{Descr}\,(d)) = \bigcup_{R \in R_0(d)} \text{Code}\,(R\,|T_0(d)\,) \cup \bigcup_{C \in C_0(d)} \text{Code}\,(C) \cup \bigcup_{S \in S_0(d)} \text{Code}\,(S\,|\text{Descr}\,(d)\,)$$

Afterwards, we will note Δ the union of the encoding of the description of all devices.

$$\Delta = \bigcup_{d \in D} \text{Code}\,(\text{Descr}\,(d))$$

Encoding of queries

A query is defined by a set of rules. Thus, the encoding of a query is the union of the encoding of each rule from its definition:

$$\text{Code}\,(Q) = \bigcup_{R \in \text{Def}(Q)} \text{Code}\,(R)$$

4.2.2 Properties

The following properties of the encoding are going to be useful to the transfer property.

Property 1 : This property shows that for a predicate offered in conclusion of a first-order rule, the associated out-propositional variable entails the corresponding in-propositional variable. It expresses the relay race composition of functionalities in the propositional side.

If there exists $r \in \text{Descr}\,(d)$ such that $\pi(t_1, t_2) \in \text{Conclusion}\,(r)$ with τ_1 (resp. τ_2) being the type of t_1 (resp. t_2), then

$$\Delta, \pi.\tau_1.\tau_2{}^{\text{o}} \models \pi.\tau_1.\tau_2{}^{\text{i}}$$

Proof:

Let θ_1 be a strict super-class of τ_1. Then

$$\text{Code}\,(subclassof\,(\tau_1,\ \theta_1)\,|\text{Descr}\,(d)\,) \in \Delta$$

Now,

$$\begin{gathered} \pi(t_1, t_2) \in \text{Descr}\,(d) \\ type(t_1,\ \tau_1) \in \text{Descr}\,(d) \\ type(t_2,\ \tau_2) \in \text{Descr}\,(d) \\ subclassof\,(tau_1,\ \theta_1) \in \text{Descr}\,(d) \end{gathered}$$

So, from the definitions we have:

$$\pi.\tau_1.\tau_2{}^{\mathrm{o}} \rightarrow \pi.\tau_1.\tau_2{}^{\mathrm{i}} \in \mathrm{Code}\,(\mathit{subclassof}(\tau_1,\ \theta_1)\,|\{\pi(t_1,t_2), \mathit{type}(t_1,\ \tau_1)\}\,)$$

$\diamondsuit$

Property 2 : We extend the previous property throughout subclass relations.
Let τ_1, τ_2, θ_1, θ_2 be class names such that $\tau_1 \sqsubseteq \theta_1$ or $\theta_1 \sqsubseteq \tau_1$, and $\tau_2 \sqsubseteq \theta_2$ or $\theta_2 \sqsubseteq \tau_2$. Then,

$$\Delta, \pi.\tau_1.\tau_2{}^{\mathrm{o}} \models \pi.\theta_1.\theta_2{}^{\mathrm{i}}$$

Proof:

First case, $\tau_1 \sqsubseteq \theta_1$ and $\tau_2 \sqsubseteq \theta_2$:
From the definition of the encoding we have

$$\Delta, \pi.\tau_1.\tau_2{}^{\mathrm{o}} \models \pi.\tau_1.\tau_2{}^{\mathrm{i}}$$

and $$\Delta, \pi.\tau_1.\tau_2{}^{\mathrm{i}} \models \pi.\theta_1.\theta_2{}^{\mathrm{i}}$$ Q.E.D.

Second case, $\theta_1 \sqsubseteq \tau_1$ and $\theta_2 \sqsubseteq \tau_2$:
From the definition of the encoding we have

$$\Delta, \pi.\tau_1.\tau_2{}^{\mathrm{o}} \models \pi.\theta_1.\theta_2{}^{\mathrm{o}}$$

and $$\Delta, \pi.\theta_1.\theta_2{}^{\mathrm{o}} \models \pi.\theta_1.\theta_2{}^{\mathrm{i}}$$ Q.E.D.

Third case, $\theta_1 \sqsubseteq \tau_1$ and $\tau_2 \sqsubseteq \theta_2$:
From the definition of the encoding we have

$$\Delta, \pi.\tau_1.\tau_2{}^{\mathrm{o}} \models \pi.\theta_1.\tau_2{}^{\mathrm{o}}$$

and $$\Delta, \pi.\theta_1.\tau_2{}^{\mathrm{o}} \models \pi.\theta_1.\tau_2{}^{\mathrm{i}}$$

and $$\Delta, \pi.\theta_1.\tau_2{}^{\mathrm{i}} \models \pi.\theta_1.\theta_2{}^{\mathrm{i}}$$ Q.E.D.

The last case and the previous one are symmetrical.

$\diamondsuit$

Property 3 : Let A be a conjunction of atoms, T a set of typing atoms and σ any substitution. Then

$$\Delta, \mathrm{Code_o}\,(A\,|T\,) \models \mathrm{Code_o}\,(A\sigma\,|T\,)$$
$$\Delta, \mathrm{Code_i}\,(A\sigma\,|T\,) \models \mathrm{Code_i}\,(A\,|T\,)$$

Proof:

For an atom $\pi(t_1, t_2)$, where π is not the predicate type, as we treat terms in the encoding regardless of the presence of variables, we have the equality between encoding with or without substitutions. Thus, for the atom $\pi(t_1, t_2)$, the property is straightforward.

For an atom $\mathit{type}(t,\ \mathrm{X})$ where the class name is a variable, a substitution may affect the variable X. And as $X\sigma \sqsubseteq \mathit{top}$ and

$$\mathrm{Code_o}\,(\mathit{type}(t,\ \mathrm{X})\,|T\,) = \mathit{type}.\mathit{top}.\mathbb{T}^{\mathrm{o}}$$
$$\mathrm{Code_o}\,(\mathit{type}(t,\ X\sigma)\,|T\,) = \mathit{type}.X\sigma.\mathbb{T}^{\mathrm{o}}$$
$$\mathrm{Code_i}\,(\mathit{type}(t,\ \mathrm{X})\,|T\,) = \mathit{type}.\mathit{top}.\mathbb{T}^{\mathrm{i}}$$

$$\mathrm{Code_i}\,(\mathit{type}(t,\ X\sigma)\,|T\,) = \mathit{type}.X\sigma.\mathbb{T}^{\mathrm{i}}$$

we have the result.

$\diamondsuit$

Property 4 : Let A, B and C be conjunctions of atoms. Then

$$\mathrm{Code_i}\,(A \wedge B\,|C\,) = \mathrm{Code_i}\,(A\,|B \cup C\,) \wedge \mathrm{Code_i}\,(B\,|A \cup C\,)$$

Proof:

Straightforward with the definition of the encoding for conjunction of atoms.

$\diamondsuit$

Property 5 : Let A, B and C be conjunctions of atoms. Then

$$\Delta, \mathrm{Code_i}\,(A\,|B \cup C\,) \models \mathrm{Code_i}\,(A\,|C\,)$$

Proof:

Let us prove it for an atom a and two conjunctions of atoms B and C.
First case: $a = \mathit{type}(t,\ \tau)$. In this case we have,

$$\mathrm{Code_i}\,(\mathit{type}(t,\ \tau)\,|B \cup C\,) = \mathrm{Code_i}\,(\mathit{type}(t,\ \tau)\,|C\,)$$

Second case: $a = \pi(t_1, t_2)$, with θ_1 (resp. θ_2) the domain (resp. the range) of the predicate π.

Sub-case 1:
for all k if $\quad \mathit{type}(t_k,\ \tau_k) \in C$
or $\quad \mathit{type}(t_k,\ \mathrm{X}), \mathit{subclassof}(\mathrm{X},\ \tau_k) \in C$
then $\quad \mathrm{Code_i}\,(\pi(t_1, t_2)\,|B \cup C\,) = \pi.\tau_1.\tau_2{}^{\mathrm{i}} = \mathrm{Code_i}\,(\pi(t_1, t_2)\,|C\,)$

Sub-case 2:
for all k if $\quad \mathit{type}(t_k,\ \tau_k) \notin B \cup C$
or $\quad \mathit{type}(t_k,\ \mathrm{X}), \mathit{subclassof}(\mathrm{X},\ \tau_k) \notin B \cup C$
then $\quad \mathrm{Code_i}\,(\pi(t_1, t_2)\,|B \cup C\,) = \pi.\theta_1.\theta_2{}^{\mathrm{i}} = \mathrm{Code_i}\,(\pi(t_1, t_2)\,|C\,)$

Sub-case 3:
for all k if $\quad \mathit{type}(t_k,\ \tau_k) \in B \setminus C$
or $\quad \mathit{type}(t_k,\ \mathrm{X}), \mathit{subclassof}(\mathrm{X},\ \tau_k) \in B \setminus C$
then $\quad \mathrm{Code_i}\,(\pi(t_1, t_2)\,|B \cup C\,) = \pi.\tau_1.\tau_2{}^{\mathrm{i}}$
$\quad \mathrm{Code_i}\,(\pi(t_1, t_2)\,|C\,) = \pi.\theta_1.\theta_2{}^{\mathrm{i}}$

Sub-case 4:
if $\quad \mathit{type}(t_1,\ \tau_1) \in B \setminus C$
or $\quad \mathit{type}(t_1,\ \mathrm{X}), \mathit{subclassof}(\mathrm{X},\ \tau_1) \in B \setminus C$
and
if $\quad \mathit{type}(t_2,\ \tau_2) \in C$
or $\quad \mathit{type}(t_2,\ \mathrm{X}), \mathit{subclassof}(\mathrm{X},\ \tau_2) \in C$
then $\quad \mathrm{Code_i}\,(\pi(t_1, t_2)\,|B \cup C\,) = \pi.\tau_1.\tau_2{}^{\mathrm{i}}$
$\quad \mathrm{Code_i}\,(\pi(t_1, t_2)\,|C\,) = \pi.\theta_1.\tau_2{}^{\mathrm{i}}$

Sub-case 5: Similar to the sub-case 4
Finally, as for all k we have $\tau_k \sqsubseteq \theta_k$ we can conclude for an atom a:

$$\Delta, \mathrm{Code_i}\,(a\,|B \cup C\,) \models \mathrm{Code_i}\,(a\,|C\,)$$

The extension for a conjunction of atoms is straightforward.

$\diamondsuit$

Property 6 : Let A, B and C be a conjunction of atoms. Then

$$\Delta, \mathrm{Code_i}\,(A \wedge B\,|C\,) \models \mathrm{Code_i}\,(A\,|C\,) \wedge \mathrm{Code_i}\,(B\,|C\,)$$

Proof:

This property is a corollary of the property 4 and property 5.

$\diamondsuit$

The definition 14 introduces the notion of proper prime implicants which is central for the transfer propoerty.

Definition 14 : *Proper prime implicant with respect to a theory.* Let Γ be a clausal theory and q be a cube (conjunction of literals). A cube m is said to be:

- a prime implicant of q with respect to Γ if and only if $\Gamma, m \models q$ and for any other cubes m', if $\Gamma, m' \models q$ and $m \models m'$ then $m \equiv m'$.
- a proper prime implicant of q with respect to Γ if and only if it is a prime implicant of q with respect to Γ and $\Gamma \not\models \neg m$.

The set of proper prime implicants of q with respect to Γ is denoted by

$$\mathrm{Ppi}\,(q, \Gamma)$$ $\square$

Theorem 2 : *Transfer property*
Let Q be a query, D a set of devices and Δ the union of the encoding of the descriptions of all device in D. Let $\bar{t}$ an answer to Q, and $P = (r_1, \ldots, r_n)$ be a success path of the derivation tree of $Q(\bar{t})$. Let d_i be the device that stores the rule r_i.

Then, $\left(\bigwedge_{i=1}^{n} d_i^{\star}\right)$ is a proper prime implicant of $\mathrm{Code}\,(Q)$ with respect to Δ

Proof:

Let us proceed by induction on the length of the path $(r_1, \ldots, r_n)$. We consider the propositional variable q, the encoding of the query Q

Basis:
Let us assume that the length of the path P is 1. It means that $Q(\bar{t})$ unifies with a fact $F = \pi(t_1, t_2)$ stored in the device d_1, where the type of t_1 is τ_1 and the type of t_2 is τ_2.

$$\{\pi(t_1, t_2), \mathit{type}(t_1,\ \tau_1), \mathit{type}(t_2,\ \tau_2)\} \in \mathrm{Descr}\,(d_1)$$

Thus $\mathrm{Code}\,(Q)$ is of the form $\pi.\theta_1.\theta_2{}^{\mathrm{i}}$ and as $Q(\bar{t})$ unifies with F we have $\tau_1 \sqsubseteq \theta_1$ and $\tau_2 \sqsubseteq \theta_2$.

Then, with the property 2, we have:

$$\Delta, \pi.\tau_1.\tau_2{}^{\mathrm{i}} \models \pi.\theta_1.\theta_2{}^{\mathrm{i}}$$

It exists in the encoding of the description of d_1 the following rule $\pi.\tau_1.\tau_2{}^{\mathrm{o}} \rightarrow \pi.\tau_1.\tau_2{}^{\mathrm{i}}$.

Then $$\Delta, \pi.\tau_1.\tau_2{}^{\mathrm{o}} \models \pi.\theta_1.\theta_2{}^{\mathrm{i}}$$

and as F belongs to $\mathrm{Descr}\,(d_1)$, the following rule $d_1^\star \rightarrow \pi.\tau_1.\tau_2{}^{\mathrm{o}}$ belongs also to Δ, so:

$$\Delta, d_1^\star \models \pi.\tau_1.\tau_2{}^{\mathrm{o}}$$

and finally $$\Delta, d_1^\star \models \mathrm{Code}\,(Q)$$

Therefore, d_1 is the device that stores the rule r and $d^\star$ is a proper prime implicant of $\mathrm{Code}\,(Q)$ with respect to Δ.

Inductive step:

Let us assume that for a certain integer n, for any query Q and for any success paths $P = (r_1, \ldots, r_n)$ in the derivation tree of $Q(\bar{t})$ so that the length of P is equal to n, if $d_i = \mathrm{Descr}^{-1}\,(r_i)$, we have that

$$\left(\bigwedge_{i=1}^{n} d_i^\star\right) \in \mathrm{Ppi}\,(\mathrm{Code}\,(Q)\,, \Delta)$$

Let $P = (r_1, \ldots, r_{n+1})$ be a success path of length $n+1$ in the derivation tree of $Q(\bar{t})$. Let $A_1, \ldots, A_n$ the set of atoms of the successor of the root node in the success path. Let us consider $\mathrm{Code}\,(\bigwedge_{i=1}^{n} A_i)$. Afterwards, $d_i = \mathrm{Descr}^{-1}\,(r_i)$.

We have $$r_1 = F' \wedge T' \rightarrow F \wedge C$$

and

$\mathrm{Code}\,(r_1\,|T_0(d_1)) =$

$$d_1^\star \wedge \mathrm{Code_i}\,(F' \wedge T'\,|T_0(d_1)) \rightarrow \mathrm{Code_o}\,(F \wedge C\,|T_0(d_1) \cup T')$$

in particular, one of the atoms in the conclusion of r_1 (say $F_1 = \pi(t_1, t_2)$) unifies with A_1 (say $\pi(u_1, u_2)$) and their MGU is σ. Then after the deletion of A_1 by the rule r_1 we have

$$F'\sigma, T'\sigma, A_2\sigma, \ldots, A_n\sigma$$

Moreover, in our model everything is typed at least implicitly, for each i, in $F'\sigma, T'\sigma, A_2\sigma, \ldots, A_n\sigma$, either there is $\mathit{type}(t_i\sigma,\ \tau_i)$ and $\mathit{type}(u_i\sigma,\ \tau_i')$ or they are implicit. As it is a success path, both hold. And as $u_i\sigma = t_i\sigma$, it results that it is the same object that is typed twice. So for each i either $\tau_i \sqsubseteq \tau_i'$ or $\tau_i' \sqsubseteq \tau_i$. Thus, in the propositional side, the property 1 assures that it exists a set of rules in Δ such that:

$$\Delta, \mathrm{Code_o}\,(F_1\sigma\,|T' \cup C \cup T_0(d_1)) \models \mathrm{Code_i}\,(A_1\sigma\,|\textstyle\bigcup_{i=2}^{n} A_i)$$

With the property 3 we can remove the substitution and we have:

$$\Delta, \mathrm{Code_o}\,(F_1\,|T' \cup C \cup T_0(d_1)\,) \models \mathrm{Code_o}\,(F_1\sigma\,|T' \cup C \cup T_0(d_1)\,)$$
$$\Delta, \mathrm{Code_i}\,(A_1\sigma\,|\bigcup_{i=2}^{n} A_i\,) \models \mathrm{Code_i}\,(A_1\,|\bigcup_{i=2}^{n} A_i\,)$$

and because $\mathrm{Code}\,(r_1\,|T_0(d_1)\,)$ belongs to Δ, we have

$$\Delta, d_1^\star \wedge \mathrm{Code_i}\,(F' \wedge T'\,|T_0(d_1)\,) \models \mathrm{Code_o}\,(F_1\,|T' \cup C \cup T_0(d_1)\,)$$

and so, $$\Delta, d_1^\star \wedge \mathrm{Code_i}\,(F' \wedge T'\,|T_0(d_1)\,) \models \mathrm{Code_i}\,(A_1\,|\bigcup_{i=2}^{n} A_i\,)$$

Now according to the property 4 we have

$$\mathrm{Code_i}\,(A_1\,|\bigcup_{i=2}^{n} A_i\,) \wedge \mathrm{Code_i}\,(\bigwedge_{i=2}^{n} A_i\,|A_1\,) = \mathrm{Code_i}\,(\bigwedge_{i=1}^{n} A_i)$$

Therefore, $\mathrm{Code_i}\,(F' \wedge T'\,|T_0(d_1)\,) \wedge \mathrm{Code_i}\,(\bigwedge_{i=2}^{n} A_i\,|A_1\,) \wedge d_1^\star$ is an implicant of $\mathrm{Code_i}\,(\bigwedge_{i=1}^{n} A_i)$ with respect to Δ.

Now, the set of atoms $F'\sigma, T'\sigma, A_2\sigma, \ldots, A_n\sigma$ resulting from the deletion of A_1 is encoded into the set of propositional variables

$$\mathrm{Code_i}\,(F'\sigma \wedge T'\sigma \wedge \bigwedge_{i=2}^{n} A_i\sigma)$$

According to the property 3 we have:

$$\mathrm{Code_i}\,(F'\sigma \wedge T'\sigma \wedge \bigwedge_{i=2}^{n} A_i\sigma) = \mathrm{Code_i}\,(F' \wedge T' \wedge \bigwedge_{i=2}^{n} A_i)$$

According to the property 6 we have:

$$\Delta, \mathrm{Code_i}\,(F' \wedge T' \wedge \bigwedge_{i=2}^{n} A_i) \models \mathrm{Code_i}\,(F' \wedge T') \wedge \mathrm{Code_i}\,(\bigwedge_{i=2}^{n} A_i)$$

Until now we have proved the following:

$$\Delta, d_1^\star, \mathrm{Code_i}\,(F'\sigma \wedge T'\sigma \wedge \bigwedge_{i=2}^{n} A_i\sigma) \models \mathrm{Code_i}\,(\bigwedge_{i=1}^{n} A_i)$$

Now, $P' = (r_2, \ldots, r_{n+1})$ the sub-path of P satisfies the inductive hypothesis. Thus, we have: $\left(\bigwedge_{i=2}^{n+1} d_i^\star\right)$ is a proper prime implicant of $\mathrm{Code}\,(F'\sigma \wedge T'\sigma \wedge \bigwedge_{i=2}^{n} A_i\sigma)$ with respect to Δ ■

4.2.3 Examples

An example with two computers $\mathtt{pc_1}$ — which is located in a room `roomD` — and $\mathtt{srv_1}$ — which is located in a building `batC` — is described in table 4.1. The encoding of their description is represented figure 4.3 by a graph where vertices are propositional variables and arrows between two propositional variables a and b denotes a propositional rule $a \rightarrow b$ in the encoding. From the example, a query Q — asked to the device d_1 — in logic can be:

$$\mathit{located}(\mathrm{X},\ \mathrm{Y}) \wedge \mathit{type}(\mathrm{Y},\ \mathit{room}) \rightarrow Q(\mathrm{X})$$

and its encoding:

$$d_1^\star \wedge \mathit{located.device.room}^{\mathrm{i}} \rightarrow q$$

Figure 4.4 shows the derivation tree of the above first-order query Q with respect to the first-order description shown in table 4.1. Thus, using the encoding, the set of device propositional variable implicants

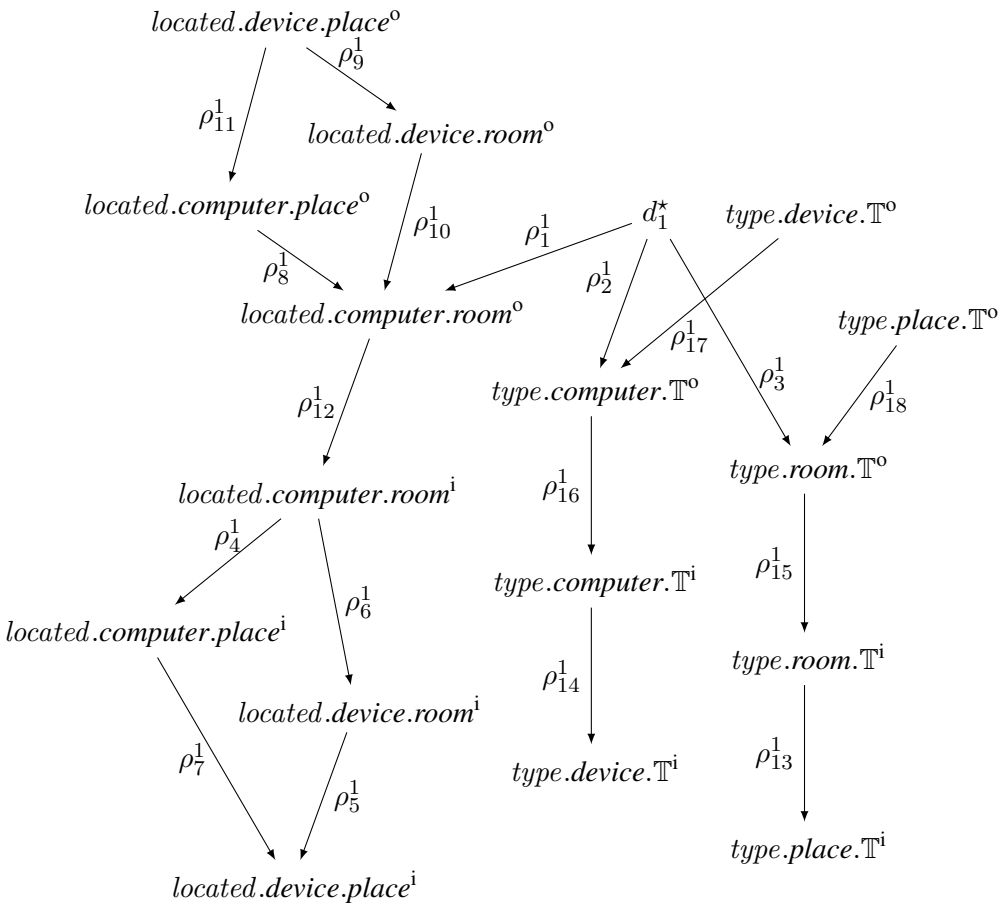


Figure 4.3: The encoding of a description shown in a graph

Table 4.1: Descriptions of two devices and their encoding

(a) Rules that enable the inheritance of types

R_0: $\mathit{subclassof}(\mathrm{C},\ \mathrm{D}), \mathit{type}(\mathrm{X},\ \mathrm{C}) \rightarrow \mathit{type}(\mathrm{X},\ \mathrm{D})$

(b) Description of a computer

R_1^1:	$\mathit{type}(\mathtt{pc_1},\ \mathit{computer})$
R_2^1:	$\mathit{type}(\mathtt{roomD},\ \mathit{room})$
R_3^1:	$\mathit{subclassof}(\mathit{room},\ \mathit{place})$
R_4^1:	$\mathit{subclassof}(\mathit{computer},\ \mathit{device})$
R_5^1:	$\mathit{located}(\mathtt{pc_1},\ \mathtt{roomD})$

(c) Description of a server

R_1^2:	$\mathit{type}(\mathtt{srv_1},\ \mathit{computer})$
R_2^2:	$\mathit{type}(\mathtt{batC},\ \mathit{building})$
R_3^2:	$\mathit{subclassof}(\mathit{building},\ \mathit{place})$
R_4^2:	$\mathit{subclassof}(\mathit{computer},\ \mathit{device})$
R_5^2:	$\mathit{located}(\mathtt{srv_1},\ \mathtt{batC})$

of the encoding of Q is $d_1^\star$ as shown in figure 4.5

4.3 Look-up by decentralised propositional reasoning

We exploit the transfer property of the propositional encoding to retrieve a (possibly strict) subset of the relevant devices for a query Q. In section 4.3.1 we show (Theorem 3) that such a subset, denoted $\mathrm{Lookup}\,(Q)$ can be obtained by decoding the proper prime implicant of the encoding of Q that are made of device propositional variables only. In section 4.3.2, we show how to use SOMEWHERE for a decentralised implementation of the computation of $\mathrm{Lookup}\,(Q)$.

4.3.1 Look-up: definition and properties

Definition 15 : Let $\mathrm{Ppi}^\star\,(q, \Delta)$ be the set of proper prime implicants of q made of device propositional variables.

$$\mathrm{Ppi}^\star\,(q, \Delta) = \{\bigwedge_{i=1}^{n} d_i^\star \in \mathrm{Ppi}\,(q, \Delta)\}$$

Let $\mathrm{Lookup}\,(Q)$ be the set of devices decoded from $\mathrm{Ppi}^\star\,(\mathrm{Code}\,(Q)\,, \Delta)$.

$$\mathrm{Lookup}\,(Q) \;=\; \bigcup_{P \in \mathrm{Ppi}^\star(\mathrm{Code}(Q), \Delta)} \mathrm{Decode}\,(P) \qquad \square$$

The following theorem states that the look up method consisting in computing $\mathrm{Lookup}\,(Q)$ is sound and complete. That is to say, for a query Q, an answer is found using only the description of the devices of $\mathrm{Lookup}\,(Q)$ if and only if it is a correct answer that would have been found also using the whole set of available devices.

Theorem 3 : Let Q be a query and D a set of devices

$$\mathrm{Answer}\,(Q,\ D) = \mathrm{Answer}\,(Q,\ \mathrm{Lookup}\,(Q))$$

Proof:

As the $\mathrm{Lookup}\,(Q)$ is a subset of the devices, the lookup is sound.
The other inclusion of the theorem can be rewritten as:

$$\forall \bar{t} \qquad \mathrm{Descr}\,(D) \models Q(\bar{t}) \quad \rightarrow \quad \mathrm{Descr}\,(\mathrm{Lookup}\,(Q)) \models Q(\bar{t})$$

Let us consider the derivation tree of $Q(\bar{t})$. Let $P = (r_1, \ldots, r_n)$ be the success path and d_i the device that stores the rule r_i in its description. Then according to the theorem 2

$$\mathrm{Code}\left(\bigcup_{i=1}^{n} d_i\right) \in \mathrm{Ppi}^\star\,(\mathrm{Code}\,(Q)\,, \Delta)$$

From the definition of Lookup we deduce:

$$\bigcup_{i=1}^{n} d_i \in \mathrm{Lookup}\,(Q)$$

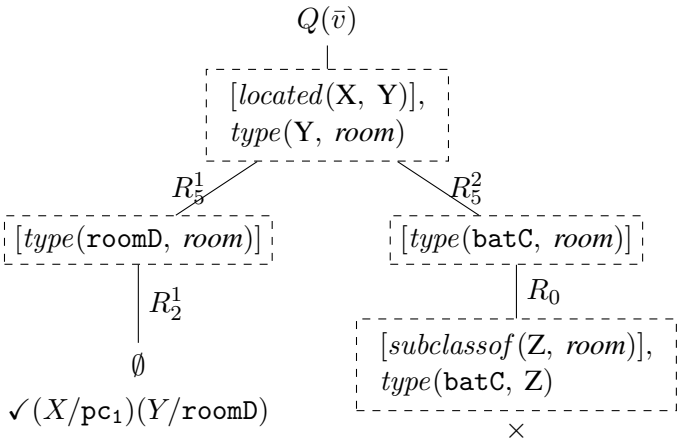


Figure 4.4: Derivation tree of the query Q with respect to the rules in table 4.1

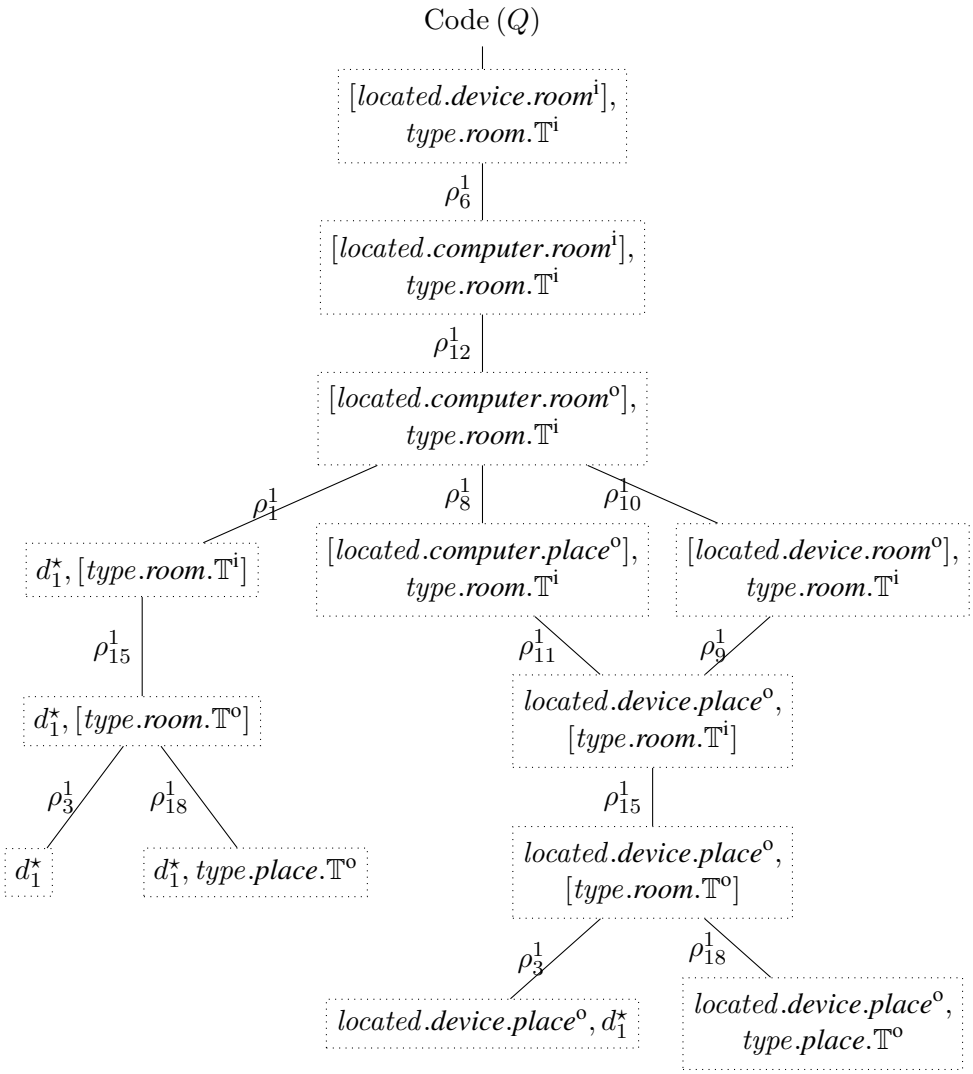


Figure 4.5: Propositional implicants of Code (Q) with respect to the encoding of the rules

and thus, $$\text{Descr}\left(\bigcup_{i=1}^{n} d_i\right) \subset \text{Descr}\left(\text{Lookup}\left(Q\right)\right)$$

Now we are in a success path so we have

$$\text{Descr}\left(\bigcup_{i=1}^{n} d_i\right) \models Q(\bar{t})$$

And therefore, $$\text{Descr}\left(\text{Lookup}\left(Q\right)\right) \models Q(\bar{t})$$ ■

The important point is that, while the look-up method is complete, $\text{Lookup}\left(Q\right)$ can provide a strict subset of $\text{Relevant}\left(Q,\ D\right)$. It is shown using the example in table 4.1. From figure 4.4 we see that $\text{Relevant}\left(Q,\ D\right)$ is the set of d_1 and d_2 — because the rules appearing in the label of the edges of the derivation tree come from d_1 and d_2 — while in figure 4.5, we can see that $\text{Lookup}\left(Q\right)$ is the singleton d_1 — because $d_1^\star$ is the only proper prime implicants of the query made of device propositional variables.

4.3.2 Decentralised computation of $\text{Lookup}\left(Q\right)$ using SOMEWHERE

Figure 4.6 illustrates the possible implementation of the look-up method in a decentralised setting by adding layers on top of SOMEWHERE deployed in each device. the process in the centralised setting. The top layer, deals with the user interface. The step 1 (resp. step A) represents the translation of the description (resp. a query) from the high-level language to the first-order logic. The intermediate layer deals with the first-order logic description and reasoner. The step 2 (resp. step B) represents the encoding

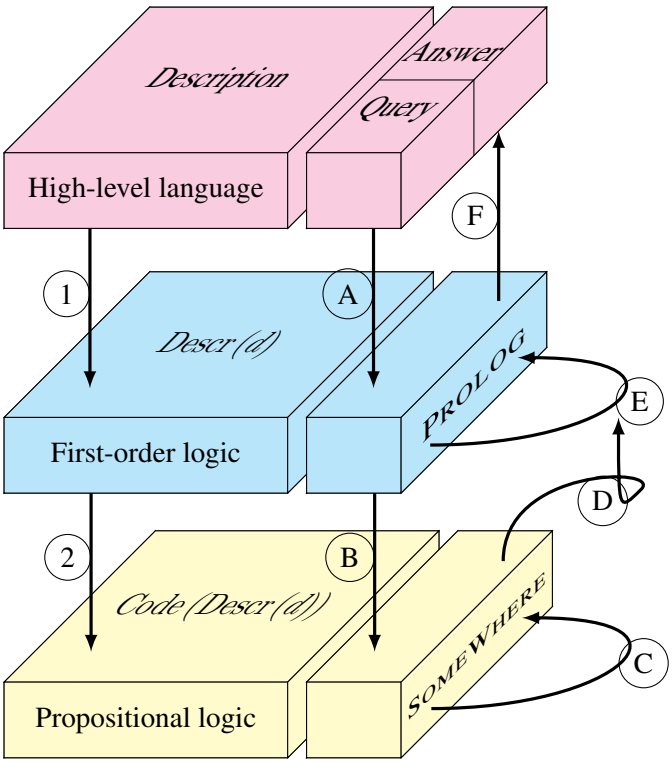


Figure 4.6: Schema of the whole process in a device

of first-order description (resp. first-order query). The bottom layer is the propositional logic encoding and reasoner handled by SOMEWHERE.

SOMEWHERE is a fully decentralised propositional logic reasoner. It is deployed on each device of the opportunistic network in a form of a piece of JAVA code. The SOMEWHERE layer computes the proper prime implicants of a conjunction of propositional variables with respect to a decentralised propositional theory. In our context, the conjunction of propositional variables is the encoded query and the propositional theory is the global theory Δ. Some of the propositional variables can be defined in SOMEWHERE as *target variables* and we can restrict the computation of proper prime implicants to those target variables. In our context, device propositional variables are the target variables. Therefore, the look-up can be achieved by the SOMEWHERE platform (step C).

According to the transfer property (Theorem 2 in section 4.3.1), the decoding of the result returned by SOMEWHERE triggered on the queried device provides the result of $\mathrm{Lookup}\,(Q)$. Therefore, the queried device can download the first-order description of the devices selected by the look-up process (step D), and locally computes using the PROLOG engine the answers to the query with respect to to the selected descriptions (step E).

4.4 Heterogeneous descriptions

So far in this chapter, we have assumed that the descriptions between devices were homogeneous, *i.e.* using a common taxonomy shared by every devices. We describe in this section what changes must be done in the encoding and the look-up process if we allow heterogeneous descriptions. Throughout this section, we assume that class names appearing in the taxonomy of a device are prefixed with the identifier of that device, so that similar class names between taxonomies are disambiguated.

Mappings between heterogeneous taxonomies have been introduced in section 2.7. They establish a subclass relation between class names in the taxonomies of two different devices. In a decentralised setting, we just have to add the mappings in the device description: in the description of a device d, we add the mappings, denoted by $M_0(d)$ the subset of $\mathrm{Descr}\,(d)$ made of the predicate $subclassof$ and contains foreign class names. $M_0(d)$ is a subset of $S_0(d)$ and therefore it is encoded the same way. Therefore, the lookup method presented in section 4.3 based on propositional encoding applies without changes to the case of heterogeneous descriptions.

Bernard de Chartres said that we are like dwarfs sitting on the shoulders of giants. We see more, and things that are more distant, than they did, not because our sight is superior or because we are taller than they, but because they raise us up, and by their great stature add to ours.

Metalogicon III, Jean de Salisbury (1115–1180)

Chapter 5

Related Work and Conclusion

Contents

5.1 Semantic Approach of Dynamic Web Services Composition **64**

5.2 Web Service Composition via Planning **67**

5.3 Logic based language for Web services composition **68**

5.4 Conclusion **71**

We focus this chapter on comparing our work to previous existing work on dynamic Web service composition. A Web service [7] is an application that can be described, published, and invoked over the network by using standards network technologies. Those characteristics match the characteristics of devices and functionalities. In our model Web services can be treated as logical devices while embedded services are equivalent to functionalities.

In the last decade, Web service composition problems and Web service discovery problems have been addressed (*e.g.* see [8, 19] for surveys). The Web service discovery problem is to fulfil a request of a Web service that has initial input parameters and desired output. The Web service composition problem occurs when no single Web service can satisfy the user request whereas a composite Web service— made of a combination of either simple or composite Web services — could satisfy it. To achieve this goal, a large number of research works has been carried out :

- using planning [15, 25];
- given a model to express composition between Web services to verify compositions [10, 18];
- to effectively build composition of Web services down to a lower granularity of the descriptions.

Some works (*e.g.* [4, 6, 25]) compose Web services by mingling their behaviour and execution, which needs to describe the Web services at a low level.

Several works on Web service composition are based on predefined patterns [10, 26] or bindings between services that are known beforehand [3]. A more challenging problem, which is the one we address, is to compose services dynamically, on demand. Few works address it. We structure this chapter accordingly, by first situating our approach with respect to semantic approaches, then with respect to approaches based on planning and finally with respect to logic-based approaches. Note that those three approaches are not disjoint and some of the works detailed here can be ranked among more than one section.

5.1 Semantic Approach of Dynamic Web Services Composition

The semantic approach aims to give resources well-defined meaning and to make them machine-understandable. Then, resources can be shared and processed by automated tools as well as by human users. The introduction of semantics in Web service composition is the continuity of the Semantic Web, which vision was first introduced by [5].

The semantics is described by standard languages including :

- Uniform Resource Identifier (URI) a fundamental component of current Web. It allows to uniquely identify resources as well as relations among resources;
- eXtensible Markup Language (XML) provides a syntactical interoperability on Web. It is the universal format for structured documents and data on the Web;
- RDF leverages URI and XML to allow documents being described in the form of metadata by means of resources, properties, and statements;
- RDFS is an extension of RDF, which defines a simple modeling language on top of RDF. It enables the representation of class, property and constraint while RDF allows the representation of instances and facts, thus making it a lightweight ontology language. While RDF and RDFS are different, they are combined together to form the basic language for knowledge representation denoted as RDF(S);
- Web Ontology Language (OWL) use the RDF(S) as a starting point and extend it to a more expressive powered ontology specification language for the description of the semantic Web.

Our language is an extension of RDF(S) that make uses of function. In chapter 2 we expressed our language in RDF(S) whenever it was possible.

We compare our work to some previous work introducing semantics for data in Web service composition. The first one introduce the concept of semantic value, and the second builds an ontology upon which the composition is made.

In [13], Web service composition satisfy two conditions:

- Web services must agree on the meaning of the exchanged data in order to build consistent composition;
- semantic-data conflicts — such as incompatibility of types or differences in the unit used — must be resolved using context.

The data conflicts arisen by the composition make [13] resolve these conflicts by using the concept of semantic value introduced in [24]. A semantic value is the association of a simple value (like '5' as an integer) and a semantic information — named the context — to ease the contextual understanding (*e.g.* "5(currency = euro)" describes 5 units of currency of type euros). The context is a set of properties where each property is assigned to a semantic value. This definition is recursive. For instance a rent can be expressed like:

$$640\ (\textit{Periodicity} = \text{'monthly'}(\textit{FirstIssueDate} = \text{'15 Jan'}),\ \textit{Currency} = \text{'Euros'}).$$

Here, the value 640 has two properties: *Periodicity* and *Currency*. The value of the former property is also a semantic value having the property *FirstIssueDate*. The semantics of 640 can thus be interpreted as a monthly rent of 640 euros with a beginning on January 15. Upon such a definition, the concept of semantic mediator is introduced in [17]. A semantic mediator is triggered during a Web service

composition and execution and it is not embedded in any of the Web services involved in the composition. It is rather an independent component that permits conversion and comparison between semantic values.

In our approach, data are typed and can have properties (*e.g.* stored) . They may have a property that value them. For instance the taxonomy of data-type can contain "Euro" as a sub-type of "Currency" and a data object — whose value-property is for instance '5' — can be declared to be an instance of "Euro". Finally, the semantic mediator is covered by the mappings between taxonomies when there is no conversion of values or by a functionality that makes the conversion.

To make sure that different users (machine or human) have a common understanding of the terms, ontologies in which terms are described, and which establish a joint terminology are usually adopted.

For a particular form of Web services called 'Data-providing' Web services, [4] addresses the problem of composition by defining an ontology. Data-providing Web services (as opposed to Effect-providing Web services) corresponds to our functionalities without post-condition. They define an ontology by a 6-tuple:

1. a set of classes C;
2. a set of data-types L;
3. a set of data-type properties DP;
4. a set of object properties OP;
5. a sub-class relationship between classes ($C \times C$);
6. and a sub-property relationship between homogeneous properties ($OP \times OP \cup DP \times DP$).

A query is defined on the instance graph of an ontology by a 3-tuple:

1. a backbone : a sequence of the form

$$?c_1(C_1).\Psi.p_{1.2}.?c_2(C_2).\Psi.p_{2.3} \ldots \Psi.p_{n-1.n}.?c_n(C_n)$$

   where

   - c_i is a variable of type C_i, which is a class of the ontology;
   - $p_{i,j}$ is an object property involving $?c_k \quad 1 \leq k \leq n$;
   - Ψ is a linking operator. Instances of $?c_i$ must satisfy all of the conditions specified by Ψ;

2. the constraints set imposed on data-type properties of the variables in the backbone;
3. the output set including output variables.

Implemented in the RDF query language SPARQL, an example of a query that means *'For a patient P_1 who takes a medication termed "Some Stuff", what are the tests they had to perform'* could be :

$Q1:$
$Backbone =$
$?T_1(type\colon Test).[HasTest]^{-1}.?P_1(type\colon Patient).[TakeDrugs].?M_1(type\colon Drugs)$
$Ct = \{\$M_1(Name =$ "Some Stuff$'')\}$,
$Out = \{?T_1(Result)\}$

They model the Data-Providing Web services as RDF Parametrised views over an OWL ontology. A parametrised view is a predicate $WS_i(C_i)$ called the view head and a 4-tuple called the view body:

1. a backbone that includes both the variable set C (of classes types) linking input and the output of the service, and the object properties set OP relating the different variables in C;
2. A set of constraints imposed on the data-type properties of C such that they are not required inputs of the service;
3. Necessary literals for the service invocation;
4. Output literals.

The parametrised view permits to catch the semantic relation holding between the inputs (literals for the service invocation) and the outputs. For instance, a Web services that provides the name of the patients that take a certain medication is written as follow :

$$\begin{aligned} &WS4(Patient(Name)) : - \\ &\qquad \{?Patient.[TakeDrugs].?Drug\} \\ &\qquad Ct : \{\emptyset\} \\ &\qquad In : \{\$Drug(Name)\} \\ &\qquad Out : \{?Patient(Name)\} \end{aligned}$$

The resolution of the query is based on the parametrised views of the services. In order to satisfy a query, a set of services is selected and the union of their backbones has to cover the backbone of the query. It means that the requested data items are provided and the query's constraints list is satisfied with the union of the constraints of the selected services. As a matter of fact, the composition is guided by the structure of the query's backbone. The form of the composed Web services expected as an answer to the query is known beforehand. Thus, with this approach, one may fail finding composition even if there is possible composition that would provide the data items requested without satisfying the constraints of the backbone. In our approach the query is more flexible as it is expressed directly in first-order logic (without the constraint of the backbone, a sequence of a particular form) used here.

A rule based composition is done in [16] by generating composite services from high-level declarative description. Rules (called composability rules) are used to determine whether two services are likely to be composed. The approach consists in four phases:

1. Specification: a high-level description of the desired compositions using Composite Service Specification Language (CSSL), allowing semantic features and specification of the control flow between composite service operations.
2. Match-making: composability rules are used to generate composition plans conform to the requests. This phase checks *purpose composability* and *category composability* between a composite service operation and a component service operation. It checks also *quality composability* and *message compatibility* (leading to a one-to-one mapping between messages).
3. Selection: if more than one plan is generated, a selection is made based on quality of composition parameters like ranking, relevance (also called soundness) and completeness;
4. Generation phase.

The composability rules checks different aspects of the composition that are equivalent to the assembly line composition and relay race composition in our approach and they are expressed and checked by the reasoner during the answering of a query.

5.2 Web Service Composition via Planning

There are many research tackling Web service composition problem via planning. A planning problem can be described by a set S of all possible states of the world, a set S_0 of initial state, a set G of goals states the planning system attempts to reach. a set A of actions planner can perform changing one state to another in the world and a relation $\Gamma \subset S \times A \times S$ which defines the preconditions and effects for the execution of each action. In the terms of Web services, S_0 (resp. G) is the initial states (resp. the goal states) specified in the requirement of Web services. A represents a set of available services and Γ denotes the state change function of each service.

An automated composition task using planning methods is also used in [25] that build a new composed service (*i.e.* the state transition system), given the OWL-S process model description of n available services.

A state transition system is a 6-tuple:

1. a set S of states;
2. a set A of actions;
3. a set O of observations;
4. a set I of initial states, subset of S ($I \neq \emptyset$);
5. a transition function $T : S \times A \mapsto 2^S$ associating the set of next states $T(s, a)$ after applying an action a to a state s;
6. an observation function $X : S \mapsto O$.

A Composition goals express requirements for the service to be automatically generated. They should represent conditions on the temporary evolution of services, and, as shown by the next example, requirements of different strengths and preference conditions.

The planner in charge of the composition has two inputs: the composition goal and the planning domain Σ which represents all the ways in which the services represented by $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ can evolve. The automated composition task consists in finding a plan that satisfies the composition goal G over a domain Σ. A plan may encode sequential, conditional and iterative behaviors. Therefore a plan is modelled as an automaton:

A plan for planning domain $\Sigma = \{S, A, O, I, T, X\}$ is a 4-tuple where:

- C is the set of plan contexts.
- $c_0 \in C$ is the initial context.
- $\alpha : C \times O \mapsto A$ is the action function; it associates to a plan context c and an observation o an action $a = \alpha(c, o)$ to be executed.
- $\epsilon : C \times O \mapsto C$ is the context evolutions function; it associates to a plan context c and an observation o a new plan context $c = (c, o)$.

The generation of a new composition is achieved by translating a process models to non-deterministic and partially observable state transition systems and by generating automatically a plan that can express conditional and iterative behaviours of the composition.

5.3 Logic based language for Web services composition

In this section, we compare our work to situation calculus and some existing works that extend it.

From our approach, a functionality can be seen as an action with parameters, preconditions and effects on the current state. The situation calculus [14] gives a representation of actions and permits to detail preconditions and effects like the following. First, one rule for the action preconditions that tells if an action is executable in a given situation:

$$Poss(a(\vec{\imath}), s) \leftrightarrow Prec(\vec{\imath}, s)$$

where $Prec(\vec{\imath}, s)$ is the preconditions of the action. Then there are as many rules as fluents to change with the performance of the action:

$$Poss(a(\vec{\imath}), s) \rightarrow F(\vec{x}, do(a(\vec{\imath}), s))$$

where $do(a(\vec{\imath}), s))$ denotes the new situation resulting from the performance of an action $a(\vec{\imath})$ on a situation s; where $F(\vec{x}, s')$ denotes the fluents that undergo the effects of the action.

Some existing works [15, 18] extend situation calculus to make Web service composition. The authors conceive a Web service as either a primitive action or a complex action and therefore describe a Web service down to its components. An adapted version of Golog [12] deductive machinery is used to address the Web service composition problem. Golog is a high-level logic programming language built on top of the situation calculus, a first-order logical language for reasoning about action and change. They extend Golog to allow Golog programs to be used by a variety of different users. For instance, making a travel plans is a common task, yet it is difficult to task another person — to task a computer even more so — to make it for you due to your individual constraints and preferences. The adaptation and extension of the logic programming language in [15] are designed to allow programs to be written but yet being customisable by indivual users with their own constraints. In their approach a composite service is a set of atomic services which are connected using procedural constructs (if-then-else, while *etc.*).

A set of extra-logical constructs to assemble primitive actions, which are defined in the situation calculus, into complex actions are the following;

- a : primitive actions
- $\delta_1; \delta_2$: sequences
- $\phi?$: tests
- $\delta_1 | \delta_2$: non deterministic choice of actions
- $(\pi x)\delta(x)$: non deterministic choice of acrguments
- δ^* : non deterministic iteration
- $\textbf{if} \quad \phi \quad \textbf{then} \quad \delta_1 \quad \textbf{else} \quad \delta_2 \quad \textbf{endif} = [\phi?; \delta_1] | [\neg\phi?; \delta_2]$: conditionals
- $\textbf{while} \quad \phi \quad \textbf{do} \quad \delta \quad \textbf{endwhile} = [\phi?; \delta]^*; \neg\phi?$: loops

Given a domain theory D and a program δ written in the language of the domain theory D and using the above constructs, program execution must find a sequence of actions $\vec{a}$ such that:

$$D \models do(\delta, S_0, do(\vec{a}, S_0))$$

where $do(\vec{a}, S_0)$ stands for $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$

In addition to the fluent $Poss(a, s)$ in situation calculus which expresses that the action a is physically possible in the situation s, they define $Desirable(a, s)$ to constrains the search space for actions.

In the following we will show how our approach can be adapted to comply with situation calculus. In our approach, if we want to represent functionalities associated with devices as actions from the situation calculus, an adaptation of the function symbol do is needed:

$$do(f(d), [I_1, \ldots, I_n], s)$$

denotes the new situation resulting from the performance of the functionality of the device d, denoted by $f(d)$, fed with the inputs $I_1, \ldots, I_n$. $f(d)$ is a skolem function with a function symbol f. While the function symbol f has no semantics, the semantics of $f(d)$ is defined by associating it with a class name in a taxonomy of functionalities.

The same way an adaptation of the predicat $Poss$ in order to reflect the representation of the functionalities is needed:

$$Poss(f(d), [I_1, \ldots, I_n], s)$$

Some functionalities have inherently a result that depends on their inputs. Therefore, it is very useful to link the result of a functionality with the inputs. For instance, when a compilation process transforms a source file into a file comprehended by the computer, we have to keep the piece of information that the compiled file comes from a particular source. Furthermore, to type the result of the functionality over a given situation there is a function symbol out (that refers to 'output'):

$$\mathrm{out}(f(d), [I_1, \ldots, I_n], s)$$

which denotes the output of the functionality $f(d)$ of the device d fed with the inputs $I_1, \ldots, I_n$ and performed in a situation s.

The introduction of the function do together with the situation permit us to express the assembly line composition. The assembly line composition can be seen as a sequence of functionalities (actions) where there is possibly no data exchanged between them. The following example illustrates the expression of such a composition:

Imagine a mobile phone which provides the functionality of sending a short message to another mobile phone knowing its number. It may be the case that more than one carrier can be used to send the message, each one billing different prices depending on the date, the time of the day and the plan subscribed to the carrier. So the phone can select a carrier. The short messages service functionality gets the number of a mobile phone and a short message to send. As a precondition, the carrier has to be selected in order to send the message. Thus, we have a mobile phone $\mathtt{phn_1}$ with two functionalities:

- $f_1(\mathtt{phn_1})$ sends a short messages to a phone number;
- $f_2(\mathtt{phn_1})$ provides the cheapest carrier among the carriers available, and as a side-effect the carrier is selected.

Now suppose we have a Personal Digital Assistant (PDA) which contains our address book and which has a functionality to retrieve someone's phone number knowing its name. We want to express the relay race composition which is done with the function out as seen in section 2.4, and the assembly line composition which is done with the function do as we are going to see it now.

First we select the carrier:

$$s_1 = do(f_2(\mathtt{phn_1}), [], s_0)$$

Next, in the situation s_1 where the carrier is selected, we look for the phone number in the PDA. This is the expression of the phone number:

$$N = \mathrm{out}(f_1(\mathtt{pda_1}), [\text{"Peter"}, \text{"Zweistein"}], s_1)$$

Yet, the new situation is the following:

$$s_2 = do(f_1(\mathtt{pda_1}), [\text{"Peter"}, \text{"Zweistein"}], s_1)$$

Therefore, the fact that the message is sent is expressed by the following statement:

$$\mathrm{out}(f_1(\mathtt{phn_1}), [N, \text{"Hello!"}], s_2)$$

This extension permits us to express the assembly line composition while situation calculus in itself has no mean to express relay race composition.

5.4 Conclusion

In the context of an opportunistic network, we address three problems that come along with dynamic composition.

1. To express a composition of functionalities that meet certain criteria such as the type of the output, the properties of the devices, *etc*.
2. To look for functionalities or devices available through the opportunistic network.
3. To describe devices and their functionalities to make the search and the composition possible.

We have built a logical class-based language upon the first-order logic to describe the devices and the functionalities. Every object (devices, functionalities, data, *etc*) is assigned to a set of user-defined classes which are organised in taxonomies. An important characteristic of the language is the use of functions to link an output of a functionality to its input. It follows that composition of functionalities are naturally expressed in the language by composition of functions.

The first-order logical class-based language makes description be a knowledge base upon which we can reason. Then, thanks to a reasoner, searching for devices (resp. functionalities) according to their properties (location, type, functionalities' type, *etc*.) (resp. input types, output types, *etc*.) is achieved by answering to queries.

The main effect of the combination of the language and a reasoner lies in the compositions of functionalities being directly provided as answers to queries. As a composed functionality is considered as a functionality, an answer to a query looking for a functionality can be indiscriminately a simple functionality or a composed functionality.

As there is no fully distributed PROLOG to shift the problem of dynamic composition in an opportunistic network, we addressed this problem by breaking it down to two steps. According to a query, a look up process through the network is made thanks to SOMEWHERE. This demands an encoding of the description and the queries into propositional logic. The look up provides a list of devices' description that are proved to be sufficient, once retrieved, for the second step which is to compute locally the answer of the query.

As perspective, we outline three future works that would worth to be investigated. First, experimentation of the decentralised deployment in a real case and impact on the performance due to an heterogeneous description can rely on the experimentation made on SOMEWHERE [1] that shows the scalability of the approach. Their consequence finding algorithm, deployed on a real cluster of heterogeneous computers, scales well even on queries that solicit a large number of neighbours to be processed and even on networks with lots of shared variables between couple of neighbours.

Then, an extension of the description language can be investigate to use negation. It would be interesting

- to expect that a certain property does not hold, in a precondition of a functionality;
- to express that a device does not belong to a certain class;

If negation is treated by failure, the reasoner will wrongly answers to some queries involving unbound variables as shown in the following minimalistic example. "Alice is a woman" is expressed by the first statement, and "A man is someone who is not a woman" is expressed by the second statement.

$$woman(alice).$$
$$man(X) \ :- \neg woman(X).$$

Therefore, even if $man(bob)$ succeeds, $man(X)$ will fail because $woman(X)$ succeeds. Negation as failure rely on the close world assumption but the knowledge base in our case is not complete. In our case, the order of evaluation of the predicate is not significant. So a possible workaround could be to postpone the evaluation of a negation until all the variables involved are instantiated.

Finally, a decentralised PROLOG to compute answers to a query on a distributed Prolog program can be an improvement. Instead of retrieving relevant description and compute answers locally, we could try to partially answer to a query and ask the neighbourhood about the unresolved part of the query and gather all this partial answers to build up a complete answer to the initial query. Another approach could be to extend SOMEWHERE algorithm to first-order logic.

Appendix A

Glossary

CSSL Composite Service Specification Language 66
MGU Most General Unifier 42
OWL Web Ontology Language 64
P2P Peer-to-peer 1
PDA Personal Digital Assistant 69
PDF Portable Document File 4
RDF Resource Document Framework 6
RDFS Resource Document Framework Schema 7
SPARQL SPARQL Protocol and RDF Query Language
URI Uniform Resource Identifier 64
WSC Web service composition
WSD Web service discovery
XML eXtensible Markup Language 64

Appendix B

Index

A

Assembly line composition, *see* Composition

C

Composition, 5, 7, 20, 27
- assembly line, 17, 66, 69, 70
- ordered tree of, 16–18
- relay race, 16, 17, 52, 66, 69, 70

Composite Service Specification Language, 66

D

Derivation tree, 42, 43, 46, 47, 55–57, 59–61
Description
- First-order, 40, 57, 62

Devices
- Display, 4
- Mixing console, 3
- Printer, 4
- Screen, 4
- Webcam, 3

F

First-order, *see* Description, *see* Logic
Functionality
- Assertion, 13, **17**
- Signature, 12, **16**, 21
- Virtual functionality, 4, 5, 21, 22

G

Graph
- of connections, 26, 27, 29, 31, 33, 34, 37

H

Heterogeneous descriptions, 22, 62
High-level language, 8, 9, 11, 12, 14, 15, 17–21, 61

J

JAVA, 62

L

Logic
- First-order, 5, 6, 8–10, 14, 15, 21–25, 40, 42, 50, 61, 66, 71, 72
- Propositional, 5, 40, 48, 50, 61, 62, 71

M

Most General Unifier, 42, 56

O

Opportunistic network, 2, 3, 5, 6, 62, 71
Ordered tree of composition, *see* Composition
OWL, 64, 67

P

Peer-to-peer, 1, 39
Personal Digital Assistant, 69, 70
Portable Document File, 4, 21
Post-condition, 5, 10, 11, 13–15, 17, 41, 65
Precondition, 5, 11, 13–17, 41, 67–69, 71
PROLOG, 5–7, 25, 26, 28, 40, 46, 61, 62, 71, 72
- XSB, 6, 26, 28–37

R

Resource Document Framework, 6, 12, 64, 65, 73
Resource Document Framework Schema, 7–10, 64
Relay race composition, *see* Composition

S

Skolem, 16, 40, 69
SOMEWHERE, 5, 40, 48, 59, 61, 62, 71, 72

T

Taxonomy, **5**, 44

U

Uniform Resource Identifier, 64

V

Variable of interest, 18–20, 42
Virtual object, 20
    Virtual device, 4, 5, 20, 22
    Virtual functionality, *see* Functionality

W

Web service composition, 63, 64, 67, 68
Web service discovery, 63

X

eXtensible Markup Language, 64
XSB, *see* Prolog

Appendix C

Bibliography

[1] Philippe Adjiman, Philippe Chatalic, François Goasdoué, Marie-Christine Rousset, and Laurent Simon. Scalability study of peer-to-peer consequence finding. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 351–356. Professional Book Center, 2005.

[2] Philippe Adjiman, Francois Goasdoué, and Marie-Christine Rousset. SOMERDFS in the semantic web. *Journal on Data Semantics*, 8, 2007.

[3] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services, March 2003.

[4] Mahmoud Barhamgi, Pierre-Antoine Champin, Djamal Benslimane, and Aris Ouksel. Composing Data-Providing Web Services in P2P-based Collaboration Environments. In Springer, editor, *19th International Conference on Advanced Information Systems Engineering (CAiSE'07)*, pages 513–545. Springer, June 2007.

[5] Tim Berners-Lee and Eric Miller. The semantic web lifts off. *ERCIM News*, 51:9–10, October 2002.

[6] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated web service composition by on-the-fly belief space search. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *ICAPS*, pages 358–361. AAAI, 2006.

[7] Mike Clark, Peter Fletcher, J. Jeffrey Hanson, Romin Irani, Mark Waterhouse, and Jorgen Thelin. *Web Services Business Strategies and Architectures*. Wrox Press, 2002.

[8] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[9] Mike Dean et al. Owl web ontology language reference. Online. http://www.w3.org/TR/owl-ref/.

[10] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *ADC '03: Proceedings of the 14th Australasian database conference*, pages 191–200, Darlinghurst, Australia, 2003. Australian Computer Society, Inc.

[11] Graham Klyne and Jeremy J. Carroll. Resource description framework: Concepts and abstract syntax. Online. http://www.w3.org/TR/rdf-concepts/.

[12] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, Richard B. Scherl, and Richard B. Golog: A logic programming language for dynamic domains, 1994.

[13] Zakaria Maamar, Djamal Benslimane, and Nanjangud C. Narendra. What can context do for web services? *Communications of the ACM*, 49(12):98–103.

[14] John McCarthy. Situations, actions and causal laws. *Semantic Information Processing*, pages 410–417, 1968.

[15] Shelia McIlraith and Tran Cao Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, pages 482–493, 2002.

[16] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing web services on the semantic web. *The VLDB Journal*, 12, 2003.

[17] Michael Mrissa, Chirine Ghedira, Djamal Benslimane, and Zakaria Maamar. Context and semantic composition of web services. pages 266–275, 2006.

[18] Srini Narayanan and Sheila McIlraith. Simulation, verification and automated composition of web services. 11th international conference on World Wide Web, 2002.

[19] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54, San Diego, USA, 2004. Springer.

[20] Marie-Christine Rousset, Philippe Adjiman, Philippe Chatalic, François Goasdoué, and Laurent Simon. SOMEWHERE in the semantic web. SOFSEM 2006 (International Conference on Current Trends in Theory and Practice of Computer Science), january 2006.

[21] Konstantinos Sagonas and Terrance Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM Trans. Program. Lang. Syst.*, 20(3):586–634, 1998.

[22] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 442–453, New York, NY, USA, 1994. ACM.

[23] Konstantinos Sagonas, David S. Warren, David Warren, Steve Dawson, Steve Dawson, Juliana Freire, Juliana Freire, Baoqiu Cui, Michael Kifer, Michael Kifer, Terrance Swift, Terrance Swift, Terrance Swift, Kostis Sagonas, Prasad Rao, Prasad Rao, Prasad Rao, and Juliana Freire The Breakdown. The xsb system version 2.2 volume 1: Programmer's manual, 2000.

[24] Michael Siegel, Edward Sciore, and Arnon Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems*, 19:254–290, 1994.

[25] Paolo Traverso and Marco Pistore. Automated composition of semantic web services into executable processes. pages 380–394, 2004.

[26] Moe Thandar Tut and David Edmond. The use of patterns in service composition. In Christoph Bussler, Richard Hull, Sheila A. McIlraith, Maria E. Orlowska, Barbara Pernici, and Jian Yang, editors, *WES*, volume 2512 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 2002.

Résumé

Dans des réseaux opportunistes — dont la topologie est dynamique — de dispositifs intelligents, nous traitons le problème de la recherche et de la composition dynamique de fonctionnalités à l'aide d'une description logique des dispositifs. Nous définissons un langage logique du premier ordre dans lequel les dispositifs, leurs fonctionnalités et leur propriétés sont exprimés, en utilisant une taxonomie de classes pour contraindre le type des ressources. Nous définissons conjointement un langage de requêtes basé sur celui de la description nous permettant d'utiliser des raisonneurs du type PROLOG pour répondre aux requêtes. Ces réponses sont des instanciations des variables d'intérêt présentes dans la requête et représentent des constructions de fonctionnalités composées. Dans un second temps, pour faire face aux spécificités d'un réseau dynamique, nous utilisons la plateforme SOMEWHERE — qui permet de faire du raisonnement en logique propositionnel lorsque la base de connaissances est totalement distribuée — comme un service de *lookup* récupérant un sous-ensembles des ressources dont les descriptions sont nécessaires à l'obtention de toutes les réponses à la requête donnée via le raisonneur. Dans cette optique, nous utilisons un encodage des descriptions et de la requête écrites en logique du premier ordre vers la logique propositionnelle qui conserve les bonnes propriétés de la description.

Mots-clés: composition, réseau opportuniste, Web sémantique

Abstract

In opportunistic networks — which topology is dynamic — of smart devices, we address the problem of looking for functionalities and of building a composition of functionalities with a logical description of the devices. We define a first order logic language in which the devices, their functionalities and their properties are expressed using taxonomies of classes to constrain the type of resources. We jointly define a query language based on the description language, allowing us to use a reasoner like PROLOG to answer to the queries. Those answers are instanciations of the variables of interest which belong to the query. They represent composed functionalities. Next, to deal with the dynamicity of the network, the platform SOMEWHERE — that make reasoning in propositional logic in a totally distributed manner — is used as a lookup service retrieving a subset of resources which descriptions are necessary to obtain all the answers to a certain query. For that purpose, we use an encoding of the descriptions and of the query, both written in first order logic, into propositional logic so that some good properties of the descriptions are kept.

Keywords: composition, opportunistic networks, semantic Web